软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(6):1717−1732 [doi: 10.13328/j.cnki.jos.006246]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

以太坊中间语言的可执行语义*

韩 宁[1,3], 李希萌[1,2], 张倩颖[1,2], 王国辉[1,2], 施智平[1,2], 关 永[1,3]

[1](首都师范大学 信息工程学院,北京 100048)

[2](电子系统可靠性技术北京市重点实验室(首都师范大学),北京 100048)

[3](北京成像理论与技术高精尖创新中心(首都师范大学),北京 100048)

通讯作者: 李希萌, E-mail: lixm@cnu.edu.cn

摘 要: 智能合约是实现各类区块链应用的核心软件程序.近期,以太坊区块链平台(Ethereum)上的智能合约暴露出大量错误和安全隐患,在国际上引发了智能合约形式化验证的研究热潮.为提供高可信度的验证结果,智能合约程序语言的形式化必不可少.对以太坊中间语言 Yul 进行形式化,首次给出了其类型系统和小步操作语义的形式化定义.该语义为可执行语义(executable semantics),由 120 个 Yul 语言程序组成的测试集进行测试.该工作在 Isabelle/HOL 证明辅助工具中完成,为基于定理证明的智能合约正确性、安全性验证奠定了基础.

关键词: 智能合约;Yul 语言;Isabelle/HOL;形式化语义;以太坊

中图法分类号: TP311

中文引用格式: 韩宁,李希萌,张倩颖,王国辉,施智平,关永.以太坊中间语言的可执行语义.软件学报,2021,32(6):1717−1732. http://www.jos.org. cn/1000-9825/6246.htm

英文引用格式: Han N, Li XM, Zhang QY, Wang GH, Shi ZP, Guan Y. Executable semantics of Ethereum intermediate language. Ruan Jian Xue Bao/Journal of Software, 2021,32(6):1717−1732 (in Chinese). http://www.jos.org.cn/1000-9825/6246.htm

Executable Semantics of Ethereum Intermediate Language

HAN Ning[1,3], LI Xi-Meng[1,2], ZHANG Qian-Ying[1,2], WANG Guo-Hui[1,2], SHI Zhi-Ping[1,2], GUAN Yong[1,3]

[1](College of Information Engineering, Capital Normal University, Beijing 100048, China)

[2](Beijing Key Laboratory of Electronic System Reliability and Prognostics (Capital Normal University), Beijing 100048, China)

[3](Beijing Advanced Innovation Center for Imaging Theory and Technology (Capital Normal University), Beijing 100048, China)

Abstract: Smart contracts are key software components of blockchain applications. Recently, a great number of bugs and security vulnerabilities have been exposed in smart contracts on the Ethereum blockchain. This resulted in extensive research efforts in the formal verification of smart contracts at the international level. To obtain highly trustworthy verification results, the formalization of programming languages for smart contracts is necessary. This work formalizes Yul—the Ethereum intermediate language. The core of this formalization consists of the first formal definitions of the type system and the small-step operational semantics for Yul. The semantics is executable, and it is validated using a test suite of 120 Yul programs. The proposed formalization is performed in the Isabelle/HOL proof assistant. It lays a solid foundation for the formal verification of the correctness and security of Ethereum smart contracts through theorem proving.

Key words: smart contract; Yul language; Isabelle/HOL; formal semantics; Ethereum

* 基金项目: 国家自然科学基金(61572331, 61602325, 61802375, 61876111, 61877040, 62002246); 北京市教育委员会科技计划(KM20190028005, KM202010028010); 中国科学院计算技术研究所计算机体系结构国家重点实验室开放课题(CARCH201920)

Foundation item: National Natural Science Foundation of China (61572331, 61602325, 61802375, 61876111, 61877040, 62002246); General Project of Beijing Municipal Education Commission (KM20190028005, KM202010028010); Open Project of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201920)

本文由“形式化方法与应用”专题特约编辑邓玉欣教授推荐.

收稿时间: 2020-08-30; 修改时间: 2020-10-26; 采用时间: 2020-12-19; jos 在线出版时间: 2021-02-07

区块链(blockchain)是去中心化、分布式的数字账本[1],它能够可靠记录多方事务的处理过程和结果,维护多方共识,并提供交易记录可验证、难篡改、交易历史可追溯等优良特性.以太坊(Ethereum)是目前使用最广泛的公有区块链平台之一[2].以太坊上的智能合约(smart contract)是一种约定多方任务逻辑的计算机程序,能够自动执行并产生经过共识的链上结果.近年来,智能合约已被应用在数字金融、供应链管理、在线游戏等不同领域.然而,许多链上合约暴露出编程漏洞,且针对智能合约的安全攻击屡见不鲜.例如:2016 年 6 月,The DAO 合约因重入漏洞造成超过 6 000 万美元的损失[3];2017 年 11 月,Parity 电子钱包因多重签名漏洞造成约 1.5 亿美元的资金被冻结于智能合约中;2018 年 4 月,美链(BEC)合约因整数溢出漏洞,使得市值数亿的项目惨淡收场.据 CNVD-BC 区块链漏洞子库统计[4],在 2019 年 6 月~2020 年 6 月之间,发现区块链漏洞中,关于智能合约漏洞的数量占据首位.智能合约的正确性和安全性问题,已成为制约区块链落地应用的主要因素之一.

形式化验证是为系统正确性、安全性提供高度保障的有效工具[5].近年来,国际上许多为智能合约提供正确性、安全性保障的研究工作不同程度地采用形式化验证的思想和方法[6].其中,定理证明方法通过将验证任务转化为数学证明,并使用计算机程序检查证明的基本步骤,为验证结果提供极高的可信度.若要可靠验证智能合约的正确性和安全性,仍须明确给出编写智能合约所用程序语言的语义,较为有效的手段是在软件工具(如证明辅助工具)中对语义进行形式化.

在以太坊的技术体系中,Yul 语言是一种结构化的中间语言[7].可由 Solidity[7,8]等高级语言所编写的智能合约经编译得到其 Yul 语言表示,或在 Solidity 的内联汇编中使用 Yul 语言,或直接使用 Yul 语言进行合约的编写.此外,Yul 语言能够被编译到多种低级语言(如不同版本的 EVM 字节码、EWASM 等),使合约能在以太坊平台上执行.Yul 语言具备高级语言中的函数、代码块、控制流结构等抽象,以该语言表示的智能合约相较于字节码合约更容易理解,因而易于参照代码进行形式规约.作为中间语言,Yul 语言的特性相较于 Solidity 等高级语言更为简单,因而更易于形式化.

本文在证明辅助工具 Isabelle/HOL 中,采用小步语义对以太坊中间语言 Yul 进行形式化建模.小步语义属于操作语义中的一种,能够反映程序执行的中间状态[9],因而能够为许多安全性质(security property)的验证提供支撑.本文的形式化语义是可执行语义,因而既能在证明辅助工具中模拟智能合约的执行,又能支撑对智能合约正确性、安全性的定理证明.本文工作的主要贡献有:

(1) Yul 语言类型系统的形式化;

(2) Yul 语言小步语义的定义和形式化;

(3) Yul 语言小步语义的测试.

其中,类型系统的形式化,实现了对 Yul 语言官方文档中自然语言描述的数据类型、变量作用域等规范的严格表达;小步语义的定义和形式化,实现了对 Yul 语言程序执行过程和结果(包括单步执行所消耗的 gas 量)的严格描述;在 Isabelle/HOL 中对语义的测试,保证该语义如实反映 Yul 语言规范以及实际观察的合约执行行为.

Yul 语言具有一定的复杂程度.除去在证明辅助工具中使用高阶逻辑对目标理论进行表述外,本研究的另一难点是通过自然语言文档研读、Yul 程序编译执行实验、形式化代码检视、测试等手段保障 Yul 语言各种特性形式化的准确性,所涉及的语言特性包括数据类型、变量作用域、控制流结构(if,switch,for 等)、可嵌套的函数定义(Yul 是分程序语言)、内部函数调用与外部合约调用、大部分内置函数、异常、燃料(gas)等.本工作为基于定理证明的智能合约正确性和安全性验证奠定了坚实基础.

本工作针对 2020 年初发布的 Yul 0.6.1 版本,在 Isabelle/HOL 中的形式化覆盖了 Yul 0.6.1 中除 break, continue 和 leave 语句外的所有语言特性,以及所有(78 个)内置函数中的 62 个(形式化代码请参见 https://github.com/lixm/yul-smallstep).其中:break 和 continue 用于细粒度循环控制;而 leave 是近期引入 Yul 语言的新特性,用于退出当前函数.在智能合约中,这 3 个语句的使用并不多见,且在需要的场合,常常可由其他语法元素实现相同的功能,故目前形式化中对它们的省略并不会严重影响智能合约的表达.尽管如此,后文仍会对如何在形式化语义中支持这 3 个语句进行讨论.

本文第 1 节介绍国内外关于智能合约形式化验证、程序语言形式化的相关工作,并与本工作进行比较.第 2

节简单介绍证明辅助工具 Isabelle/HOL.第 3 节介绍 Yul 语言语法的形式化.第 4 节介绍 Yul 语言类型系统的形式化.第 5 节介绍 Yul 语言小步语义的形式化以及对语义正确性的测试.第 6 节利用一个代币合约(token contract)演示 Yul 语言合约的类型检查以及依据形式化语义的执行.第 7 节总结本工作并展望未来工作.

