Received September 20, 2018, accepted November 5, 2018, date of publication November 12, 2018, date of current version December 18, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2880692

Optimization of Executable Formal Interpreters Developed in Higher-Order Logic Theorem Proving Systems

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 In recent publications, we presented a novel formal symbolic process virtual machine (FSPVM) framework that combined higher-order logic theorem proving and symbolic execution for verifying the reliability and security of smart contracts developed in the Ethereum blockchain system without suffering from standard issues surrounding reusability, consistency, and automation. A specific FSPVM, denoted as FSPVM-E, was developed in Coq based on a general, extensible, and reusable formal memory framework, an extensible and universal formal intermediate programming language, denoted as Lolisa, and a corresponding formally verified interpreter for Lolisa, denoted as FEther. However, our past work has demonstrated that the execution efficiency of the standard development of FEther is extremely low. As a result, FSPVM-E fails to achieve its expected verification effect. This paper addresses this issue by first identifying three root causes of the low execution efficiency of formal interpreters. We then build abstract models of these causes, and present respective optimization schemes for rectifying the identified conditions. Finally, we apply these optimization schemes to FEther, and demonstrate that its execution efficiency has been improved significantly.

INDEX TERMS Formal verification, smart contracts, evaluation strategy, interpreter, higher-order logic theorem proving, security.

I. INTRODUCTION

Blockchain technology [1], such as the Ethereum blockchain system, has been adopted in a wide variety of applications such as cryptocurrency [2] and distributed storage [3]. Among the most widely adopted blockchain systems, Ethereum implements a general-purpose Turing-complete programming language known as Solidity [4]. Solidity allows the development of arbitrary applications and scripts (i.e., programs) that are collectively denoted as smart contracts, which can be executed in a virtual runtime environment named as the ethereum virtual machine (EVM) to conduct blockchain transactions automatically. In addition to smart contracts, a number of other lightweight programs have been recently deployed in critical domains. The growing use of these lightweight programs has led to increased scrutiny of their security because they include properties ranging from transaction-ordering dependencies to mishandled exceptions that make them susceptible to deliberate attacks that can result in direct economic loss [5]–[7]. Therefore, it is crucial to verify the security and reliability of such programs in the most rigorous manner available. Among the available verification approaches, higher-order logic theorem proving is one of the most rigorous technologies for verifying the properties of programs. This approach involves establishing a formal model of a software system, and then verifying the system according to a mathematical proof of the formal model. However, this process suffers from problems associated with consistency, reusability, and automation. One of the available solutions for addressing these problems involves designing a formal symbolic process virtual machine (FSPVM) based on higher-order logic theorem proving technique.

The design and building of a general and powerful FSPVM for certifying and verifying smart contracts operating on multiple blockchain platforms has been an ongoing project undertaken by the present authors for some time. In our recent work [8], we presented a theoretical FSPVM framework based on our proposed extension of Curry-Howard isomorphism, known as execution-verification isomorphism (EVI), for automatically verifying lightweight programs and solving the issues associated with reusability, consistency, and automation in higher-order logic theorem proving technique. Specifically, the proposed theoretical FSPVM framework contains four key elements: (i) EVI, (ii) a general memory formal model, (iii) a high-level formal intermediate language that is equivalent to high level programming languages in the real world, and (iv) a respective formally verified definitional interpreter. Subsequently, we adopted the proposed theoretical FSPVM framework to build an FSPVM, denoted as FSPVM-E, in Coq for the verification of Ethereum smart contracts [9]. FSPVM-E was constructed using a general, extensible, and reusable formal memory (GERM) framework, an extensible intermediate programming language denoted as Lolisa [10], which is a large subset of the Solidity programming language with a mechanized syntax and semantics, and a respective formal interpreter denoted as FEther. The FEther interpreter is the final critical component of FSPVM-E that integrates the trusted core of Coq, GERM, and Lolisa. However, our past work has demonstrated that, if FEther is designed according to the standard approach recommended by most relevant research studies and tutorials regarding programming language formalism and interpreter design (e.g., [11]), its symbolic execution efficiency is extremely low. This low execution efficiency of FEther directly influences the verification efficiency and the automation level of FSPVM-E because FEther is employed for parsing the domain specific language (i.e., Lolisa in the present development), implementing program behavior, modifying the formal memory space, and generating the final logic memory state for program verification, and therefore serves as the proof engine of the overall FSPVM-E framework. As such, this is a crucial issue that must be addressed.

The present work addresses this crucial issue associated with FEther by building a general abstract formal interpreter model to analyze and optimize the design of formal interpreters built in higher-order logic theorem proving assistants. Analysis identifies three essential causes of the low execution efficiency of FEther, which are *call-by-name termination* (CBNT), *information redundancy explosion* (IRE), and *concurrent reduction* (CR). Next, we build abstract models corresponding to CBNT, IRE, and CR, and analyze the models in detail to provide respective methods for addressing each of these issues. Finally, we apply these schemes to optimize FEther, and demonstrate that the execution and verification efficiency of FSPVM-E are improved significantly.

The remainder of this paper is structured as follows. Section 2 introduces relevant past studies regarding the formal verification of virtual machines and programs. Section 3 provides some foundational concepts and definitions required for understanding the present work. Section 4 presents the respective abstract models and analyses specific to CBNT, IRE, and CR. Section 5 describes the solutions established for each issue, and presents experimental verification results based on example smart contracts obtained using FEther after optimization. Section 6 presents the conclusions of our work and the directions of our future efforts.

II. RELATED WORK

The work of this paper was primarily inspired by the symbolic process virtual machine KLEE [12], which is a well-known and successful certification tool based on symbolic execution. However, it must be noted that many recent tools are based on symbolic execution [13], but most of them adopt model checking technology as their foundation, and few are developed in a higher-order logic theorem proving system to enable real-world programs to be symbolically executed, and their properties verified automatically using the execution result. However, FSVPM-E supports the higher-order logic based verification of complex properties. In addition, while the verification efficiency of these previously developed symbolic execution tools is reasonably high, we can expect that FSVPM-E will provide a similarly high verification efficiency when properly optimized. Moreover, FEther is based on a type of higher-order predicate logic that can inductively express all execution states of a program. Therefore, FEther satisfies the condition of completeness in contrast to conventional static or dynamic testing technologies relying on an enumeration of test cases.

In addition to general verification tools, a number of well-known frameworks have been developed for the verification of Ethereum smart contracts. For example, the formal semantic known as KEVM [14] was developed for the formal low-level programming verification of Solidity bytecode on the EVM platform using the K-framework, like the formalization conducted in Lem [15]. Since KEVM is executable, it can run the validation test suite provided by the Ethereum foundation. However, the low-level verification conducted by KEVM makes it poorly suited to high-level programming languages, such as Solidity, which was a primary motivation for the development of FSVPM-E.

A number of interesting projects have been undertaken that employ higher-order logic theorem proving assistants as the fundamental platform. For example, Frama-C [16] is an extensible and collaborative platform dedicated to the source-code analysis of software written in the C programming language. In addition, VST [17] is one of the powerful program verification toolchains based on the CompCert project [18], and SMTCoq [19] is another interesting project for developing automatic theorem proving tools. Unfortunately, these platforms also fail to provide a suitable combination of symbolic execution and higher-order logic theorem proving. Moreover, these studies fail to discuss the verification efficiencies obtained and the optimization schemes employed in their work.

In light of the above analysis of the past studies, we note that the present work represents the first systematic discussion regarding the optimization of symbolic execution efficiency in higher-order logic theorem proving assistants, such as Coq and Isabelle. Moreover, the effect of these optimization schemes is also confirmed by their application to a formally verified interpreter.

$$\begin{tikzcd}
 & & FPL \arrow[d, "develop"] & FPL \arrow[d, "formalize"] & \\
\mathcal{FL} \arrow[r, "yields"] \arrow[d, leftrightarrow, "equivalent"] & FRWprogram \arrow[r, "Input"] & \mathcal{FI} \arrow[r, phantom, "+"] & FMemory \arrow[r, Rightarrow, "execute"] & Formal\ m_{state} \\
\mathcal{L} \arrow[r, "yields"] & RWprogram \arrow[r, "Input"] \arrow[u, dashed, "equivalently\ translate"] & \mathcal{I} \arrow[r, phantom, "+"] & Memory \arrow[r, Rightarrow, "execute"] \arrow[u, "virtualize"] & m_{state}
\end{tikzcd}$$

Logic operating environment

Physical operating environment

FIGURE 1. Equivalence between the execution of a real world program (*RWprogram*) written in a higher-order programming language $\mathcal{L}$ and execution in a logic environment using a high-level formal intermediate language $\mathcal{FL}$, which is equivalent to $\mathcal{L}$, to rewrite *RWprogram* as a formal *RWprogram* (*FRWprogram*) in conjunction with a formal memory framework (*FMemory*) as the memory space and a respective formally verified interpreter ($\mathcal{FI}$).

III. FOUNDATIONAL CONCEPTS AND DEFINITIONS

Real world virtual machines of high-level programming languages, such as Smalltalk, Java, and. Net, typically support bytecode as their instruction set architecture (ISA) and are implemented by translating the bytecode for commonly used code paths into native machine code. In contrast, an FSPVM takes the specification functional programming language (*FPL*) provided by a higher-order theorem proof assistant, such as Gallina in Coq, as the bytecode, the formal memory framework (*FMemory*) as the memory space, and the trusted core of the proof assistant as the CPU. However, the trusted core of a proof assistant has only two functions, i.e., evaluating and proving. Since, the fundamental environment provided by Coq is not sufficient to symbolically execute programs written by a mainstream higher-order programming language $\mathcal{L}$, and thereby obtain logic memory states. Therefore, the proof environment of higher-order theorem proof assistants must be extended. A blueprint for the previously proposed logic memory state generation process [8] is illustrated in Fig. 1, where a high-level formal intermediate language $\mathcal{FL}$, which is equivalent to $\mathcal{L}$, is adopted for rewriting a real-world program (*RWprogram*) as a formal *RWprogram* (*FRWprogram*), and the respective formally verified interpreter ($\mathcal{FI}$) is formalized in the FSPVM. The executable semantics of $\mathcal{FL}$ play the role of the ISA of $\mathcal{FI}$. In addition, $\mathcal{FI}$ plays the role of the core of the execution engine in the FSPVM whose task is to simulate the real execution process of *FRWprograms* and generate logic memory states. FSPVM-E is totally developed in Coq with *FMemory* based on the GERM model, $\mathcal{FL}$ specified as Lolisa, and $\mathcal{FI}$ specified as FEther.

The abstract syntax of Lolisa includes contract declaration (*Contract*), modifier declaration (*Modifier*), variable declaration (*Var*), structure declaration (*Struct*), assignment (*Assign*), return (*Return*), multi-value return (*Returns*), throw (*Throw*), skip (*Snil*), function definition (*Fun*), while loop ($Loop_{while}$), for loop ($Loop_{for}$), function call (Fun_{call}), conditional (*If*), and sequence (*Seq*) statements. However, the issues specific to CBNT, IRE, and CR, which form the basis of the present work, are exclusively related to only *Seq* statements. Therefore, only the syntax details of the *Seq* constructor

$$\begin{aligned}
\textit{Statement:}\quad s ::= &\ Contract \mid Modifier \mid Var \mid Struct \mid Assignv \\
&\mid Return \mid Returns \mid Throw \mid Snil \\
&\mid Fun \mid Loop_{while} \mid Loop_{for} \mid Fun_{call} \mid If \mid Seq\ (s, s')
\end{aligned}$$

FIGURE 2. Abstract syntax of Lolisa sequence (*Seq*) statements.

are explicitly defined in Fig. 2. Details regarding the other statements employed by Lolisa are reported in our previous work [10]. In addition, the development of FEther in Coq and its verification process will be simplified if the *FRWprograms* written in Lolisa are maintained as structural programs. To ensure this condition, the semantics of Lolisa are made to adhere to the following pointer counter axiom.

Axiom (Program Counter): Suppose that, for all statements s, if s is the next execution statement, it must be the head of the statement sequence in the next execution iteration.

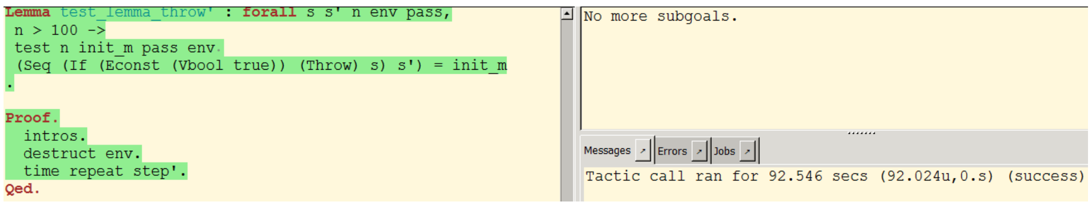
TABLE 1. State functions.

