IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 11, 2019, accepted March 12, 2019, date of publication March 15, 2019, date of current version April 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2905428

FEther: An Extensible Definitional Interpreter for Smart-Contract Verifications in Coq

ZHENG YANG, (Member, IEEE), AND HANG LEI
School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China
Corresponding author: Zheng Yang (zyang.uestc@gmail.com)

ABSTRACT Recently, blockchain technology has been widely applied in the financial field. Therefore, the security of the blockchain smart contracts is among the most popular contemporary research topics. To improve the theorem-proving technology in this field, we are developing an extensible hybrid verification proof engine, denoted as FEther, for Ethereum smart contract verification. Based on Lolisa, which is a large subset of solidity mechanized in Coq, FEther guarantees the consistency between smart contracts and its formal model. Combining symbolic execution with higher order logic theorem-proving, FEther contains a set of automatic strategies that execute and verify the smart contracts in Coq with a high level of automation. Besides, in FEther, the verified code segments also can be reused to assist in the verification of other properties. The functional correctness of FEther was verified in Coq. The execution efficiency of FEther has far exceeded that of the interpreters that are developed in Coq in accordance with the standard tutorial. To the best of our knowledge, FEther is the first definitional interpreter of the solidity language in Coq.

INDEX TERMS Symbolic execution, formal verification, smart contract, Coq, etheruem, definitional interpreter.

I. INTRODUCTION

Blockchain technology [1], which adds records to a list using cryptographic links, is among the most popular contemporary technologies. Ethereum is a widely adopted blockchain system that implements a general-purpose, Turing-complete programing language called Solidity [2]. Ethereum enables the development of arbitrary smart contracts that can automate blockchain transactions in a virtual runtime environment, namely, the Ethereum Virtual Machine (EVM). Here smart contracts refer to the applications and scripts (i.e., programs) that execute the blockchain. The growing use of smart contracts has necessitated increased scrutiny of their security. Smart contracts can include particular properties that expose them to deliberate attacks causing direct economic loss. Some of the largest attacks on smart contracts are well known, such as the attacks on decentralized autonomous organizations [3] and parity wallet contracts [4]. Many classes of subtle bugs, ranging from transaction-ordering dependencies to mishandled exceptions, exist in smart contracts [5]. Therefore, the security and reliability of smart-contract programs must be verified as rigorously as possible. The properties of programs can be rigorously verified by proving higher-order logic theorems. In the standard approach, a formal model for the target software system is manually abstracted using higher-order theorem-proving assistants. Such formal verification technology provides sufficient freedom and flexibility for designing formal models based on higher-order logic theories, and can abstract and express very complex systems. However, when applied to smart contract verification, the advantages of theorem-proving technology are suppressed by automation, reusability, consistency and efficiency problems.

The above issues can be resolved by a formal symbolic process virtual machine (FSPVM) [6], which directly and symbolically executes real-world smart-contract programs using higher-order theorem-proving assistants. The program's properties are then automatically verified by the execution result. To this end, we are developing an FSPVM named FSPVM-E [7] for smart contracts deployed on the Ethereum platform. FSPVM-E is programed in Coq (a formal proof-management system) and inspired by KLEE, a high-coverage test generator for complex-systems programs [8]. Similarly to [9], the symbolic execution of FSPVM-E is verified in FEther, a hybrid proof engine that supports multiple types of symbolic execution. FEther, however, is designed for higher-order theorem proving, and its verification process is

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

founded on Hoare [10] and reachability [11] logic. Therefore, the successful implementation of an FSPVM must overcome several challenges [6].

Some of these challenges have been addressed in our recent studies. In [6], we noted the lack of a versatile formal memory model for constructing the logic operating environment within a higher-order theorem-proving system. We thus developed a general, extensible, and reusable formal memory (GERM) framework based on higher-order logic using Coq [12]. In our previous work, we presented an application extension of Curry-Howard isomorphism (CHI) [13] to combines theorem proving and symbolic execution technology. Herein, it is denoted as execution-verification isomorphism (EVI) and takes the basic theory of FSPVM.

Finally, we developed an extensible large subset of the Solidity programing language, denoted as Lolisa [14], which equivalently formalizes real-world programing languages as an extensible intermediate programing language.

The present paper completes the FSPVM-E by overcoming the final challenge: developing its proof engine. Our contributions are as follows. First, we develop a definitional interpreter in Coq's specification language (Gallina). This interpreter symbolically executes the smart contracts of Ethereum written in Lolisa on the GERM framework. The execution results are represented by a GERM logic memory state, which can be verified in Coq. Next, we implement a set of automatic evaluation strategies based on the Ltac [12] mechanism, by which FEther finishes the execution and verification process. The correctness of FEther is then certified in Coq. The present FEther is the optimized version with higher evaluation efficiency than the interpreters developed in Coq using standard tutorial approaches. To our knowledge, FEther is the first hybrid proof engine specification that automatically and symbolically executes and verifies Ethereum smart contracts in Coq.

The remainder of this paper is structured as follows. Section 2 describes the difference between the FEther and other relevant works. Section 3 introduces the foundations of the present work, including the prototype, the basic environment of Lolisa, and the preparatory modification of the GERM framework. Section 4 describes the theoretical design and implementation of FEther, and its self-correctness certification. Section 5 verifies FEther in a real-world case study and analyzes its benefits. Section 6 discusses the advantages and limitations of FEther. The study concludes with Section 7.

II. RELATED WORK

The security of smart contracts has been seriously researched since 2015. The security of smart contracts and similar lightweight programs can be rigorously guaranteed by formal methods. Our symbolic executor has several novel features that distinguish it from other approaches. This section introduces the interesting achievements already reported in this field.

The EVM execution is formally described in Yellow Paper [15]. This official document also provides the data, algorithms, and parameters required for building consensus-compatible EVM clients and Ethereum implementations. Yellow Paper, however, does not always clarify the operational behavior of the EVM. In such cases, it is often easier to consult an executable implementation for guidance.

Most of the recent researches have concentrated on EVM security. The C++ implementation Cpp-ethereum plays a dual role of security and defector semantics in EVMs. Lem semantics [16] is a Lem [17] implementation of EVM providing executable semantics of EVM, which formally verifies smart contracts. However, the Lem semantics do not precisely capture the inter-contract execution. KEVM [18] is a formal semantics for EVMs, resembling Lem but written in the K-framework. As KEVM is executable, it can run the validation test suite provided by the Ethereum foundation. According to Hildenbrandt *et al.* [18], the KEVM reference interpreter passes the full 40,683-test EVM compliance suite. Nevertheless, self-correctness cannot be proven completely or even certified in KEVM. Moreover, none of the above approaches satisfies the de Bruijn criterion [19].

Mythril [20] is a security analysis tool for Ethereum smart contracts. Mythril detects various problems by concolic analysis, but whether the tool effectively increases the reliability of smart contracts has not been proven.

The above researches adopt the bytecode of Solidity. The consistency between Solidity and bytecode after compiling cannot be guaranteed. However, high-level formal specifications and relevant formal verification tools of Solidity, which are important for programing and debugging smart contract software, have received little attention.

Finally, some of these works focus on a specific domain. Their complex architecture is inflexible and not easily extendible to new relevant problems.

III. FOUNDATIONAL CONCEPTS AND DEFINITIONS

The present paper builds upon our recent previous works. Therefore, prior to defining the formal specifications of FEther, we first define the basic environment.

A. PREDEFINITIONS

To be specific, first of all, the EVI is an application extension of CHI. Briefly, CHI proposes that a deep correspondence exists between the worlds of logic and computation. To avoid ambiguity in the following discussion of EVI, we use *program* to represent programs written in Gallina or a similar specification language based on CHI and $program_{frw}$ to represent the formal version of real-world programs written in Lolisa. This correspondence can be expressed according to three general principles as follows:

types correspond to propositions;
proofs correspond to a program;
proofs correspond to the evaluation of programs.

The deep correspondences applied in CHI make it very useful for unifying formal proofs and *program* computation. The that *programs*, implemented in the proof assistants supporting CHI such as Coq, are first-class citizens of their logic can be directly evaluated in higher-order theorem proving system that supports CHI, and this process corresponds to the propositions proof of *programs* in corresponding theorems. However, most mainstream programming languages are not designed based on lambda calculus and cannot be analyzed in a higher-order logic environment. Programs written in these languages are very difficult or impossible to verify directly and automatically using CHI. The execution engine, such as FEther that executes $program_{frw}$, is developed in a formal system that supports CHI based on formal operational semantics and is a kind of special *program*; this forms the basis of EVI, which extends the formal relations of CHI to include three corollaries: *evaluation of program corresponds to execution of* $program_{frw}$, *properties correspond to types*, *and verifications correspond to proofs*. Based on these corollaries, the correspondences of CHI can be extended to obtain a fourth general principle: *verifications correspond to execution of* $program_{frw}$, denoted as EVI. More details are introduced in [6].

TABLE 1. Basic memory-management APIs employed in the formal memory model.

Function	Description
$read_{dir}$	Read m_{value} from a directly
$read_{chck}$	Read m_{value} from a after validation checking
$write_{dir}$	Write m_{value} at a directly
$write_{chck}$	Write m_{value} at a after validation checking
$address_{offset}$	Offset address a to a'
$allocate$	Allocate memory blocks
$free_{mem}$	Free a specified memory block
$init_{mem}$	Initialize the entire memory space

GERM The GERM is a general, extensible, and reusable formal memory framework. It simultaneously supports different formal verification specifications, particularly at the code level. This framework simulates the physical memory hardware structure, including a low-level formal memory space, and provides a set of simple, nonintrusive application programing interfaces (APIs). The proposed GERM framework is independent and customizable, and was verified entirely in Coq. Table 1 summarizes the top-level interface of FEther, where $m_{value}m_{value}$ and a represent a memory value of type *value* and a memory address of type $L_{address}$, respectively. In the specific formal specification, a formal memory state has type *memory*. Finally, b_{infor} represents the block information for environment checking.

Lolisa The $\mathcal{FL}$ is Lolisa, a large subset of the Solidity programing language. Assisted by generalized algebraic datatypes (GADTs) [21], the formal syntax of Lolisa adopts a stronger static-type system than Solidity, which enhances the type safety. Lolisa includes contract declarations (*Contract*), modifier declarations (*Modifier*), variable declarations (*Var*), structure declarations (*Struct*), assignments (*Assign*), returns (*Return*), multi-value returns (*Returns*), throws (*Throw*), skips (*Snil*), function definitions (*Fun*), while loops ($Loop_{while}$), for loops ($Loop_{for}$), function calls (Fun_{call}), and conditional (*If*) and sequence statements.

TABLE 2. State functions in the dynamic semantic definitions.

$\mathcal{E}$	environment information	$\mathcal{F}$	formal system world
M	memory space	Γ	proof context

TABLE 3. Helper functions.

set_{env}	Changes the current environment	env_{check}	Validates the current environment
id_{search}	Searches the address of array elements	$init_{var}$	Initializes variable into memory block
id_{map}	Searches the address of mapping elements	$eval_{str}$	Evaluates the struct datatype
$eval_{bop}$	Evaluates binary expression of Lolisa value	$eval_{uop}$	Evaluates unary expression of Lolisa value

Type level ⇝ *Declare type annotation*
Value level ⇝ *Represent struct value & field value*
Expression level ⇝ *Represent right value of struct type*
Statement level ⇝ *Declare new struct type*

Convention 1. Formal *Struct* datatypes.

Table 2 summarizes the helper states used in the dynamic semantic definitions, and Table 3 lists the helper functions for calculating commonly needed values in the current program state. All of these state functions will be encountered in the following discussion. The components of specific states will be denoted by appropriate Greek letters subscripted by the state of interest. In Table 2, M, σ, and $\mathcal{E}$ denote the contexts of the formal memory space, a specific memory state, and the execution environment, respectively. The proof evaluation is executed in the proof contexts, denoted as Γ, Γ_1, ...For brevity, we hereafter represent the overall formal system by $\mathcal{F}$, the current execution environment by *env*, and the super-environment of type *env* by *fenv*.

In the following sections, we introduce the relevant analysis and solutions that improve the computation efficiency of $\mathcal{FI}$ in higher-order theorem-proving assistants.

To simplify the verification process and the development of the respective formal verified Lolisa interpreter in Coq, we maintained the Lolisa programs as structural programs. For this purpose, the semantics of Lolisa were forced to adhere to the following *Program Counter* axiom. The conventions of the *Struct* datatypes are defined in Convention 1. The *Program Counter* axiom is a FEther design principle that maintains Lolisa as a structural language.

Axiom (Program Counter): If statement s is the next execution statement, it must be the head of the statement sequence in the next iteration.

To avoid infinite loops in the programs, FSPVM also imports bounded model checking (BMC) [22]. Fortunately, the EVM does not support infinite execution processes, as each execution step costs the *gas* of the smart contract owners. If the *gas* balance cannot satisfy the limitation,

TABLE 4. Encapsulation functions in the optimized FEther.

$mems_{find}$	Searches the valid struct fields	$eval_{head}$	Evaluates the struct value basic information
$lexpr_{check}$	Checks the validation of the l-position expressions	get_{bool}	Gets the Boolean value from memory value
$valid_{array}$	Check the declaration of array type	get_{stt}	Gets the statement value from memory value

the execution terminates. This design well suits the BMC concept. Therefore, our implementation uses *gas* to limit the execution of the Solidity programs.

In the following contents, we represent other arguments by the wildcard "$*$" and by the symbol $|* \longmapsto *|\}$. E is the syntax of constructor pattern matching of the λ-expression E [23]. To avoid ambiguity in the following discussion of FEther, the functions represent the programs and functions written in Gallina, and *RWprogram* represents the real-world programs written in general-purpose programing languages.

B. MODIFCATIONS FOR OPTIMIZATION

As mentioned previously, when analyzing the current problems, the computational efficiency of the definitional interpreter based on the FSPVM may be extremely low. The three essential problems are *call-by-name termination*(CBNT), *information redundancy explosion* (IRE), and *concurrent reduction* (CR). To optimize the low-level computations of the evaluation problems, we incorporate the respective solutions in [24] into the implementation details of FEther.

First, the sequence statement s is implicitly replaced by an equivalent list rather than explicitly defined, which avoids the CBNT problem. Second, the pattern matchings and reusable functions are encapsulated as optimization helper functions. Some of these functions are summarized in Table 4. To avoid the CR problem, we finally impose a limitation K (independent of the *gas* constraint) on the expression and value layers.

IV. FETHER IMPLEMENTATION

FEther is the bridge that connects the GERM framework, the Lolisa programing language, and the trusted core of Coq (TCOC). As demonstrated in our previous work and elaborated in the following subsections, FEther can be totally built in Coq.

A. ARCHITECTURE

Figure 1 shows the overall structure of the FEther framework. The whole FEther is constructed in the trusted domain of Coq, and logically comprises three main components: a parser, an ISA based on Lolisa semantics, and a validation checking mechanism (see left, center and right blocks in Figure. 1, respectively). The parser analyzes the syntax of the *FRWprograms* written in Lolisa. According to EVI theory, FEther is essentially a huge function written in Gallina. In this sense, it differs from the real-world virtual machines of high-level programing languages such as Smalltalk, Java, and .Net, which support bytecode as their ISA and are implemented by translating the bytecodes of commonly used code paths into native machine code. Instead, the ISA of FEther comprises the Lolisa semantics, which specify the semantics of the syntax tokens that govern the respective behaviors. The validation checking mechanism includes two parts: checking the result validation (including the memory states and values), and checking the execution condition. First, because all functions are vulnerable to undefined situations, they are developed with the help of effect programing. More specifically, all functions are tagged by an optional type. A valid result is returned in the form *Some* t; an invalid result is returned as an undefined value *None*. The symbol $[\![t]\!]$ denotes that term t is tagged by an optional type. In the second part, the *gas* and K limitations are validated by the helper functions env_{check} and $pump_{check}$, respectively.