1 相关工作

1.1 智能合约语言形式化及智能合约验证

本文的研究领域为智能合约语言的形式化(机械化)语义,以下讨论该方面的现有工作.此外,由于智能合约验证工作是语义形式化工作的自然延伸,且部分现有的语义形式化工作同时处理一些验证问题,故本节亦讨论一些重要的智能合约验证工作.

在高级语言方面,Bhargavan 等人给出 Solidity 语言部分核心特性的形式化,并提出一个基于 F*验证以太坊智能合约的框架[10].Zakrzewski 在 Coq 中给出 Solidity 一个子集的大步语义[11].Yang 等人在 Coq 中对 Solidity 语言的核心子集 Lolisa 进行形式化[12],并开发了关于智能合约可靠性、安全性的验证系统[13].Jiao 等人在 K framework 中提出了智能合约高级语言的一般形式语义框架[14]以及 Solidity 语言的较为完整的形式化[15].Ahrendt 等人提出在 KeY 工具中对智能合约进行演绎验证的技术[16].在中间语言方面,Sergey 等人在 Coq 中定义了一种基于通信自动机的智能合约中间语言 Scilla[17].Bernardo 等人为 Tezos 平台提出了一种智能合约中间语言 Albert,且在 Coq 中描述了 Albert 的编译器[18].Li 等人讨论了在中间语言层进行智能合约定理证明的优势,并基于中间语言 Yul 的大步语义(首先在文献[19]中给出),在 Isabelle/HOL 中,对一数字货币合约进行了正确性证明[20].在低级语言层面,Hirai 在 Lem 中对以太坊虚拟机(EVM)的指令集进行形式化建模,并利用该模型,在 Isabelle/HOL 中验证了以太坊智能合约的一些不变式和关于账户余额增减的正确性[21].Amani 等人基于该工作定义了 EVM 字节码的一个可靠程序逻辑[22].Grishchenko 等人给出了 EVM 字节码的第一个完整的小步语义,并给出了 Ethereum 智能合约的 3 种安全属性[23].Hildenbrandt 等人在 K framework 中创建了可执行的 EVM 字节码语义[24].Kasampalis 等人提出了一种以虚拟寄存器为基础的智能合约底层语言 IELE,并在 K framework 中对语义进行了形式化[25].

以上工作侧重支撑演绎推理的智能合约语言形式化.此外,Luu 等人对 EVM 的 CALL,RETURN 等指令给出了纸笔定义,并构建了基于符号执行的漏洞检测工具 Oyente[26].Permenev 等人基于软件模型检测相关技术,构建了自动验证以太坊智能合约功能属性的验证器 VerX[27].Li 等人利用 SAPIC 演算对 BNB 智能合约进行建模,并在 Tamarin 自动证明器中,通过把模型转换成重写规则来验证该合约的安全性[28].Nehai 等人在模型检测工具 NuSMV 中构建以太坊智能合约的模型,验证智能合约是否符合时序逻辑规范[29].Kalra 等人构造了借助 LLVM 中间表示验证智能合约正确性、公平性的工具 ZEUS,并分析了大量智能合约代码[30].

由上述相关研究可知:在以太坊智能合约的形式化建模方面,现有工作多为以太坊字节码语言和高级语言的形式化.但 EVM 字节码作为一种具有简单指令编码的机器语言,不具有可读性,对以字节码形式存在的智能合约难于描述所需的证明目标和证明所需的支撑性断言.而高级语言特性多且复杂,变化更新快,很难形式化全部特性来实现完整规范.已有的面向中间语言的工作[17,18]所涉及的中间语言往往并非以太坊智能合约开发所用工具链支持的中间语言.相比之下,本工作形式化的中间语言 Yul 是以太坊项目自身的中间语言,可作为 Solidity 等高级语言的翻译目标,亦可编译为 EVM1.0,EVM1.5,eWASM 等低级语言.在文献[19]中亦包含 Yul 语言的部分形式化,但其语义为大步语义,且不包括 gas 的建模.相较于该工作,本工作定义 Yul 语言的类型系统以及包含 gas 模型的小步语义.小步语义反映智能合约执行的中间过程,能够很好地支撑智能合约安全属性(security property)的验证.

1.2 其他程序语言的形式化

已有工作对各种通用编程语言进行了不同程度的形式化,以下讨论其中的主要工作.关于 C 语言,Leinenbach 等人在 Isabelle/HOL 中形式化了 C 语言子集 C0 的小步语义并实现了其编译器规范的纸笔证明[31].

Blazy 和 Leroy 在 Coq 中给出了 C 的子集 Clight 的大步语义,它支撑了 CompCert 项目的 C 语言编译器验证任务[32].Ellison 等人在 K framework 中形式化 C 语言的可执行语义,并利用 GCC 测试集进行全面测试[33].关于 C++语言,Wasserrab 等人在 Isabelle/HOL 中对 C++中的多重继承提供形式化语义和类型安全性的形式化证明[34].Norrish 在 HOL 中对 C++的多种动态语义进行了形式化建模[35].关于 Java 语言,Drossopoulou 和 Eisenbach 对 Java 子集的操作语义和类型系统进行形式化描述,并给出了类型系统可靠性的证明[36].Bogdanas 等人在 K framework 中首次完整地实现 Java 1.4 的形式化可执行语义,并开发了一个涵盖所有 Java 构造的综合测试集[37].

与本工作相比,上述工作形式化不同的程序语言,在基本思想方法上有许多共通之处,而在技术路线上亦有差异.如文献[32]中给出的 Clight 语义采用 Coq 的归纳类型定义,在形式化证明中易于使用,但不可执行.在文献[31]中给出的 C0 语义使用浅层嵌入(shallow embedding)将基本操作直接表示为 Isabelle/HOL 中的函数,而本工作的 Yul 语言形式化采用完全的深层嵌入(deep embedding),具体定义基本操作的语法和语义.在 K framework 中所定义语义均为可执行语义,但基于该工具的程序验证结果尚未达到证明辅助工具所允许的可信度,尽管由 K framework 向 Coq 等证明辅助工具生成验证结果证书(certificate)的工作正在开展.

2 Isabelle/HOL 简介

本文的形式化工作在高阶逻辑证明辅助工具 Isabelle/HOL 中完成,它支持以函数式编程方式对系统进行形式规约,以及运用证明策略(tactics)或结构化证明语言 Isar 对系统性质进行证明[38].以下通过简单例子说明 Isabelle/HOL 中进行类型定义、函数定义、函数求值的方法.

在 Isabelle/HOL 中,用关键字 datatype 进行数据类型定义.下面例子中定义了类型 *T*,具有该类型的元素须由构造子 *Ty*1,*Ty*2,*Ty*3,*Ty*4 之一进行构造.若使用 *Ty*4,则需要提供整型参量,而其他构造子无需参量.

datatype *T*=*Ty*1|*Ty*2|*Ty*3|*Ty*4 *int*

Isabelle/HOL 中定义了内建列表类型*'a* list,这里,*'a* 是类型变量,即列表为多型(polymorphic)类型,如 int list 表示整型元素的列表,*T list* 表示 *T* 类型元素的列表,等等.函数定义往往使用关键字 fun 进行.以下定义函数 *ins_lst*,用于向 *T* 类型列表中无重复地插入元素.

```
fun ins_lst::"T list⇒T⇒T list" where
  "ins_lst ts t=(
     if t∈(set ts) then ts else ts@[t]
  )"
```

本定义中,::后为函数 *ins_lst* 的类型.Isabelle/HOL 中,*A*⇒*B* 表示由类型 *A* 到类型 *B* 的函数类型,其中,⇒右结合.因此,函数 *ins_lst* 作用于给定的 *T* 类型列表和给定的 *T* 类型元素,返回新的 *T* 类型列表.关键字 where 后为该函数的具体定义:当元素为类型 *T* 的列表 *ts* 中不存在试图插入的元素 *t* 时,则在该列表的尾部添加此元素;否则,该列表保持不变.该定义中,*set* 是 Isabelle/HOL 的库函数,*set ts* 获取表 *ts* 中元素所组成的集合;@表示同类型列表之间的连接.此外,后文亦用到操作符#,用于连接单个元素与列表.

关键字 value 支持对表达式进行求值.下面给出对函数 *ins_lst* 进行求值的两个示例.本工作中,对语义的测试就是借助 value 关键字进行的.

value "*ins_lst*[ ] *Ty*1"	结果:"[*Ty*1]"::"*T list*"
value "*ins_lst*[*Ty*1,*Ty*3] *Ty*1"	结果:"[*Ty*1, *Ty*3]"::"*T list*"

本文的形式化工作中大量使用基于列表的映射类型,该类型为自定义类型.具体而言,*'a list_map* 表示由整型元素到可选的*'a* 类型元素(*Some 'a* 或 *None*)的映射.它由整型和*'a* 类型元素所组成二元组的列表实现(事实上,*'a list_map* 定义为(*int*×*'a*) *list* 的同义词).对列表中的二元组(*i*,*a*),若它后面不再有形为(*i*,*a'*)的二元组,则视为 *i* 映射到 *Some a*;若对某个整型元素 *i*,列表中不存在形为(*i*,*a'*)的二元组,则视为 *i* 映射到 *None*.函数 *lm_dom* 给出某个*'a list_map* 类型映射的定义域,函数 *lm_get* 给出某个整型元素被映射到的值.