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

Table 1 summarizes the state functions used in the dynamic semantic definitions. Table 2 lists the helper functions used to calculate commonly needed values from the current state of programs. All of these state functions are encountered in the following discussion. Components of specific states are denoted by the appropriate Greek letter subscripted by the state of interest. As shown in Table 2, the context of the formal memory space is denoted by M, where σ is employed to denote a specific memory state, and the context of the execution environment is represented as $\mathcal{E}$. Furthermore, we assign Ω as the native value set of the basic logic system. Also, the proof evaluation will execute in the proof contexts, which we will denote as Γ, Γ_1, etc. For brevity in the following discussion, we assign $\mathcal{F}$ to represent the overall formal system. All the following sections present semantics evaluation relations of the form $\sigma_0 \ \Downarrow_{stt} \ \langle\sigma_1\rangle$, where σ_0 and σ_1 are the initial and final memory states, respectively,

TABLE 2. Helper functions.

set_{env}	Changes the current environment	env_{check}	Validates the current environment
Statement outcomes: out ::=	normal	continue with the next statement	
	| stop	stop executing current statement	
	| error	stop executing current statement with error message	
	| exit	function exit	



```
Lemma test_lemma_throw' : forall s s' n env pass,
 n > 100 ->
 test n init_m pass env
 (Seq (If (Econst (Vbool true)) (Throw) s) s') = init_m
.

Proof.
  intros.
  destruct env.
  time repeat step'.
Qed.
```

```
No more subgoals.
```

```
Tactic call ran for 92.546 secs (92.024u,0.s) (success)
```

FIGURE 3. Evaluation time required for Example 1 by the non-optimized FEther.

v_0 represents the form of Lolisa syntax being defined, and the nature of v_1 depends on the precise evaluation relation being defined. The terms *env* and *fenv* represent the current execution environment and the super environment, respectively, defined in [10].

IV. PROBLEM ANALYSIS

As discussed, the computational efficiency of the previous development of FEther was not sufficiently high to execute and verify formal programs written in Lolisa. A simple example of this is illustrated by the conditional statement as follows:

$$if_{throw} \overset{\text{def}}{=} \forall \left(s, s' : statement\right), if\ (true)\ \{throw(); \}\ else\ \{s; \}\ s' \qquad \text{(Example 1)}$$

where *throw*() is a widely defined special function in programming languages like JavaScript that is called to throw out an executing program. This simple code segment will execute *throw*() to throw out an executing program and return the initial memory state m_{init}. However, as shown in the Fig. 3, executing (i.e., verifying) this very simple code segment using the non-optimized development of FEther requires an execution time of 92.546 s, which is unacceptably long.

First, we must obtain an objective appraisal of the computational efficiency of FEther. To this end, we employed the example smart contracts given in [4] as the testing data set, and evaluated the maximum execution time required by FEther. These example smart contracts include about 300 lines. We employed 5 identical personal computers with equivalent hardware of 8 G memory and a 3.20 GHz CPU, and equivalent software, including Windows 10 and CoqIDE 8.6. Each computer executed the same data set 100 times to obtain the peak execution times of the FEther evaluation process. In addition, we set an execution time limitation of 3600 s to obtain enough experimental information. The results are shown in Fig. 4. If the initial arguments of the programs are defined inductively using quantifiers (such as ∀ and ∃) to logically express all possibilities of the arguments, programs will exceed the time limitation after executing more than about 15 lines. If the initial arguments are defined as specific values, programs that are greater than 30 lines will exceed the limitation.

After analyzing the results given in Fig. 4, we can determine that the standard implementation of FEther, which requires that sequence statements be defined explicitly, will generate a volume of logic information in the current context Γ_c that will exceed the affordable range of higher-order logic theorem proving assistants, making the evaluation efficiency extremely low. As discussed, the extremely low computational efficiency is specifically caused and exacerbated by the three crucial problems CBNT, IRE, and CR. These problems are analyzed and defined in the following sections.

Problem 1 (Call-by-Name Termination): The root of CBNT is caused by the evaluation strategy of lambda calculus while FEther is evaluating the semantics of *Seq* statements.

First, we note that the essence of a formal interpreter $\mathfrak{FI}$ employed as part of an FSPVM framework is a large recursive function written in the specification language provided by a higher-order logic theorem proving system. The type of $\mathfrak{FI}$ can be abstractly defined as follows.

$$\mathfrak{FI} :: args \to optional\ memory \to FRWprogram \to optional\ memory \tag{1}$$

Because almost all mainstream higher-order logic theorem proving systems adopt higher-order lambda calculus as the fundamental theory of their specification language, the symbolic execution process P_{exe} is equivalent with the process of evaluating recursive functions P_{eval} in a theorem proving system following higher-order lambda calculus, as indicated by the following expression.

$$\Omega, M, \mathcal{F} \vdash_{ins} P_{exe} \equiv P_{eval} \tag{2}$$

Next, we note that the sequence (*Seq*) statement is one of the most essential statements that is used in formal semantics,

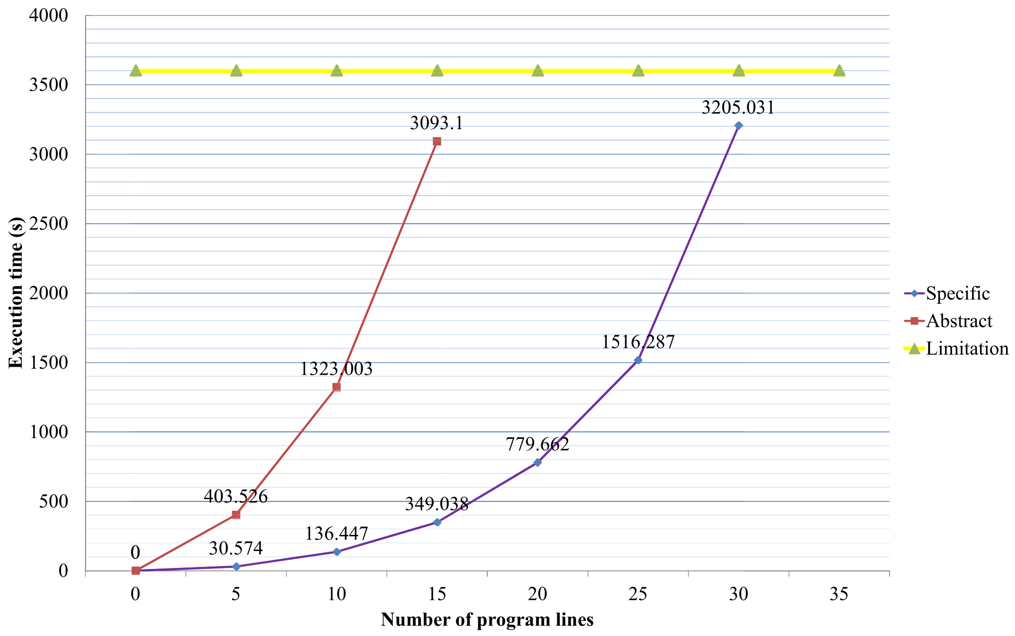


FIGURE 4. Maximum evaluation times of FEther for example smart contracts given in [4].

including operational semantics, denotational semantics and axiom semantics, to connect the remaining types of statements. In most relevant research studies and tutorials regarding programming language formalism and interpreter design (e.g., [11]), it is standard to explicitly define the abstract syntax and semantics of *Seq* statements using *Seq* constructors. For example, the formal semantics of *Seq* statements in Lolisa are defined explicitly according to rules EVAL-STT-SEQ1 and EVAL-STT-SEQ2 below [9]. In Coq, statements are mechanized as an inductive type *statement* that is constructed by specific statement constructors, representing statement tokens. Therefore, an *FRWprogram* can be summarized according to expression (3), as shown at the bottom of this page.

According to EVAL-STT-SEQ1 and EVAL-STT-SEQ2, using the standard approach to evaluate a valid *Seq* statement s, where $s \equiv Seq s_0 s_1$, s_0 and s_1 represent two arbitrary statements, and $s_0 \neq Seq\,(*)$, requires that s be processed according to the algorithm given in Table 3, where $\mathfrak{FI}$ is defined as a partial function that returns an *option* type to indicate success or failure.

Since *FRWprograms* are guaranteed to be structural programs by applying the *Program Counter* axiom, we can directly employ a pattern matching mechanism to obtain the next instruction. However, as shown in Fig. 2, *Seq* is also a constructor of the *statement* type. Therefore, $\mathfrak{FI}$ must first determine whether the head statement in the current context Γ_c is a *Seq* statement. If this is the case, $\mathfrak{FI}$ must evaluate the statement s_0 stored in *Seq* again by recursively calling itself. In higher-order logic theorem proving assistants, such as Coq, a recursive function will create a new logic proof context Γ' each time a β-reduction is applied. Therefore, the current $\mathfrak{FI}$ evaluation belongs to Γ_c, and the expression $\mathfrak{FI}\left([\![m_{state}]\!], env, fenv, args, s_0\right)$ in ***Step*₂** of the algorithm in Table 3 belongs to Γ'. Most specification languages based on higher-order lambda calculus, such as Gallina, are a type of non-Turing-complete *FPL* that supports passing functions as arguments to other functions, returning functions as the

$$\frac{\begin{array}{l} M \vdash \sigma\mathcal{E}, M, \mathcal{F} \vdash s_0, s_1 \\ s_0 \neq Seq\left(s, s'\right) \wedge \sigma \Downarrow_{s_0} \left(\sigma', normal\right) \\ env_{check}\left(env, fenv\right) \hookrightarrow true \wedge set_{gas}\left(Seq\left(s_0, s_1\right), env\right) \hookrightarrow Someenv' \end{array}}{\mathcal{E}, M, \mathcal{F} \vdash \langle \sigma, env, fenv, seq(s_0, s_1)\rangle \Rightarrow \langle \sigma' \Downarrow_{s_1}, env', fenv, normal\rangle} \quad \text{(EVAL-STT-SEQ1)}$$

$$\frac{\begin{array}{l} M \vdash \sigma\mathcal{E}, M, \mathcal{F} \vdash s_0, s_1 \\ s_0 \neq Seq\left(s, s'\right) \wedge \sigma \Downarrow_{s_0} \left(\sigma', error\right) \\ env_{check}\left(env, fenv\right) \hookrightarrow false \vee set_{gas}\left(Seq\left(s_0, s_1\right), env\right) \hookrightarrow None \end{array}}{\mathcal{E}, M, \mathcal{F} \vdash \langle \sigma, env, fenv, seq(s_0, s_1)\rangle \Rightarrow \langle \sigma', env', fenv, error\rangle} \quad \text{(EVAL-STT-SEQ2)}$$

$$FRWprogram \overset{\text{def}}{=} \left(Seq s_0 \left(Seq s_1 \left(\ldots \left(Seq s_{n-1} \left(Seq s_n Snil\right)\right)\right)\right)\right), \quad \left(s_i \neq Seq\,(*)\,, i \in \mathbb{N}\right) \tag{3}$$

TABLE 3. Standard algorithm defining $\mathcal{FI}$ evaluation.

```
Algorithm $\mathcal{FI}$_enter_point
Function: Fixpoint $\mathcal{FI}$
Input: Initial optional memory state $[\![m_{state}]\!]$, current environment $env$, super environment $fenv$, initial arguments $args$, and valid $FRWprogram$;
Output: Final memory state signed with option type;
Step_0: if $env_{check}(env, fenv) = true$, then $set_{env}(Seq(s_0, s_1), env) \hookrightarrow Some\ env'$ and go to Step_1; else, exit;
Step_1: if $FRWprogram = Seq(s_0, s_1)$, then go to Step_2, else, $(env.K, m_{state}, env, fenv, args, FRWprogram) \Downarrow_{P(FRWprogram)}$;
Step_2: if $s_0 \neq Seq(*)$, then let $[\![m'_{state}]\!] := \mathcal{FI}([\![m_{state}]\!], env, fenv, args, s_0)$ and go to Step_3, else, throw out program;
Step_3: $\mathcal{FI}([\![m'_{state}]\!], env', fenv, args, s_1)$
```

values from other functions, and storing them in data structures [20]. In other words, these specification languages treat all essential components as functions, and have no mutable state like that usually applied in imperative programming languages to store the result generated in Γ'. The standard solution for addressing this condition in functional programming is to employ a *let* expression to connect contexts Γ_c and Γ', and temporarily store the new memory state σ' generated by the process of executing s_0. In this way, σ' can be taken as the initial memory state in the next iteration cycle for executing s_1. However, implementing $\mathfrak{FI}$ according to the standard approach given in Table 3 includes a hidden problem, in that the actual order of expression evaluation is the opposite to that expected. Specifically, if *FRWprogram* is a statement stream connected by a *Seq* constructor, as defined by the expression (4), as shown at the bottom of this page.
the ideal order of *FRWprogram* evaluation should be identical with the order in the real world, as follows:

1) Get the current execution statement $s_{current}$;
2) Evaluate $s_{current}$ and generate the new memory state $[\![m'_{state}]\!]$ using $\mathfrak{FI}$;
3) Call the next statement s_1 in $\mathfrak{FI}$ recursively.