FEther inherits the low-coupling property from Lolisa. Within the same level, the executable semantics are wholly independent, and are encapsulated as modules connected by a set of interfaces. In different levels, the higher-level semantics can access the lower-level semantics only via the interfaces, and the implementation details of the lower-level semantics are transparent to the higher-level semantics (indicated by dotted lines in Figure 1). Moreover, the implementation of the higher-level semantics does not depend on the lower-level semantics.

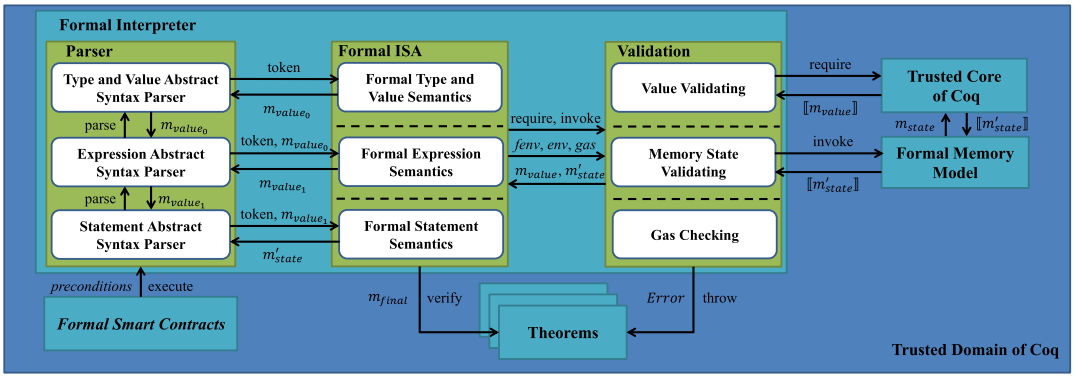


Figure 1. Architecture of FEther.

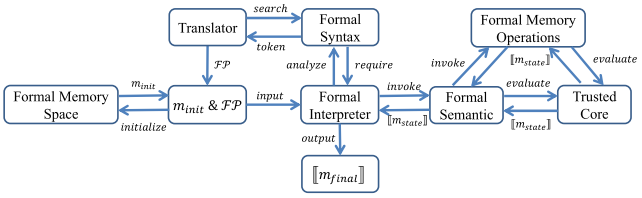


Figure 2. Workflow of FEther.

```
1function pledge(bytes32 _message) {
2    if (msg.value ==0 || complete || refunded) throw;
3    ...;pledges(numPledges) = Pledge(msg.value, msg.sender, _message);
4    numPledges++;
5}
```

(a)

```
1Definition fun_pledge :=
2(Fun (Some public) (Efun (Some pledge ) Tundef )
3  ( pcons (Epar (Some _message) Tbyte32) pnil))
4  ( nil )
5  (
6    (If ((SCField Tuint Msg (values ~> \\) None) (==) (SCInt 0)
7        (||) (SVBool complete) (||) (SVBool refunded))
8        (Throw;;;) (Snil;;;)
9    );;;
10  ...
11);;;
12.
```

(b)

Figure 3. Translation process from Solidity into Lolisa.

The workflow of FEther is outlined in Figure 2. The user first sets the initial memory state and the target *FRWprograms* by initializing the formal memory space of GERM and applying the translator. Note that the translator is an auxiliary component of Lolisa introduced in [14]. As shown in Figure 3, it converts the Solidity program of (a) into the FRWprograms written in Lolisa of (b) by searching the abstract syntax tree of Lolisa, binding the variable identifiers with a unique memory address, and declaring the ML modules. Next, the FEther parser analyzes the formal model of smart contracts according to the Lolisa abstract syntax tree, and invokes the respective executable semantics. The TCOC handles the evaluation requirements, and the results are validated by the validation mechanism. Although the validation module is logically independent of the other parts (as mentioned above), it is implemented separately in the *Formal Interpreter* and *Formal Semantic* modules in real cases. Therefore, the validation module is not explicitly defined in Figure 3. The final formal memory state will be assumed in the property theorems.

Lolisa is defined by GADTs, which guarantees well-formed constructions of the syntax specifications. Thus, the side conditions of syntax correctness do not need checking by help functions defined in FEther. The type safety can be checked by the type-checking mechanism of Coq. The complete workload of constructing a FEther framework with 100 memory blocks is itemized in Table 5.

B. INSTRUCTION SET ARCHITECTURE OF FETHER

The ISA of FEther is the core of the proof engine, which follows the big-step operational semantics of Lolisa. As shown in Figure 1, the FEther ISA is separated into three layers. FEther is implemented as described in Appendix A.

TABLE 5. Workload statistics for constructing the FEther framework with 100 memory blocks.

	Objects	Lines in Coq
Formal Type and Value layer	58	1,347
Formal Expression layer	46	1,503
Formal Statement layer	40	741
Automatic tactic	26	344
Correctness Lemmas	74	3,746
Total	244	7,681

Value layer This project aims to formalize a mechanized syntax and semantics for a subset of the Solidity language, which can be directly executed and verified in higher-order logic theorem-proving assistants. Therefore, the Solidity values must be evaluated like the native values in the formal system. Ideally, the values of Solidity or some mainstream high-level programing language would be explicitly employed in the formal system. Due to the strict typing system of the trusted core and the adoption of different paradigms, however, Gallina (Coq) does not directly support array, mapping, and other complex values. Therefore, we must define an interlayer between the values of the real-world language and the native values of the formal system. This interlayer directly represents the real world-values by an equivalent syntax, and translates them into the native values using formal semantics.

After evaluating the Lolisa value by formal executable semantics, the native value information is computed or derived in the base formal system, and the respective GERM memory values are determined. In the following sections, $\mathcal{ESV}$ represents the entry point of calling the value semantics, which is abstractedly defined by Rule 1 below.

$$\begin{aligned}\mathcal{ESV}\mathbb{Z} \rightarrow\ & (\forall \tau : type, val\,\tau) \\ & \rightarrow memory \rightarrow Blc \rightarrow Env \rightarrow option\ value \quad (1)\end{aligned}$$

Here, the metavariable val incorporates the Lolisa value val and the mapping value val_{map}. Apart from lacking definitions related to mapping type, val_{map} has the same static typing rule as val, so the two values can be combined. The mapping relation between each Lolisa value and its unique memory value is expressed as $\approx$.

First, we define the computational semantics of the constant values. In Lolisa, the constant values are the set of normal-form values, and the set of metavariables v_{const}. The v_{const} evaluation process generates the respective memory values for directly recording the native value information of the formal system. For example, consider the constant variable $Vbool\ (b)\ :\ Tbool\ \in\ v_{const}$. The computational semantics of $Vbool\ (b)$ are defined by Rule 2. We then define $\mathcal{ESV}_{const_{bool}}\left(n, env, b_{infor}\right) \approx Vbool\ (n)$.

$$\begin{aligned}&\mathcal{ESV}_{const_{bool}} \\ &\equiv \lambda\,(n : bool)\,.\lambda\,(env : Env)\,.\lambda\left(b_{infor} : Blc\right).Vbool\ (n) \\ &\Rightarrow Some\ Bool\left(n, env, b_{infor}\right) \quad (2)\end{aligned}$$

The computational semantics $\mathcal{ESV}_{const}$ are summarized in Table 11 of A. The correctness of $\mathcal{ESV}_{\text{const}}$ is certified by Theorem 1 (the constant-mapping theorem).

Theorem 1 (Constant Mapping): For all Lolisa values $v_{const}\ (n)$, environment values *env*, and block information b_{infor}, the mapping $\mathcal{ES}_{const}\left(n, env, b_{infor}\right) \approx v_{const}\ (n)$ holds.

We then define the semantics of the reference values (the array, mapping, structure and field access values), which are needed for accessing the formal memory space and match indexes. The respective values are defined as follows, and the semantics $\mathcal{ESV}_{array}$ of the array values are defined in Table 12 of A. Particularly, id_{search} is a subsidiary function for index evaluation. To be specific, the Lolisa language supports an *n*-dimensional array (by Rule 4), so id_{search} is a special subsidiary function that searches the memory block indexed by the current *n*-dimensional array index. id_{search} is also used in the $init_{var}$ function. Below we introduce the specific implementations of id_{search}. The abstract function of id_{search} implementations is given by Rule 3.

$$id_{search} :: type \rightarrow L_{address} \rightarrow emory \rightarrow Env \rightarrow option\ address \quad (3)$$

$$Tarray\left[id_0 Tarray\left[id_1 Tarray\left[\ldots\left[Tarrayid_n\tau_{final}\right]\right]\right]\right] \quad (4)$$

The mapping value is stored as a singly linked list structure in the GERM memory model, and in the form $Map : L_{address} \rightarrow option\ \left(prodvalue_{map}value\right) \rightarrow type_{map} \rightarrow type \rightarrow option\ address \rightarrow Env \rightarrow Blc \rightarrow value$ in Lolisa. In the above expression, the first parameter stores the initial address, the second parameter stores the paired key and indexed values, the third and fourth parameters record the key value and indexed value types, respectively, and the fifth parameter represents the next address. The *Map* can be briefly abstracted as Figure 4. In this design, the structure supports the *n*-dimensional mapping datatype.

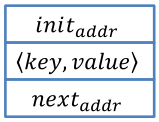


Figure 4. Abstract structure of a mapping-type memory block.

At the value level, a *Struct* memory value is represented by a struct datatype. Therefore, it resembles a normal-form value, and can be extracted directly by $read_{check}$ (see Table 14).

The semantics of field access are very flexible in Solidity and comprise contract member access and struct field access. If the contract member access is derived from an inheritance relationship or a special identifier, such as *this*, contract members can be directly accessed based on the ML module system. For the second part of the field access semantics, Lolisa supports all kinds of struct field access. However, as convention 1 is introduced, wherein middle members cannot be functions. To simulate the behavior of the functions for user node communication, such as *send* and *call*, these communication functions must modify the corresponding communication logic addresses. For example given as follows, the filed member *address* is an implicit parameter of *send* function.

$$pledges[i].address.send(pledges[i].amount) \equiv send(pledges[i].address, v, mss).$$

Therefore, in Table 15, if $mems_{find}$ successfully evaluates variable a_{init} with type a_{type}, it returns the pair (Dad, m_v), where *Dad* refers to the address of member mem_{n-1}.

Expression Layer The executable semantics of expressions are the rules that acquire the results of the value layer and evaluate the Lolisa expression in the memory value of GERM. The evaluation requires the left-value (l-value) and right-value positions, representing the memory addresses and the specific memory value, respectively. In the following contents, the entered pointer of the expression layer $\mathcal{ESE}$ is defined by Rules 5 and 6.

$$\mathcal{ESE}_l :: \mathbb{Z} \rightarrow (\forall \tau_0 \tau_1 : type, expr\,\tau_0\tau_1) \rightarrow memory \rightarrow Blc \rightarrow Env \rightarrow option\ L_{address} \quad (5)$$

$$\mathcal{ESE}_r :: \mathbb{Z} \rightarrow (\forall \tau_0 \tau_1 : type, expr\,\tau_0\tau_1) \rightarrow memory \rightarrow Blc \rightarrow Env \rightarrow option\ value \quad (6)$$

In formal Lolisa semantics, the modifier expression is a special one that cannot be evaluated in the expression layer (as will be later explained in the statement semantics). The computational semantics are defined as $\mathcal{ESE}_{lmodi} \equiv \mathcal{ESE}_{rmodi} \equiv Error$.

Expressions in the l-value position: The following rules define the semantics of evaluating the expressions in the l-value position (i.e., the corresponding memory address). The expressions in the l-value position, which can be constructed by the *Econst* constructor, represent Lolisa values at the expression level. Specifically, the left values can be assigned as the *Econst* specified by *Varray* and *Vmap*. As mentioned previously, *Varray* and *Vmap* are address pointers to values stored in specific memory blocks. For instance, as shown in rule 7, an array used as the left value in an assignment statement is commonly used in most general-purpose programing languages.

$$A\,[i] = a. \quad (7)$$

Thus, *Varray* and *Vmap* can represent not only memory values, but also memory addresses. Note that the remaining values (*Vstruct* and *Vfield*) are also address pointers by specifying the *Econst* constructor, but cannot represent the expressions in the l-value position. Because *Evar* can represent any variables address using any types, including structure and field access values. To avoid confusion between *Evar* and *Econst* specified by *Vstruct* and *Vfield*, we set the convention that *Vstruct* and *Vfield* represents only the memory value at the value level.

In both Solidity and Lolisa, *Vfield* admits many special structures, such as *msg* and *block*, whose members cannot

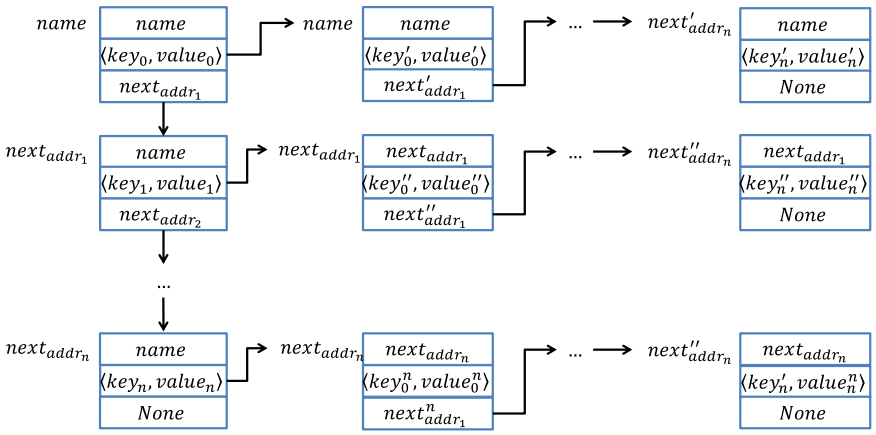


Figure 5. Structure of a 2-dimensional mapping value stored in GERM.

be changed at will. In rare cases, Solidity allows field-access expressions in the left position. Therefore, to ensure that Lolisa remains well-formed and well-behaved, the left value in expressions cannot be evaluated by *Vfield*. The fields of structures can be changed by invoking *Estruct* to change all fields, or by declaring a new field, as explained below. Although the limitations of *Vstruct* and *Vfield* are inconvenient for programmers and verifiers, they avoid any potential risk.

Besides, if the constructor is *Varray*, the semantics are chosen as $\mathcal{ESE}_{lexpr_{array}}$ (defined in Table 16); if the constructor is *Vmap*, they are chosen as $\mathcal{ESE}_{lexpr_{map}}$ (Table 17). The other constructors return *None* directly. The semantics of the left constant value $\mathcal{ESE}_{lexpr_{const}}$ at the expression level are then given in Table 18 of A. The IRE problem is avoided by introducing a special helper function $lexpr_{check}$, which encapsulates the matching tree for obtaining the value information recorded in valid constructors.

The reference expressions *Evar*, *Efun*, *Econ*, and *Epar* need only to return their addresses directly. In Table 19 of A, these reference expressions are summarized as $Eaddr\ (\llbracket name \rrbracket) : expreaddr\ (\llbracket name \rrbracket)\ eaddr\ (\llbracket name \rrbracket)$. The *Estruct*, *Ebop*, and *Euop* expressions can only be assigned as right expression values, so their semantics are banned by the $lexpr_{check}$ function (which returns an undefined result *None*).

Expressions in the r-value position: The following functions describe the semantics of evaluating expressions in the r-value position (i.e., the respective memory values). The evaluation of constant expressions is given in Table 20 of A. Assisted by $\mathcal{ESV}_{value}$, the $\mathcal{ESE}_{rexpr_{const}}$ directly provides the respective memory value.

According to Convention 1, the struct constructor *Estruct* represents an expression value at the right position, which is the only way to initialize or modify struct-type terms. The semantics of the right struct value are defined in Table 21 of A. The helper function $eval_{str}$ contains the type matching and value evaluation. The type matching part checks whether the type of each value satisfies the respective field. The second part recursively invokes $\mathcal{ESV}_{value}$ to evaluate the values in the respective memory values. If the evaluation process yields a *None* message, the *Estruct* evaluation has failed. Otherwise, the members' value set is retrieved and the respective struct memory value is returned.