3 Yul 语言语法的形式化

Yul 语言程序包括代码块、语句、表达式、变量、字面值等构成单位.由于在 Yul 中语句可作为表达式使用,所以表达式和语句均用类型 *expr* 进行形式化.该处理简化形式化定义而不影响表达能力.

智能合约在 Yul 语言中表示为代码块.一个代码块由一系列表达式组成.代码块类型 *block* 由 *Blk* 构造,提供一表达式列表作为参量.其形式化定义如下面第一行所示.为突出重点,略去了关键字 *datatype* 以及类型定义中的部分内容(后文中往往作类似简化).

block = *Blk* "*expr list*"
expr = *EId id*0|*ELit literal*|*EFunCall id*0 "*expr list*"|
*EFunDef id*0 "(*id*0**type_name*) *list*" "(*id*0**type_name*) *list*"*block*|
EAssg "*id*0 *list*" *expr*|*EIf expr block*|*EFor block expr block block*|...
—‹*Intermediate expressions*›
EStop|*ERetId* "(*id*0**type_name*)"|*EImLit literal*|
*EImFunCall id*0 "*expr list*"|*ECond expr block*|*EGasPop*|…

表达式类型 *expr* 有多种构造方式,其中,变量类型 *EId id*0 表示标识符为(*v*::*id*0)的变量.这里,*id*0 为 *int* 的别名,即不同的变量标识符基于不同整数进行构造.字面值类型 *ELit literal* 表示数值或布尔值常量.其中,*literal* 类型包含字面值内容(*lit_content* 类型)和类型名称(*type_name*)两部分信息.例如,(*TL*:*L* Bool)表示其字面值内容为 *TL*(真),类型名为 Bool.函数调用类型 *EFunCall id*0 "*expr list*"表示对标识符为(*f*::*id*0)的函数以某个参数列表(*es*::"*expr list*")进行调用.函数调用分为内置函数调用和自定义函数调用,根据标识符 *f* 进行区分.函数定义类型 *EFunDef id*0 "(*id*0**type_name*) *list*" "(*id*0**type_name*) *list*" *block* 中的 *id*0 为所定义函数标识符的类型,两个"(*id*0**type_name*) *list*"分别为形式参数列表类型和返回值列表的类型,而 *block* 为函数体的类型(函数体为一代码块).赋值类型 *EAssg* "*id*0 *list*" *expr* 表示将某表达式(*e*::*expr*)的值赋给某个变量列表(*xs*::"*id*0 *list*")中变量的赋值语句.条件表达式类型 *EIf expr block* 表示在某条件表达式(*e*::*expr*)为真时执行某代码块(*blk*::*block*)的条件语句.循环表达式类型 *EFor block expr block block* 表示具有初始化代码块、条件表达式及后处理代码块的循环语句(类似 C 语言中的 for 循环).此外,关于变量声明、赋值、分支(switch)等 Yul 语言语法特性的形式化不再赘述.

除 Yul 语言本身的表达式外,本工作定义一系列中间表达式,以支撑语义的形式化.其中,*EStop* 表示执行结束;调用返回表达式 *ERetId* "(*id*0**type_name*)"表示函数调用的返回值需要写入的变量(*v*::*id*0)及其类型名(*t*::*type_name*);其他中间表达式 *EImLit literal*,*EImFunCall id*0 "*expr list*",*ECond expr block*,*EGasPop* 等的引入,是为了保证动态语义中能够正确计算 *gas* 消耗量.这些中间表达式在 Yul 语言的语法中并不存在,合约编写者无法直接使用.

4 Yul 语言类型系统的定义和形式化

在 Yul 语言官方文档中,给出了一系列数据类型、函数类型以及变量、函数的直观作用域规则.本工作结合该文档和实际编译 Yul 程序的实验结果,给出 Yul 语言类型系统(或称静态语义)的完整定义,及其在 Isabelle/HOL 中的形式化.

4.1 类型检查函数

Yul 语言类型系统的形式化是通过定义类型检查函数进行的.函数 *type_e* 对表达式进行类型检查,该函数在给定变量类型环境(*vte*::*type_env*)、外层声明变量集合(*xs*::*id*0 *set*)和函数类型环境(*fte*::*ftype_env*)下,判断给定表达式(*e*::*expr*)是否为良类型(well-typed),并给出表达式的具体类型以及更新的变量类型环境.函数 *type_es* 对表达式列表进行类型检查,该函数在给定变量类型环境(*vte*::*type_env*),外层声明变量集合(*xs*::*id*0 *set*)和函数类型环境(*fte*::*ftype_env*)下,判断给定表达式列表(*es*::*expr list*)是否为良类型.函数 *type_b* 在给定变量类型环境(*vte*::*type_env*)、外层声明变量集合(*xs*::*id*0 *set*)和函数类型环境(*fte*::*ftype_env*)下判断给定代码块(*blk*::*block*)是否为良

类型.这 3 个类型检查函数在 Isabelle/HOL 中的类型如表 1 上半部分所示,3 个函数的具体定义相互递归.

表 1 下半部分给出了变量类型环境、函数类型环境和表达式类型检查结果在 Isabelle/HOL 中的形式化类型.变量类型环境(*vte*::*type_env*)将变量标识符映射为变量类型 *vtype*,包括布尔型、不同位数的无符号整型、有符号整型等.函数类型环境(*fte*::*ftype_env*)将函数标识符映射为函数类型 *ftype*.其中,函数类型包含函数输入参数和输出参数类型的列表.

Table 1 Type checking functions and structures

表 1 类型检查函数和函数中包含的结构

type_es	::	"*type_env*⇒*id*0 *set*⇒*ftype_env*⇒*expr list*⇒(*bool*×*type_env*)"	
type_b	::	"*type_env*⇒*id*0 *set*⇒*ftype_env*⇒*block*⇒*bool*"	
type_e	::	"*type_env*⇒*id*0 *set*⇒*ftype_env*⇒*expr*⇒*expr_type_res*"	
type_env	=	"vtype *list_map*"	(变量类型环境)
ftype_env	=	"ftype *list_map*"	(函数类型环境)
expr_type_res	=	*ETypable etype type_env*|*EUntypable*	(表达式类型的结果)
vtype	=	*VType type_name*	(变量类型)
ftype	=	*FType* "*type_name list*" "*type_name list*"	(函数类型)
etype	=	*DataType* "*type_name list*"|*StmtType*	(表达式类型)

表达式结果的类型 *expr_type_res* 首先包含表达式是否为良类型的信息:若表达式为良类型,则进一步给出其类型(*t*::*etype*)和更新的变量类型环境(*vte′*::*type_env*).表达式类型 *etype* 可以为数据类型(*DataType* "*type_name list*")或语句类型(*StmtType*).更新的变量类型环境中包括原有变量到其类型的映射以及在所检查表达式中新声明变量到其类型的映射.

最后,Yul 语言是分程序语言,允许函数定义的嵌套(在该方面类似 Pascal 语言).对于某个函数定义,其外层所声明变量在函数体内既不能重新声明,亦不能使用.这与普通代码块嵌套中变量作用域规则不同.因此,为各个类型检查函数提供外层所声明变量的集合,以帮助进行内层函数定义的类型检查.

4.2 类型检查规则

Isabelle/HOL 中的函数定义可包括多个子句,每个子句对应于所提供参数值的一种情形.类型检查函数 *type_e* 定义中的一种情形可视为一条类型规则.以下对函数调用表达式的类型规则(如下所示)进行说明.

```
1.  type_e vte xs fte (EFunCall f es)=(
2.  case (lm_get(fte@builtin_fte) f) of
3.    Some (FType ptl rtl)⇒(
4.      if |ptl|=|es| then (
5.        let es_no_elist=(foldl(λacc e.(acc∧(not_elist e))) True es) in
6.        let es_pts=zip es ptl in
7.        let es_res=(foldl(λacc(e,t).
8.          (if acc then type_e vte xs fte e=ETypable(DataType[t]) vte else False)) True es_pts)
9.        in (if es_no_elist∧es_res then
10.         (if |rtl|=0 then ETypable StmtType vte else ETypable(DataType rtl) vte)
11.       else EUntypable))
12.     else ...)
13. |None⇒EUntypable)
```

在给定变量类型环境 *vte*、外层声明变量集合 *xs* 以及函数环境 *fte* 下,对函数调用 *EFunCall f es* 进行类型检查.首先判断函数类型环境 *fte* 或内置函数类型环境 *builtin_fte* 是否将被调函数的标识符 *f* 映射到某个函数类型 *FType ptl rtl*:若该条件满足,进而检查实参列表 *es* 和形参列表 *ptl* 的长度是否一致(第 4 行,其中,|*ptl*|是本文中表示列表 *ptl* 长度的直观记号).若此二列表长度相同,则判断实参列表 *es* 中每个表达式 *e* 是否都为良类型,且具有形参列表中所规定的数据类型(第 6 行~第 9 行).本步的实现方法为:先生成列表 *es* 与 *ptl* 中对位元素所构成

二元组的列表 *es_pts*,再用库函数 *foldl* 遍历该列表,检查每个二元组(*e*,*t*)中实参表达式 *e* 的类型检查结果是否与类型名称 *t* 相对应.若本步检查通过,则函数调用表达式为良类型.此时,若函数无输出参数(|*rtl*|=0,亦即无返回值),则可确定函数调用的类型为语句类型(*ETypable StmtType vte*);否则,函数调用的类型为数据类型(*ETypable* (*DataType rtl*) *vte*).