Because of the fundamental processing of $\mathfrak{FI}$, the ideal evaluation order is that the next instruction will not be executed until the current instruction is simplified as a β-normal form [21], where a λ-expression cannot be further simplified by β-reduction. According to the operational semantics, the β-normal form of a instruction in this ideal evaluation process refers to a basic intermediate state that is represented by a formal memory value m_{state} in $\mathfrak{FI}$.

Nonetheless, the standard solution of employing a *let* expression to obtain an ideal evaluation order encounters difficulties with respect to the evaluation strategy of lambda calculus adopted as the fundamental theory by most higher-order logic theorem proving systems. Here, the *let* expression is defined in lambda calculus as a lambda abstraction [22]. For example, ***let*** $fx = y$ ***in*** z means that a function f is defined by $fx = y$, which is equivalent with the lambda expression $(\lambda f.z)\,(\lambda x.y)$, where λ represents the abstraction. The formal definition of the *let* expression is based on the following rule.

$$\textbf{\textit{let}}\ x : T\ \textbf{\textit{in}}\ y \equiv (\lambda x.y) \qquad \text{(LET-ABS)}$$

In addition, the *let* expression allows application and substitution to be applied to other expressions according to the respective rules as follows:

$$x \notin FV\,(y) \Longrightarrow ((\textbf{\textit{let}}\ (x : T) := N\ \textbf{\textit{in}}\ y) \Leftrightarrow (\textbf{\textit{let}}\ x := N\ \textbf{\textit{in}}\ y) \equiv (\lambda x.y)\,N) \quad \text{(LET-APP)}$$

$$x \notin FV\,(y) \Longrightarrow (\textbf{\textit{let}}\ (x : T) := N\ \textbf{\textit{in}}\ y) \equiv (\lambda x.y)\,N \Longrightarrow y[x := M] \quad \text{(LET-SUB)}$$

In rule LET-APP, if $x \notin FV\,(y)$, where $\forall E.FV\,(E)$ represents the free variable set of expression E, then expression N can be applied to expression $(x : T)$ bound in expression y. According to the substitution rule of lambda calculus [23], we can simplify rule LET-APP to obtain rule LET-SUB. Thus, the computational formal semantics given in Table 3 can be abstracted as Table 4, and, according to rules LET-ABS, LET-APP, and LET-SUB, the implementations in Tables 3 and 4 are identical.

As discussed, $\mathfrak{FI}$ is essentially a large recursive function written in a specification language. Therefore, according to rules LET-APP and LET-SUB, the *let* expression for substatement s_0 given in (5) below can be converted into (6), and then evaluated as (7) using the evaluation tactic *simpl* or *cbn* provided by the Coq *tactic* mechanism.

$$\begin{aligned} &\textbf{\textit{let}}\ m'_{state} \\ &\quad ::= \mathfrak{FI}\,(m_{state}, env, fenv, parss, s_0)\ \textbf{\textit{in}} \\ &\qquad \mathfrak{FI}\left(m'_{state}, , env, fenv, parss, s_1\right) \qquad (5) \\ &\left(\lambda\left(m'_{state} : memory\right).\mathfrak{FI}\left(m'_{state}, set_{env}\,(env)\,, fenv, input, s_1\right)\right) \end{aligned}$$

$$\frac{\dfrac{\begin{array}{c} M \vdash m_{state}\mathcal{E}, M, \mathfrak{F} \vdash FRWprogram\mathcal{E}, M, \mathfrak{F} \vdash pars\mathfrak{F} \vdash K \\ FRWprogram \xrightarrow{matches} Seq\,(s_{current}, s_1) \xrightarrow{matches} s_{current} \wedge s_1 \end{array}}{(env.K, [\![m_{state}]\!], env, fenv, pars, FRWprogram) \Downarrow_{P(s_{current})} \xrightarrow{yields} [\![m'_{state}]\!]}}{\mathfrak{FI}\left([\![m'_{state}]\!], set_{env}\,(env)\,, fenv, pars, s_1\right)}, \qquad (4)$$

TABLE 4. Abstract definition of $\mathcal{FI}$ evaluation in lambda form.

$$
\begin{aligned}
&\boldsymbol{fix}\,\mathcal{FI}\ \equiv\\
&\lambda\,(\mathcal{FI}\colon option\ memory \to list\ value \to Env \to Env \to statement \to option\ memory).\\
&\quad \lambda\,(s\colon statement).\,\lambda\,(input\colon list\ value).\,\lambda\,(env, fenv\colon Env).\,\lambda\,(m_{state}\colon memory)..\\
&\quad \{|\ true \mapsto\\
&\qquad \{|\ Some\ env' \mapsto\\
&\qquad\quad \{|\ Seq\ s_0\ s_1 \mapsto\\
&\qquad\qquad \left(\lambda\,(m'_{state}\colon memory).\,\mathcal{FI}(m'_{state}, e', fenv, input, s_1)\right)\\
&\qquad\qquad\quad \left(\mathcal{FI}(m_{state}, env, fenv, input, s_0)\right);\\
&\qquad\qquad \_ \mapsto Some\ m_{state}\ |\}.\,s;\\
&\qquad None \mapsto Some\ m_{state}\ |\}.\,set_{env}(env)\\
&\quad false \mapsto Some\ m_{state}\ |\}.\left(env_{check}(env, fenv)\right)
\end{aligned}
$$

$$(\mathcal{FI}\ (m_{state}, env, fenv, input, s_0)) \tag{6}$$

$$(\mathcal{FI}\ ((\mathcal{FI}\ (gas, m_{state}, input, set_{env}\ (env)\ , fenv, s_0))\ ,\ env, fenv, input, s_1)) \tag{7}$$

TABLE 5. Simple example of a *Call-by-name* evaluation strategy.

$$
\begin{aligned}
&(\lambda\, y\colon int.\,(\lambda\, x\colon int.\, y + x))\ (1+3)\\
&\Rightarrow_{\beta}\ (\lambda\, y\colon int.\,(\lambda\, x\colon int.\, y + x))\ [y \coloneqq (1+3)]\\
&\Rightarrow_{\beta}\ \lambda\, x\colon int.\,1 + 3 + x\\
&\not\Rightarrow_{\beta}\ \lambda\, x\colon int.\,4 + x
\end{aligned}
$$

Here, we can determine that the expression $(\mathcal{FI}(m_{state}, env, fenv, input, s_0))$ is applied directly instead of unfolding $\mathcal{FI}$ and reducing this expression to a normal form as a new memory state. Therefore, the ideal evaluation order is violated. As such, the root cause of results like (7) is the evaluation order of lambda calculus. In Coq, the evaluation tactics *cbn* and *simpl* adopt the *call-by-name* evaluation strategy [24], which means that the λ-expressions under lambda abstraction will not be reduced, and the arguments to a function call will not be evaluated, even though the λ-expressions are not normal forms. A simple example of the *call-by-name* evaluation strategy is illustrated in Table 5, which demonstrates that the expression $(\lambda y : int.\,(\lambda x : int.y + x))\ (1 + 3)$ cannot be reduced to $\lambda x\ :\ int.4 + x$ using this strategy. The only method of simplifying the expression $\lambda x\ :\ int.1 + 3 + x$ is to specify x in such a way as to make the expression free of λ-expressions. Moreover, according to [25], the evaluation tactics *cbn* and *simpl* will attempt first to apply β-reduction and ι-reduction, and will then attempt to apply σ-reductions if necessary. Notice that only transparent constants whose identifier can be reused in the recursive calls are possibly unfolded by *simpl* and *cbn*. Accordingly, the evaluation process is illustrated in Table 6. We note from the table that the logical expression generated by the recursive function will not be simplified until the entire *FRWprogram* connected by the *Seq* constructor has been unfolded completely, which yields the following expression:

$$(\mathcal{FI}_n\ (\mathcal{FI}_{n-1}\ (\ldots (\mathcal{FI}_1\ (\mathcal{FI}_0\ (m_0, s_0, *)))))) \tag{8}$$

TABLE 6. Iterations associated with $\mathcal{FI}$ evaluation.

$$
\begin{aligned}
&\mathcal{FI}(m_{state}, FRWprograms, *)\\
&\xRightarrow{unfold} \mathcal{FI}\left(m_{state}, \left(Seq\ s_0\ \left(Seq\ s_1\ \left(\ldots\left(Seq\ s_{n-1}(Seq\ s_n\ Snil)\right)\right)\right)\right), *\right)\\
&\overset{cbn}{\Rightarrow}_{\beta\iota\sigma} \mathcal{FI}\left(\mathcal{FI}(m_{state}, s_0, *), \left(Seq\ s_1\ \left(\ldots\left(Seq\ s_{n-1}(Seq\ s_n\ Snil)\right)\right)\right), *\right)\\
&\overset{cbn^*}{\Rightarrow}_{\beta\iota\sigma} \cdots\\
&\overset{cbn}{\Rightarrow}_{\beta\iota\sigma} \mathcal{FI}\left(\mathcal{FI}\left(\ldots\left(\mathcal{FI}(\mathcal{FI}(m_{state}, s_0, *), s_1, *)\right)\right), \left(Seq\ s_{n-1}(Seq\ s_n\ Snil)\right), *\right)\\
&\overset{cbn}{\Rightarrow}_{\beta\iota\sigma} \mathcal{FI}\left(\mathcal{FI}\left(\left(\ldots\left(\mathcal{FI}(\mathcal{FI}(m_{state}, s_0, *), s_1, *)\right)\right), s_{n-1}, *\right), (Seq\ s_n\ Snil), *\right)\\
&\overset{cbn}{\Rightarrow}_{\beta\iota\sigma} \mathcal{FI}\left(\mathcal{FI}\left(\left(\ldots\left(\mathcal{FI}(\mathcal{FI}(m_{state}, s_0, *), s_1, *)\right)\right), s_{n-1}, *\right), s_n, *\right)
\end{aligned}
$$

where $\mathcal{FI}_i (i\ \in\ \mathbb{N})$ represents the i^{th} recursive call of $\mathcal{FI}$, and the wildcard $*$ represents irrelevant parameters. Here, all *let* expressions of $\mathcal{FI}$ $(m_{state}, FRWprograms, *)$ have been applied and λ-abstraction has been eliminated. Therefore, expression (8) is free of λ-expressions, and can be unfolded and simplified from the outside to the inside.

According to the above analysis, the result cannot be simplified and evaluated in the actual execution process until all recursive calls of $\mathcal{FI}$ are completed. Therefore, the actual evaluation order is given (9), as shown at the bottom of the next page.

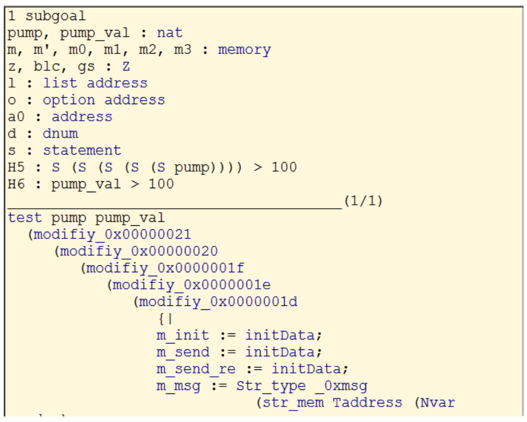
This order can be explicitly stated as follows:

1) Get the current execution statement $\mathrm{s_{current}}$;
2) Call the next statement $\mathrm{s_1}$ in $\mathcal{FI}$ recursively with the function call of $\mathrm{s_{current}}$;
3) Evaluate the entire FRWprogram and generate the final memory state $[\![\mathrm{m_{final}}]\!]$ with $\mathcal{FI}$.

Obviously, in the actual evaluation process, $\mathcal{FI}$ takes the entire *FRWprogram* rather than a single statement as an evaluation unit. Thus, the information generated from the iterations cannot be directly simplified in normal form, and the volume of information of the current context will be too great to simplify. The majority of this information is the statements and respective arguments stored in the massive expression defined by expression (8). The length of this expression is equivalent to the number of statements in an *FRWprogram*. Therefore, we define the size of an *FRWprogram* and the arguments that will occupy the computing resource as *size* (*FRWprogram*), which can be abstracted as follows.

$$infor_{size} \equiv size\,(FRWprogram) \tag{10}$$

In addition, the above analysis indicates that an arbitrary statement within an *FRWprogram* composed of n statements can be evaluated after $2n$ iterations. Thus, the average number of iterations is $n * \frac{2n}{n}$. Hence, the time complexity of this process is $O(n)$.



```
1 subgoal
pump, pump_val : nat
m, m', m0, m1, m2, m3 : memory
z, blc, gs : Z
l : list address
o : option address
a0 : address
d : dnum
s : statement
H5 : S (S (S (S (S pump)))) > 100
H6 : pump_val > 100
______________________________________(1/1)
test pump pump_val
  (modifiy_0x00000021
     (modifiy_0x00000020
        (modifiy_0x0000001f
           (modifiy_0x0000001e
              (modifiy_0x0000001d
                 {|
                 m_init := initData;
                 m_send := initData;
                 m_send_re := initData;
                 m_msg := Str_type _0xmsg
                             (str_mem Taddress (Nvar
```

FIGURE 5. Embedded execution result of FEther in Coq.