Finally, the semantics of the binary and unary operations are defined in Tables 22 and 23 of A, respectively. Because of the static-type limitation in the formal abstract syntax definition based on GADTs, all expressions, sub-expressions and operations are well-formed, and the semantics do not need to check the type dependence relation. Therefore, subsidiary assist functions are not required. The functions $eval_{bop}$ and $eval_{uop}$ take the results of the expression evaluations and the required operations as arguments, and combine them to generate new memory values. In the present version of FEther, the above definition forbids mixed arithmetic operations, such as "int + float," because Solidity does not completely support the float datatype, and float values are rarely employed in smart contract programs. Therefore, mixed-arithmetic operations would add unnecessary complexity and computational burden when implementing the formal interpreter.

Statement Layer Having defined the semantics, we can now define the statement layer. Statement semantics parse the *FRWprograms* written by Lolisa, and evaluate the new memory states. The semantics of the sequence statements are not explicitly defined, and the relevant statement definitions are modified to improve the extremely low computational efficiency of solving the CBNT problem. We express the evaluation process of a statement as$\mathcal{ESS}$, and give its abstract definition as Rule 8.

$$\begin{aligned}\mathcal{ESS} :: \mathbb{Z} \to\ & memory \to option\ (list\ value) \\ & \to Env \to Env \to statement \to option\ memory\end{aligned} \tag{8}$$

Most statement evaluations employ the helper function $\text{env}_{\text{check}}$, which takes the current environment env and the super-environment fenv as arguments, and checks the

conditions (gas limitation and execution-level validity). For example, if the domains in env and fenv are equal but have different execution levels, the program is terminated and env is reset by fenv. If $\text{env}_{\text{check}}$ returns a true result, the current statements are executed; otherwise, the program is terminated and the initial memory state is restored.

Contract declarations are among the most important Solidity statements. Contract declaration in Lolisa involves two operations. First, the consistency of the inheritance information is checked using the helper function $\text{inherit}_{\text{check}}$, which determines whether the current inheritance relation inherits is stored in the current module context C. The function $\text{inherit}_{\text{check}}$ is defined as a sum type in Rule 9.

$$\begin{aligned} inherit_{check} \equiv \forall\, (inherits\ inhertis_c : list\ address)\,, \\ \{inherits = inhertis_c\} + \{inherits \neq inhertis_c\} \end{aligned} \tag{9}$$

Second, the initial contract information, including all member identifiers, is written into a designated memory block by the assistant function $write_{dir}$. The formal semantics of the contract declaration are defined in Table 24 of A.

Variable declaration is a basic task in Lolisa. The function $init_{var}$ is a special case of $write_{dir}$, with type given by Rule 10.

$$\begin{aligned} init_{var} :: memory \to Env \to Blc \to option\ access \\ \to type \to address \to option\ memory \end{aligned} \tag{10}$$

This function takes the current memory state, variable type, indexed address, and environment information as parameters, and initializes the respective memory block. Being based on the GERM memory model, the initialization and location processes of this term with the array datatype differ from those in standard researches on formalizing array types. The function $init_{var}$ calls the $init_{array}$ function to initialize the respective terms.

In [14], *normal types* are datatypes whose typing rules disallow recursive definition. A normal type is assigned as τ_{final}. The type τ_{final} is the recursive base of a multidimensional array. In other words, if the τ of the current element is τ_{final}, it represents the final dimension of the recursive definition of the current multidimensional array.

Particularly, because each memory block in the GERM memory model directly stores all logic information with type *value* [6], regardless of the sizes of the array elements, we only need to calculate the number of logic elements in the array. In order to illustrate the evaluation process, we take the following three-dimensional array $a\,[2]\,[3]\,[2]$ with type $[iAconst_{id\,(2)} Tarray[iAconst_{id\,(3)} Tarray[iAconst_{id\,(2)} \tau_{final}]]]$ as a simple example. The full tree structure of this array is given in Figure 6, and the mathematical evaluation process is shown as follows:

$$\begin{aligned} &(size_1 + size_1 * (size_2 + size_2 * size_3)) \\ &= (2 + 2 * (3 + 3 * 2)) \\ &= 20. \end{aligned}$$

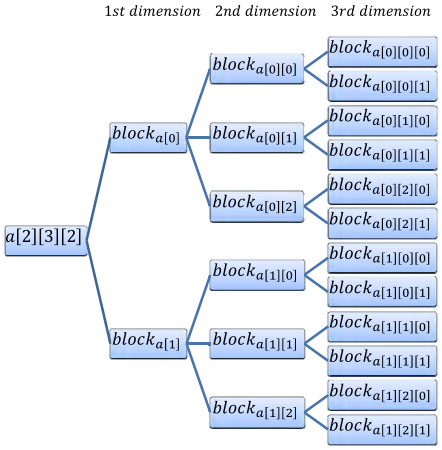


Figure 6. Example of a 3-dimensional array.

Note that this array requires a memory allocation of 20 blocks.

The array-size calculation can be summarized as Rule 12. Using this formula, we can implement the subsidiary function id_{array} to calculate and return the number of blocks allocated to arrays in each dimension. The abstract of this assignment is defined in Rule 13. Note that $size_1$ in Rules 11 and 13 represents the size of the current dimension rather than the size of the first dimension. For example, to calculate the $array_{size}$ of the second dimension of $a\,[2]\,[3]\,[2]$, we should replace $size'_1 = 2$ in Rules 12 and 13 with $size'_2 = 3$.

$$\begin{aligned} array_{size} &\equiv size_1 + size_1 * (size_2 + size_2 \\ &\quad * (\ldots (size_{n-1} + size_{n-1} * size_n))) \\ &\equiv \sum_{i=1}^{n} \prod_{j=1}^{i} size_j \end{aligned} \tag{11}$$

$$group_{size} \equiv array_{size}/size_1 \equiv \left(\sum_{i=1}^{n} \prod_{j=1}^{i} size_j\right)/size_1 \tag{12}$$

$$id_{array} :: index_{array} \to memory \to Env \to option\ \mathbb{Z} \tag{13}$$

Figure 7 shows the initialization process of $a\,[2]\,[3]\,[2]$, which follows its tree structure. In step (1), FEther searches a continuous memory space with a total size of 20 blocks, according to Rule 11. The algorithm *Tree Initialization* then classifies $a\,[2]\,[3]\,[2]$ as two initial trees, indexed by $a\,[0]$ and $a\,[1]$. The elements in both groups are then recursively initialized by $init_{arrray}$ in sequence. For example, in the recursion of $a\,[0]$, id_{array} calculates the size of the group indexed by $a\,[0]$ as $\left(size'_1 + size'_1 * \left(size'_2 + size'_2 * size'_3\right)\right)/size'_1 = 10$ blocks from $Block_0$ to $Block_9$. Step (2) allocates the memory blocks. Because $a\,[0]$ is also the beginning address of the whole array, it is allocated to $Block_0$ (deep blue block in Step (2) of Figure 7). To allocate the memory blocks of the second dimension, $init_{arrray}$ must proceed to the next level and recursively initialize the sub-groups indexed by $a\,[0]\,[0]$, $a\,[0]\,[1]$, and $a\,[0]\,[2]$. In Step (3), the information of the group indexed by $a\,[0]\,[0]$ is stored in $Block_1$, which requires $\left(size'_2 + size'_2 * size'_3\right)/size'_2 = 3$ blocks from $Block_1$

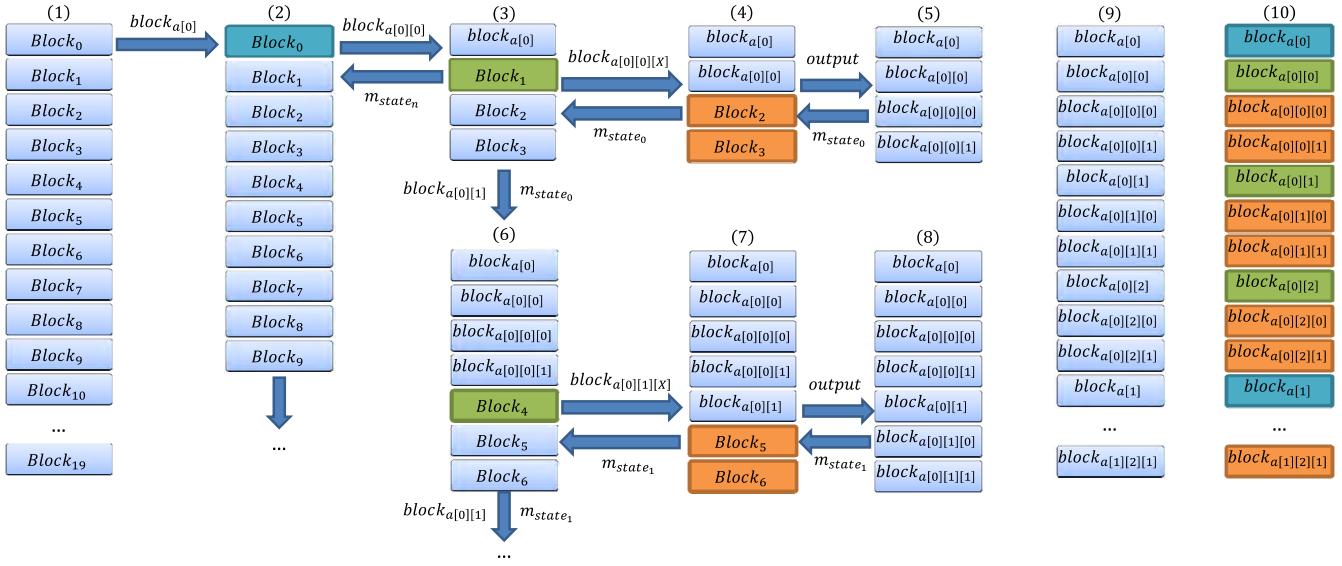


Figure 7. Array value initializing process (see text for details), and final structure of the a[2][3][2] array in GERM memory space.

to $Block_3$ (green block in Step (3) of Figure 7). To allocate the memory blocks of the third dimension, $init_{arrray}$ continues the deep recursion in Steps (4) and (5), which initialize $a\,[0]\,[0]\,[X]\,, X \in \{0, 1\}$. The elements in $a\,[0]\,[0]\,[X]$ are of type τ_{final}, so this is the leaf-node level with a group size of $size'_3/size'_3 = 1$. In other words, $a\,[0]\,[0]\,[0]$ and $a\,[0]\,[0]\,[1]$ are single-element groups. Both groups are contiguously stored in $Block_2$ and $Block_3$ (orange block in Step (4) of Figure 7). Steps (6)–(8) restore the memory state m_{state_0} to the recursion level of the second dimension, and repeat Steps (3)–(5) for $a\,[0]\,[1]\,[X]$. This process repeats for the remaining groups, until the whole array information has been stored into respective memory blocks. The final structure of $a\,[2]\,[3]\,[2]$ in the memory space, from $Block_0$ to $Block_{19}$ in Figure 7. Here, the left column (9) is the real structure and the right column (10) is the group classification.

The id_{search} can be implemented by a similar algorithm. However, as id_{search} directly locates the indexed group rather than searching each group, its core procedure is $address_{offset}\,(+, offset, init_{addr})$. The offset formula is given as Rule 14. For example, when locating the block of $a\,[0]\,[1]\,[1]$, the offset is calculated as $0 * 10 + 1 * 3 + 1 * 1 + 2 = 6$, and the initial address is $Block_0$. Therefore, the information of $a\,[0]\,[1]\,[1]$ is stored in $Block_6$.

$$offset \equiv \sum_{i=1}^{n} index_i * group_{size_i} + (n-1) \qquad (14)$$

Generally, if $valid_{array}\,(\tau) \wedge alloc\,(m_{state}, array_{size}) = Some\ init_{addr}$, the array space can be initialized by Table 6, called *Tree Initialization*.

After running this algorithm, the symbolic execution in FEther more accurately simulates the initialization and allocation behavior of an n-dimensional array in real hardware than other formalizations using the *list* datatype. An array can be abstracted by a number of interesting algorithms, such as

TABLE 6. Algorithm of the $init_{array}$ function.

Algorithm *Tree Initialization*
Function: *Fixpoint* $init_{array}$
Input: Initial K steps, initial memory state m_{state}, current initial address, array type, current environment *env*;
Output: The final memory state signed with optional type;
$\boldsymbol{Step_0}$: if $\tau = Tarray\ [id\ \tau_{recursive}] \wedge index_i < group_{size_i}$, then move to $\boldsymbol{Step_1}$, else return to $Step_2$;
$\boldsymbol{Step_1}$: ***let*** $m'_{state} \coloneqq write_{dir}(m_{state}, name, v_{array_i}), init_{array}\,(K-1, m'_{state}, (address_{offset}(+, offset, name)), \tau_{recursive}, env)$;
$\boldsymbol{Step_2}$: if $\tau \in \tau_{final} \wedge index_i < group_{size_i}$, then move to $\boldsymbol{Step_3}$, else return m_{state};
$\boldsymbol{Step_3}$: ***let*** $m''_{state} \coloneqq write_{dir}(m_{state}, name, v_{array_i}), init_{array}\,(K-1, m''_{state}, (address_{offset}(+, offset, name)), \tau_{recursive})$;

tree structure mapping [25] or graphic mapping [26], but the advantages of these algorithms are partially offset by disadvantages. For example, although they can represent an infinite memory space, their specifications and formal structures are very complex and difficult to extend. Moreover, to modify an array element, an operation must search each node one by one, and the overflow problem is difficult to check without a dependent type. In an algorithm based on the GERM memory model, the array is stored in a fixed-size contiguous memory space without assistance by a dependent type [27]. Verifiers can formally simulate the address offsetting process, check the array overflow problem by checking the head Flag stored in the memory block, and modify an array block directly by indexing the respective memory address. Consequently, the verification process becomes easier and more accurate.

Assuming that the current logic context based on GERM has sufficient logic memory space, and that each identifier has a valid and free address, $init_{var}$ represents the first time of setting the indexed memory block, and $write_{dir}$ is

always successful. The variable declaration semantics in this scenario are defined in Table 25 of A.

The semantics of the structure datatype declaration are defined in Table 26 of A. By Convention 1, the structure declaration at the statement level declares a new structure type with address identifier str_τ. The field member list of str_τ is $mems_\tau$. As an example, Figure 8 defines the built-in *address* datatype of Solidity rewritten by Lolisa. The *_0xaddress* is the str_τ, and the remaining fields are the $mems_\tau$. The $\mathcal{ESS}_{str}$ records the struct type information directly into the memory block with address str_τ.

$$\begin{aligned} &Address \stackrel{\text{def}}{=} Str_{type}\ \_0xaddress\ (str_{mem}\ TInt\ (Nvar\ addr)\\ &\qquad str_{mem}\ TInt\ (Nvar\ balance)\\ &\qquad str_{mem}\ (Tfid\ (Some\ \_Send))\ (Nvar\ send)\\ &\qquad str_{mem}\ TInt\ (Nvar\ gas)\ (str_{nil}))))\ \_0xaddress\ 2\ occupy) \end{aligned}$$

Figure 8. Address type declaration in Solidity, and its equivalent special struct type in Lolisa syntax.

In Lolisa, a function call statement unfolds the function body stored in the respective memory address. The semantics of a function call $\mathcal{ESS}_{call}$ are given in Table 27 of A. In the first step, the function call attempts to read the function declaration statements stored in the respective memory address. If the readout is successful, the second step sets the current execution environment level to 0, and (with the assistance of set_{env}) sets the domain as the called function identifier. In the final step, the function body is executed with the new *env'*.