以上定义中,第 5 行所涉及的技术细节无关对该类型规则的直观理解,故关于其意义不再赘述.

代码块的类型规则借助表达式列表的类型检查函数 *type_es* 实现如下:

type_b vte xs fte (*Blk es*)=(**case** (*type_es vte xs fte es*) of (*b*,*vte′*)⇒*b*).

其中,*type_es vte xs fte es* 在代码块类型检查所用的变量类型环境 *vte*、外层声明变量集合 *xs* 和函数类型环境 *fte* 下,给出类型检查结果 *b*(True 或 False)以及经过 *es* 中所有表达式更新的变量类型环境 *vte′*,而 *b* 被直接用作代码块的类型检查结果.类型检查函数 *type_es* 具有如下定义.

1. *type_es vte xs fte es*=
2. *foldl*(*λacc e*.(**if** (*fst acc*) **then**
3. (**case** *type_e*(*snd acc*) *xs fte e* **of** (*ETypable StmtType vte′*)⇒(True,*vte′*)|_⇒(False,(*snd acc*)))
4. **else** *acc*))
5. (True,*vte*) *es*

本定义利用库函数 *foldl* 对表达式列表 *es* 中的各个表达式逐一进行类型检查,要求各个表达式均为语句类型(*StmtType*).对 *es* 中的每个表达式 *e*,若其前面的表达式均通过检查,则 *e* 的类型检查所使用的变量类型环境由 *e* 前面所有表达式的类型检查更新得到.

5 Yul 语言小步语义的定义和形式化

5.1 语义函数

Yul 语言小步语义的形式化通过定义语义函数进行.语义函数 *step* 和 *step_ctx* 在 Isabelle 中类型见表 2.函数 *step* 给出在当前交易环境(*tre*::*trans_env*)下,当前栈(*gstk*::*gstack*)执行一步后的结果(*gstk′*::*gstack*).函数 *step_ctx* 表示表达式(*e*::*expr*)在上下文(*ec*::*ectx*)中的单步执行.两个语义函数的定义相互递归.

Table 2 Semantic functions and structures in Yul

表 2 Yul 的语义函数和函数中包含的结构

step	::	"*trans_env*⇒*gstack*⇒*gstack*"	
step_ctx	::	"*trans_env*⇒*expr*⇒*ectx*⇒*gstack*⇒*gstack*"	
trans_env	=	(|*oaddr*::*address* *gprice*::"256 word" *bheader*::*blk_header*|)	(交易环境)
gstack	=	"*gstack_frame list*"	(全局栈)

表 2 中,交易环境类型 *trans_env* 记录交易过程中初始交易账户地址 *oaddr*,每单位 *gas* 的价格 *gprice* 和区块头 *bheader* 的信息.这些信息在同一交易的执行过程中保持不变.

全局栈类型 *gstack* 用来跟踪合约所作外部调用,其每一个栈帧通常包含一个被调合约执行时的控制状态(仍需执行的程序代码)、变量状态、函数标识符和函数定义的绑定状态、链上所有账户的状态(包括余额、存储等)、被调合约和调用者的地址、交易转移的金额、输入数据、*gas* 余量等信息.其中,被调合约的控制状态、变量状态、函数标识符和函数定义的绑定状态这 3 项信息对被调合约所作的每一个内部调用以及所进入的每一个局部代码块分别保存,形成一个局部栈(类型为 *lstack*).全局栈帧也可反映合约发生异常或终止执行的状态,从而包含其他信息,详见下文.

全局栈形式化为全局栈帧列表,表头对应栈顶.全局栈帧由符号 *gfrm* 表示,有 3 种可能的形态:表示正常执行状态的全局栈帧$\langle lstk,gs,ck\rangle_N$、表示异常状态的全局栈帧$\langle ck\rangle_E$和表示合约执行终止的全局栈帧$\langle gs,ret_data,ck\rangle_H$.表示正常执行状态的全局栈帧包括局部栈(*lstk*::*lstack*)、全局状态(*gs*::*gstate*)和外部调用类型(*ck*::*call_kind*);表

示异常状态的全局栈帧只包括外部调用类型(*ck*::*call_kind*);表示合约执行终止状态的全局栈帧包括全局状态(*gs*::*gstate*)、返回数据(*ret_data*::"(8 word) *list*")和外部调用类型(*ck*::*call_kind*).以下依次介绍全局栈帧的 3 个组成部分,即局部栈、全局状态和外部调用类型.

1) 局部栈(类型为 *lstack*)用来跟踪同一合约内部因函数调用或执行局部代码块所产生的作用域变化.每一个局部栈帧(类型为 *lstack_frame*)记录当前作用域相关信息.局部栈帧由符号 *lfrm* 表示,有两种可能形态:包含表达式形态的局部栈帧$(e,ls,fs)_E$ 和包含代码块形态的局部栈帧$(blk,ls,fs,cf)_B$.包含表达式的局部栈帧记录的信息有表达式(*e*::*expr*)、变量状态(*ls*)和函数状态(*fs*);包含代码块的局部栈帧记录的信息有代码块(*blk*::*block*)、变量状态(*ls*)、函数状态(*fs*)和判断当前是否为自定义函数调用的标志(*cf*::*cflag*).其中,变量状态 *ls* 记录每个变量的当前值,形式化为变量标识符和值所组成二元组的列表,其类型为"*literal list_map*".函数状态 *fs* 将当前作用域内可调用的函数标识符映射到其函数定义(表示为 *fvalue* 类型的值),将其形式化为函数标识符和值(即函数的语义表示)所组成二元组的列表,类型为"*fvalue list_map*".由于可调用的内置函数不随作用域变化,故函数状态 *fs* 中不包括内置函数信息.标志(*cf*::*cflag*)有两种可能的取值:若本局部栈帧记录由函数调用所产生的局部作用域相关信息,则 *cf* 为被调函数的 *fvalue* 值;若本局部栈帧记录由执行局部代码块所形成的局部作用域相关信息,则 *cf* 为空;
2) 全局状态(类型为 *gstate*)记录属于整个合约调用、而非合约的某个内部调用的状态信息.具体包括当前地址 *addr*、发送方地址 *saddr*、交易时的转移金额 *money*、交易的输入数据 *input*、本地内存 *mem*、内存中的活跃字数 *naws*、当前账户中可用的 *gas* 数量 *gas*、记录当前字栈的大小 *ct*、所有账户的状态 *accs*(由账户地址到账户状态的映射,反映每个账户的余额、代码、存储)、退还的余额 *refund*、日志列表 *logs* 和自杀集 *ss*;
3) 外部调用类型(*ck*::*call_kind*)给出了形成全局栈帧的外部调用的类型,包括 *CKCall*,*CKDelCall*,*CKCallCode* 和 *CKDummy*,分别对应于以太坊指令 *CALL*,*DELEGATECALL*,*CALLCODE* 所引发调用以及当前全局栈帧为栈底元素的情况.

5.2 语义规则

本节围绕赋值、条件语句、循环语句、代码块以及函数调用和返回说明 Yul 语言语义规则的要点.

赋值的语义规则如下.

```
1 step tre [⟨((xs::=e,ls,fs)E#lstk′),gs,ck⟩N]=(
2   case e of
3     EImLit lit⇒(
4       let new_lstk=upd_var_in_lstack((EStop,ls,fs)E#lstk′) (xs!0) lit in
5         if (valid (gas_of gs) (5*(size xs)) (ct_of gs))
6         then (let gs′=gs (| gas:=(gas gs)−(word_of_int(5*(int(size xs)))) |) in [⟨new_lstk,gs′,ck⟩N])
7         else [⟨ck⟩E])
8   |EList es⇒…
9   |_⇒step_ctx tre e(ECAssg xs Hole) [⟨((EStop,ls,fs)E#lstk′),gs,ck⟩N])
```

其中,表达式 *e* 形如 *EImLit lit* 对应于赋值右侧表达式已经求值完毕、且结果为单个字面值的情形(第 3 行开始).此时使用辅助函数 *upd_var_in_lstack* 将右侧表达式的求值结果 *lit* 写入局部栈最靠近栈顶且定义域中含有变量 *xs*!0(*xs* 中的首个变量)的变量状态中,形成更新的局部栈 *new_lstk*.若当前 *gas* 剩余能够支撑赋值操作的 *gas* 消耗,则从前者中扣除后者,形成新的全局状态 *gs*′;反之,抛出异常.其次,表达式 *e* 形如 *EList es* 的情形对应于将值列表赋给变量列表的情形(第 8 行).值列表中实际包含某函数返回的一系列字面值.此时,对局部栈以及 *gas* 剩余的更新操作类似于第 1 种情况.若表达式 *e* 不具备上述两种形式中的任何一种,则 *e* 为尚未求值完全的表达式(第 9 行).此时,借助 *step_ctx* 函数首先在赋值上下文 *ECAssg xs Hole* 中对 *e* 进行小步执行.