The results presented in Fig. 5 verify that the execution process in Coq is identical to the above analysis. Accordingly, the space resource of higher-order logic theorem proving assistants, such as Coq, will be consumed by a large volume of non-normal form logic expressions like that given in (8). Therefore, the size of an *FRWprogram* directly decreases the evaluation efficiency of the proof engine, and may even result in the failure of symbolic execution due to overload.

$$\begin{aligned}
&Inductive\ ident\colon \left[[binder]\ \overline{\tau}\right] \rightarrow\ sort\colon\ = \\
&\quad |\ base_0\colon \left[[binder_0]\ \overline{\tau_0}\right] \rightarrow ident \\
&\quad |\ base_1\colon \left[[binder_1]\ \overline{\tau_1}\right] \rightarrow ident \\
&\quad \ldots \\
&\quad |\ cons_i\colon \left[[binder_i]\ \overline{\tau_\iota}\right] \rightarrow ident \rightarrow \\
&\qquad\qquad ident \rightarrow [\ldots] \rightarrow ident \\
&\quad \ldots
\end{aligned}$$

FIGURE 6. An abstract data type that illustrates the ***call-by-name termination*** (CBNT) problem.

To further clarify this issue, we define an abstract recursive type *ident* in Fig. 6. Here, the identifier *ident* is its name and *sort* is its type. The identifiers $base_0$ to $base_n$ are the names of the *ident* recursive base constructors, and $cons_i$ is the rule that reduces all other cases toward the base constructors. The binders $binder_0$ to $binder_n$ are the quantifiers (such as $\forall$ and $\exists$), and $\overline{\tau}_i$ represents the type set of the other inductive types of respective quantifiers. These terms are optional, which is indicated by placing the terms within square brackets. An inspection of Fig. 6 reveals that $Seq : statement \rightarrow statement \rightarrow statement$ is obviously a special case of *ident*, in that $cons_i : [[binder_i]\,\overline{\tau_i}] \rightarrow ident \rightarrow ident \rightarrow [\ldots] \rightarrow ident$ is specified as the form $cons_{seq} : ident_{statement} \rightarrow ident_{statement} \rightarrow ident_{statement}$. As discussed above, the absence of a mutable state in most *FPLs* requires that, if the parameters in the constructor, which must be evaluated in Γ_c, and are therefore denoted as base parameters, cannot be evaluated, we can only transmit the base parameters into the next recursive circle or discard them. However, it is also clear that the transmission of base parameters is limited because, if the current recursion period transmits n base parameters into the next recursion period, then the next recursion period must in turn transmit $2n$ base parameters. Therefore, the m^{th} recursion period will need to transmit $m*n$ base parameters into the next recursion period. The strict type system employed by higher-order lambda calculus requires that the number of parameters and the respective types of each function must be defined explicitly, and the functions can receive only a fixed number of parameters. However, according to the above analysis, the number of remaining base parameters is dynamic. Hence, it is impossible to transmit the remaining base parameters into the next recursion period.

Unfortunately, most higher-order logic theorem proving assistants adopt *call-by-name* as their essential evaluation strategy. Thus, CBNT is a common problem in all studies where researchers have followed the standard approach for designing a computational proof engine in these higher-order logic theorem proving assistants to evaluate formal programs at the code level, or when researchers have defined very large recursive functions to evaluate recursive datatypes like the abstract datatype given in Fig. 6.

Problem 2 (Information Redundancy Explosion): The cause of IRE is primarily the result of a common programming style, and IRE is exacerbated by CBNT. In most cases, the major component of functions written in *FPLs* consists of conditional statements defined by a pattern matching mechanism. In general, the programming style involves defining these conditions explicitly in the function body rather than encapsulating these conditions as a function interface. In fact, higher-order logic theorem proving assistants actually encourage users to apply this type of programming style in programs. To this end, proof assistants provide wildcard

$$\frac{\dfrac{M \vdash m_{state} \quad \mathcal{E}, M, \mathfrak{F} \vdash FRWprogram \quad \mathcal{E}, M, \mathfrak{F} \vdash par \quad \mathfrak{F} \vdash K \qquad FRWprograms \xrightarrow{matches} Seq\,(s_{current}, s_1) \xrightarrow{matches} s_{current} \wedge s_1}{set_{args}(par) \hookrightarrow par'}}{\mathfrak{F}\mathfrak{I}\,(\mathfrak{F}\mathfrak{I}\,([\![m_{state}]\!], env, fenv, par, s_{current}), env, fenv, par', s_1)}. \tag{9}$$

TABLE 7. Simple example of pattern matching definition in Coq. The definition with wildcards is given on the left, while the actual definition completed by the core of Coq is given on the right.

$Definition\ test\ (a{:}\ T){:}\ bool \coloneqq$	$Definition\ test\ (a{:}\ T){:}\ bool \coloneqq$
$match\ a\ with$	$match\ a\ with$
$\mid A \Rightarrow true$	$\mid A \Rightarrow true$
$\mid \_ \Rightarrow false$	$\mid B \Rightarrow false \mid C \Rightarrow false$
$end.$	$end.$

syntactic sugar to simplify the manual definition, and the built-in interpreter will automatically fill the wildcard during the evaluation process. A simple example of this process is given in Table 7 for an inductive type T that has three constructors A, B, and C. Here, wildcards have been used to define a function on the left side of the table, and the trusted core of the proof assistant automatically fills the wildcards, as shown on the right side of the table.

In addition, as discussed above, the evaluation tactics *cbn* and *simpl* can only be successfully applied to transparent logic expressions free of λ-abstraction [25]. In other words, proof assistants like Coq only apply β-reduction rules for λ-expressions when the top-level structure [26] of the terms of the λ-expressions is deconstructed as specific constructors without λ-abstraction. This is illustrated by the simple example given in Table 7 based on the process. Here, researchers seeking to prove the theorem $\forall\,(a : T)\,,test\ a \wedge false = false$ in a proof assistant like Coq requires the completion of two basic steps. First, according to higher-order lambda calculus, the theorem $\forall\,(a : T)\,,test\ a \wedge false = false$ is equivalent with $\lambda\,(a : T)\,,test\ a \wedge false = false$. Hence, the λ-abstraction should be specified with a term of type T. To avoid confusion, the specific term is bound with the name a_0. Therefore, $\lambda\,(a : T)\,,test\ a \wedge false = false$ is transformed as $(a_0 : T) \vdash test\ a_0 \wedge false = false$. Second, the *test* will be unfolded in Γ_c as follows.

$$(match\ a_0\ with|A \Rightarrow true|\_ \Rightarrow false\ end) \wedge false = false \quad (11)$$

Here, a_0 is the guard clause (also denoted as the matching guard) [27] of the pattern matching in *test*. Because a_0 is a top-level structure term, the expression given in expression (11) cannot be simplified directly. Therefore, a_0 should be deconstructed according to its constructors A, B, and C, which generates three subgoals of the proof, respectively. Each subgoal can be proven easily, where the subgoal of A can be evaluated as true, while the subgoals of B and C can be evaluated as false.

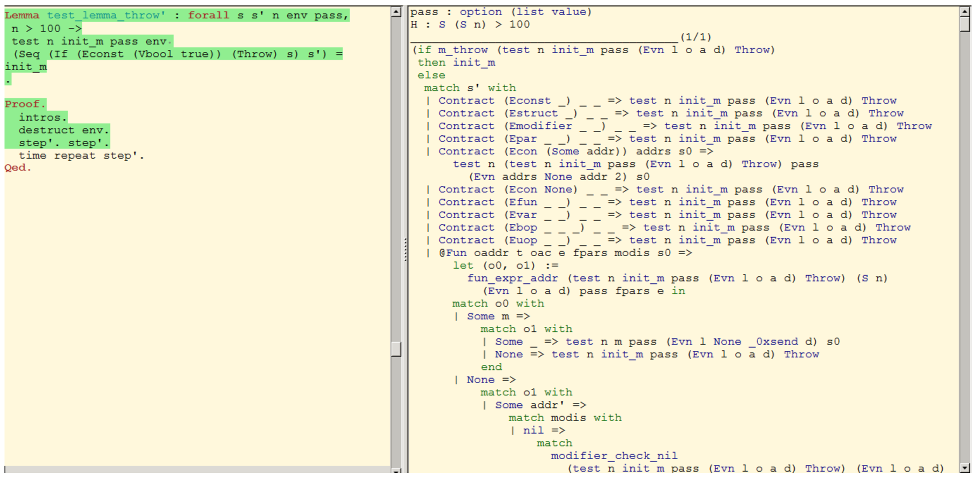
The evaluation and verification process illustrated by the above example is equivalent to that conducted by the proposed FSPVM-E. As discussed previously, FEther is a recursive function that is constructed entirely based on the GERM framework in Coq, so the sum total of logic information, such as λ-expressions and proof terms, must be evaluated and verified in Γ_c by the trusted core of the proof assistant, which contrasts with the process conducted in a real world virtual machine using actual hardware. Therefore, during the ideal process of evaluating *FRWprograms* and generating the new memory state $[\![m'_{state}]\!]$ using $\mathfrak{FI}$, the entire $\mathfrak{FI}$ structure is treated as a function definition, and unfolded in Γ_c.

However, this process includes a hidden problem. A higher-order logic theorem proving system must display all logic information in the same level context to maintain logical consistency. In other words, all logic expressions defined explicitly according to the standard programming style, rather than being encapsulated as a function interface, will be displayed in Γ_c by the proof assistant. Therefore, as discussed above, all wildcards in the definition of $\mathfrak{FI}$ will be automatically filled, and pattern matching conducted without definition encapsulation will be unfolded in Γ_c. However, because the logic information in Γ_c cannot be simplified and evaluated by tactics prior to deconstructing the top-level structure terms of the λ-expressions as specific constructors without λ-abstraction, very large formal programs like the non-optimized version of FEther will generate more than 5000 lines of logic expressions in Γ_c during a single recursive cycle.

In addition, as shown in expression (8), the first step is to recursively apply $\mathfrak{FI}$ for the next statement and nestedly unfold $\mathfrak{FI}_i$ in Γ_c, rather than simplifying the terms of Γ_c. Therefore, pattern matching in the function body cannot be simplified promptly, and all definitions in all branches will be unfolded in Γ_c. Unfortunately, most branches need not be unfolded in Γ_c because the natural deduction systems of higher-order logic theorem proving theory can prune irrelevant branches. Specifically, according to the inductive datatype principle, all constructors of a datatype are mutually exclusive with each other. This is illustrated by expression (11), where it is impossible to generate constructors A and B by evaluating the matching guard of *test* simultaneously. Hence, a single branch at most of a pattern matching process can be derived by a deterministic state Φ_i of Γ_c during the forward reasoning deduction process, and other branches will be filtered out. This is abstracted as follows.

$$\frac{\Gamma_c\,(\Phi_i) \rhd (B_0, B_1, \ldots, B_n)}{[\![B_i]\!]} \quad (12)$$

As such, the definitions belonging to these irrelevant branches will not be unfolded in Γ_c. This condition is illustrated by the *test* shown in Fig. 7. Here, when if_{throw} is evaluated in Coq, the statement s is bound with a universal quantifier, which is an unknown top-level value term. Therefore, for s, which has type *statement*, all its possible pattern matching combinations, including all branches and



```
Lemma test_lemma_throw' : forall s s' n env pass,
 n > 100 ->
 test n init_m pass env
 (Seq (If (Econst (Vbool true)) (Throw) s) s') =
init_m
.

Proof.
  intros.
  destruct env.
  step'. step'.
  time repeat step'.
Qed.
```

```
pass : option (list value)
H : S (S n) > 100
______________________________________(1/1)
(if m_throw (test n init_m pass (Evn l o a d) Throw)
 then init_m
 else
  match s' with
  | Contract (Econst _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Estruct _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Emodifier _ _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Epar _ _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Econ (Some addr)) addrs s0 =>
      test n (test n init_m pass (Evn l o a d) Throw) pass
        (Evn addrs None addr 2) s0
  | Contract (Econ None) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Efun _ _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Evar _ _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Ebop _ _ _) _ _ => test n init_m pass (Evn l o a d) Throw
  | Contract (Euop _ _) _ _ => test n init_m pass (Evn l o a d) Throw
  | @Fun oaddr t oac e fpars modis s0 =>
      let (o0, o1) :=
        fun_expr_addr (test n init_m pass (Evn l o a d) Throw) (S n)
          (Evn l o a d) pass fpars e in
      match o0 with
      | Some m =>
          match o1 with
          | Some _ => test n m pass (Evn l None _0xsend d) s0
          | None => test n init_m pass (Evn l o a d) Throw
          end
      | None =>
          match o1 with
          | Some addr' =>
              match modis with
              | nil =>
                  match
                    modifier_check_nil
                      (test n init_m pass (Evn l o a d) Throw) (Evn l o a d)
```

FIGURE 7. Example illustrating irrelevant logic information unfolding in the proof context.

sub-branches, will be explicitly presented in the current logic context of Coq. This process generated 10,736 lines of logic information in the current proof context. However, according to the logic process of if_{throw}, if the condition expression is true, the result of if_{throw} is fixed as $init_m$ and independent of s. As such, the frustrating truth is that most of the logic information details in the 10,736 lines are redundant and irrelevant. Therefore, although the conditional statement given in Example 1 is very simple, s is a top-level term that cannot be simplified directly, and, because s is unfolded first, but is evaluated last, CBNT exacerbates the problem of IRE, resulting in an excessive volume of unnecessary logic expressions that must be handled by the trusted core of the proof assistant.