Modifier declarations are special function declarations requiring three steps, and including a single limitation. The parameter values are set by the set_{par} predicate. As defined by Table 28 in A, the first step initializes and sets the parameters. The second step stores the modifier body into the respective memory block, and the third step attempts to initialize the return address Λ_{fun}. Under the rules of Solidity, the modifier body can return the checking flag, but cannot change the memory states. Therefore, in FEther, we add a special Boolean-type memory block in the GERM framework, indexed by *_0xmodifer*. If the modifier checking is successful, the block is set to *true* and assigned as σ_{mtrue}; otherwise, it is set to *false* and assigned as σ_{mfalse}, meaning that other blocks cannot be modified.

To guarantee the type safety, Lolisa separately defines the single- and multi-return value functions. As shown in Table 29 of A, however, we combine them such that the return type and modifier limitation are both defined as lists. The evaluation is completed by the *repeat* function. Unlike modifier semantics, the function semantics check the modifier limitations restricting the function. Specifically, all modifiers restricting the function are executed before the function is invoked. If the $modif_{chck}$ result of a modifier evaluation is true, the function is executed; otherwise, it is terminated. Particularly, if $modif_{chck}$ finds a modified memory state, the execution is discarded.

Assignment-statement semantics are based on the expression-evaluation semantics. If the result of evaluating an r-value expression is a function pointer generated by a field access, then the return values are evaluated by function call semantics. The semantics of assignment statements are defined in Table 30 of A.

C. THE FETHER PARSER

To analyze the syntactic units of *FRWprograms*, the semantics must be integrated into a parser that is easily implemented on the ISA. As shown in Figure 2, the parser has three layers for parsing the three syntax layers. The functions of these layers are validating the environment, deconstructing the input syntactic units, mapping the syntactic units $\mathcal{S}_i$ into the respective semantics $\mathcal{ES}_i$, and transmitting the information stored in the $\mathcal{S}_i$ to the $\mathcal{ES}_i$. As an example, consider the value layer in Table 7. First, the $\mathcal{ESV}$ checks the K limitation. It then deconstructs the input value v into specific constructors by pattern matching. Finally, the logic data are transmitted into their respective semantics.

TABLE 7. Simple example of the value-layer parser.

$$\begin{aligned} &\mathcal{ESV} :: \mathbb{Z} \to (\forall\ \tau\text{:}\ type, val\ \tau) \to memory \to Blc \to Env \\ &\qquad \to option\ value \\ &\mathcal{ESV} \equiv \\ &\lambda\ (K\text{:}\mathbb{Z}).\lambda\big(v\text{:}\ (\forall\ \tau\text{:}\ type, val\ \tau)\big).\lambda\ (env\text{:}\ Env).\lambda\ (\sigma\text{:}\ memory).\lambda\ \big(b_{infor}\text{:}\ B \\ &\ \{|\ valid \mapsto \\ &\quad \{|\ Vconst(\tau, n) \mapsto \big(\lambda\ \big(b_{infor}\text{:}\ Blc\big).\mathcal{ESV}_{const}\big(n, env, b_{infor}\big)\big) \\ &\quad Varray(index, \tau, name) \mapsto \big(\lambda\ \big(b_{infor}\text{:}\ Blc\big).\mathcal{ESV}_{array}\big(env, \sigma, b_{infor}\big)\big) \\ &\quad |\}.v \\ &\ invalid \mapsto None \\ &\ |\}.K_{check}(K) \end{aligned}$$

Therefore, the parsers can be summarized as the typing judgements 15 and 16, where *valid* denotes the validation process.

$$\frac{\begin{array}{l}\mathcal{E} \vdash env, fenvM \vdash \sigma, b_{infor} \\ valid\ (K, env, fenv) = true \\ \mathcal{S}_i \approx \mathcal{ES}_i\end{array}}{\mathcal{E}, M, \mathcal{F} \vdash \mathcal{S}_i\ (args) \Rightarrow \mathcal{ES}_i\ \big(\sigma, env, fenv, b_{infor}, args\big)} \quad (15)$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash env, fenvM \vdash \sigma, b_{infor} \mathcal{F} \vdash K \\ valid\ (K, env, fenv) = false \\ \mathcal{S}_i \approx \mathcal{ES}_i\end{array}}{\mathcal{E}, M, \mathcal{F} \vdash \mathcal{S}_i\ (args) \Rightarrow None} \quad (16)$$

The information in $\mathcal{S}_i$ needs to be partially preprocessed before transmission to $\mathcal{ES}_i$. First, we must check whether the constructor of $\mathcal{ES}_{rexpr_{eaddr}}$ is *Efun*. If true, we must transmit the respective Λ_{fun} instead of *name*. This action is recorded as $|Efun\ (oaddr, \tau, *) \longmapsto read_{chck}\ \big(\sigma, env, b_{infor}, \Lambda_{fun}\big)\ |\}.e$. As mentioned above, after evaluating $\mathcal{ESS}_{re}$ and $\mathcal{ESS}_{res}$, we must then change the current environment into a super-environment for stopping the function execution. Moreover, as semantics such as $\mathcal{ESS}_{call}$, $\mathcal{ES}_{rexpr_{bop}}$ and $\mathcal{ES}_{rexpr_{uop}}$

TABLE 8. Algorithm of the FEther entering point.

Algorithm FEther_enter_point
Function: *Fixpoint* FEther
Input: Initial K steps, optional initial memory state $[\![m_{state}]\!]$, current environment *env*, and super-environment *fenv*; initial arguments $args$, and valid $FRWprogram$;
Output: The final memory state signed with optional type;
$\boldsymbol{Step_0}$: if $env_{check}(env, fenv) = true \wedge [\![m_{state}]\!] \neq None$, then move to $\boldsymbol{Step_1}$, else return $[\![m_{state}]\!]$;
$\boldsymbol{Step_1}$: if $FRWprogram = s_0 :: s_1$, set $env' = set_{env}(s_0 :: s_1, env)$ and move to $\boldsymbol{Step_2}$, else return $[\![m_{state}]\!]$;
$\boldsymbol{Step_2}$: if $\mathcal{S}_0 \approx \mathcal{ES}_0$ then $(K, m, env, fenv, \mathcal{ES}_0) \Downarrow_{P(s_0)}^{yields} [\![m'_{state}]\!]$ and move to $\boldsymbol{Step_3}$, else return $None$.
$\boldsymbol{Step_3}$: $FEther([\![m'_{state}]\!], env', fenv, args, s_1)$

recursively invoke $\mathcal{ESE}_{rexpr}$, the specific $\mathcal{ESE}_{rexpr}$ and $\mathcal{ESS}$ must be defined as recursive functions. Finally, the parser statement level integrates the two lower levels, and also defines the entering point of FEther (see the *FEther_enter_point* algorithm given as Table 8).

The rules governing the execution of a Lolisa program in FEther are defined by the rules (EXE-F) and (EXE-IF), as shown at the bottom of this page, where the symbol ∞ refers to infinite execution, and T is the termination condition set of a finite execution.

D. AUTOMATION TACTICS

Automated theorem proving is a core topic in formal verification research. Many higher-order theorem-proving assistants provide tactics or similar mechanisms that simplify the program evaluation process and construct proofs automatically. With manual modeling technology, different formal models with significantly different structures and verification processes can be constructed in various programs. Hence, designing a set of tactics that automatically verifies models in different programs is nearly impossible.

The above problem is circumvented by FEther. According to EVI theory, the FEther symbolic execution corresponds to both the function evaluation and the program verification (see Rule 17). In other words, it unifies the verification processes of different programs in higher-order theorem-proving assistants by simplifying the program evaluation process in FEther. Because the situations of FEther execution constitute a fixed and finite set $\{s_0, s_1, \ldots, s_m\}$, we can design

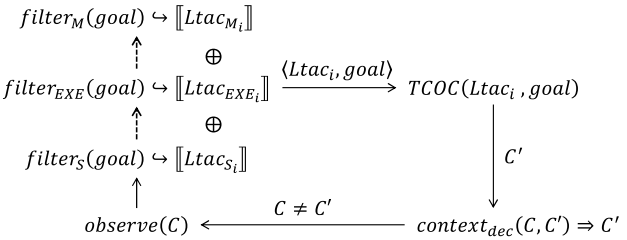


Figure 9. An abstract automatic tactic working process.

sufficiently many sub-tactics for all situations. Exploiting this advantage and assisted by the *Ltac* mechanism, we designed primitive automatic tactics for the FEther. The tactic strategy model is constructed from three parts: memory operating, K costing and semantics simplifying.

$$\Omega, M, \mathcal{F} \vdash_{ins} P_{exe} \equiv P_{eval} \equiv P_{verify} \tag{17}$$

The workflow is defined in Figure 9. When the proof universe of Coq is open, the *observe* function scans the current context C to obtain the current goal. In sequence, each part attempts to capture the operation characteristic of the current goal and choose the matching tactics. The selected tactics are combined into a solution tactic $Ltac_i$ that solves the goal in TCOC. The new context C' is compared with C in $context_{dec}$. If C' and C are identical, the current tactics cannot solve the goal automatically, and the tactic model is terminated. Otherwise, the tactic model continuously attempts to simplify the goal of C'.

The expression 18 states the *unfold_modify* tactic, a sub-tactic of the memory operation part. This sub-tactic captures parts of the operation characteristic of the $write_{dir}$ function, and evaluates the scanned $write_{dir}$ using basic built-in tactics.

Ltac *unfold_modify* :=
match goal with
|[| − context[?Y(?X : memory)(?Z : value)]]
⇒ unfold Y in*; cbn in*
end. (18)

The average ratios of contract size to proof size are shown in Table 9. Smart contracts exceeding 500 lines were excluded from this analysis, because the size of large contracts were

$$\frac{\begin{array}{l}\mathcal{E} \vdash env, fenvM \vdash \sigma, b_{infor}\mathcal{F} \vdash opars\mathcal{E}, M, \mathcal{F} \vdash P(stt)\,\mathcal{E}, M, \mathcal{F} \vdash lib \\ env = set_{gas}(init_{env}(P(stt)))\, fenv = init_{env}(P(stt)) \\ \sigma = init_{mem}(P(stt), lib)\end{array}}{\mathcal{E}, M, \mathcal{F} \vdash FEther\left(\left[\!\left[m'_{state}\right]\!\right], env', fenv, args, P(stt)\right) \overset{execute,T}{\Longrightarrow} \langle \sigma', env', fenv \rangle} \quad \text{(EXE-F)}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash env, fenvM \vdash \sigma, b_{infor}\mathcal{F} \vdash opars\mathcal{E}, M, \mathcal{F} \vdash P(stt)\,\mathcal{E}, M, \mathcal{F} \vdash lib \\ env = set_{gas}(init_{env}(P(stt)))\, fenv = init_{env}(P(stt)) \\ \sigma = init_{mem}(P(stt), lib)\end{array}}{\begin{array}{l}\mathcal{E}, M, \mathcal{F} \vdash FEther\left(\left[\!\left[m'_{state}\right]\!\right], env', fenv, args, P(stt)\right) \overset{execute,\infty}{\Longrightarrow} \langle \sigma', env', fenv \rangle \vee env'.(gas) \\ \quad \rightarrow (\neg fenv.(gasLimit)) \overset{execute,T}{\Longrightarrow} \langle \sigma', env', fenv \rangle\end{array}}. \quad \text{(EXE-IF)}$$

TABLE 9. Ratios of proof size to contract size in theorem-proving tactics.

Contract size	Built-in tactics	Automatic tactic
≤ 100 lines	1.27	0.16
≤ 200 lines	2.33	0.13
≤ 300 lines	5.26	0.17
≤ 400 lines	7.5	0.22
≤ 500 lines	10.3	0.17

limited by the *gas* cost. The second and third columns of Table 9 list the ratios using Coq's built-in tactics and our automatic tactics, respectively. Obviously, the automatic tactics reduce much of the proof workload. Moreover, according to our experimental results, the ratio floats in a range is influenced by the complexity of the target contract. Specifically, the ratios obtained by the built-in tactics range from approximately -0.5 to $+10.0$, whereas those of the automatic tactics range from approximately -0.1 to $+0.3$. Therefore, the automatic tactics possess a better universal property than the directly applied built-in tactics.

E. SELF-CORRECTNESS CERTIFICATION

The FEther interpreter is entirely constructed in Coq, which confers a natural advantage over other program verifications and analysis tools. The core of Coq is the trusted computation base (TCB) [28], which satisfies the de Bruijn criterion. In almost all program analysis tools, TCB self-verification is arguable and paradoxical, so whether the TCB of a program verification (analysis) tool satisfies the de Bruijn criterion is an important indicator of the trustworthiness of the verification.

The correctness of FEther is certified by its consistency between the relational and computational definitions, the correctness of its essential properties, and the meta-properties of its semantics.

First, we must prove that the operational semantics [29] of Lolisa (the inductive relational forms) are equivalent to the operational semantics (the executable function forms). As desired in the CompCert project [25], we check whether each evaluation in the relation semantics corresponds to the symbolic execution in the executable semantics. For this purpose, we construct a simulation diagram. Under identical conditions, the relational and executable semantics must have the same observable effect (same traces of the evaluation process). This requirement is embodied in the following *simulation diagram* theory.

Theorem (simulation diagram) Let $\mathcal{E}, M, \mathcal{F} \vdash_{ins} \sigma, opars, env, fenv, b_{infor}$ be the initial evaluation environment, and let R_{eq} represent the equivalence relationship between two terms. Then, any relational semantic S_{rel} and executable semantic S_{exe} must satisfy the following simulation diagram:

$$\begin{array}{ccc} S_{rel} & \xrightarrow[\langle \sigma, env, fenv, b_{infor}, opars, \Downarrow_{S_{rel}} \rangle]{\mathcal{E}, M, \mathcal{F} \vdash_{ins} \sigma, opars, env, fenv, b_{infor}} & [\![result]\!] \\ \Big\updownarrow R_{eq} & & \Big\updownarrow R_{eq} \\ S_{exe} & \xrightarrow[S_{exe}(n, \sigma, opars, env, fenv, b_{infor})]{\mathcal{E}, M, \mathcal{F} \vdash_{ins} \sigma, opars, env, fenv, b_{infor}} & [\![result']\!] \end{array}$$

Second, we must certify the correctness of the foundation behavior of the executable semantics. As a simple instance, we construct Lemma *test_lemma_if_false*, which certifies the correctness of the following execution: For all statements s and s', if the *if* statement condition is false, FEther must execute the statement s' of the false branch. By a similar process, we certified that almost all of the executable semantics exhibit standard behaviors.

Lemma (Test_Lemma_If_False):

$$\begin{aligned} &\forall\ if_{false}\ if_{state}\ s\ s'\ n\ env\ pass, \\ &\qquad if_{false} = Econst\ \ (Vbool\ false) \rightarrow \\ &\qquad if_{state} = \big(If\ \ if_{false}\ s\ s'\big) \rightarrow \\ &\qquad n > 0 \rightarrow \\ &\qquad \big(FEther\ n\ init_m\ pass\ env\ env\ if_{false}\big) = \\ &(FEther\ n\ init\_m\ pass\ set_{gas}\ (env)\ env\ s'). \end{aligned}$$

Finally, we prove the meta-properties of these semantics. The most basic properties in each layer are the *progress* and *preservation* properties, which maintain the static-type safety of the specification. For example, the *progress* and *preservation* of the expression layer are defined in Lemma *expression type safety*. Because Lolisa is a strongly typed language defined in terms of GADTs, the *progress* and *preservation* properties of expressions are easily proven by simplifying the semantics function. The *progress* and *preservation* properties of other layers are certified similarly. Besides the meta-properties, we proved the execution determinism of all semantics in Coq. The Lemma *execution determinism* is one example of the relevant proofs.

Lemma (Expression Type Safety):

1. If $e : expr_{\tau_0\tau_1}$ and $e \longmapsto e'$, then $e' : expr_{\tau_0'\tau_1}$.
2. If $e : expr_{\tau_0\tau_1}$, then either $e\,(v)$ or some e' exists such that $e \longmapsto e'$.

