软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007222] [CSTR: 32375.14.jos.007222]

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

Solidity 到 MSVL 转换的等价性研究*

王小兵, 常家俊, 李春奕, 杨潇钰, 赵 亮

(西安电子科技大学 计算机科学与技术学院, 陕西 西安 710071)

通信作者: 赵亮, E-mail: lzhao@xidian.edu.cn

摘 要: 智能合约是运行在以太坊区块链上的脚本, 能够处理复杂的业务逻辑. 大多数的智能合约采用 Solidity 语言开发. 近年来智能合约的安全问题日益突出, 为此提出了一种采用时序逻辑程序设计语言 (MSVL) 与命题投影时序逻辑 (PPTL) 的智能合约形式化验证方法, 开发了 SOL2M 转换器, 实现了 Solidity 程序到 MSVL 程序的半自动化建模, 但是缺乏对 Solidity 与 MSVL 操作语义等价性的证明. 首先采用大步语义的形式, 从语义元素、求值规则、表达式以及语句 4 个层次详细定义了 Solidity 的操作语义. 其次给出了 Solidity 与 MSVL 的状态、表达式和语句之间的等价关系, 并基于 Solidity 与 MSVL 的操作语义, 使用结构归纳法对表达式操作语义进行等价证明, 同时使用规则归纳法对语句操作语义进行等价证明.

关键词: 智能合约; Solidity; 程序转换; 操作语义; 等价性证明

中图法分类号: TP311

中文引用格式: 王小兵, 常家俊, 李春奕, 杨潇钰, 赵亮. Solidity到MSVL转换的等价性研究. 软件学报. http://www.jos.org.cn/1000-9825/7222.htm

英文引用格式: Wang XB, Chang JJ, Li CY, Yang XY, Zhao L. Research on Equivalence of Solidity to MSVL Conversion. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7222.htm

Research on Equivalence of Solidity to MSVL Conversion

WANG Xiao-Bing, CHANG Jia-Jun, LI Chun-Yi, YANG Xiao-Yu, ZHAO Liang

(School of Computer Science and Technology, Xidian University, Xi'an 710071, China)

Abstract: Smart contracts are scripts running on the Ethereum blockchain capable of handling intricate business logic with most written in the Solidity. As security concerns surrounding smart contracts intensify, a formal verification method employing the modeling, simulation, and verification language (MSVL) alongside propositional projection temporal logic (PPTL) is proposed. A SOL2M converter is developed, facilitating semi-automatic modeling from the Solidity to MSVL programs. However, the proof of operational semantic equivalence of Solidity and MSVL is lacking. This study initially defines Solidity's operational semantics using big-step semantics across four levels: semantic elements, evaluation rules, expressions, and statements. Subsequently, it establishes equivalence relations between states, expressions, and statements in Solidity and MSVL. In addition, leveraging the operational semantics of both languages, it employs structural induction to prove expression equivalence and rule induction to establish statement equivalence.

Key words: smart contract; Solidity; operational semantics; program conversion; equivalence proof

2008 年, 比特币概念首次在中本聪的文章《比特币: 一种点对点的电子现金系统》[1]中被提出, 中本聪认为互联网上的交易几乎完全依赖于可信任的第三方 (如金融机构) 来处理电子支付问题, 由于第三方机构不可避免需要调解纠纷, 这导致在交易过程中成本上升, 因此他提出了一种基于加密证明的无需可信第三方的电子交易系统, 该交易系统中使用的货币就是比特币. 2009 年比特币正式上线, 比特币是第一个实现去中心化和去信任的数字货币, 逐渐被多个大型商务网站作为支付方式之一[2], 同时区块链 1.0 也随之诞生, 作为区块链技术的初级版本, 区块

* 基金项目: 国家自然科学基金 (61972301, 61672403); 陕西省重点研发计划 (2023-YBGY-229); 西安市科技计划项目 (22GXFW0025)
收稿时间: 2023-09-21; 修改时间: 2024-01-18, 2024-04-02; 采用时间: 2024-05-10; jos 在线出版时间: 2024-07-03

链 1.0 能够验证交易过程中的合理性而无需再依赖第三方机构的监督. 比特币自 2009 年以来的成功刺激了其他基于区块链应用程序的发展, Buterin 在 2013 年提出了以太坊[3]区块链平台, 标志着区块链进入 2.0 时代, 以太坊作为分布式计算平台[4], 用户可以在其中创建和执行高级脚本, 这些脚本被称为智能合约[5], 它们运行在以太坊虚拟机 (Ethereum virtual machine, EVM) 上, 智能合约[6]大多是基于 Solidity 编写的计算机程序, 可自动执行或强制执行其合同条款[7].

由于智能合约具有去中心化、去信任等特殊性质, 不同于使用其他编程语言 (例如 C/Java), 如果对底层语义模型理解有误, 编写程序时容易产生安全漏洞, 且 Solidity 语言的多种设计 (例如 fallback 回退函数) 进一步恶化了这种情况. 近年来智能合约的安全问题日益突出, 并且通常管理价值数百万美元的数字资产, 因此其中的安全漏洞会导致巨大的损失. 一个著名的智能合约安全问题是 DAO 攻击[8], 攻击者利用了 DAO 合约中与回退函数和重入属性相关的漏洞[9]非法获得了 6 000 万美元. 此外, 由于区块链的特点, 智能合约一旦部署就不能被篡改, 因此在将智能合约部署到区块链之前对其进行验证显得非常重要.