We also note that, in addition to the size of *FRWprograms*, the size and complexity of the $\mathfrak{FI}$ structure are also elements influencing the computational efficiency because the function evaluation process typically generates a large volume of logic information combinations, particularly for a very large recursive function like $\mathfrak{FI}$. Therefore, we redefine (10) as follows.

$$infor_{size} \equiv (num\,(FRWprogram) * size\,(\mathfrak{FI})) + size\,(FRWprogram) \quad (13)$$

As mentioned previously, the number of recursion steps required by $\mathfrak{FI}$ is equivalent to the number of statements in an *FRWprogram*, so $num\,(FRWprogram)$ represents the nested depth of $\mathfrak{FI}$ iterations. In addition, $size\,(\mathfrak{FI})$ is mainly determined by the branches of pattern matching clauses. Thus, it can be summarized as follows.

$$size\,(\mathfrak{FI}) \equiv \big(c_{nosub_1} + c_{sub_1} * \big(c_{nosub_2} + c_{sub_2} * (\ldots(c_{nosub_i} + c_{sub_i} * (\ldots))))\big) \quad (14)$$

Here, c_{nosub} represents the average number of constructors without sub-branches, c_{sub} represents the average number of constructors with sub-branches, and c_{sub_j} and c_{nosub_j} represent the number of sub-branches of c_{sub_i} $(i, j \in \mathbb{N}, 0 \le i \le j)$. Obviously, (13) can be summarized as follows:

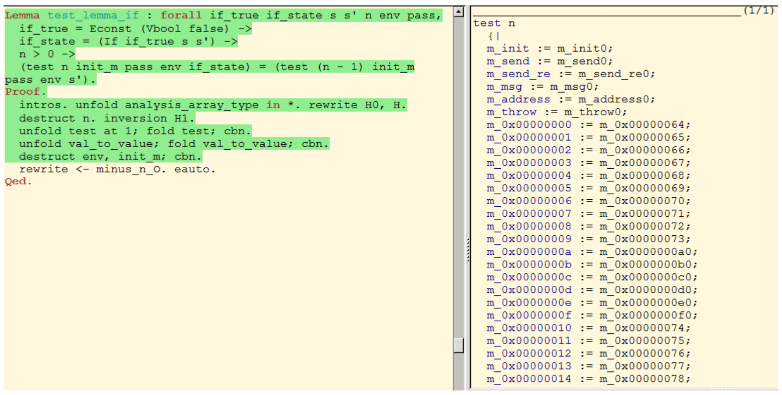
$$infor_{size} \equiv \Big(num(FRWprogram) * \Big(\sum_{i=1}^{n} c_{nosub_i} * c_{sub_{i-1}}!\Big)\Big) + size\,(FRWprogram) \quad (15)$$

where we use $c_{sub_n}!$ to represent the factorial expression $c_{sub_1} * c_{sub_2} * \ldots * c_{sub_n}$, and $c_{sub_0}! = 1$.

In addition, for the FSPVM-E framework proposed herein, FEther is based on our GERM framework, which is defined as a large data structure with a memory space abstracted as a special record type *memory*. Therefore, a memory state σ is treated as a very large record value with a *memory* type in the trusted core of Coq. As such, σ is treated in the Coq computation process as an unknown top-level structure that must be unfolded and deconstructed according to its specific constructors during its evaluation process, and all information stored in the current memory state will be shown in the current context while waiting to be simplified. This is illustrated by the Coq evaluation process shown in Fig. 8 for a simple example. According to the calculus of inductive construction, all identical value terms in the same level context must be deconstructed or unfolded simultaneously to maintain logical consistency. This is defined for FEther as follows.

$$\begin{aligned} &size\,(FEther) \\ &\quad \equiv c_{nosub_1} + (m_1 + a_1) * m_{size} + c_{sub_1} \\ &\qquad * \big(c_{nosub_2} + (m_2 + a_2) * m_{size} + c_{sub_2} \\ &\qquad * \big(\ldots\big(c_{nosub_i} + (m_i + a_i) * m_{size} + c_{sub_i} * (\ldots)\big)\big)\big) \end{aligned} \quad (16)$$

Here, the values in all branches, such as memory space terms m and memory address terms a, which are respectively defined as a massive record type and enumeration type, and which must be generally used in nearly all function modules, will be unfolded simultaneously. We use m_i and a_i to represent the average number of memory space and address value



```
Lemma test_lemma_if : forall if_true if_state s s' n env pass,
  if_true = Econst (Vbool false) ->
  if_state = (If if_true s s') ->
  n > 0 ->
  (test n init_m pass env if_state) = (test (n - 1) init_m
pass env s').
Proof.
  intros. unfold analysis_array_type in *. rewrite H0, H.
  destruct n. inversion H1.
  unfold test at 1; fold test; cbn.
  unfold val_to_value; fold val_to_value; cbn.
  destruct env, init_m; cbn.
  rewrite <- minus_n_O. eauto.
Qed.
```

```
                                                    (1/1)
test n
  {|
  m_init := m_init0;
  m_send := m_send0;
  m_send_re := m_send_re0;
  m_msg := m_msg0;
  m_address := m_address0;
  m_throw := m_throw0;
  m_0x00000000 := m_0x00000064;
  m_0x00000001 := m_0x00000065;
  m_0x00000002 := m_0x00000066;
  m_0x00000003 := m_0x00000067;
  m_0x00000004 := m_0x00000068;
  m_0x00000005 := m_0x00000069;
  m_0x00000006 := m_0x00000070;
  m_0x00000007 := m_0x00000071;
  m_0x00000008 := m_0x00000072;
  m_0x00000009 := m_0x00000073;
  m_0x0000000a := m_0x000000a0;
  m_0x0000000b := m_0x000000b0;
  m_0x0000000c := m_0x000000c0;
  m_0x0000000d := m_0x000000d0;
  m_0x0000000e := m_0x000000e0;
  m_0x0000000f := m_0x000000f0;
  m_0x00000010 := m_0x00000074;
  m_0x00000011 := m_0x00000075;
  m_0x00000012 := m_0x00000076;
  m_0x00000013 := m_0x00000077;
  m_0x00000014 := m_0x00000078;
```

FIGURE 8. Example of the unfolding of memory space terms in the proof context.

terms in a branch, respectively. Because types *memory* and *address* have the same number of constructors, their sizes are identically defined as m_{size}. Therefore, the size of FEther in the current logic context during the evaluation process is summarized as follows.

$$
\begin{aligned}
&infor_{FEther}\\
&\quad \equiv \left(num\,(FRWprogram) * \left(\sum_{i=0}^{n} \left(c_{nosub_i} + (m_i + a_i) \right.\right.\right.\\
&\qquad \left.\left.\left. * m_{size}) * c_{sub_{i-1}}! \right) \right) + size\,(FRWprogram) \qquad (17)
\end{aligned}
$$

Clearly, this represents an exponential growth in the volume of information in a given logic context, which results in a very large volume of logic information that must be evaluated in a current proving context by proof assistants. Therefore, the burden of computation is very large, even if *FRWprogram* is a very simple code segment.

Finally, because IRE is caused by the programming style and the basic features of higher-order proof assistants, we note that normal large programs developed in higher-order logic theorem proving assistants will also cause this problem. Hence, the final $infor_{size}$ can be abstracted as follows.

$$
\begin{aligned}
&infor_{size}\\
&\quad \equiv \left(num\,(FRWprogram) * \left(\sum_{i=0}^{n} \left(es_i + ds_i + c_{nosub_i} \right.\right.\right.\\
&\qquad \left.\left.\left. + \begin{bmatrix} r_0 & \dots & r_i \end{bmatrix} * \begin{bmatrix} size_0 \\ \dots \\ size_i \end{bmatrix} \right) * c_{sub_{i-1}}! \right) \right)\\
&\qquad + size\,(FRWprogram) \qquad (18)
\end{aligned}
$$

Here, r_0 to r_i represent the number of values constructed by different datatypes, and $size_0$ to $size_i$ represent the number of constructors for each respective datatype. In addition, $infor_{size}$ also contains basic expressions and definitions that can be evaluated directly, and the average number of these basic expressions and definitions are defined as es_i and ds_i, respectively.

Problem 3 (Concurrent Reduction): The present implementation of FSPVM-E seeks to combine the advantages of model checking and theorem proving in proof loops and avoid halting problems. To this end, we have employed Bounded Model Checking (BMC) [28] in EVI by allowing FEther to unfold and execute *FRWprograms* only K times. This approach, which is known as *fuel* or *pump*, avoids functions invoking infinite loops, and corresponds to the *gas* approach employed by Ethereum [29], where the evaluation process of semantics are halted if the level of *gas* fails to pass the *gas* checking function.

In the standard FEther design, an equivalent K-limitation is employed to limit the symbolic execution in the statement, expression, and value semantic layers simultaneously, rather than using different K values in different layers. However, as discussed previously, all identical value terms in the same context must be deconstructed or unfolded simultaneously to maintain logical consistency. Therefore, the value of K will be modified and shared among all layers, and the layers that await the execution result will also be forced to be unfolded, as illustrated in Fig. 9. While this process will not cause data corruption owing to the forward reasoning of higher-order logic theorem proving, it will cause IRE to occur more frequently because c_{nosub_i} will become c_{sub_i} in (14).

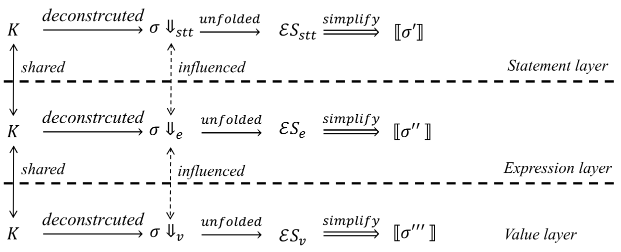
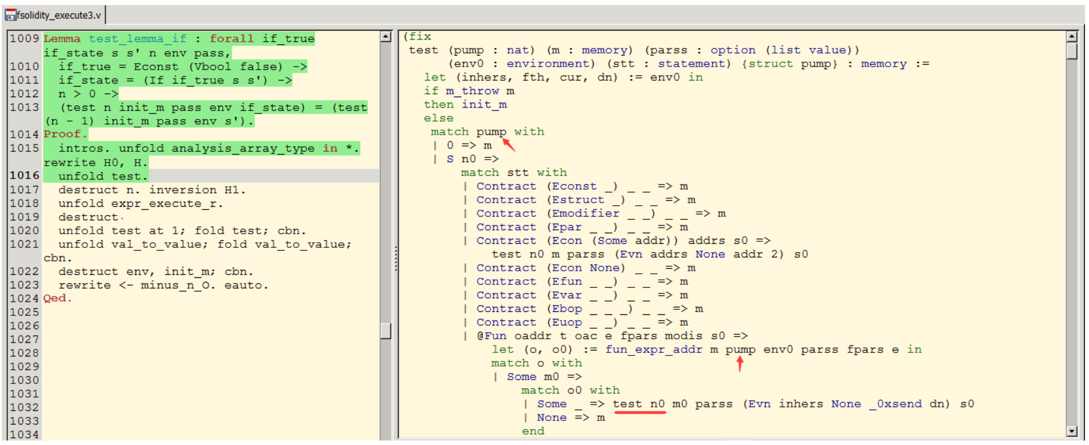


FIGURE 9. Illustration of how the shared *K*-limitation influences all layers simultaneously.