Lemma (Execution Determinism):

$$\begin{aligned} &\forall sm\ [\![m_{final}]\!]\ [\![m'_{final}]\!]\ nenvpass, \\ &\qquad FEther\ n\ m\ pass\ env\ env\ s = [\![m_{final}]\!] \rightarrow \\ &\qquad FEther\ n\ m\ pass\ env\ env\ s = [\![m'_{final}]\!] \rightarrow \\ &\qquad m_{final} = m'_{final} \end{aligned}$$

At present, the core functions have been completely verified. The correctness certification includes 74 theorems and lemmas, and approximately 4000 lines of Coq proof code.

V. FORMAL VERIFICATION OF SMART CONTRACT BY FETHER

To demonstrate the power of FEther in real-world practice, this section takes a smart contract extracted from the contract demonstration [2] as a case study to illustrate the verification process and features using FEther. Next, we will compare FEther with other similar works. The experimental environment was five identical personal computers with equivalent hardware of 8 GB memory and a 3.20 GHz CPU. All computers were run on Windows 10 and CoqIDE 8.8.

test_ICO.v

```
Definition wallet :=
   (Var (Some public) (Evar (Some open) Tuint));; (Var (Some public) (Evar (Some close) Tuint));;
   (Var (Some public) (Evar (Some quota) Tuint));; (Var (Some public) (Evar (Some rate) Tuint));;
   (Var (Some public) (Evar (Some partiLimit) Tuint));; (Var (Some public) (Evar (Some totalLimit) Tuint));;
   (Var (Some public) (Evar (Some finalLimit) Tuint));; (If (Econst (@Vmap Iaddress Tbool priviledges (Mstr_id Iaddress msg (sender ~>> \\\\)) None))
    ((Assignv (Evar (Some open) Tuint) (Evar (Some privilegeOpen) Tuint));;
     (Assignv (Evar (Some close) Tuint) (Evar (Some privilegeClose) Tuint));;
     (Assignv (Evar (Some quota) Tuint) (Evar (Some privilegeQuota) Tuint));;
     (Assignv (Evar (Some rate) Tuint) RATE_PRIVILEGE);; nil)
    ((Assignv (Evar (Some open) Tuint) (Evar (Some ordinaryOpen) Tuint));;
     (Assignv (Evar (Some close) Tuint) (Evar (Some ordinaryClose) Tuint));;
     (Assignv (Evar (Some quota) Tuint) (Evar (Some ordinaryQuota) Tuint));;
     (Assignv (Evar (Some rate) Tuint) RATE_ORDINARY);; nil ));;
   (If ((Evar (Some now) Tuint) (<) (Evar (Some open) Tuint) (||)
        (Evar (Some now) Tuint) (>) (Evar (Some close) Tuint))
    (Throw;; nil) (Snil;; nil));;
   (If ((Evar (Some subscription) Tuint) (+) (Evar (Some rate) Tuint) (>) TOKEN_TARGET_AMOUNT)
    (Throw;; nil) (Snil;; nil));;
   (If ((Evar (Some index) Tuint) (==) (Econst (Vint (INT I64 Unsigned 0))))
    (Throw;; nil) (Snil;; nil));;
   (If ((Econst (@Vmap Iuint Tuint deposits (Mvar_id Iuint index) None)) (>=) (Evar (Some quota) Tuint))
    (Throw;; nil) (Snil;; nil));;
   (If ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (==) (Econst (Vint (INT I64 Unsigned 0))))
    (Throw;; nil) (Snil;; nil));;
   (If ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (%) (Econst (Vint (INT I64 Unsigned 10000000000000000000))) (!=) (Econst (Vint (INT I64 Unsigned 0))))
    (Throw;; nil) (Snil;; nil) );;
   (Assignv (Evar (Some partiLimit) Tuint) ((Evar (Some quota) Tuint) (-) (Econst (@Vmap Iuint Tuint deposits (Mvar_id Iuint index) None))));;
   (Assignv (Evar (Some totalLimit) Tuint)
    ( (TOKEN_TARGET_AMOUNT (-) (Evar (Some subscription) Tuint)) (-)
      (TOKEN_TARGET_AMOUNT (-) (Evar (Some subscription) Tuint)) (%)
      (Evar (Some rate) Tuint) (/) (Evar (Some rate) Tuint) (x) (Econst (Vint (INT I64 Unsigned 10000000000000000000))) ));;
   (If ((Evar (Some partiLimit) Tuint) (<=) (Evar (Some totalLimit) Tuint))
    ((Assignv (Evar (Some totalLimit) Tuint) (Evar (Some partiLimit) Tuint)) ;; nil)
    ((Assignv (Evar (Some totalLimit) Tuint) (Evar (Some totalLimit) Tuint)) ;; nil));;
   (If ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (<=) (Evar (Some finalLimit) Tuint))
    ((Fun_call (Econst (Vfield (Tfid (Some safe)) (Fstruct _0xaddress safe) (send ~> \\) None))
      (pccons (Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) pcnil));;
     (Assignv (Econst (@Vmap Iuint Tuint deposits (Mvar_id Iuint index) None)) ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (+) (Econst (@Vmap Iuint Tuint
deposits (Mvar_id Iuint index) None))));
     ((Assignv (Evar (Some subscription) Tuint) ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (+) (Evar (Some finalLimit) Tuint) (/) (Econst (Vint (INT I64
Unsigned 10000000000000000000))) (x) (Evar (Some rate) Tuint)))) :: nil
    )
    (((Fun_call (Econst (Vfield (Tfid (Some _0xsend)) (Fstruct _0xaddress safe) (send ~> \\) None))
      (pccons (Evar (Some finalLimit) Tuint) pcnil))));;
     (Assignv (Econst (@Vmap Iuint Tuint deposits (Mvar_id Iuint index) None)) ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (+) (Econst (@Vmap Iuint Tuint
deposits (Mvar_id Iuint index) None))));;
     (Assignv (Evar (Some subscription) Tuint) ((Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None)) (+) (Evar (Some finalLimit) Tuint) (/) (Econst (Vint (INT I64
Unsigned 10000000000000000000))) (x) (Evar (Some rate) Tuint))));;
     (Fun_call (Econst (Vfield (Tfid (Some _0xsend)) (Fstruct _0xmsg msg) (sender ~> send ~> \\) None))
      (pccons (Econst (Vfield Tuint (Fstruct _0xmsg msg) (values ~> \\) None) (-) (Evar (Some finalLimit) Tuint)) pcnil));; nil
   )));; nil.
```

Figure 10. Formal version of the wallet function.

A. CASE STUDY: HYBRID VERIFICATION

At the beginning, the *.sol* file of smart contract is automatically translated line-by-line from Solidity into Lolisa with the assistant of translator. As a simple example, we consider the *wallet* function encoded in Appendix B. This function, which executes initial coin offering, is a segment of the Solidity contract extracted from [2], and its formal model is translated into Figure 10. As the comparison between Figure 10 and Appendix B, the consistency of smart contract is guaranteed strictly.

One of the most important requirements (*not in time*) of *wallet* is the application time validation. Clearly, it is invalid that the current time *now* is below *privilegeOpen* or above *privilegeClose*. Hence, if the current time *now* in the *wallet* function is outside the range *open* to *close*, the smart contract must be discarded.

According to EVI theory, verification in the proposed FSPVM is founded on simultaneous Hoare logic and reachability logic. Meanwhile, verification in FEther combines higher-order theorem proving and symbolic execution. By virtue of FEther, programmers can mechanically define the Hoare style properties following the formula abstract (19), where the wildcard "$*$" represents other specific arguments.

$$\mathrm{P}\,\{\mathrm{m_{init}}\}\,\mathrm{FEther}\,(\mathrm{m_{init}},\,\mathrm{FRWprograms},\,*)\,\mathrm{Q}\,\{\mathrm{m_{final}}\} \quad (19)$$

According to the reachability logic, the Hoare logic derivation is equivalent to the trusted operational semantics execution. Therefore, the execution of FEther can be seen as a derivation based on Hoare logic. The inference process is given by expression (20). The specific initial memory state m_{init} is the precondition of the program verification. Guided by the semantics of each statement $\mathrm{c_i c_i}$, FEther logically modifies the current memory state m_{i-1} to a new post-condition $Q_i\,\{m_i\}$ (i.e., the precondition of $\mathrm{c_i c_i}$). The theorems need only judge wither the final output memory state m_n after executing the final statement matches the correct memory state m_{final}. Most importantly, this verification procedure is automated in the proposed FSPVM.

$$\begin{aligned}\mathrm{P}\,\{\mathrm{m_{init}}\}\,\mathrm{c_0} &\xrightarrow{\mathrm{FEther}(\mathrm{m_{init}},\mathrm{c_0},*)} \mathrm{Q_0}\,\{\mathrm{m_0}\}\,\mathrm{c_1}\\ &\xrightarrow{\mathrm{FEther}(\mathrm{m_0},\mathrm{c_1},*)} \mathrm{Q_1}\,\{\mathrm{m_1}\}\,\mathrm{c_2} \twoheadrightarrow \mathrm{c_n Q_n}\,\{\mathrm{m_n}\}\\ &\xleftrightarrow{?} \mathrm{Q}\,\{\mathrm{m_{final}}\}\end{aligned} \quad (20)$$

During this process, verifiers can alter the verification patterns (including static, concolic, and selective symbolic execution) by defining the preconditions in different ways. For example, programmers can vary the *wallet* function by the following three approaches.

1) STATIC SYMBOLIC EXECUTION

The basic verification pattern is static symbolic execution. When the initial arguments are inductively defined with quantifiers such as $\forall$ and $\exists$, the traditional symbolic execution will traverse all cases. For example, the Lemma *no_in_time* defined in Figure 11 marked by red box defines (INT I64 Unsigned?X) and (INT I64 Unsigned?Y) by defining ?X and ?Y as inductive values representing all possible situations of time as $\forall$ (x : Int) (INT I64 Unsigned x) and $\forall$ (y : Int) (INT I64 Unsigned y). And they will be written into the initial memory state m_3 defined in the figure. Next, the requirement on time of wallet function should be defined under first-order logic as $(t < x \wedge y > t) \vee (t > x \wedge y < t)$.

```
Lemma no_in_time : forall pump pump_val m m' m0 m1 m2 m3 t x y z blc gs (env : environment) (s :
statement),
  let (_, _, cur, dn) := env in
  m = init_msg init_m z 19 blc gs IcoController msg ->
  m' = write_by_address m (Map priviledges
          (Some (iStr _0xaddress (Some (cons (iInt (Some (INT I64 Signed 19)) public occupy)
                  (cons (iFid (fid (Some _0xsend)) (Some nil) public occupy)
                  (cons (iInt (Some (INT I64 Signed blc)) public occupy)
                  (cons (iInt (Some (INT I64 Signed gs)) public occupy)
                   nil))))) public occupy, Bool (Some true) cur 2 public occupy))
                   Iaddress Tbool None cur dn public occupy) priviledges ->
  m0 = write_by_address m' (Int (Some (INT I64 Unsigned x)) cur 2 public occupy) privilegeOpen ->
  m1 = write_by_address m0 (Int (Some (INT I64 Unsigned y)) cur 2 public occupy) privilegeClose ->
  m2 = write_by_address m1 (Int (Some (INT I64 Unsigned t)) cur 2 public occupy) now ->
  m3 = mem_address (mem_msg m2) ->
  ((t <? x)%Z = true /\ (y <? t)%Z = false)  \/ ((t <? x)%Z = false /\ (y <? t)%Z = true) ->
  pump > 100 ->
  pump_val > 100 ->
  test pump pump_val m3 (Some nil) env env wallet
    = Some init_m'
.
Proof.
  time(
  intros; init_memory_state env; rewrite_mem'; step_multi).
Qed.
```

```
No more subgoals.
```

```
Messages | Errors | Jobs
Tactic call ran for 5.772 secs (2.792u,0.015s) (success)
```

Figure 11. Execution and verification of wallet with abstract symbol arguments using.

```
Lemma no_in_time : forall pump pump_val m m' m0 m1 m2 m3 t x y z blc gs (env : environment) (s :
statement),
  let (_, _, cur, dn) := env in
  m = init_msg init_m z 19 blc gs IcoController msg ->
  m' = write_by_address m (Map priviledges
          (Some (iStr _0xaddress (Some (cons (iInt (Some (INT I64 Signed 19)) public occupy)
                  (cons (iFid (fid (Some _0xsend)) (Some nil) public occupy)
                  (cons (iInt (Some (INT I64 Signed blc)) public occupy)
                  (cons (iInt (Some (INT I64 Signed gs)) public occupy)
                   nil))))) public occupy, Bool (Some true) cur 2 public occupy))
                   Iaddress Tbool None cur dn public occupy) priviledges ->
  m0 = write_by_address m' (Int (Some (INT I64 Unsigned x)) cur 2 public occupy) privilegeOpen ->
  m1 = write_by_address m0 (Int (Some (INT I64 Unsigned y)) cur 2 public occupy) privilegeClose ->
  m2 = write_by_address m1 (Int (Some (INT I64 Unsigned t)) cur 2 public occupy) now ->
  m3 = mem_address (mem_msg m2) ->
  ((t <? x)%Z = true /\ (y <? t)%Z = false)  \/ ((t <? x)%Z = false /\ (y <? t)%Z = true) ->
  pump > 100 ->
  pump_val > 100 ->
  test pump pump_val m3 (Some nil) env env wallet
    = Some init_m'
.
Proof.
  destruct env0; intros; unfold mem_msg, mem_address in *.
  rewrite H4; clear H4.
  rewrite H3; clear H3.
  rewrite H2; clear H2.
  rewrite H1; clear H1.
  rewrite H0; clear H0.
  rewrite H; clear H.
  destruct H5; destruct H;
  do 7 (step; try unfold_modify_s);
  step;
  next pump_val;
  push; unfold init_addr_str; cbn in *;
  do 5 (next pump_val);
  push;
  do 4 (step; unfold_modify_s);
  repeat step.
  push; repeat unfold ltInt, gtInt; push;  eauto.
  rewrite H, H0; eauto.
  push; repeat unfold ltInt, gtInt; push;  eauto.
  rewrite H, H0; eauto.
Qed.
```

Figure 12. Manual verification of the wallet function.

Obviously, the expected final memory state should be smart contract termination.

Finally, as shown in Figure 11, using the automatic tactics designed for FEther, the *wallet* function and corresponding lemma can be automatically executed and verified under only 1 line proof code within 5.772 s. As shown in Figure 12, if users directly invoke the built-in tactics of Coq, the lemma is verified with 21 lines proof code, even though the proof code has been simply optimization. Compared with the manually verification using the built-in tactics provided by Coq, FEther has a higher level automated degree.