条件语句和循环语句的语义均借助中间表达式 *ECond e blk* 进行定义.类似分支语句 *EIf e blk*,中间表达式

ECond e blk 实现“若表达式 *e* 求值为真则执行代码块 *blk*”的分支控制逻辑,但忽略部分 *gas* 消耗.中间表达式 *ECond* e *blk* 的语义规则如下.

1. *step tre* [⟨(((*ECond e blk*,*ls*,*fs*)$_E$#*lstk*′),*gs*,*ck*⟩$_N$]=(
2. **case** *e* **of**
3. (*EImLit*(*TL*:*L* Bool))⇒
4. **if** (*valid* (*gas gs*) (3+3+10) (*ct gs*))
5. **then** (**let** *gs*′=*gs* (| *gas*:=(*gas gs*)−(*word_of_int* 16) |) **in**
6. [⟨(((*blk*,[ ],*get_func_values blk*,*None*)$_B$#(*EStop*,*ls*,*fs*)$_E$#*lstk*′),*gs*′,*ck*⟩$_N$])
7. **else** [⟨*ck*⟩$_E$]
8. |(*EImLit*(*FL*:*L* Bool))⇒
9. **if** (*valid* (*gas gs*) (3+3+10) (*ct gs*))
10. **then** (**let** *gs*′=*gs* (| *gas*:=(*gas gs*)−(*word_of_int* 16) |) **in** [⟨(((*EGasJumpDest*,*ls*,*fs*)$_E$#*lstk*′),*gs*′,*ck*⟩$_N$])
11. **else** [⟨*ck*⟩$_E$]
12. |_⇒*step_ctx tre e* (*ECCond Hole blk*) [⟨(((*EStop*,*ls*,*fs*)$_E$#*lstk*′) *gs ck*⟩$_N$]
13.)

故若表达式 *e* 已求值为字面值(第 3 行、第 8 行),则检查全局状态中 *gas* 剩余是否能够支持判断条件表达式真值以及执行条件转移所消耗的 *gas* 量.若该检查通过,则从全局状态中扣除该消耗量,并在所形成的全局状态下视 *e* 为布尔值“真”或“假”执行不同操作:若 *e* 为布尔值“真”,则在新的局部栈帧(*blk*,[ ],*get_func_values blk*,*None*)$_B$ 中执行代码块 *blk*;若 *e* 为布尔值假,则执行表达式 *EGasJumpDest*,以产生分支逻辑结尾标签(字节码 *JUMPDEST*)所对应的 *gas* 消耗.若表达式并非字面值(第 12 行),则借助 *step_ctx* 函数首先在条件分支上下文 *ECCond Hole blk* 中对 *e* 进行小步执行.

条件语句的语义规则将 *EIf e blk* 的执行转化为 *ECond e* (*blk*++$_{es}$[*EGasJumpDest*])的执行,其中,++$_{es}$ 在代码块 *blk* 结束位置添加列表[*EGasJumpDest*]中的表达式.结合 *ECond* 表达式的语义,这保证无论条件表达式 *e* 是否求值为真,条件分支结束后均发生字节码 *JUMPDEST* 所引起的 *gas* 消耗.而循环语句的语义规则将 *EFor blk*0 *e blk*1 *blk* 的执行转化为下列表达式在新局部栈帧中的执行:

*blk*0++$_{blk}$[*EGasJumpDest*,*ECond e* (*blk*++*blk*1++$_{es}$[*EGasPush*,*EGasJump*,*EFor*(*Blk*[ ]) *e blk*1 *blk*])].

故循环初始化代码块 *blk*0 首先执行,其中声明的变量在本循环内(对应于新建的局部栈帧)有效.而后产生字节码标签 *JUMPDEST*(用于继续循环所需的跳转)所引起的 *gas* 消耗.此后,若表达式 *e* 求值为真,则继续执行循环体 *blk* 以及进行后处理(post-processing)的代码块 *blk*1,产生跳转回循环表达式 *e* 开始处所引起的 *gas* 消耗,并继续执行初始化代码块为空的循环 *EFor*(*Blk*[ ]) *e blk*1 *blk*.

代码块的语义规则给出全局栈 *gstk*0=⟨(*Blk es*,*ls*,*fs*,*fg*)$_B$#*lstk*′,*gs*,*ck*⟩$_N$#*gstk*′的单步执行结果.若代码块不为空(即 *es* 为非空表达式列表),则根据含有单个栈帧的全局栈[⟨(*e*,*ls*,*fs*)$_E$#*lstk*′,*gs*,*ck*⟩$_N$]的单步执行结果更新全局栈 *gstk*0,其中,*e* 为 *es* 中的第 1 个表达式.若代码块为空(即 *es* 为空的表达式列表),而局部栈 *lstk*′不为空,则弹出局部栈帧(*Blk es*,*ls*,*fs*,*fg*)$_B$,表示代码块及其作用域的结束.若代码块为空,且局部栈 *lstk*′亦为空,则将全局栈帧改为⟨*gs*1,[ ],*ck*⟩$_H$,表示对当前合约调用的结束,其中,*gs*1 是由全局状态 *gs* 消耗一定量 *gas* 得到的全局状态.代码块的完整语义规则较为复杂,故在此不作展示.

以下对函数调用中栈的变化进行说明.图 1 所示为形如⟨*lfrm*#*lstk*,…⟩$_N$#*gstk* 的全局栈完成一次成功的自定义函数调用,到被调函数执行结束并返回,全局栈和局部栈的主要变化情况.单步执行由实线箭头表示,多步执行由虚线箭头表示.调用栈的变化部分粗体显示.

考虑局部栈帧 *lfrm* 中的调用 *EFunCall f es* 被执行的情形.首先,在第一步执行 *A*→*B* 中,实际调用并不发生,而是将 *EFunCall f es* 转化为中间表达式 *EImFunCall f es*,并更新全局状态,扣除实际调用发生前,返回地址入栈引起的 *gas* 消耗量(仅在调用自定义函数时为非零值).此时,含有 *EImFunCall f es* 的局部栈帧为 $lfrm'_0$.而后,视该

调用为对同一合约内部函数的调用或对其他合约的外部调用,调用栈的变化情况分别如 $B\to C$ 和 $B\to F$ 所示.若为内部调用($B\to C$),则形成新的局部栈帧 $lfrm''$,并在原局部栈帧的代码部分中用形如 *ERetId*(_,_)的返回值占位符替换表达式 *EImFunCall f es*,形成局部栈帧 $lfrm'$.被调函数可正常执行后返回,弹出局部栈帧 $lfrm''$,并使用返回值替换 *ERetId*(_,_),形成局部栈帧 $lfrm'_1$ ($C\to D$);被调函数亦可由于 *gas* 耗尽等原因抛出异常,从而销毁整个全局栈帧$\langle lfrm''\#lfrm'\#lstk,\ldots\rangle_N$,并代之以表示异常的全局栈帧$\langle\ldots\rangle_E$($C\to E$).直观上,后一种情况表示当前合约中所有的内部调用均异常终止.若表达式 *EImFunCall f es* 进行外部合约调用(即 *f* 为某个实现外部调用的内置函数的标识符),则形成新的全局栈帧 *gfrm*,并在原局部栈帧 $lfrm'_0$ 中用形如 *ERetId*(_,_)的返回值占位符替换表达式 *EImFunCall f es*,形成局部栈帧 $lfrm''_0$ ($B\to F$).被调合约可正常终止于形如$\langle\ldots\rangle_H$的全局栈帧(如图 1(G)所示),并向调用者返回 1;被调合约亦可异常终止于形如$\langle\ldots\rangle_E$的全局栈帧(如图 1(I)所示),并向调用者返回 0.

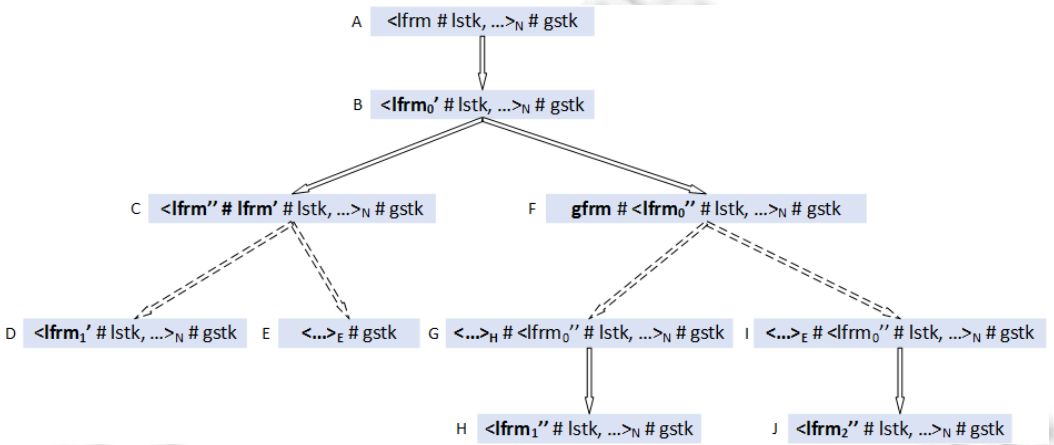

Fig.1 Changes of the semantic configuration with function calls and returns

图 1 函数调用和返回相关状态变化

以下说明函数调用和返回所涉及的关键语义规则.函数调用表达式 *EFunCall f es* 的第 1 步执行如下.

```
1.  step tre ([⟨(EFunCall f es,ls,fs)E#lstk',gs,ck⟩N])=(
2.    if (f∉lm_dom builtin_ctx)∧(valid (gas gs) 3 (ct gs))
3.    then (let gs'=gs (| gas:=(gas gs)−3 |)
4.      in [⟨(EImFunCall f es,ls,fs)E#lstk',gs',ck⟩N])
5.    else (if (f∈lm_dom builtin_ctx) then ([⟨(EImFunCall f es,ls,fs)E#lstk',gs,ck⟩N]) else [⟨ck⟩E])
6.  )
```

为了对 *gas* 消耗量进行描述,*EFunCall f es* 首先转化为中间表达式 *EImFunCall f es*.自定义函数调用和内置函数调用在 *gas* 消耗方面存在差异.当判断该语句是自定义函数调用时,该步执行需要消耗 3 个单位的 *gas* 值(第 3 行);当判断该语句为内置函数的调用时,则该步执行无需消耗 *gas*(第 5 行 **then** 中的部分).