The simple example given as Example 1 above is employed to illustrate the CR problem in Fig. 10. Here, because the different layers share an equivalent K-limitation (i.e., *pump*), when *pump* in the first matching guard, which belongs to the statement layer, is deconstructed, the function*fun_expr_addr*,



fsolidity_execute3.v

```
Lemma test_lemma_if : forall if_true
if_state s s' n env pass,
  if_true = Econst (Vbool false) ->
  if_state = (If if_true s s') ->
  n > 0 ->
  (test n init_m pass env if_state) = (test
(n - 1) init_m pass env s').
Proof.
  intros. unfold analysis_array_type in *.
rewrite H0, H.
  unfold test.
  destruct n. inversion H1.
  unfold expr_execute_r.
  destruct.
  unfold test at 1; fold test; cbn.
  unfold val_to_value; fold val_to_value;
cbn.
  destruct env, init_m; cbn.
  rewrite <- minus_n_O. eauto.
Qed.
```

```
(fix
 test (pump : nat) (m : memory) (parss : option (list value))
      (env0 : environment) (stt : statement) {struct pump} : memory :=
   let (inhers, fth, cur, dn) := env0 in
   if m_throw m
   then init_m
   else
    match pump with
    | 0 => m
    | S n0 =>
        match stt with
        | Contract (Econst _) _ _ => m
        | Contract (Estruct _) _ _ => m
        | Contract (Emodifier _ _) _ _ => m
        | Contract (Epar _ _) _ _ => m
        | Contract (Econ (Some addr)) addrs s0 =>
            test n0 m parss (Evn addrs None addr 2) s0
        | Contract (Econ None) _ _ => m
        | Contract (Efun _ _) _ _ => m
        | Contract (Evar _ _) _ _ => m
        | Contract (Ebop _ _ _) _ _ => m
        | Contract (Euop _ _) _ _ => m
        | @Fun oaddr t oac e fpars modis s0 =>
            let (o, o0) := fun_expr_addr m pump env0 parss fpars e in
            match o with
            | Some m0 =>
                match o0 with
                | Some _ => test n0 m0 parss (Evn inhers None _0xsend dn) s0
                | None => m
                end
```

FIGURE 10. Example based on Example 1 illustrating the *concurrent reduction* (CR) problem.

which takes *pump* as its limitation in the *Fun* branch, will be unfolded at the same time. However, the second matching guard cannot simplify the *stt* in this step. Therefore, *fun_expr_addr* also cannot be simplified after unfolding, and the logic information will remain in the proof context. Unfortunately, this condition will exist in other branches, but most of the branches are irrelevant after the *stt* is deconstructed. This means that a large volume of irrelevant logic information will be generated before the *stt* is deconstructed.

Finally, it should be noted that all functions developed according to BMC in higher-order logic theorem proving assistants will also suffer from CR.

V. OPTIMIZATION

A. SOLUTIONS FOR EACH PROBLEM

According to the above analysis, $\mathcal{FI}$, like all large programs written by a specification language in higher-order logic theorem proving assistants that adopt the *call-by-name* strategy and BMC, will be subject to CBNT, IRE, and CR problems. We therefore present three general optimization methods for addressing each of these problems respectively.

Solution 1 (call-by-Name Termination): To clarify the present discussion, we summarize the cause of CBNT as follows. If the *Seq* cell constructor is defined explicitly in $\mathcal{FI}$, the execution of $\mathcal{FI}$ requires that the pattern matching mechanism successively obtain the *Seq* result and s_0 in the current context Γ_c. Subsequently, CBNT occurs during the evaluation process where s_0 is evaluated in Γ' and the result is bound in Γ_c by the *let* expression. However, this summary indicates that the CBNT problem can be solved directly if s_0 is directly evaluated in Γ_c rather than in Γ'. One of the available solutions is that *Seq* can be defined implicitly. Specifically, this solution involves removing the *Seq* constructor from the *statement* inductive datatype, and defining the sequence statement implicitly using the *list* datatype. The new semantics of the proposed sequence statement are defined in rules NEW-STT-SEQ1 and NEW-STT-SEQ2 below. Here, we use *list* to connect the value constructed by the new statement type *statement'*, which does not have the *Seq* constructor. Because no other semantics are modified, the $\sigma \Downarrow_{stt}$ process is still adopted in the new sequence statement semantics. Based on the above definitions, the equivalence between the standard sequence statement semantics and the new semantics is proven by the Theorem *Sequence Equivalence* given below. However, we first require an intermediate function T to equivalently translate an *FRWprogram* from the *statement* type to the *statement'* type. The abstract definition of the translation function is assigned as $\mathcal{T} :: statement \rightarrow liststatement'$. Its expected function is redefining, where, if *FRWprogram* is $(Seqs_0\ (Seqs_1\ (\ldots(Seqs_{n-1}\ (Seqs_n snil)))))$, the new *FRWprogram* should be $s_0 :: s_1 :: \ldots :: s_{n-1} :: s_n :: nil$. The correctness of T can be guaranteed by the following lemma, where we assume that the implementation of T is correct.

Lemma (Translation):

$$
\begin{aligned}
&\forall\, (K : nat)\, (FRWprogram : statement)\\
&\quad (m : memory)\, (env, fenv : environment).\\
&\quad \forall\, (s_0 s_1 : statement), \quad FRWprogram = Seq\,(s_0, s_1)\\
&\qquad\qquad \rightarrow s_0 \neq Seq\,(*).
\end{aligned}
$$

Goal: $FRWprogram = T^{-1}\ (T\ (FRWprogram))$.

Theorem (Sequence Equivalence): $\forall$ *(K: nat) (FRW program: statement) (m: memory) (env, fenv: environment).*

Judgment 1: $(\forall (s_0 s_1 : statement), FRWprogram = Seq(s_0, s_1) \rightarrow s_0 \neq Seq(*))$.

Goal: $\mathcal{E}_{stt}(m, env, fenv, FRWprogram) = \mathcal{E}'_{stt}(m, env, fenv, T(FRWprogram))$.

Proof: An *FRWprogram* is defined with an inductive type. Therefore, it can be inducted as the basic *Seq* statement s_b, where $s_b \neq Seq\,(*)$ and $Seq\,(s_0, s_1)$. According to the definition of T, $T\ (s_b)$ can be evaluated as $s_b :: nil$, and $T\ (Seq\ (s_0, s_1))$ can be evaluated as $s_0 :: s'_1$, where s'_1 has type *list statement'*.

First, the proof goal above is converted for s to prove $\mathcal{E}_{stt}\ (m, env, fenv, s_b) = \mathcal{E}'_{stt}\ (m, env, fenv, s_b :: nil)$. For the left side, $\mathcal{E}_{stt}\ (m, env, fenv, s_b) = m \Downarrow_{s_b}$

according to the rule EVAL-STT-SEQ1. For the right side, $\mathcal{E}'_{stt}(m, env, fenv, T(FRWprogram)) = m \Downarrow_{(s_b::nil).head} = m \Downarrow_{s_b}$ according to rule NEW-STT-SEQ1. Therefore, $\mathcal{E}_{stt}(m, env, fenv, s_b) = \mathcal{E}'_{stt}(m, env, fenv, s_b :: nil)$ is true, which yields the following judgment.

Judgment2: Judgment1

$$\vdash \forall (s : statement), \mathcal{E}_{stt}(m, env, fenv, s) = \mathcal{E}'_{stt}(m, env, fenv, T(s)).$$

Second, the original proof goal is converted as follows.

Goal': $\forall (ss_2 : statement), \mathcal{E}_{stt}(m, env, fenv, Seq(s_2, s)) = \mathcal{E}'_{stt}(m, env, fenv, T(Seq(s_2, s)))$.

We simplify *Goal'* according to the definition of T as $\forall (ss_2 : statement), \mathcal{E}_{stt}(m, env, fenv, Seq(s_2, s)) = \mathcal{E}'_{stt}(m, env, fenv, s_2 :: s)$. For the left side of *Goal'*, $s_2 \neq Seq(*)$ according to Judgment 1. Therefore, the left side can be evaluated according to the rule EVAL-STT-SEQ1 as follows.

$$\mathcal{E}_{stt}\left(\mathcal{E}_{stt}(m, env, fenv, s_2), env', fenv, s\right) \quad \text{(H1)}$$

Similarly, we can follow the process defined in the rule EVAL-STT-SEQ2 to evaluate the right side of *Goal'* as follows.

$$\mathcal{E}'_{stt}\left(\mathcal{E}'_{stt}(m, env, fenv, s_2 :: nil), env', fenv, s\right) \quad \text{(H2)}$$

According to *Judgment* 2, s can be specified as s_2. Hence, $H_1 : \mathcal{E}_{stt}(m, env, fenv, s_2) = \mathcal{E}'_{stt}(m, env, fenv, s_2 :: nil)$. If the output state of H_1 is an error, $\mathcal{E}_{stt}\left(\mathcal{E}_{stt}(m, env, fenv, s_2), env', fenv, s\right) = \mathcal{E}'_{stt}\left(\mathcal{E}'_{stt}(m, env, fenv, s_2 :: nil), env', fenv, s\right) = error$. Otherwise, H_1 can be assigned as m'. Hence, the left side of *Goal'* is $\mathcal{E}_{stt}\left(m', env', fenv, s\right)$ and the right side is $\mathcal{E}'_{stt}\left(m', env', fenv, s\right)$. Furthermore, s' can be specified as s according to *Judgment* 2. Therefore, $\mathcal{E}'_{stt}\left(m', env', fenv, s\right) = \mathcal{E}_{stt}\left(m', env', fenv, s\right)$. Hence, we successfully prove the Theorem *Sequence Equivalence*.

Because the old semantics are equivalent to the new semantics, and the new semantics also satisfy the Axiom *Pointer Counter*, the new evaluation algorithm for $\mathcal{FI}$ can be redefined according to that given in Table 8 based on the rules NEW-STT-SEQ1 and NEW-STT-SEQ2, where $\mathcal{FI}'$ represents the optimized interpreter. Because the semantics of all other statement types are not modified, the $\Downarrow_{P(FRWprogram)}$ process still represents the process of evaluation.

First and foremost, this modification solves the CBNT problem. This is illustrated in Table 8 by the fact that the evaluation unit at each step is a single statement rather than the entire *FRWprogram*. The *list* datatype is an individual polymorphic recursive type, so the *list* datatype can take the *statement* datatype as its parameter and be specified as a *list* {*statement*} datatype whose list elements are values with the *statement* datatype. If an *FRWprogram* is a statement list, the head of *FRWprogram* is the next executed statement, and it can be evaluated by $\Downarrow_{P(s)}$ directly in Γ_c, rather than first employing $\Downarrow_{P(s)}$ to evaluate sequence statement semantics. Therefore, $\mathcal{FI}'$ will not be employed again to evaluate s_0, and this process will also not be bounded by λ-abstraction due to the *let* expression. Thus, as illustrated in Table 9, the actual evaluation order is the same as the expected order, which takes a statement as an evaluation unit. Therefore, the specification *num* (*FRWprogram*) in (18) is simplified to the specification *num* (*statement*), where *num* (*statement*) can be viewed as the special case of *num* (*FRWprogram*) for an *FRWprogram* with only a single statement. As such, *num* (*statement*) = 1. Hence, the new $infor_{size}$ is optimized as follows.

Here, we note that the component *size* ($\mathcal{FI}$) corresponding to the first term on the right will not be recursively called, and the time complexity for evaluating a sequence statement is reduced as $O(1)$.

Second, the new definition not only reduces $infor_{size}$, but it also strengthens the typing judgment of sequence statements. This can be explained as follows. In contrast to the original definition, the new definitions NEW-STT-SEQ1 and NEW-STT-SEQ2 are not essential for defining the side condition $\forall s : statmnt, \textit{if } s = Seq(s_0, s_1) \textit{ then } s_0 \neq Seq(s, s')$ because the *Seq* constructor is removed from the new *statement*, and the remaining statement constructor is a typing parameter of *list* type, where, according to the *list* type [30], the connection constructor is *cons* {*statement*} :: *statement* → *list statement* → *list statement*. Hence, if the first parameter is not a statement type, the list typing judgment aids the type-checking mechanism in the trusted core of proof assistants to locate errors, such that the side condition need not be defined in the new sequence statement typing judgment.