2) CONCOLIC EXECUTION

Second, FEther supports concolic symbolic execution that gets real inputs. To accurately simulate execution processes on real world hardware, FEther is built in a virtual execution environment. Therefore, a FEther execution can be regarded as a special dynamic analysis. As shown in Figure 13, the entering points *test* and code *wallet* are unmodified, and *privilegeOpen*, *proviledgeClose* and *now* are replaced with specific values 0, 3, and 4, marked by blue box. The other constraints are still inductively defined as abstract symbols. The function correctness of concolic execution with specific inputs is then proven by the *no_in_time* lemma. Because the

```
Lemma no_in_time : forall pump pump_val m m' m0 m1 m2 m3 t x y z blc gs (env : environment) (s :
statement),
  let (_, _, cur, dn) := env in
  m = init_msg init_m z 19 blc gs IcoController msg ->
  m' = write_by_address m (Map priviledges
          (Some (iStr _0xaddress (Some (cons (iInt (Some (INT I64 Signed 19)) public occupy)
                  (cons (iFid (fid (Some _0xsend)) (Some nil) public occupy)
                  (cons (iInt (Some (INT I64 Signed blc)) public occupy)
                  (cons (iInt (Some (INT I64 Signed gs)) public occupy)
                   nil))))) public occupy, Bool (Some true) cur 2 public occupy))
                   Iaddress Tbool None cur dn public occupy) priviledges ->
  m0 = write_by_address m' (Int (Some (INT I64 Unsigned 0)) cur 2 public occupy) privilegeOpen ->
  m1 = write_by_address m0 (Int (Some (INT I64 Unsigned 3)) cur 2 public occupy) privilegeClose ->
  m2 = write_by_address m1 (Int (Some (INT I64 Unsigned 4)) cur 2 public occupy) now ->
  m3 = mem_address (mem_msg m2) ->
  pump > 100 ->
  pump_val > 100 ->
  test pump pump_val m3 (Some nil) env env wallet
    = Some init_m'
.
Proof.
  time(
  intros; init_memory_state env; rewrite_mem'; step_multi).
Qed.
```

```
No more subgoals.
```

```
Messages | Errors | Jobs
Tactic call ran for 5.534 secs (2.652u,0.046s) (success)
```

Figure 13. Concolic verification of the wallet function.

```
Definition msp' :=
    If
     (Evar (Some now) Tuint (<) Evar (Some open) Tuint
      (||) Evar (Some now) Tuint (>) Evar (Some close) Tuint) (Throw;; nil)
     (Snil;; nil);; nil
.
Lemma msp_correct : forall pump pump_val m m' m0 m1 m2 m3 z blc gs (env : environment) (s : statement),
  let (_, _, cur, dn) := env in
  m = init_msg init_m z 19 blc gs IcoController msg ->
  m' = write_by_address m (Map priviledges
          (Some (iStr _0xaddress (Some (cons (iInt (Some (INT I64 Signed 19)) public occupy)
                 (cons (iFid (fid (Some _0xsend)) (Some nil) public occupy)
                 (cons (iInt (Some (INT I64 Signed blc)) public occupy)
                 (cons (iInt (Some (INT I64 Signed gs)) public occupy)
                  nil))))) public occupy, Bool (Some false) cur 2 public occupy))
                  Iaddress Tbool None cur dn public occupy) priviledges ->
  m0 = write_by_address m' (Int (Some (INT I64 Unsigned 0)) cur 2 public occupy) open ->
  m1 = write_by_address m0 (Int (Some (INT I64 Unsigned 3)) cur 2 public occupy) close ->
  m2 = write_by_address m1 (Int (Some (INT I64 Unsigned 4)) cur 2 public occupy) now ->
  m3 = mem_address (mem_msg m2) ->
  pump > 100 ->
  pump_val > 100 ->
  (test pump pump_val (mem_address (mem_msg m3)) (Some nil) env env msp')
    = Some init_m'
.
Proof.
  destruct env0; intros. unfold mem_msg, mem_address in *. initmem.
  rewrite H4; clear H4; cbn in *.
  step.
  step.
Qed.

  Definition wallet :=
    (Var (Some public) (Evar (Some open) Tuint));; (Var (Some public) (Evar (Some close) Tuint));;
    (Var (Some public) (Evar (Some quota) Tuint));; (Var (Some public) (Evar (Some rate) Tuint));;
    (Var (Some public) (Evar (Some partiLimit) Tuint));; (Var (Some public) (Evar (Some totalLimit) Tuint));;
    (Var (Some public) (Evar (Some finalLimit) Tuint));; (If (Econst (@Vmap Iaddress Tbool priviledges
(Mstr_id Iaddress msg (sender ~>> \\\\)) None))
     ((Assignv (Evar (Some open) Tuint) (Evar (Some privilegeOpen) Tuint));;
      (Assignv (Evar (Some close) Tuint) (Evar (Some privilegeClose) Tuint));;
      (Assignv (Evar (Some quota) Tuint) (Evar (Some privilegeQuota) Tuint));;
      (Assignv (Evar (Some rate) Tuint) RATE_PRIVILEGE);; nil)
     ((Assignv (Evar (Some open) Tuint) (Evar (Some ordinaryOpen) Tuint));;
      (Assignv (Evar (Some close) Tuint) (Evar (Some ordinaryClose) Tuint));;
      (Assignv (Evar (Some quota) Tuint) (Evar (Some ordinaryQuota) Tuint));;
      (Assignv (Evar (Some rate) Tuint) RATE_ORDINARY);; nil ));;
    msp' ++ (If ((Evar (Some subscription) Tuint) (+) (Evar (Some rate) Tuint) (>) TOKEN_TARGET_AMOUNT)
      (Throw;; nil) (Snil;; nil));;
```

Figure 14. Selective symbolic execution of the wallet function.

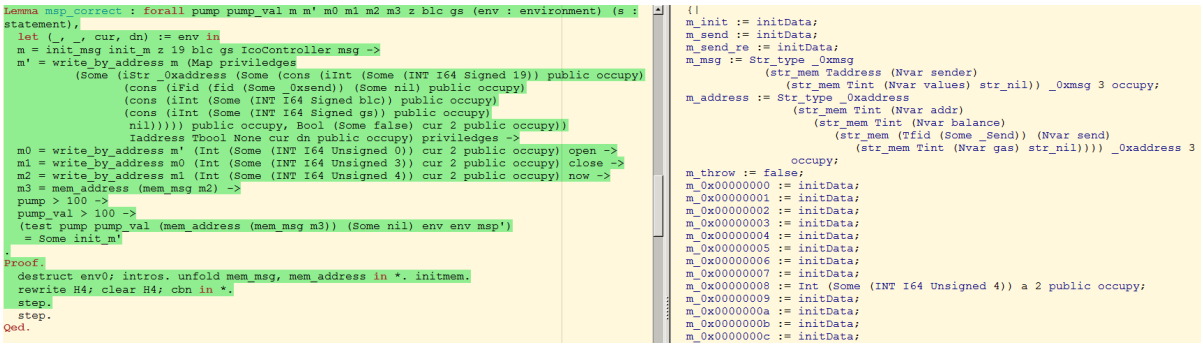
inputs are specified, the number of possible execution paths is limited, and the execution time reduces to 5.534 s.

This pattern leaves the extensible space for testing. Programmers can develop assistant tools to generate automatic test scripts that modify the input values.

3) SELECTIVE SYMBOLIC EXECUTION

Third, the *wallet* function can be varied by exploiting the selective symbolic execution of FEther. As shown in Figure 14, programmers can extract the core code segment *if*$(now < open||now > close)\{throw();\}$ from the *wallet* function and represent it by a new definition such as *msp'* in red box, which can be individually verified by the *msp_correct* lemma. After combining the verified *msp'* into the *wallet* function, the verification of the *no_in_time* lemma can be finished by invoking the *msp_correct* lemma. Clearly, the *msp_correct* can also assist the proofs that use the *msp'* code segment.

In this manner, the verified code segment can be invoked repeatedly by new properties' verification and improve the reusability of theorem proving technology. Besides, the FEther can also simplify loop proofs. In the standard approach of higher-order theorem proving, program loops are proven by manually identifying the invariants. Searching the loop invariants of simple loops, however, is a tedious process. By combining symbolic execution and higher-order theorem proving, we simultaneously facilitate the use of BMC and the search for loop invariants. Employing BMC, we first limit



```
Lemma msp_correct : forall pump pump_val m m' m0 m1 m2 m3 z blc gs (env : environment) (s :
statement),
  let (_, _, cur, dn) := env in
  m = init_msg init_m z 19 blc gs IcoController msg ->
  m' = write_by_address m (Map priviledges
          (Some (iStr _0xaddress (Some (cons (iInt (Some (INT I64 Signed 19)) public occupy)
                    (cons (iFid (fid (Some _0xsend)) (Some nil) public occupy)
                    (cons (iInt (Some (INT I64 Signed blc)) public occupy)
                    (cons (iInt (Some (INT I64 Signed gs)) public occupy)
                     nil))))) public occupy, Bool (Some false) cur 2 public occupy))
                      Iaddress Tbool None cur dn public occupy) priviledges ->
  m0 = write_by_address m' (Int (Some (INT I64 Unsigned 0)) cur 2 public occupy) open ->
  m1 = write_by_address m0 (Int (Some (INT I64 Unsigned 3)) cur 2 public occupy) close ->
  m2 = write_by_address m1 (Int (Some (INT I64 Unsigned 4)) cur 2 public occupy) now ->
  m3 = mem_address (mem_msg m2) ->
  pump > 100 ->
  pump_val > 100 ->
  (test pump pump_val (mem_address (mem_msg m3)) (Some nil) env env msp')
    = Some init_m'
.
Proof.
  destruct env0; intros. unfold mem_msg, mem_address in *. initmem.
  rewrite H4; clear H4; cbn in *.
  step.
  step.
Qed.
```

```
{|
m_init := initData;
m_send := initData;
m_send_re := initData;
m_msg := Str_type _0xmsg
           (str_mem Taddress (Nvar sender)
              (str_mem Tint (Nvar values) str_nil)) _0xmsg 3 occupy;
m_address := Str_type _0xaddress
               (str_mem Tint (Nvar addr)
                  (str_mem Tint (Nvar balance)
                     (str_mem (Tfid (Some _Send)) (Nvar send)
                        (str_mem Tint (Nvar gas) str_nil)))) _0xaddress 3
                  occupy;
m_throw := false;
m_0x00000000 := initData;
m_0x00000001 := initData;
m_0x00000002 := initData;
m_0x00000003 := initData;
m_0x00000004 := initData;
m_0x00000005 := initData;
m_0x00000006 := initData;
m_0x00000007 := initData;
m_0x00000008 := Int (Some (INT I64 Unsigned 4)) a 2 public occupy;
m_0x00000009 := initData;
m_0x0000000a := initData;
m_0x0000000b := initData;
m_0x0000000c := initData;
```

Figure 15. Debugging of the wallet function in Coq.

TABLE 10. Feature comparison of FEther semantics and existing software quality tools.

Tool	Spec.	Exec.	Certif.	Verif.	Debug.	Gas.	Level	Logic.	Hybrid.
Yellow Paper	Yes	No	No	No	No	No	None	None	No
Lem spec	Yes	Yes	Testing	Yes	No	No	Byte Code	Higher order	No
Mythril	Yes	Yes	Testing	Yes	Yes	Yes	Byte Code	First order	No
Hsevm	No	Yes	Testing	No	Yes	No	Byte Code	None	No
Scilla [32]	Yes	No	Testing	Yes	No	No	Intermediate	Higher order	No
Cpp-ethereum	No	Yes	Testing	No	No	No	Byte Code	None	No
KEVM	Yes	Yes	Testing	Yes	Yes	Yes	Byte Code	First order	No
FEther	Yes	Yes	Verifying	Yes	Yes	Yes	Solidity	Higher order	Yes

FEther to K or fewer executions of FRWprogram. In general, if L executions (where $L \leq KL \leq K$) of an FRWprogram can generate the corresponding final memory state, the loops existing in the FRWprogram can be directly unfolded as a set of identical normal-sequence statements within finite time, as inferred from Rule 21. If the *FRWprogram* fails to generate the corresponding final memory state after K executions, we can set the loop statement as a breakpoint (by virtue of the selective symbolic execution) and separate the *FRWprogram* into two parts, denoted as the head and tail parts. Next, we must locate the loop invariants and encapsulate them into an invariant memory state $I\{m_i\}$, which serves as the final memory state of the head part and the initial memory state of the tail part. This procedure is embodied in Rule 21 below.

$$P\{m_{init}\}\, c_0 \twoheadrightarrow c_i I\{m_i\}\ (\text{head})$$

and

$$c_i I\{m_i\} \twoheadrightarrow c_n Q\{m_{final}\}\ (\text{tail}) \tag{21}$$

Under the composition rule of Hoare logic, we have $P\{m_{init}\}\, c_0 \twoheadrightarrow c_n Q\{m_{final}\}$. In this way, simple loops can be proven automatically, reducing the workload of searching loop invariants. Moreover, complex loops that cannot be verified by model checking and symbolic execution technology can be proved by higher-order theorem-proving technology.

4) DEBUG MECHANISM

Finally, the FEther provides a debug mechanism for users. Because FEther is developed in the GERM memory model, it provides debug tactics such as *step*, which enables step-by-step debugging of a smart contract. The formal intermediate memory states obtained during the execution and verification process of a Lolisa program using FEther are shown in the proof context (right panel of Figure 15). In this manner, programmers can follow the intermediate memory states to locate the bugs.

Clearly, users of FEther can flexibly choose the most suitable method for verifying their programs.

B. FEATURE COMPARISON OVERVIEW

FEther is the first hybrid symbolic execution engine for Ethereum smart contracts. To illustrate the advantages of FEther over the solvers of other tools, we require a compelling benchmark, such as a testing suite or analysis time. Given that FEther is constructed on Coq, however, and directly executes and verifies the Solidity source code of smart contracts rather than compiling Solidity at the bytecode level, such a benchmark is difficult to find. For a fair comparison, we instead compared the presence and absence of various features in FEther and in other tools. The compared features are listed and defined below:

- Spec.: Suitable as a formal specification of the EVM language
- Exec.: Executable on concrete tests
- Certif.: Certifiable self-correctness
- Verif.: Verifiable properties of EVM programs
- Debug.: Provision of an interactive EVM debugger
- Gas.: Tools for analyzing the gas complexity of an EVM program
- Level.: Analysis or verification level of code
- Logic.: Type of essential logic supported
- Hybrid.: Support for hybrid verification methods

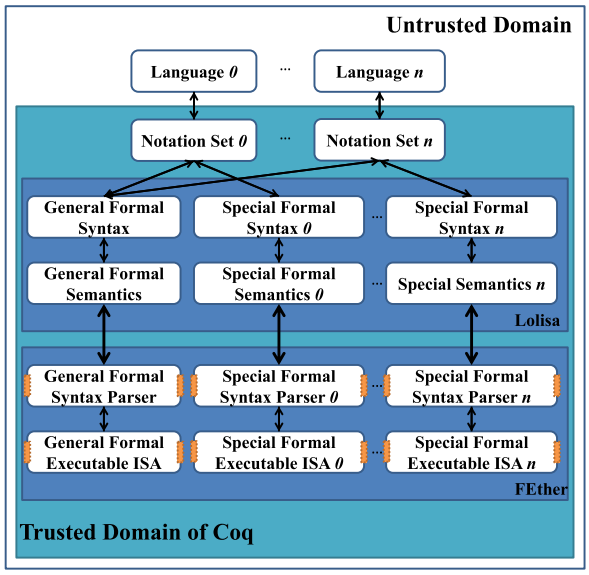


Figure 16. Detailed architecture for extending Lolisa to other general-purpose programing languages.

Table 10 overviews the results of the feature comparison. Obviously, only FEther, the core of KEVM, and Mythril support the Spec, Exec, Verif, Debug, and Gas features. The Certif feature of FEther is "verifying" rather than "testing," which improves the reliability of FEther (at least in theory) over testing methods such as KEVM and Mythril. Moreover, the execution and verification level of FEther is "Solidity" rather than "byte code," which avoids the error risk during compiling. FEther also supports higher-order logic, which improves the expressive ability. Moreover, the fundamental verification theory of FEther is the calculus of inductive construction instead of the satisfiability modulo theories or Boolean satisfiability problem. Therefore, the situations that cannot be evaluated and verified do not exist. Finally, FEther is the only tool that supports hybrid formal verification.

According to our previous experimental results [24], the symbolic execution time of the optimized current version of FEther is approximately 0.03 s per statement when the initial arguments are specified and approximately 0.07 s when the initial arguments are inductively defined by quantifiers. The execution efficiency of FEther far exceeded that of the interpreters that are developed in Coq in accordance with the standard tutorial developed in Coq. The current version also supports the verification of smart contract models adhering to the Ethereum ERC20 standard.

VI. DISCUSSION

A. CONTRIBUTIONS

This article overcomes the final challenge noted in our previous work: completing the proof engine of FSPVM-E. We now highlight the significant contributions of the present work. First, we confirmed that FEther maintains consistency between the Solidity source code and the respective formal specifications. To our knowledge, FEther is the first proof engine of Ethereum that supports the hybrid verification technology of Coq. Second, it provides a debug mechanism by which programmers can directly debug target smart contracts in Coq. Third, the correctness of FEther has been completely certified in Coq, implying that FEther is a reliable proof engine. Fourth, we provided a proprietary set of automatic tactics for FEther, which will help programmers to finish their property verifications with a high degree of automation. Finally, we optimized the high-level evaluation efficiency of FEther. We confirmed the utility of our previous works in building a certified executable proof engine in Coq.