中间表达式 *EImFunCall f es* 的语义规则如下所示.

```
1.  step tre [⟨(EImFunCall f es,ls,fs)E#lstk',gs,ck⟩N]=(
2.  let (idx,found)=lst_find_idx(rev es) not_lit in (
3.    if found then (
4.      step_ctx tre (es!(|es-idx-1|)) (ECFunCall f(take|es-idx-1| es) Hole (map peel(drop|es-idx|es)))
5.        [⟨(EStop,ls,fs)E#lstk',gs,ck⟩N]
6.  ) else (
7.    case lm_get((aggr_fs((EImFunCall f es,ls,fs)E#lstk'))@builtin_ctx) f of
8.      Some (FunctionV f ptl rtl blk)⇒(…
```

```
9.    let blk′=(n_expr_lst[EGasPush,EGasJump,EGasJumpDest] EGasPush|rtl|) ++blk blk ++es post_es in
10.     [⟨(((blk′,(zip(map fst ptl) (map peel es))@rzl,fs′,Some (FunctionV f ptl rtl blk))B
11.       #((if (|rtl|≠0) then (EList(map(λxt.(ERetId xt)) rtl)) else EStop),ls,fs)E#lstk′),gs,ck⟩N])
12.   |Some (CallBuiltinV callgb)⇒(
13.     if (callgb∈{Call,CallCode,DelegateCall,Return,Revert,Selfdestruct,Stop,Invalid})
14.     then ((step_callbuiltin⟨(EImFunCall f es,ls,fs)E#lstk′,gs,ck⟩N callgb lits))
15.     else [ ]
16.   …
17.   )
18. ))
```

表达式 *EImFunCall f es* 的执行首先考虑 *es* 中是否有未完全求值的参数表达式:若有,则在相应上下文中执行最右侧的未完全求值表达式(第 4 行);否则,按照所调用函数是否为自定义函数作不同描述.

若函数标识符在环境(*aggr_fs*((*EImFunCall f es*,*ls*,*fs*)$_E$ # *lstk′*))*@builtin_ctx* 中被映射为形如 *FunctionV f ptl rtl blk* 的值(第 8 行),说明所调用函数为自定义函数.此种情况下,向被调函数的函数体 *blk* 前后分别添加表示调用时进行跳转和参数入栈所消耗 *gas* 的表达式以及表示返回时调整栈布局(包括参数和局部变量出栈等操作)所消耗 *gas* 的表达式,形成代码块 *bl′*.在 *blk′*定义中,使用了代码块与表达式列表的连接符++$_{es}$ 以及表达式列表与代码块的连接符++$_{blk}$.代码块 *blk′*放在新的局部栈帧中,准备执行.而调用者栈帧中的表达式置为 **if** (|*rtl*|≠0) **then** (*EList*(*map*(*λxt*.(*ERetId xt*)) *rtl*)) **else** *EStop*,表示若返回值个数不为零,则设置相应个数的用于接收返回值的占位符;否则,函数调用在该栈帧再次成为栈顶时已结束(*EStop*).

若函数标识符在环境(*aggr_fs*((*EImFunCall f es*,*ls*,*fs*)$_E$ # *lstk′*))*@builtin_ctx* 中被映射为形如 *CallBuiltinV callgb*(第 12 行),*OpBuiltinV callgb*,*RBuiltinV callgb* 等值,则表示所调用函数为相应类型的内置函数.其中,形如 *CallBuiltinV callgb* 的值表示与外部调用和返回相关的内置函数,具体是哪个函数,由 *callgb* 表示.若 *callgb* 在 Yul 所支持的此类内置函数范围内,则借助辅助函数 *step_callbuiltin* 给出调用的执行结果(略去进一步技术细节);否则,给出表示出错的空全局栈.其余种类的内置函数调用作类似描述.

若外部调用发生后,被调合约正常终止,则全局栈栈顶为形如⟨*gs′*,*ret_data*,*ck′*⟩$_H$ 的栈帧(对应图 1(G)).在语义中,该外部调用返回后的全局栈由 *step tre* (⟨*gs′*,*ret_data*,*ck′*⟩$_H$#*gstk′*)给出(对某个 *gstk′*).而该表达式的定义根据外部调用的种类(对应于太坊中的 *CALL* 指令、*CALLCODE* 指令还是 *DELEGATECALL* 指令),借助不同辅助函数给出.当外部调用由 *CALL* 指令触发,即 *ck′*由 *CKCall* 构造时,*step tre* (⟨*gs′*,*ret_data*,*ck′*⟩$_H$#*gstk′*)定义为 *callret* (⟨*gs′*,*ret_data*,*ck′*⟩$_H$#*gstk′*),其中,辅助函数 *callret* 定义如下.

```
1.  callret (⟨gs′,ret_data,(CKCall g to val io is oo os)⟩H#⟨(Blk(e#es),ls,fs,cf)B#lstk′,gs,ck⟩N#gstk′)
2.    =(let … in
3.      let flag=if (accs gs to=None) then 0 else 1 in
4.      let c_call=Cgascap val flag g (gas gs) in
5.      let c=(Cbase val flag)+(Cmem(uint(naws gs)) (uint naws))+(sint c_call) in
6.        (⟨(Blk((expr_fill_retids e [(extcall_ret_id,((NL 1):L U256))])#es),ls,fs,cf)B#lstk′,
7.          (gs (| gas:=(gas gs′)+(gas gs)−(word_of_int c),… |)),ck⟩N#gstk′))
```

由于被调合约正常终止,返回值为 1.上述定义中,利用函数 *expr_fill_retids* 将该返回值写入此前发生调用时留在全局栈次栈顶中局部栈栈顶表达式中的辅助变量 *extcall_ret_id*(第 6 行).定义中,第 3 行~第 5 行计算调用的总 *gas* 成本 *c*.返回后,全局状态中的 *gas* 值更新为当前全局栈中的 *gas* 值与次栈顶中的 *gas* 值(即被调合约终止时的 *gas* 余量)的和,减去一次调用所消耗的成本 *c* 所得结果(第 7 行).

关于其他种类外部调用的正常返回、外部调用的异常返回以及所有其他语义规则,不再详细阐释.

5.3 测 试

本工作中,开发了由 120 个简单 Yul 语言程序组成的测试集(test suite),对形式化语义进行了测试.测试用例的分布覆盖了已形式化的 Yul 语言的全部语言特性(字面值、变量、变量声明和赋值、函数定义和调用、各种控制流结构、代码块等)以及内置函数.内置函数的测试分为 5 类,分别针对关于逻辑操作的内置函数、关于算术操作的内置函数、关于内存与存储的内置函数、关于执行与控制的内置函数和关于区块链状态查询与哈希值计算的内置函数.

采取白盒测试的思路,使测试用例覆盖语义函数 *step* 和 *step_ctx* 定义中的所有顶层情形,并尽可能覆盖每种情形中的子情形以及 *step*,*step_ctx* 中所调用辅助函数中的各种情形.例如,if 语句的测试覆盖 3 种情况:条件成立时、条件不成立时、条件表达式不是布尔字面值因而需要进一步求值时.再如函数调用的测试覆盖内置函数调用、自定义函数调用、外部函数调用、内部函数调用及调用函数后正常返回和异常返回.

对每个测试用例,在 Isabelle/HOL 中使用 *value* 命令,在某个交易环境下,从某个全局栈开始对语义进行执行.为观察多步执行的结果,基于语义函数 *step* 定义了多步执行函数 *multi_step*.一个在测试中对表达式进行多步执行的例子是:

value "*multi_step tre*0 *gstk_EIf* 4".

该命令在交易环境 *tre*0 下,从全局栈 *gstk_EIf* 开始进行 4 步执行,其中,全局栈 *gstk_EIf* 形如:

$$[\langle[(EIf(ELit(TL{:}L\ \text{Bool}))(Blk[\,]),ls0,[\,])_E],gs0,CKDummy\rangle_N].$$

最后,在 Isabelle/HOL 中定义辅助函数,对每个测试中 Yul 语言程序的执行结果进行检查,包括结果的全局栈帧形式、所计算数值、*gas* 消耗量等方面.本工作所定义的形式化语义通过了 120 个测试用例的测试.