It should also be noted that another available solution would appear to be defining pattern matching for each parameter of the *Seq* constructor explicitly as

$$\frac{\begin{array}{l} M \vdash \sigma\mathcal{E}, M, \mathcal{F} \vdash s : liststatement' \\ \sigma \Downarrow_{s.head} (\sigma', normal) \\ env_{check}(env, fenv) \hookrightarrow true \wedge set'_{gas}(s.head :: nil, env) \hookrightarrow Some\ env' \end{array}}{\mathcal{E}, M, \mathcal{F} \vdash \langle \sigma, env, fenv \rangle \Rightarrow \langle \sigma' \Downarrow_{s.tail}, env', fenv, normal \rangle} \quad \text{(NEW-STT-SEQ1)}$$

$$\frac{\begin{array}{l} M \vdash \sigma\mathcal{E}, M, \mathcal{F} \vdash s : liststatement' \\ \sigma \Downarrow_{s.head} (\sigma', error) \\ env_{check}(env, fenv) \hookrightarrow truefalse \vee set'_{gas}(s.head :: nil, env) \hookrightarrow None \end{array}}{\mathcal{E}, M, \mathcal{F} \vdash \langle \sigma, env, fenv, s \rangle \Rightarrow \langle \sigma', env', fenv, error \rangle} \quad \text{(NEW-STT-SEQ2)}$$

TABLE 8. New algorithm defining evaluation in the revised $\mathfrak{FI}$ (i.e., $\mathfrak{FI}'$).

```
Algorithm $\mathcal{FI}$_enter_point
Function: Fixpoint $\mathcal{FI}'$
Input: Initial K steps, optional initial memory state $om_{state}$, current environment env, super environment fenv, initial arguments $args$, and valid $FRWprogram'$.
Output: Final memory state signed with $option$ type.
$\boldsymbol{Step_0}$: if $env_{check}(env, fenv) = true$, then go to $\boldsymbol{Step_1}$, else throw out program;
$\boldsymbol{Step_1}$: if $FRWprogram = s_0 :: s_1$, then $env' = set_{env}(s_0 :: s_1, env)$ and go to $\boldsymbol{Step_2}$, else $[\![m_{state}]\!]$;
$\boldsymbol{Step_2}$: if $(m, env, fenv, \varepsilon_0) \Downarrow_{P(s_0)} \xrightarrow{yields} [\![m'_{state}]\!]$, then go to $\boldsymbol{Step_3}$, else throw out program;
$\boldsymbol{Step_3}$: $\mathcal{FI}'([\![m'_{state}]\!], env', fenv, args, s_1)$.
```

TABLE 9. Abstract evaluation process of an FRWprogram in the revised $\mathfrak{FI}$ (i.e., $\mathfrak{FI}'$).

$$\begin{aligned}
&\mathcal{FI}'([\![m_{state}]\!], FRWprograms, *)\\
&\xrightarrow{unfold} \mathcal{FI}'([\![m_{state}]\!], s_0 :: s_1 :: \cdots :: nil, *)\\
&\overset{cbn}{\Longrightarrow}_{\beta\iota\sigma} \mathcal{FI}'\left((env.K, [\![m_{state}]\!], env, fenv, \varepsilon_0) \Downarrow_{P(s_0)}, s_1 :: \cdots :: nil, *\right)\\
&\overset{cbn}{\Longrightarrow}_{\beta\iota\sigma} \mathcal{FI}'([\![m_{state_0}]\!], s_1 :: \cdots :: nil, *)\\
&\overset{cbn^*}{\Longrightarrow}_{\beta\iota\sigma} \cdots\\
&\overset{cbn}{\Longrightarrow}_{\beta\iota\sigma} [\![m_{state_n}]\!]
\end{aligned}$$

follows.

$$\begin{aligned}
&match\ s\ with\ \left|stt_{c_0} \Rightarrow ES_0\left(m_{input}, \ldots, s_0\right)\right| stt_{c_1}\\
&\Rightarrow ES_1\left(m_{input}, \ldots, s_0\right) |\ldots| Seq s_0 s_1\\
&\Rightarrow match\ s_0\ with\\
&\quad \left|stt_{c_0} \Rightarrow \mathfrak{FI}\left(ES_0\left(m_{input}, \ldots, s_0\right), \ldots, s_1\right)\right| stt_{c_1}\\
&\Rightarrow \mathfrak{FI}\left(ES_1\left(m_{input}, \ldots, s_0\right), \ldots, s_1\right) | \ldots
\end{aligned} \tag{EXP-PATTERN}$$

However, this scheme will actually make the problem more serious. This can be illustrated by the abstract recursive datatype given in Fig. 6. Compared with *Seq*, the number of parameters is arbitrary rather than including only s_0, and, if the number of parameters is n, the function must explicitly define pattern matching similar to the pattern matching expression of s_0 in EXP-PATTERN for $n-1$ parameters in the function body. However, the pattern matching for each parameter is identical to the pattern matching results of all other parameters. Therefore, the new size of $\mathfrak{FI}$ can be expressed as $size_{new}(\mathfrak{FI}) \equiv size(\mathfrak{FI}) * n$, which correspondingly increases $infor_{size}$, and makes this solution counterproductive.

Finally, we note that Solution 1 is also a generic solution for the CBNT problem for all large programs. The abstract recursive datatype given in Fig. 6 can further illustrate how our proposed solution of defining a new higher level connection datatype is one of the best solutions for the CBNT problem. As mentioned in the discussion of Problem 1, the feature of functional programming provided by higher-order logic theorem proving assistants requires that the base parameters in the constructor $cons_i : [[binder_i]\,\overline{\tau_i}] \rightarrow ident \rightarrow ident \rightarrow [\ldots] \rightarrow ident$ be evaluated in the current context; otherwise, the information will be lost. However, because the term $cons_i\ id_0\ id_1\ id_2 \ldots ids$ can be equivalently redefined as the term $cons\,(id_0\ cons\ (id_1\ cons\ (id_2 \ldots nil))) \oplus ids$, where the symbol $\oplus$ represents combination, for all datatypes specified from the abstract datatype *ident* in Fig. 6, the CBNT problem can be avoided by employing the $list\,\{ident\}$ datatype to replace $cons_i$. As such, we need only evaluate the head of the $list\,\{ident\}$ in the current recursion period, and the remaining base parameters can be completely transmitted to the next recursion period within the list. Although this increases the number of recursion periods, it solves Problem 1 completely without any other side effects, and the correctness of the translation can be easily proven by defining a lemma like *Translation*.

Solution 2 (Information Redundancy Explosion): While CBNT is solved by Solution 1, the $size\,(\mathfrak{FI})$ component given in (19), as shown at the bottom of this page, remains very large, and this will import an excessive volume of logic information in Γ_c, resulting in IRE.

Under ideal conditions, only necessary information would be included in Γ_c, and the pattern matching results in Γ_c

$$\begin{aligned}
infor_{size} &\equiv (size\,(statement) * num\,(\mathfrak{FI})) + size\,(FRWprogram)\\
&\equiv \left(size\,(statement) * \left(\sum_{i=0}^{n}\left(es_i + ds_i + c_{nosub_i} + \begin{bmatrix} r_0 & \ldots & r_i \end{bmatrix} * \begin{bmatrix} size_0\\ \ldots\\ size_i \end{bmatrix}\right) * c_{sub_{i-1}}!\right)\right) + size\,(FRWprogram)\\
&\equiv \left(\sum\nolimits_{i=0}^{n}\left(es_i + ds_i + c_{nosub_i} + \begin{bmatrix} r_0 & \ldots & r_i \end{bmatrix} * \begin{bmatrix} size_0\\ \ldots\\ size_i \end{bmatrix}\right) * c_{sub_{i-1}}!\right) + size\,(FRWprogram)
\end{aligned} \tag{19}$$

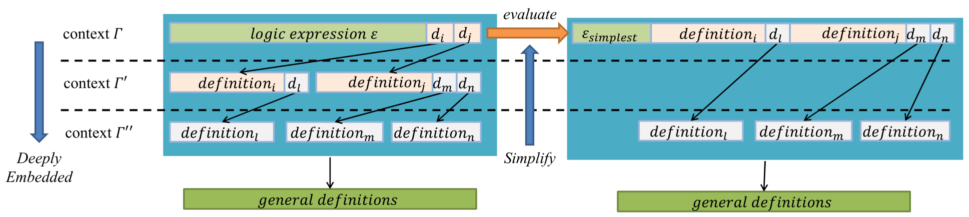


FIGURE 11. Deeply embedded structure for large functions.

could be simplified directly, rather than being held in Γ_c. This would eliminate the factorial term in (19), and thereby maintain a manageable volume of information in Γ_c. These ideal conditions can be achieved through classification and prioritization, and the best scheme to achieve this end is dynamic programming based on the *call-by-name* evaluation strategy. This is an interesting finding, in that, although CBNT is caused by the evaluation order of the *call-by-name* strategy used by tactics in higher-order logic theorem proving assistants, this strategy can be used to solve the IRE problem. According to this strategy, the bodies of all definitions, including functions and values, are stored in their own contexts, and are not evaluated until they are needed in Γ_c. Therefore, this feature can be taken advantage of to hide information not required for use in Γ_c.

The specific process for achieving this end is illustrated in Fig. 11, where the surface context Γ, which is usually Γ_c, consists only of basic expressions ε whose results are the basic constructors that can be evaluated in pattern matching directly. The matching results, their sub-matching results, functions, and special values, such as memory states and addresses, are separately encapsulated into definitions. In particular, the super-matching results must be separated from the sub-matching results. For example, if $definition_i$ is a definition body that contains two pattern matching levels, the sub-pattern matching should be separately defined in a new definition d_l. If the super-pattern matching of $definition_j$ contains two results of sub-pattern matching that are at identical level, the two sub-pattern matching results should encapsulated in definitions d_m and d_n separately. In this way, an optimal evaluation process for $\mathfrak{FI}$ can be obtained based on dynamic programming by breaking the process into sub-processes, and then recursively evaluating the simple results of all sub-processes. This can be abstracted as follows.

$$\frac{\Gamma \triangleright \mathfrak{D}\left(d_\Gamma, \Gamma' \triangleright \mathfrak{D}\left(d_{\Gamma'}, \Gamma'' \triangleright \mathfrak{D}(d_{\Gamma''}, \ldots)\right)\right) \vdash \varepsilon \xrightarrow{reduction} \varepsilon_{nf}}{\Gamma' \triangleright \mathfrak{D}\left(d_{\Gamma'}, \Gamma'' \triangleright \mathfrak{D}\left(d_{\Gamma''}, \ldots\right)\right) \vdash \varepsilon_{nf} \oplus d_\Gamma} \tag{20}$$

Because of the particular feature associated with the *call-by-name* evaluation strategy, the process $\Gamma' \triangleright \mathfrak{D}\left(d_i, d_j, \ldots, \Gamma'' \triangleright \mathfrak{D}\left(d_m, d_n, \ldots\right)\right)$ in (20) is not applied until all ε in Γ have been eliminated completely.

In addition, we also note that Solution 2 improves the level of reusability because general definitions, such as address mapping, can be reused by other definitions rather than being redefined. The rules for building definitions can be summarized by the following Principle.

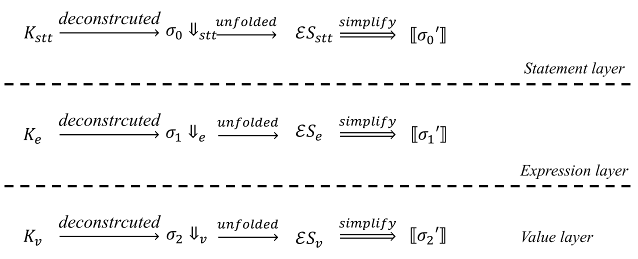

$K_{stt} \xrightarrow{deconstrcuted} \sigma_0 \Downarrow_{stt} \xrightarrow{unfolded} \mathcal{ES}_{stt} \overset{simplify}{\Longrightarrow} [\![\sigma_0']\!]$
Statement layer
$K_e \xrightarrow{deconstrcuted} \sigma_1 \Downarrow_e \xrightarrow{unfolded} \mathcal{ES}_e \overset{simplify}{\Longrightarrow} [\![\sigma_1']\!]$
Expression layer
$K_v \xrightarrow{deconstrcuted} \sigma_2 \Downarrow_v \xrightarrow{unfolded} \mathcal{ES}_v \overset{simplify}{\Longrightarrow} [\![\sigma_2']\!]$
Value layer

FIGURE 12. New *K*-limitation structure for FEther to alleviate the CR problem.

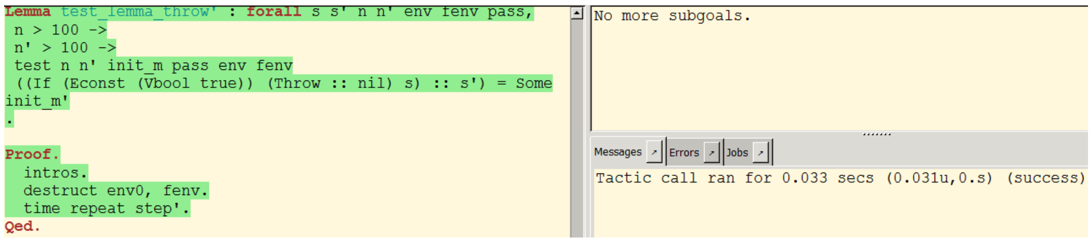
Principle (Classification): A pattern matching result should not contain any explicit sub-matching results, and any top-level structure terms should not be transformed as parameters in function calls.

According to *Classification*, the surface pattern matching results can be evaluated directly, so the surface pattern matching results are classified into the basic expression. The sub-pattern matching results are also encapsulated into definitions in different contexts. Hence, c_{nosub_i} and c_{sub_i} are eliminated from $size\,(\mathfrak{FI})$. Similarly, the essential values r_i should be encapsulated into the general definitions, and es_i, ds_i, and e_i are classified into different context levels. The effect of this process on $size\,(\mathfrak{FI})$ is illustrated as follows.