B. EXTENSIBILITY AND UNIVERSALITY

Obviously, the definitional interpreter of an intermediate must faithfully capture the intended behaviors of programs written in real-world programing languages. From a flexibility perspective, the same interpreter should also be applicable to multiple programing languages. Therefore, extensibility and universality were considered in the FEther design from the beginning of its development.

As mentioned in [14], we deliberately incorporated extensible space in Lolisa. This space is sufficient for expanding features such as pointer formalization and for implementing independent operator definitions. It can easily incorporate the features of mainstream programing languages by adding new

TABLE 11. Semantics of constant Lolisa values.

$$\mathcal{ESV}_{const} :: (\forall\, \tau_{const} : type, val\ \tau_{const}) \to Env \to Blc \to option\ value$$
$$\mathcal{ESV}_{const} \coloneqq \lambda\left(v_{const}(n) : (\forall\, \tau_{const} : type, val\ \tau_{const})\right).\lambda\,(env : Env).\lambda\left(b_{infor} : Blc\right).$$
$$Some\ m_{value_{const}}(n, env, b_{infor})$$

TABLE 12. Semantics of array types at the value layer.

$$\mathcal{ESV}_{array} :: type \to index_{array} \to L_{address} \to Env \to memory \to Blc \to option\ value$$
$$\mathcal{ESV}_{array} \coloneqq$$
$$\lambda\,(\tau : type).\lambda\,(name : L_{address}).\lambda\,(env : Env).\lambda\,(\sigma : memory).\lambda\left(b_{infor} : Blc\right).$$
$$\{|\ Some\ addr \mapsto \left(\lambda\,(addr : L_{address}).\,read_{chck}(\sigma, env, b_{infor}, addr)\right)$$
$$Error \mapsto Error\ |\}.\left(id_{search}(\tau, name, \sigma, env)\right)$$

TABLE 13. Semantics of mapping values.

$$
\begin{array}{l}
\mathcal{ESV}_{mapping} :: type \to type_{map} \to index_{map} \to option\ (val\ Tmap) \to Env \to memory \to Blc \to option\ value \\
\mathcal{ESV}_{mapping} := \\
\ \lambda\ (\tau : type)\ .\lambda\ (\tau_{map} : type_{map})\ .\lambda\ (id : index_{map})\ .\lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ (b_{infor} : Blc)\ . \\
\ \lambda\ \Big(Vmap\big(name\ [id]\ (\tau_{map} \Rightarrow \tau)\ snd\big) : val\ (Tmap\ it\ t)\Big). \\
\quad \{|\ Some\ addr \longmapsto \\
\quad\ \Big(\lambda\ (addr : L_{address})\ .read_{chck}\big(\sigma, env, b_{infor}, addr\big)\Big)\ addr \\
\quad\ Error \longmapsto Error\ |\}\ .\Big(id_{map}\big(id, env, \sigma, name, \tau, \tau_{map}, snd\big)\Big)
\end{array}
$$

TABLE 14. Semantics of struct at the value layer.

$$
\begin{array}{l}
\mathcal{ESV}_{str} :: L_{address} \to Env \to memory \to Blc \to option\ memory \\
\mathcal{ESV}_{str} := \\
\ \lambda\ (str_v : L_{address}).\lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big). \\
\ read_{chck}\big(\sigma, env, b_{infor}, str_v\big)
\end{array}
$$

TABLE 15. Semantics of field access at the value layer.

$$
\begin{array}{l}
\mathcal{ESV}_{field} :: L_{address} \to list\ struct_{name} \to list\ ival \to Env \to memory \to Blc \to option\ value \\
\mathcal{ESV}_{field} := \\
\ \lambda\ (head : L_{address}).\lambda\ (mems : list\ struct_{name}).\lambda\ (opars : list\ ival).\lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big). \\
\ \{|\ Some\ \ \big(a_{init}, a_{type}\big) \longmapsto \\
\quad \{|\ Some\ (Dad, m_v) \longmapsto \\
\quad\ \{|\ T_{Fid} \longmapsto\ set_{dad}(Dad, m_v, opars) \\
\quad\ |\ F_{Fid} \longmapsto m_v\ |\}.Fid_{chck}(m_v) \\
\quad |\ Error \longmapsto Error\ |\}.mems_{find}\big(\sigma, mems, env, b_{infor}, a_{init}, a_{type}\big) \\
\ |\ Error \longmapsto Error\ |\}.eval_{head}\big(\sigma, env, b_{infor}, head\big)
\end{array}
$$

TABLE 16. Semantics of left array values at the expression layer.

$$
\begin{array}{l}
\mathcal{ESE}_{lexpr_{array}} :: type \to index_{array} \to L_{address} \to Env \to memory \to option\ L_{address} \\
\mathcal{ESE}_{lexpr_{array}} := \lambda\ (\tau : type)\ .\lambda\ (name : L_{address}).\lambda\ (env : Env).\lambda\ (\sigma : memory)\ .id_{search}(\tau, name, \sigma, env)
\end{array}
$$

TABLE 17. Semantics of field access at the value layer.

$$
\begin{array}{l}
\mathcal{ESE}_{lexpr_{map}} :: index_{map} \to Env \to memory \to option\ L_{address} \\
\mathcal{ESE}_{lexpr_{map}} := \lambda\ \big(id : index_{map}\big)\ .\lambda\ (env : Env).\lambda\ (\sigma : memory). \\
\Big(id_{map}\big(id, env, \sigma, name, \tau, \tau_{map}, snd\big)\Big)
\end{array}
$$

TABLE 18. Semantics of left constant values at the expression layer.

$$
\begin{array}{l}
\mathcal{ESE}_{lexpr_{const}} :: (\forall\ \tau : type, val\ \tau) \to Env \to memory \to option\ L_{address} \\
\mathcal{ESE}_{lexpr_{const}} := \lambda\ \big(\ v : (\forall\ \tau : type, val\ \tau)\big).\lambda\ (env : Env).\lambda\ (\sigma : memory) \\
\ \{|\ \ Some\ Varray(index, \tau, name) \longmapsto \mathcal{ESE}_{lexpr_{array}}(Tarray\ index\ \tau, name, \sigma, env) \\
\quad Some\ Vmap\big(name\ \big[id_{map}\big]\ (\tau_{map} \Rightarrow \tau)\ snd\big) \longmapsto \mathcal{ESE}_{lexpr_{map}}\big(id_{map}, env, \sigma, name, \tau, \tau_{map}, snd\big) \\
\quad None\ \longmapsto None \\
\ |\}.lexpr_{check}(v)
\end{array}
$$

TABLE 19. Semantics of reference expressions Evar, Efun, Econ, and Epar at the expression layer.

$$
\begin{array}{l}
\mathcal{ESE}_{lexpr_{eaddr'}} :: \big(\forall\ a : option\ L_{address} : expr\ eaddr(a)\ eaddr(a)\big) \to option\ L_{address} \\
\mathcal{ESE}_{lexpr_{eaddr'}} := \lambda\ \big(Eaddr([\![name]\!]) : expr\ eaddr([\![name]\!])\ eaddr([\![name]\!])\big)\ .[\![name]\!]
\end{array}
$$

typing rule constructors in the formal abstract syntax and the respective formal semantics. Moreover, the formal syntax of Lolisa is simplified by encapsulating it in syntax sugar notations $\mathcal{N}$. As shown in Rules 22 and 23, Lolisa is treated as the core formal language, which is transparent to real-world users. The formal syntax and semantics of Lolisa are logically classified into a general component $\mathcal{G}$ and n special components $\mathcal{S}_i$ (see Rule 22 below). A general-purpose

TABLE 20. Semantics of right constant values at the expression layer.

$$\begin{array}{l}\mathcal{ESE}_{rexpr_{const}} :: (\forall\ \tau : type, val\ \tau) \to Env \to memory \to Blc \to option\ value \\ \mathcal{ESE}_{rexpr_{const}} \coloneqq \lambda\big(v : (\forall\ \tau : type, val\ \tau)\big).\lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big). \\ \quad \mathcal{ESV}_{value}(K, v, env, \sigma, b_{infor})\end{array}$$

TABLE 21. Semantics of right struct values at the expression layer.

$$\begin{array}{l}\mathcal{ESE}_{rexpr_{str}} :: \big(\forall\ a : L_{address}, struct_{par}\ a\big) \to Env \to memory \to Blc \to option\ value \\ \mathcal{ESE}_{rexpr_{str}} \coloneqq \\ \lambda\ (K : \mathbb{Z}).\lambda\ \big(head\ \{v_0; v_1; \ldots; v_n\} : struct_{par}\ head\big).\lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big) \\ \quad eval_{str}\big(K, head\ \{v_0; v_1; \ldots; v_n\}, \sigma, env, b_{infor}\big)\end{array}$$

TABLE 22. Semantics of right binary operations at the expression layer.

$$\begin{array}{l}\mathcal{ES}_{rexpr_{bop}} :: \forall(\tau\ \tau'\tau_1\ \tau_2 : type), bop_{\tau_1\ \tau_2} \to expr_{\tau\ \tau_1} \to expr_{\tau'\ \tau_1} \to Blc \to Env \to memory \to option\ value \\ \mathcal{ES}_{rexpr_{bop}} \equiv \\ \forall(\tau\ \tau'\tau_1\ \tau_2 : type), \lambda\ \big(op_2 : bop_{\tau_1\ \tau_2}\big).\lambda\ \big(e_0 : expr_{\tau\ \tau_1}\big).\lambda\ \big(e_1 : expr_{\tau'\ \tau_1}\big).\lambda\ (env : Env).\lambda\ (\sigma : memory). \\ \quad eval_{bop}\Big(\sigma, op_2, \mathcal{ESE}_{rexpr}(n, e_0, \sigma, blc, env), \mathcal{ESE}_{rexpr}(n, e_1, \sigma, blc, env)\Big)\end{array}$$

TABLE 23. Semantics of right unary operations at the expression layer.

$$\begin{array}{l}\mathcal{ES}_{rexpr_{uop}} :: \forall(\tau_0\ \tau_1\ \tau_2 : type), uop_{\tau_1\ \tau_2} \to expr_{\tau_0\ \tau_1} \to Blc \to Env \to memory \to option\ value \\ \mathcal{ES}_{rexpr_{uop}} \coloneqq \\ \lambda\ (\tau_0\ \tau_1 : type).\lambda\ \big(op_1 : uop_{\tau_1\ \tau_2}\big).\lambda\ \big(e : expr_{\tau_0\ \tau_1}\big).\lambda\ (env : Env).\lambda\ (\sigma : memory). \\ \quad eval_{uop}\Big(\sigma, op_1, \mathcal{ESE}_{rexpr}(n, e, \sigma, blc, env)\Big)\end{array}$$

TABLE 24. Semantics of contract declarations.

$$\begin{array}{l}\mathcal{ESS}_{con} :: option\ L_{address} \to list\ (prod\ value\ L_{address}) \to list\ L_{address} \to list\ L_{address} \to Env \to memory \to Blc \to option\ memory \\ \mathcal{ESS}_{con} \coloneqq \lambda\ (oname : option\ L_{address}).\lambda\ \Big(con_{infor} : list\ (prod\ value\ L_{address})\Big).\lambda\ (inherits, inhertis_c : list\ L_{address}). \\ \lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big). \\ \quad \{|\ Some\ addr \longmapsto \\ \quad\quad \{|\ left\ \_ \longmapsto Some\ write_{dir}\Big(\big(\sigma, a, Cid\ \big((cid\ oaddr), con_{infor}, env, b_{infor}\big)\big)\Big) \\ \quad\quad\ right\ \_ \longmapsto Some\ \sigma\ |\}.\big(inherit_{check}(inherits, inhertis_c)\big) \\ \quad\quad None \longmapsto None\ |\}.oname\end{array}$$

TABLE 25. Semantics of variable declarations.

$$\begin{array}{l}\mathcal{ESS}_{var} :: option\ L_{address} \to type \to option\ access \to Env \to memory \to Blc \to option\ memory \\ \mathcal{ESS}_{var} \coloneqq \lambda\ (oname : option\ L_{address}).\lambda\ (\tau : type).\lambda\ (oacc : option\ access).\lambda\ (env : Env). \\ \quad\quad \lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big). \\ \quad \{|\ Some\ addr \longmapsto init_{var}(\sigma, env, b_{infor}, oacc, \tau, name) \\ \quad\quad None \longmapsto Error \\ \quad |\}.oname\end{array}$$

TABLE 26. Semantics of struct declarations.

$$\begin{array}{l}\mathcal{ESS}_{str} :: L_{address} \to struct_{mem} \to option\ memory \\ \mathcal{ESS}_{str} \coloneqq \lambda\ (str_\tau : L_{address}).\lambda\ (mems_\tau : struct_{mem}).\lambda\ (env : Env).\lambda\ (\sigma : memory).\lambda\ \big(b_{infor} : Blc\big). \\ \quad write_{dir}\Big(\sigma, str_\tau, Str_{type}\big(str_\tau, mems_\tau, env, b_{infor}\big)\Big)\end{array}$$

programming language $\mathcal{L}_i$ can be formalized identically to the Lolisa subset $\mathcal{G} \cup \mathcal{S}_i$, where $\mathcal{L}_i$ is symbolically represented by the syntax sugar notation $\mathcal{N}_i$. Here, the syntax symbols are nearly identical to the original syntax symbols of $\mathcal{L}_i$. This method assigns each $\mathcal{L}_i$ with a respective notation set $\mathcal{N}_i$ that satisfies $\mathcal{N}_i \subseteq Lolisa$. This relation, defined by Rule 23 below, also improves the extendibility of Lolisa.

$$Lolisa \stackrel{def}{=} \mathcal{G} \cup \Big(\bigcup_{i=0}^{n} \mathcal{S}_i\Big) \tag{22}$$

$$\forall i \in \mathbb{N}.\mathcal{L}_i \leftrightarrow \mathcal{N}_i \equiv \mathcal{G} \cup \mathcal{S}_i \tag{23}$$

As the respective definitional interpreter of Lolisa, FEther inherits the extensibility advantages of Lolisa, and supports

TABLE 27. Semantics of function call statements.

$$
\begin{array}{l}
\mathit{ESS}_{call} :: \mathbb{Z} \to L_{address} \to option\ (list\ value) \to statement \to Env \to memory \to Blc \to option\ memory \\
\mathit{ESS}_{call} := \lambda\ (pump : \mathbb{Z}).\ \lambda\ (name : L_{address}).\ \lambda\ \big(args : option\ (list\ value)\big).\ \lambda\ (s : statement). \\
\quad \lambda\ (env : Env).\ \lambda\ (\sigma : memory).\ \lambda\ \big(b_{infor} : Blc\big). \\
\{|\ Some\ m_v \mapsto \\
\quad \{|\ Some\ stt \mapsto \mathit{ESS}\left(pump, \sigma, args, set_{env}\left(env, Fun_{call}\left(\big(Efun(Some\ name)\big), args\right)\right), env, (stt + + s)\right) \\
\quad\ Error \mapsto None \\
\quad |\}.\ get_{stt}(m_v)\ \ set_{par}(\sigma', fpars, inputs) \\
\quad Error \mapsto None \\
|\}.\ read_{chck}\big(\sigma, env, b_{infor}, \alpha\big)
\end{array}
$$

TABLE 28. Semantics of modifier statements.