5.4 对break,continue和leave的支持

本工作对 Yul 语言的形式化不包含用于细粒度循环控制的 break,continue 语句以及用于退出当前函数的 leave 语句.本节讨论如何在语义形式化中加入对这 3 个语句的支持.

首先,leave 语句的语义规则可将当前代码块置为空,从而引发当前局部栈帧的弹出,以反映从当前函数退出的动作;其次,为形式化 break 的语义,可在局部栈帧中记录当前最内层循环退出后需要执行的代码,并在 break 的语义规则中将当前控制状态改为该代码;最后,为形式化 continue 的语义,可在局部栈帧中记录当前最内层循环的后处理(post-processing)部分开始时仍需执行的代码,并在 continue 的语义规则中将当前控制状态改为该代码.在语义形式化中,支持 break,continue,leave 不会造成实质性的技术难度,但会使语义状态的构成及其在语义规则中的维护更加复杂.

6 代币合约案例

本节通过一个用于管理代币(token)的实用智能合约,演示 Yul 语言合约的类型检查和小步执行.在以太坊中,为了给代币智能合约提供一个特征与接口的标准而推出 ERC20 标准[39].本节案例中的合约遵循该标准,故可称为 ERC20 代币合约.

6.1 代币合约

用于演示的代币合约名为 MyToken,其主要功能包括代币余额查询、代币转账等.该合约总共有 19 个函数,其中有 6 个接口函数,以下是对接口函数的介绍,其中所涉及的用户账户由账户地址表示.

(1) 函数 *totalSupply*:返回当前的代币发行总量;
(2) 函数 *balanceOf*:返回给定用户账户 account 的代币余额;
(3) 函数 *allowance*:返回当前允许指定账户 by 从指定账户 from 取出代币的额度;
(4) 函数 *approve*:将允许指定账户 spender 从本账户(即调用该函数的账户)提取代币的总额度更新为指定值 amount;
(5) 函数 *transfer*:从本账户(即调用者账户)向指定账户 to 转移数量为 amount 的代币,并在本账户余额不

足时抛出异常;

(6) 函数 *transferFrom*:从指定账户 from 向指定账户 to 转移数量为 amount 的代币,若账户 from 不允许本账户提取额度 amount、from 余额不足或账户 to 接受数量为 amount 的代币会引起溢出,则抛出异常.

6.2 代币合约的形式化及类型检查

本节在 Isabelle/HOL 中对 MyToken 代币合约进行形式化建模,并利用已形式化的类型检查函数对其进行类型检查.代币合约的形式化代码 *mytoken_code* 如下所示,该代码由 1 段派遣程序(dispatcher)、6 个接口函数、13 个辅助函数的形式化构成:

definition *mytoken_code*::*block* **where**

"*mytoken_code=Blk*(*dispatch_logic_my_token*@[*total_supply_func*,...,(*函数定义*),...,*log_func*])"

派遣程序是由 if 表达式和 switch 表达式组成的表达式列表,如下所示,其主要功能是根据用户调用合约时所提供的输入数据将调用转到合约中具体的接口函数.

```
1.  definition dispatch_logic_my_token::"expr list" where
2.    "dispatch_logic_my_token=[
3.      EIf (EFunCall b_gt_id[EFunCall b_callvalue_id[ ],lit_zero]) (Blk[revert_zz_call]),
4.      ESwitch (EFunCall f_selector_id[ ])
5.        [
6.          (((NL 0x095ea7b3):L U256),
7.            Blk[(EFunCall f_approve_id
8.              [EFunCall f_decode_as_address_id[lit_zero],EFunCall f_decode_as_uint_id[lit_one]])]),
9.          …
10.         (((NL 0xa9059cbb):L U256),
11.           Blk[(EFunCall f_transfer_id
12.             [EFunCall f_decode_as_address_id[lit_zero],EFunCall f_decode_as_uint_id[lit_one]])]),
13.       ]
14.       (Some (Blk[revert_zz_call]))
15.   ]"
```

该部分(形式化)代码首先判断调用合约时是否存在以太坊原生数字货币的转账.由于代币合约管理代币而非以太坊原生数字货币,故不接受原生数字货币转账,当该转账数额大于 0 时,使合约异常终止,回滚状态(第 3 行).若无原生数字货币转移发生,则正式进入派遣逻辑,利用 *selector* 函数得到全局状态中交易输入数据 input 的前 4 个字节,并把该 4 个字节作为 switch 表达式中的变量表达式(第 4 行),利用变量表达式分别与 switch 中各个匹配项进行匹配:匹配成功,则执行该匹配项中的代码;否则,执行默认项中的代码.每个匹配项中的常量值(如第 6 行的 0x095ea7b3)为对应函数签名的哈希值,这是对 Solidity 语言实现中所用约定的反映.实际上,使用 Yul 语言直接编写合约,不一定遵循该规范进行.

在 Isabelle/HOL 中执行代码块类型检查函数 *type_b* 所得结果"True"表明,MyToken 合约为良类型,即该合约的派遣程序以及所有函数定义均满足类型规则.

value "type_b [] {} fte_blk_mytoken mytoken_code"	结果:("True":: "bool")

6.3 接口函数执行的模拟

本节演示 MyToken 合约在 Isabelle/HOL 中按照小步语义的执行,具体考虑用户输入引起合约中 *transfer_func* 函数执行的情况.

定义初始全局栈 *gstk_token_contract*,其中:唯一的全局栈帧含有全局状态 *gs_init* 和一局部栈;唯一的局部栈帧含有 MyToken 合约代码 *mytoken_code*.

definition *gstk_token_contract* **where**

"*gstk_token_contract*=⟨([(*mytoken_code*,*ls_init*,(*get_func_values mytoken_code*),*None*)$_B$]),*gs_init*, *CKDummy*⟩$_N$#[ ]"

全局状态 *gs_init* 中,交易发送方地址(*saddr gs_init*)为 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;交易输入数据(input *gs_init*)的前 4 字节分别为 0xa9,0x05,0x9c 和 0xbb,使得派遣程序能够将对合约的调用转向其中的 *transfer_func*.由第 5 字节到第 36 字节均为 0x0,表示 *transfer_func* 的第 1 个实参为 0x0,向该地址转账,而由第 36 字节到第 68 字节为 0x0 … 0x0 0x1 0x0 0x0 0x0 0x0(即只有第 64 字节为 0x1,其余字节为 0x0),表示 *transfer_func* 的第 2 个实参为 2^{32}=4294967296,亦即转移代币的数量.此外,全局状态 *gs_init* 中当前账户地址指向合约账户 *mytoken_account*.

该账户的存储中,对应于代币发送方账户地址 $addr_0$=0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c 的代币余额为 99 999 999 999,对应于代币接收方账户地址 0x0 的代币余额为 10 000 000 000.

定义两个辅助函数帮助观察合约执行后发生的代币余额变化,其中,函数 *get_val_from_storage* 实现从全局栈顶取出指定存储地址所存放的值的功能,函数 *get_balance_offset* 获取指定账户的代币余额在 *mytoken_account* 账户存储中的偏移量.初始时,代币发送和接收账户的代币余额如下所示.

value "*get_val_from_storage* (*multi_step tre*0_*ex*1 *gstk_token_contract* 0) (*get_balance_offset* $addr_0$)"	结果:("Some 99999999999":: "256 word option")
value "*get_val_from_storage* (*multi_step tre*0_*ex*1 *gstk_token_contract* 0) (*get_balance_offset* 0x0)"	结果:("Some 10000000000":: "256 word option")

函数 *transfer_func* 执行后,发送和接收账户的代币余额如下所示:

value "*get_val_from_storage* (*multi_step tre*0_*ex*1 *gstk_token_contract* 270) (*get_balance_offset* $addr_0$)"	结果:("Some 95705032703":: "256 word option")
value "*get_val_from_storage* (*multi_step tre*0_*ex*1 *gstk_token_contract* 270) (*get_balance_offset* 0x0)"	结果:("Some 14294967296":: "256 word option")

发送方代币余额为 99999999999–4294967296=95705032703,而接收方代币余额为 10000000000+4294967296=14294967296.这印证了 *transfer_func* 函数的代币转移功能.

7 结论与展望

为支撑面向中间语言层、基于定理证明的智能合约形式化验证,本工作在 Isabelle/HOL 中给出了以太坊中间语言 Yul 的形式化,包括该语言类型系统和小步语义的首个已知的形式化,从而形成了 Yul 语言的形式化规范.通过创建由 120 个 Yul 语言程序所组成的测试集对形式化语义进行了系统测试,印证了形式化语义对 Yul 语言文档的直观规范以及实际观察的合约执行行为的如实反映.在本文中,通过一个 ERC20 代币合约的案例演示了类型系统和语义在 Isabelle/HOL 中的执行.除去测试与案例,本工作包含约 2 800 行形式化代码.

本项目当前尚未完成的工作主要为 Yul 语言类型系统保型性(type preservation)的形式化证明,即,在 Isabelle/HOL 中证明表达式和代码块的良类型被小步执行所保持.目前已给出保型性所涉及的关于执行状态的不变式,并基本完成了保型性的纸笔验证.后续工作中,我们将首先在 Isabelle/HOL 中完成保型性证明,而后,基于 Yul 语言形式化规范开展中间语言层以太坊智能合约的形式化证明工作——重点为安全属性的验证.