Here, we assign es_{Γ_i} as the number of basic optimal expressions in the context Γ_i, and assign ds_{Γ_i} as the number of all bound names of definitions, which are the entry points of the respective definition bodies. The term ds_g represents the number of general definitions called by the current context, and es_g represents the respective definition bodies of ds_g. Finally, $size\,(\mathfrak{FI})$ is further simplified using $\overline{\Gamma_d}$, which represents the set of deep contexts given in expression (21), as shown at the bottom of the next page.

According to the *call-by-name* evaluation strategy, definitions in context Γ (d_Γ) do not occupy the computing resource. Moreover, the body of the definitions d_{Γ_i}, which is defined in the $\curvearrowleft_{\Gamma_i \triangleleft \overline{\Gamma_d}}$ process, will not be unfolded until either the expressions of Γ_i (e_{Γ_i}) are eliminated as the β-normal form or the definitions in d_{Γ_i} are necessary. This is defined as follows.

$$infor_\Gamma \equiv e_\Gamma \oplus d_\Gamma \oplus d_g \oplus statement \tag{22}$$



```
Lemma test_lemma_throw' : forall s s' n n' env fenv pass,
 n > 100 ->
 n' > 100 ->
 test n n' init_m pass env fenv
 ((If (Econst (Vbool true)) (Throw :: nil) s) :: s') = Some
init_m'
.

Proof.
  intros.
  destruct env0, fenv.
  time repeat step'.
Qed.
```

```
No more subgoals.
```

```
Tactic call ran for 0.033 secs (0.031u,0.s) (success)
```

FIGURE 13. Simple example for testing the execution efficiency of FEther after optimization.

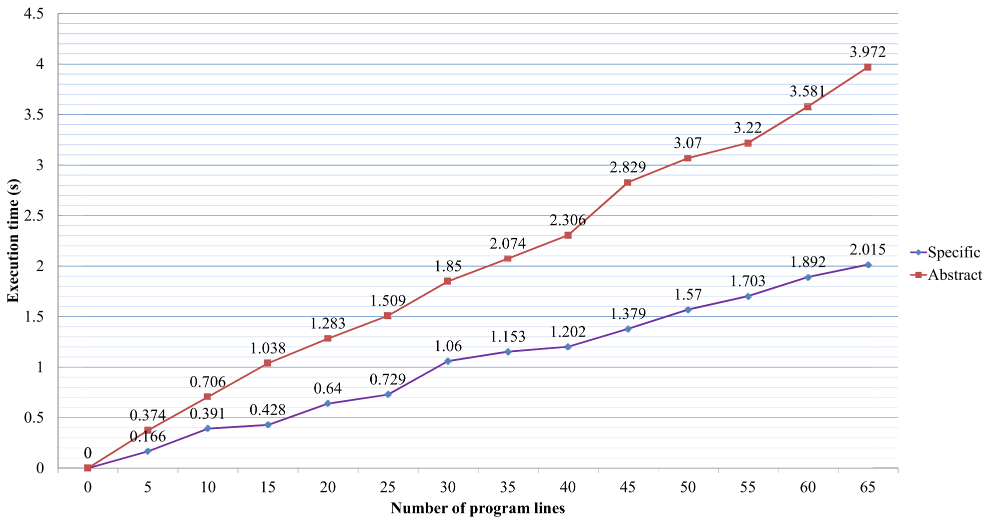


FIGURE 14. Maximum evaluation times of FEther after optimization for example smart contracts given in [4].

In this way, only the basic expressions e_{Γ_i}, the definition entry points d_{Γ_i} in context Γ_i, and the necessary general definitions will be evaluated in the current context, and only e_{Γ_i} will occupy the computing resource. The bodies of the definitions in d_{Γ_i} and deeper level logic terms, such as $d_{\Gamma_{i+n}}$ and $e_{\Gamma_{i+m}}$, will be recursively hidden in the deeper contexts $\overline{\Gamma_d}$, and the trusted core of proof assistants needs only evaluate e_{Γ_i}. Accordingly, the value of $infor_{size}$ for Γ_c can be simplified as follows.

$$infor_{size} \equiv es_{\Gamma} + ds_{\Gamma} + ds_g + size\,(statement) \tag{23}$$

Although this process will increase the number of times the *unfold* operation must be conducted, the influence of this increase on the computational efficiency of $\mathfrak{FI}$ is negligible because the *unfold* operation is one of the simplest atomic operations. Therefore, the computational load of the *unfold* operation in Γ_c on the computing resource is much less than the original evaluation process. In addition, the number of times the *unfold* operation must be conducted is less than or equal to the number of definitions d_Γ in Γ_c, and the number of $e_\Gamma \oplus d_\Gamma$ operations in Γ_c for any $\mathfrak{FI}$ is practically fixed at a constant value. Thus, the value of $infor_{size}$ for Γ_c is influenced only by the complexity of a single statement, which is denoted by $size\,(statement)$. Moreover, the requirement that e_Γ be simplified as the normal form eliminates irrelevant logic expressions from the computational load. Therefore, $infor_{size}$ is significantly reduced, such that the computational load of the trusted core remains within a manageable range. Thus, the IRE problem is solved.

Solution 3 (Concurrent Reduction): Finally, the cause of CR can be easily solved by defining different *pump* limitations for every layer, as shown in Fig. 12. Here, we modify

$$\begin{aligned}
size\,(\mathfrak{FI}) \;\equiv\; & \left(\sum_{i=0}^{n}\left(es_i + ds_i + c_{nosub_i} + \left[\, r_0 \;\ldots\; r_i \,\right] * \begin{bmatrix} size_0 \\ \ldots \\ size_i \end{bmatrix}\right) * c_{sub_{i-1}}! + \sum_{i=0}^{n} e_i\right) \\
\Rightarrow & \left(\left[es_{\Gamma_0} + ds_{\Gamma_0}\right] \curvearrowleft_{\Gamma_0 \lhd \Gamma_1} \left(\left[es_{\Gamma_1} + ds_{\Gamma_1}\right] \curvearrowleft_{\Gamma_1 \lhd \Gamma_2} \left(\ldots\left(\left[es_{\Gamma_i} + ds_{\Gamma_i}\right] \curvearrowleft_{\Gamma_i \lhd \Gamma_{i+1}} (\ldots)\right)\right)\right)\right) + ds_g \curvearrowleft_{\Gamma_{call} \lhd \Gamma_g} (es_g) \\
\Rightarrow & \left(\left[es_{\Gamma_0} + ds_{\Gamma_0}\right] \curvearrowleft_{\Gamma_0 \lhd \overline{\Gamma_d}}\right) + \overline{ds_g} \curvearrowleft_{\Gamma_{call} \lhd \Gamma_g}
\end{aligned} \tag{21}$$

the K-limitation structure adopted in each layer, respectively, which solves the CR problem completely.

B. CASE STUDY

We applied the three proposed optimization schemes in the development process of a new version of FEther for FSPVM-E, and employed FSPVM-E to execute (i.e., verify) the very simple code segment in Example 1. Compared with the result given in Fig. 3 for the non-optimized version of FEther, the results in Fig. 13 indicate that the execution time decreased substantially from 92.546 s to 0.033 s for the optimized version of FEther. As such, the optimized version requires just 3/10000 of the time required by the non-optimized version. The details regarding this new version of FEther will be introduced in a subsequent technological report.

In addition, we also tested the optimized version of FEther under an identical experimental environment and with an equivalent data set as those employed for the results obtained in Fig. 4 by the non-optimized version of FEther. As shown in Fig. 14, the purple line is the peak execution times of *FRWprograms* constructed using specific instructions, and the red line is the peak execution times of *FRWprograms* constructed using abstract instructions. Compared with the results in Fig. 4, we note that both program types exhibit a linear increase in execution time with respect to an increasing number of lines, rather than exponentially, as was obtained using the non-optimized version of FEther.

These experimental results verify that the optimization schemes provide results that conform with our analyses of the causes of CBNT, IRE, and CR. Moreover, the experiments certify that these schemes can optimize FEther, and improve the execution efficiency of the proposed FSPVM-E significantly.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented analyses of the issues denoted as *call-by-name termination*, *information redundancy explosion*, and *concurrent reduction* that reduce the evaluation efficiency of formal interpreters adopted in an FSPVM framework and other large programs developed in higher-order logic theorem proving assistants. We then built abstract models based on these analyses, and developed respective optimization schemes for each issue. Finally, we applied the proposed schemes to optimize the FEther interpreter employed in FSPVM-E. Experimental results verified that the execution efficiency of FEther was improved significantly. We are presently pursuing the optimization and certification of FSPVM-E. Then, we will extend FSPVM-E to support the formal verification of smart contracts on the EOS blockchain platform, which includes the formalization of a subset of the C++ language and the respective interpreter based on the GERM framework.

ACKNOWLEDGMENTS

The authors wish to thank Marisa for the kind assistance in the verification of this experiment, and LetPub (www.letpub.com) 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] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton Univ.Press, 2016.

[3] S. Wilkinson, T. Boshevski, J. Brandoff, and V. Buterin. *Storj A Peer-to-Peer Cloud Storage Network*. Accessed: Dec. 2016. [Online]. Available: https://storj.io/storj.pdf

[4] *Ethereum Solidity Documentation*. Accessed: Jul. 2, 2017. [Online]. Available: https://solidity.readthedocs.io/en/develop/

[5] (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/

[6] *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/

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. CCS*, New York, NY, USA, 2016, pp. 254–269.

[8] Z. Yang, H. Lei, and X. Yang. *A General Formal Memory Framework in Coq for Verifying the Properties of Programs Based on Higher-Order Logic Theorem Proving With Increased*. Accessed: Mar. 2018. [Online]. Available: https://arxiv.org/abs/1803.00403

[9] Z. Yang and H. Lei. *Formal Process Virtual Machine for Smart Contracts Verification*. Accessed: May 2018. [Online]. Available: https://arxiv.org/abs/1805.00808

[10] Z. Yang and H. Lei. *Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language*. Accessed: Apr. 2018. [Online]. Available: https://arxiv.org/abs/1803.09885

[11] S. Blazy and X. Leroy, "Mechanized semantics for the clight subset of the C language," *J. Autom. Reason.*, vol. 43, no. 3, pp. 263–288, Jul. 2009.

[12] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, 2008, pp. 209–224.

[13] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking path explosion in constraint-based test generation," in *Proc. TACAS*, Budapest, Hungary, 2008, pp. 351–366.

[14] E. Hildenbrandt *et al.*, "KEVM: A complete semantics of the ethereum virtual machine," in *Proc. 31st IEEE Comput. Secur. Found. Symp.*, 2018, pp. 204–217.

[15] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Proc. FC*, Curaçao, The Netherlands, 2017, pp. 520–535.

[16] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *Proc. SEFM*, Toulouse, France, 2012, pp. 233–247.

[17] A. W. Appel, "Verified software toolchain," in *Proc. ESOP*, Saarbrücken, Germany, Mar. 2011, pp. 1–17.

[18] *The CompCert C Verified Compiler: Documentation and User's Manual*. Accessed: Apr. 23, 2018. [Online]. Available: http://compcert.inria.fr/man/manual.pdf

[19] B. Ekici *et al.*, "SMTCoq: A plug-in for integrating SMT solvers into Coq," in *Proc. CAV*, Heidelberg, Germany, 2017, pp. 126–133.

[20] R. Burstall, "Christopher strachey—Understanding programming languages," *Higher-Order Symbolic Comput.*, vol. 13, nos. 1–2, pp. 51–55, Aug. 2000.

[21] B. Henk, *The Lambda Calculus: Its Syntax and Semantics*. Amsterdam, The Netherlands: North Holland, 1984.

[22] G. D. Plotkin, "LCF considered as a programming language," *Theor. Comput. Sci.*, vol. 5, no. 3, pp. 223–255, Dec. 1977.

[23] R. Nederpelt and H. Geuvers, *Type Theory and Formal Proof: An Introduction*. Cambridge, U.K.: Cambridge Univ. Press, 2014.

[24] G. D. Plotkin, "Call-by-name, call-by-value and the λ-calculus," *Theor. Comput. Sci.*, vol. 1, no. 2, pp. 125–159, 1975.

[25] *The Coq Proof Assistant Reference Manual*. Accessed: Jul. 23, 2018. [Online]. Available: https://coq.inria.fr/distrib/current/refman/

[26] A. Chlipala, *Certified Programming With Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. Cambridge, MA, USA: MIT Press, 2013.

[27] D. Turner, "Some history of functional programming languages," in *Proc. TFP*, St. Andrews, U.K., 2012, pp. 1–20.

[28] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*, Amsterdam, The Netherlands, 1999, pp. 193–207.

[29] *Ethereum Gas Documentation*. Accessed: Jul. 2, 2017. [Online]. Available: http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html

[30] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

ZHENG YANG received the bachelor's degree from the School of Information and Software Engineering, University of Electronic Science and Technology of China, 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 system, operating system security, and program verification, as a Professor with the Department of Computer Science, University of Electronic Science and Technology of China. He is currently a Professor (doctoral supervisor) with the School of Information and Software Engineering, University of Electronic Science and Technology of China. His research interests include big data analytics, machine learning, and program verification.