$$
\begin{array}{l}
\mathit{ESS}_{modif} :: statement \to Env \to Env \to memory \to Blc \to option\ memory \\
\mathit{ESS}_{modif} := \lambda\ (s : statement).\ \lambda\ (env : Env).\ \lambda\ (fenv : Env).\ \lambda\ (\sigma : memory).\ \lambda\ \big(b_{infor} : Blc\big). \\
\{|\ Some\ fun_{infor}\big(Efun\big(Tbool, (Some\ a)\big), [fpar_0, fpar_1, \ldots, fpar_n], \Lambda_{fun}, s_{body}\big) \mapsto \\
\quad \{|\ Some\ \sigma' \mapsto \\
\quad\ \{|\ Some\ \sigma'' \mapsto write_{chck}\big(\sigma'', env, b_{infor}, name, s_{body}\big) \\
\quad\ \ None \mapsto None\ |\}.\ init_{re}(\sigma', \Lambda_{fun}, [Tbool]) \\
\quad None \mapsto None\ |\}.\ repeat\ (init_{var}(\sigma, env, fenv, oacc, \tau, fpar_i), [fpar_0, fpar_1, \ldots, fpar_n]) \\
\quad None \mapsto None\ |\}.\ get_{fun}(s)
\end{array}
$$

TABLE 29. Semantics of function statements.

$$
\begin{array}{l}
\mathit{ESS}_{fun} :: statement \to Env \to Env \to memory \to Blc \to option\ memory \\
\mathit{ESS}_{fun} := \lambda\ (s : statement).\ \lambda\ (env : Env).\ \lambda\ (fenv : Env).\ \lambda\ (\sigma : memory).\ \lambda\ \big(b_{infor} : Blc\big). \\
\{|\ Some\ [\sigma_0, \sigma_1, \ldots, \sigma_n] \mapsto \\
\quad \{|\ Some\ true \mapsto \\
\quad\ \{|\ Some\ fun_{infor}\big(Efun\big([\tau_0, \tau_1, \ldots, \tau_n], (Some\ a)\big), [fpar_0, fpar_1, \ldots, fpar_n], \Lambda_{fun}, s_{body}\big) \mapsto \\
\quad\ \ \{|\ Some\ \sigma' \mapsto \\
\quad\ \ \ \{|\ Some\ \sigma'' \mapsto write_{chck}\big(\sigma'', env, b_{infor}, name, s_{body}\big) \\
\quad\ \ \ \ None \mapsto None\ |\}.\ init_{re}(\sigma', \Lambda_{fun}, [\tau_0, \tau_1, \ldots, \tau_n]) \\
\quad\ \ None \mapsto None\ |\}.\ repeat\ (init_{var}(\sigma, env, fenv, oacc, \tau, fpar_i), [fpar_0, fpar_1, \ldots, fpar_n]) \\
\quad\ None \mapsto None\ |\}.\ get_{fun}(s) \\
\quad Some\ false \mapsto \sigma \\
\quad None \mapsto None\ |\}.\ modif_{chck}([\sigma_0, \sigma_1, \ldots, \sigma_n], \sigma_{mtrue}, \sigma) \\
None \mapsto None\ |\}.\ repeat\ (\mathit{ESS}_{call}(\sigma, modi_i), [modi_0, modi_1, \ldots, modi_n])
\end{array}
$$

TABLE 30. Semantics of assignment statements.

$$
\begin{array}{l}
\mathit{ESS}_{assign} :: \mathbb{Z} \to \big(\forall\ \tau_0, \tau_1 : type, expr_{\tau_0\,\tau_1}\big) \to \big(\forall\ \tau_0, \tau_1 : type, expr_{\tau_0\,\tau_1}\big) \to statement \to statement \to option\ statement \\
\mathit{ESS}_{assign} := \lambda\ (K : \mathbb{Z}).\ \lambda\ \big(e_r : (\forall\ \tau : type, expr_{\tau\,Tbool})\big).\ \lambda\ (s_0\ s_1 : statement).\ \lambda\ (\sigma : memory).\ \lambda\ \big(b_{infor} : Blc\big).\ \{|\ Some\ v \mapsto \\
\quad \{|\ Some\ name \mapsto \\
\quad\ \{|\ Some\ v \mapsto \\
\quad\ \ \{|\ true \mapsto \mathit{ESS}\left(K, \sigma, args, env, fenv, Fun_{call}\left(\big(Efun(o\alpha)\big), inputs\right)\right) \\
\quad\ \ false \mapsto write_{chck}(\sigma', env', b_{infor}, addr, v')\ |\}.\ check_v(Fid, v) \\
\quad Error \mapsto Error\ |\}.\ \mathit{ESE}_{lexpr}(K, e, env, \sigma) \\
\quad Error \mapsto Error \\
|\}.\ \mathit{ESE}_{rexpr}(K, e, env, \sigma)
\end{array}
$$

all of its syntaxes and semantics. Moreover, at the same level, any executable semantic $\mathscr{S}_i$ is independent of any other semantic, and all same-level semantics are encapsulated into an independent module $\mathcal{M}$ (see Rules 24 and 25 below). Higher-level semantics can access the APIs of lower-level semantics in different $\mathcal{M}$s, but the implementation details are transparent among the levels. Therefore, as shown in Figure 16, FEther is also easily extendible to new executable semantics in Lolisa without affecting the old semantics.

$$\mathcal{M}odule_K\left[\forall i, j \in \mathbb{N}, i \neq j.\mathscr{S}_i \bigcap \mathscr{S}_j = \emptyset\right] \tag{24}$$

$$\mathscr{S}_{h_m} := \mathcal{M}_a.\ (I_i\ ]\mathscr{S}_i])\oplus\mathcal{M}_b.\ \left(I_j\ \right]\mathscr{S}_j]\big)\ldots\oplus\mathcal{M}_q.\ (I_n\ ]\mathscr{S}_n]) \tag{25}$$

C. LIMITATIONS

Although the novel features in the current version of FEther confer many advantages, some limitations remain.

First, the FEther operates at the Solidity source-code level. Although it will not import vulnerabilities in the compiling process, it cannot guarantee the correctness of the bytecode when the compiler is untrusted. One possible solution is developing a low-level version of FEther that executes the bytecode generated by the compilation. One must then prove equivalence between the Solidity execution results and the respective execution results of the bytecode.

Second, similar to other symbolic execution tools, the FEther traverses all possible execution paths, which risks the path explosion problem. Given that Ethereum smart contracts are lightweight or even featherweight programs, however, the path explosion problem is almost precluded. Moreover, in situations that do meet the path explosion problem, the executions can be merged as invariants by the theorem-proving technology, and proven manually. This solution would exploit the selective symbolic execution pattern of FEther.

Finally, although the current version of FEther achieves property verifications by a few simple automatic tactics, it is not yet fully automated. In occasional situations, programmers must analyze the current proof goal and choose suitable verification tactics. Fortunately, this goal can be achieved by optimizing the design of the tactic evaluation strategies.

VII. CONCLUSIONS AND FUTURE WORK

This paper tackled the final challenge of the FSPVM blueprint: developing a definitional interpreter in Coq. The interpreter, called FEther, supports hybrid symbolic executions of Ethereum smart-contract formal verifications. Based on the GERM memory model, FEther accurately simulates the execution behaviors of Solidity in Coq, and guarantees the consistency between source code and corresponding formal models. For evaluating complete situations during the FEther execution process, we also designed a set of tactics based on the Ltac mechanism of Coq, and combined them into a huge automatic tactic. Using this tactic, the FEther can semi-automatically execute and verify different smart contracts in a symbolic virtual machine with high-level automation and reusability. To demonstrate the power of FEther in the real world, a sample smart contract was verified by conventional symbolic executions in FEther (simultaneous concolic and selective symbolic executions). We also compared the essential features of FEther and the cores of relevant tools. The self-correctness of FEther had been already confirmed by certifying the main functions in Coq. The current version of FEther supports the verification of smart contracts following the ERC20 standard. Finally, we discussed the extensibility and universality of FEther, and proposed an initial scheme for systematically simplifying and extending it, thus supporting the formalization of multiple general-purpose programming languages.

We hope that FSPVM-E will become sufficiently powerful and user-friendly for easy program verification by general programmers. Currently we are formalizing higher-level smart-contract development languages of the EOS blockchain platform [30]. We are also aiming to extend and optimize the current version of FEther. Future versions will support the assembly language of Solidity and corresponding bytecode such as [31]. Next, we will extend the FSPVM-E to support the Ethereum and EOS simultaneously. A formal verified interpreter of these languages will be developed based on the GERM platform. We will then build a general formal verification toolchain for blockchain smart contracts based on the EVI. Finally, we will build a general formal verification toolchain for blockchain smart contracts based on EVI, with the ultimate goal of automatic smart-contract verification.

Algorithm 1 Partial Source Code of Wallet Smart Contract.

```
Function wallet() public payable {
 uint index= indexes[msg.sender];
 uint open; uint close; ui nt quota; uint rate; uint partiLimit;
 uint totalLimit; uint finaiLimit;

 if (privileges[msg.sender]) {
  open= privilegeOpen;
  close = privilegeCiose;
  quota= privilegeQuota;
  rate = RATE_PRf VlLEGE;
 } else {
  open = ordinaryOpen;
  close = ordinaryCiose;
  quota= ordinaryQuota;
  rate = RATE_ORDINARY;
 }

 if(now < open II now > close) {
  revert();
 }
 if (subscription >= TOKEN_TARGET_AMOUNT) {
  revert();
 }
 if (index= 0) {
  revert();
 }
 if(deposits[index]>= quota) {
  revert();
 }
 if(msg.val ue = O) {
  revert();
 }
 if(msg.val ue% 1000000000000000000 != 0) {
  revert();
 }
 partiLimit =quota- deposits[index];
 total Limit = ((TOKEN_TARGET_AMOUNT - subscription)
 - (TOKEN_TARGET_AMOUNT- subscription)% rate) / rate*
1000000000000000000;

 if(parti Limi t <= totaiLimit) {
  final Limit = partiLimit;
 } else {
 final Limit = total Limit;
 }

 if(msg.value <= final limit) {
  safe.transfer(msg.value);
  deposits[index] += msg.value;
  subscription += msg.val ueI I 000000000000000000 * rate;
  Transfer(msg.sender, msg.value);
 } else {
  safe.transfer(fi naiLi mit);
  deposits[index] += finallimit;
  subscription += final Limit I I 000000000000000000 *rate;
  Transfer(msg.sender, finaiLimit);
  msg.sender.transfer(msg.vaI ue- fi naI Limit);
 }
}
```

APPENDIX A

The executable semantics of FEther are given in Table 11 to 30.

APPENDIX B

See Algorithm 1.

ACKNOWLEDGMENTS

The authors wish to thank Marisa for the kind assistance in the verification of this experiment, and Enago for its linguistic assistance during the preparation of this manuscript.

REFERENCES

[1] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Accessed: 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[2] *Ethereum Solidity Documentation*. Accessed: Jul. 2, 2018. [Online]. Available: https://solidity.readthedocs.io/en/develop/
[3] (2016). *The DAO Attacked: Code Issue Leads to $60 Million Ether Theft*. Accessed: Jun. 17, 2017. [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/
[4] *Ethereum Parity Hack May Impact ETH 500,000 or $146 Million*. Accessed: Dec. 2, 2017. [Online]. Available: https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/
[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Securit*, New York, NY, USA, Oct. 2016, pp. 254–269.
[6] Z. Yang and H. Lei. "A general formal memory framework in Coq for verifying the properties of programs based on higher-order logic theorem proving with increased." Accessed: Mar. 27, 2018. [Online]. Available: https://arxiv.org/abs/1803.00403
[7] Z. Yang and H. Lei, "Formal process virtual machine for smart contracts verification," *Int. J. Performability Eng.*, vol. 14, no. 8, pp. 1726–1734, Aug. 2018.
[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, 2008, pp. 209–224.
[9] B. Ekici *et al.*, "SMTCoq: A plug-in for integrating SMT solvers into Coq," in *Proc. Int. Conf. Comput. Aided Verification*, 2017, pp. 126–133.
[10] V. R. Pratt, "Semantical consideration on floyo-hoare logic," in *Proc. SFCS*, Oct. 1976, pp. 109–121.
[11] G. Roşu and A. Ştefănescu, "From hoare logic to matching logic reachability," in *Proc. Int. Symp. Formal Methods*, vol 7436, 2018, pp. 387–402.
[12] *The Coq Proof Assistant Reference Manual*. Accessed: Jul. 23, 2018. [Online]. Available: https://coq.inria.fr/distrib/current/refman/
[13] P. Wadler, "Propositions as types," *Commun. ACM*, vol. 58, no. 12, pp. 75–84, Dec. 2015.
[14] Z. Yang and H. Lei. "Lolisa: Formal syntax and semantics for a subset of the solidity programming language." Accessed: Apr. 1, 2018. [Online]. Available: https://arxiv.org/abs/1803.09885
[15] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Accessed: Apr. 2018. [Online]. Available: http://yellowpaper.io/
[16] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, vol. 10323, vol. 2017, pp. 520–535.
[17] D. P. Mulligan, S. Owens, K. E. Gray, L. Ridge and P. Sewell, "Lem: Reusable engineering of real-world semantics," in *Proc. ICFP*, Sep. 2014, vol. 49, no. 9, pp. 175–188.
[18] E. Hildenbrandt *et al.*, "KEVM: A complete formal semantics of the ethereum virtual machine," in *Proc. IEEE 31st Comput. Secur. Found. Symp. (CSF)*, Jul. 2018, pp. 204–217.
[19] H. Barendregt and E. Barendsen, "Autarkic computations in formal proofs," *J. Autom. Reasoning*, vol. 28, no. 3, pp. 321–336, Apr. 2002.
[20] *Mythril Documentation and User's Manual*. Accessed: Apr. 23, 2018. [Online]. Available: https://github.com/b-mueller/mythril/
[21] H. Xi, C. Chen, and G. Chen, "Guarded recursive datatype constructors," *ACM SIGPLAN*, vol. 38, no. 1, pp. 224–235, Jan. 2003.
[22] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*, Amsterdam, The Netherlands, 1999, pp. 193–207.
[23] P. Barbara, "Semantics of typed lambda-calculus with constructors," in *Logical Methods in Computer Science*. Braunschweig, Germany: Technische Universität Braunschweig, 2011, pp. 561–576. [Online]. Available:https://lmcs.episciences.org/page/lmcs-ev
[24] Z. Yang and H. Lei, "Optimization of executable formal interpreters developed in higher-order logic theorem proving systems," *IEEE Access*, vol.5, pp. 70331–70348, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8531607
[25] *The CompCert C verified Compiler: Documentation and User's Manual*. Accessed: Apr. 23, 2018. [Online]. Available: http://compcert.inria.fr/man/manual.pdf
[26] R. O'Connor, "Simplicity: A new language for blockchains," in *Proc. Workshop Program. Lang. Anal. Secur.*, New York, NY, USA, Oct. 2017, pp. 107–120.
[27] A. Bove and P. Dybjer, "Dependent types at work," in *Proc. LNCS*, 2009, pp. 57–99.
[28] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 265–310, Nov. 1992.
[29] D. Miller, "Formalizing operational semantic specifications in logic," *Electron. Notes Theor. Comput. Sci.*, vol. 246, pp. 147–165, Aug. 2009.
[30] *EOS Blockchain Platform*. Accessed: Apr. 23, 2018. [Online]. Available: https://eos.io/
[31] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in Isabelle/HOL," in *Proc. 7th ACM SIGPLAN Int. Conf. Certified Programs Proofs*, Jan. 2018, pp. 66–77.
[32] I. Sergey, A. Kumar, and A. Hobor. (2018). "Scilla: A smart contract intermediate-level language." [Online]. Available: https://arxiv.org/abs/1801.00687

ZHENG YANG received the bachelor's degree from the School of Information and Software Engineering, University of Electronic Science and Technology of China, in 2017, where he is currently pursuing the Ph.D. degree. His research interests include programming language theory, formal methods, and program verification.

HANG LEI received the Ph.D. degree in computer science from the University of Electronic Science and Technology of China, China, in 1997. After graduation, he conducted research in the fields of real-time embedded operating systems, operating system security, and program verification, as a Professor with the Department of Computer Science, University of Electronic Science and Technology of China, where he is currently a Professor (a Doctoral Supervisor) with the School of Information and Software Engineering. His research interests include big data analytics, machine learning, and program verification.