References:

[1] Yaga D, Mell P, Roby N, Scarfone K. Blockchain technology overview. U.S., National Institute of Standards and Technology, 2018: 1−6. [doi: 10.6028/NIST.IR.8202]

[2] Wood G. Ethereum: A secure decentralized generalized transaction ledger. Ethereum Project Yellow Paper, 2014,151(2014):1−32.

[3] Li XQ, Jiang P, Chen T, Luo XP, Wen QY. A survey on the security of blockchain systems. Future Generation Computer Systems, 2020,107:841−853. [doi: 10.1016/j.future.2017.08.020]

[4] CNVD blockchain vulnerability sub-library. 2020. https://bc.cnvd.org.cn/statistics

[5] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):33−61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[6] Tolmach P, Li Y, Lin SW, Liu Y, Li ZX. A survey of smart contract formal specification and verification. arXiv:2008.02712, 2020.

[7] Yul-Solidity documentation (v0.6.1). https://solidity.readthedocs.io/en/v0.6.1/yul.html

[8] Bartoletti M, Galletta L, Murgia M. A minimal core calculus for solidity contracts. In: Pérez-Solà C, Navarro-Arribas G, Biryukov A, García-Alfaro J, eds. Proc. of the Cryptocurrencies and Blockchain Technology on Data Privacy Management. Springer-Verlag, 2019. 233–243. [doi: 10.1007/978-3-030-31500-9_15]

[9] Nielson HR, Nielson F. Semantics with applications: An appetizer. In: Proc. of the Undergraduate Topics in Computer Science. Springer-Verlag, 2007. 19–41. [doi: 10.1007/978-1-84628-692-6]

[10] Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N, Zanella-Béguelin S. Formal verification of smart contracts: Short paper. In: Murray TC, Stefan D, eds. Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security. 2016. 91–96. [doi: 10.1145/2993600.29 93611]

[11] Zakrzewski J. Towards verification of Ethereum smart contracts: A formalization of core of solidity. In: Piskac R, Rümmer P, eds. Proc. of the Working Conf. on Verified Software: Theories, Tools and Experiments. Springer-Verlag, 2018. 229–247. [doi: 10. 1007/978-3-030-0 3592-1_ 13]

[12] Yang Z, Lei H. Lolisa: Formal syntax and semantics for a subset of the solidity programming language in mathematical tool coq. In: Proc. of the Mathematical Problems in Engineering. 2020. [doi: 10.1155/2020/6191537]

[13] Yang Z, Lei H, Qian WZ. A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts. IEEE Acess, 2020,8:21411–21436. [doi: 10.1109/ACCESS.2020.2969437]

[14] Jiao J, Lin SW, Sun J. A generalized formal semantic framework for smart contracts. In: Wehrheim H, Cabot J, eds. Proc. of the Fundamental Approaches to Software Engineering. Springer-Verlag, 2020. 75–96. [doi: 10.1007/978-3-030-45234-6_4]

[15] Jiao J, Kan S, Lin SW, Sanan D, Liu Y, Sun J. Semantic understanding of smart contracts: Executable operational semantics of solidity. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. IEEE, 2020. 1695–1712. [doi: 10.1109/SP40000.2020. 00066]

[16] Ahrendt W, Pace GJ, Schneider G. Smart contracts: A killer application for deductive source code verification. In: Müller P, Schaefer I, eds. Proc. of the Principled Software Development. Springer-Verlag, 2018. 1–18. [doi: 10.1007/978-3-319-98047-8_1]

[17] Sergey I, Kumar A, Hobor A. Scilla: A smart contract intermediate-level language. arXiv:1801.00687, 2018.

[18] Bernardo B, Cauderlier R, Pesin B, Tesson J. Albert, an intermediate smart-contract language for the tezos blockchain. arXiv:2001. 02630, 2020.

[19] Eth-isabelle. 2018. https://github.com/pirapira/eth-isabelle

[20] Li XM, Shi ZP, Zhang QY, Wang GH, Guan Y, Han N. Towards verifying Ethereum smart contracts at intermediate language level. In: Ameur YA, Qin SC, eds. Proc. of the Int'l Conf. on Formal Engineering Methods. Springer-Verlag, 2019. 121–137. [doi: 10. 1007/978-3-030-32409-4_8]

[21] Hirai Y. Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner M, Rohloff K, Bonneau J, Miller A, Ryan PYA, Teague V, Bracciali A, Sala M, Pintore F, Jakobsson M, eds. Proc. of the Int'l Conf. on Financial Cryptography and Data Security. Springer-Verlag, 2017.520–535. [doi: 10.1007/978-3-319-70278-0_33]

[22] Amani S, Bégel M, Bortin M, Staples M. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Andronick J, Felty AP, eds. Proc. of the 7th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs. ACM, 2018. 66–77. [doi: 10.1145/3167084]

[23] Grishchenko I, Maffei M, Schneidewind C. A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer L, Küsters R, eds. Proc. of the 7th Int'l Conf. on Principles of Security and Trust. Springer-Verlag, 2018. 243–269. [doi: 10.1007/ 978-3-319-89722-6_10]

[24] Hildenbrandt E, Saxena M, Rodrigues N, Zhu XR, Daian P, Guth D, Moore B, Park D, Zhang Y, Ştefănescu A, Roşu G. KEVM: A complete formal semantics of the Ethereum virtual machine. In: Proc. of the 31st Computer Security Foundations Symp. IEEE Computer Society, 2018. 204–217. [doi: 10.1109/CSF.2018.00022]

[25] Kasampalis T, Guth D, Moore B, Şerbănuţă TF, Zhang Y, Filaretti D, Serbanuta V, Johnson R, Roşu G. IELE: A rigorously designed language and tool ecosystem for the blockchain. In: Beek MH, McIver A, Oliveira JN, eds. Proc. of the Int'l Symp. on Formal Methods. Spinger-Verlag, 2019. 593–610. [doi: 10.1007/978-3-030-30942-8_35]

[26] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Weippl ER, Katzenbeisser S, Kruegel C, Myers AC, Halevi S, eds. Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2016. 254–269. [doi: 10.1145/2976749.2978309]

[27] Permenev A, Dimitrov D, Tsankov P, Drachsler-Cohen D, Vechev M. VerX: Safety verification of smart contracts. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. 2020. 1661–1677. [doi: 10.1109/SP40000.2020.00024]

[28] Li XY, Su C, Xiong Y, Huang WC, Wang WS. Formal verification of BNB smart contract. In: Proc. of the Int'l Conf. on Big Data. IEEE, 2019. 74–78. [doi: 10.1109/BIGCOM.2019.00021]

[29] Nehai Z, Piriou PY, Daumas F. Model-Checking of smart contracts. In: Proc. of the 2018 IEEE Int'l Conf. on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data. IEEE, 2018. 980−987. [doi: 10.1109/Cybermatics_2018.2018.00185]

[30] Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing safety of smart contracts. In: Proc. of the Network and Distributed System Security Symp.. The Internet Society, 2018. [doi: 10.14722/NDSS.2018.23082]

[31] Leinenbach D, Paul W, Petrova E. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: Aichernig BK, Beckert B, eds. Proc. of the IEEE Int'l Conf. on Software Engineering and Formal Methods. IEEE Computer Society, 2005. 2−12. [doi: 10.1109/SEFM.2005.51]

[32] Blazy S, Leroy X. Mechanized semantics for the clight subset of the c language. Journal of Automated Reasoning, 2009,43(3): 263−288. [doi: 10.1007/s10817-009-9148-3]

[33] Ellison C, Roşu G. An executable formal semantics of C with applications. In: Field J, Hicks M, eds. Proc. of the 39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 2012. 533−544. [doi: 10.1145/2103656. 2103719]

[34] Wasserrab D, Nipkow T, Snelting G, Tip F. An operational semantics and type safety proof for multiple inheritance in C++. In: Tarr PL, Cook WR, eds. Proc. of the 21th Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications. ACM, 2006. 345−362. [doi: 10.1145/1167473.1167503]

[35] Norrish M. A formal semantics for C++. Technical Report, Australia: NICTA, 2008. 1−124.

[36] Drossopoulou S, Eisenbach S. Towards an operational semantics and proof of type soundness for Java. In: Proc. of the Formal Syntax and Semantics of Java. 1998. 1523.

[37] Bogdanaş D, Roşu G. K-Java: A complete semantics of Java. In: Rajamani SK, Walker D, eds. Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 2015. 445−456. [doi: 10.1145/2676726. 2676982]

[38] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag, 2002. 1−223. [doi: 10.1007/ 3-540-45949-9]

[39] ERC-20 token standard. 2019. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

附中文参考文献:

[5] 王戟,詹乃军,冯新宇,刘志明.形式化方法概貌.软件学报,2019,30(1):33−61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

韩宁(1996−),女,硕士生,主要研究领域为智能合约,形式化验证.

李希萌(1987−),男,博士,讲师,CCF 专业会员,主要研究领域为形式化验证,区块链系统可靠性,信息流安全.

张倩颖(1986−),女,博士,讲师,CCF 专业会员,主要研究领域为嵌入式操作系统,系统安全,形式化验证.

王国辉(1984−),男,博士,实验师,CCF 专业会员,主要研究领域为高可靠嵌入式系统与定理证明.

施智平(1974−),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为形式化方法,人工智能.

关永(1966−),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为形式化验证,高可靠嵌入式系统,机器人.