对智能合约安全分析方法主要有 5 种, 包括形式化验证法、符号执行法、模糊检测法、中间表示法和深度学习法[10], 其中形式化验证法是通过形式化语言, 把合约中的概念、判断、推理转化成形式化模型, 从而消除合约中的歧义性和不通用性, 配合严谨的逻辑和证明, 验证智能合约函数功能的正确性和安全性. 目前有学者采用形式化方法对面向合同的智能合约进行形式化定义, 进而消除智能合约与商务合同之间的歧义[11]. 但这些方法存在一种问题: 如何保证形式化模型与智能合约之间的等价性.

在以往的工作中, 基于 MSVL (modeling, simulation and verification language) 与 PPTL (propositional projection temporal logic) 进行了智能合约的形式化验证工作[12,13], 开发了自动化建模工具 SOL2M 将 Solidity 程序转换为 MSVL 程序, 并在 UMC4M[14]中进行验证, 实现了对 Solidity 可重入漏洞的检测. 不过该工作缺乏对 Solidity 语义以及 Solidity 与 MSVL 语义等价性的研究. 因此, 本文研究了 Solidity 的语法, 定义了 Solidity 表达式及语句的操作语义, 并基于 Solidity 与 MSVL 的操作语义, 证明了 Solidity 程序与转换后的 MSVL 程序之间的等价性, 为基于 MSVL 的验证提供理论基础, 保证了建模过程的可信性和可验证性. 通过建立形式化语言和 Solidity 之间基于语义的严格映射关系, 也为基于形式化方法的智能合约安全验证奠定了更形式化的理论基础.

本文提出了一种基于大步语义的 Solidity 子集的操作语义, 并给出了其与 MSVL 操作语义的等价性证明. 首先分析 Solidity 语言的语法结构, 给出了语义元素与求值规则的定义, 进一步给出了表达式与语句的操作语义定义. 然后, 定义了 Solidity 与 MSVL 的状态、表达式和语句之间的等价关系, 并通过结构归纳法证明了表达式操作语义的等价性, 进一步通过规则归纳法证明了语句操作语义的等价性.

本文第 1 节将回顾 MSVL 语法与操作语义. 第 2 节在回顾 Solidity 语法基础上给出 Solidity 语言特性以及操作语义. 第 3 节基于 SOL2M 的转换规则给出 Solidity 与 MSVL 操作语义的等价性证明. 第 4 节给出相关工作, 第 5 节是总结及展望.

1 MSVL 语法与操作语义

MSVL 是一种时序逻辑程序设计语言, 初始版本是 Framed Tempura[15], 在引入了等待语句和非确定选择语句之后, 得到了建模、仿真与验证语言. MSVL 是投影时序逻辑 (projection temporal logic, PTL) 的可执行子集, 具有与高级程序设计语言类似的语法. MSVL 操作语义借鉴于参考文献[16−18].

1.1 MSVL 语法

MSVL 的语法类似 C 语言, 除了顺序语句、循环语句等基本语句外, 还加入了时序操作和投影相关语句, 相较 PTL 对软硬件系统有了更强的表达能力. 与其他高级编程语言一样, MSVL 提供了各种数据类型, 包含了无符号字符 (char)、无符号整数 (int)、和浮点数 (float), 此外还定义了数组 (array)、结构体 (struct)、指针 (pointer) 等. 详情见文献 [19]. MSVL 的表达式归纳定义如下:

$$
\begin{aligned}
var &::= id|id[ra]|id[ra_1][ra_2]|la.var|pt \to var\\
pt &::= la|\&|(\tau)ra|ra_1+ra_2|ra_1-ra_2|\text{ext } g(ra_1,\ldots,ra_n)|\text{ext } h(ra_1,\ldots,ra_n,RVal)\\
la &::= var|*pt\\
ra &::= c|la|\&|(\tau)ra|mop_1\ ra|ra_1\ mop_2\ ra_2|\text{if}(b)\ \text{then } ra_1 \text{ else } ra_2|\ominus ra|\text{ext } g(ra_1,\ldots,ra_n)|\text{ext } h(ra_1,\ldots,ra_n,RVal)\\
mop_1 &::= +|-|\sim\\
mop_2 &::= +|-|*||\%|\ll|\gg|\&||\wedge\\
b &::= \text{true}|\text{false}|ra_1\ mrop\ ra_2|\neg b|b_1\wedge b_2|b_1\vee b_2\\
mrop &::= <|>|<=|>=|=|!=
\end{aligned}
$$

其中, var、 id 为变量, pt 为指针表达式, τ 为类型, (τ) 为类型转化 , la 为左值表达式, ra 为右值表达式 $\ominus ra$ 表示前一状态 ra 的值, mop_1 为一元操作符, mop_2 为二元操作符, b 为布尔表达式, $mrop$ 为关系操作符, $\text{ext } g(ra_1,\ldots,ra_n)$ 与 $\text{ext } h(ra_1,\ldots,ra_n,RVal)$ 均表示外部调用, g 表示外部函数, h 表示用户自定义函数.

下面介绍 MSVL 的基本语句:

(1) 区间长度语句: $\text{empty}, \text{skip}, len(n)$;

(2) 区间框架语句: $frame(x)$;

(3) 并行语句: $p\|q$;

(4) 顺序语句: $p;q$;

(5) 非确定选择语句: p or q;

(6) 合取语句: p and q;

(7) 等待语句: $await(b)$;

(8) 立即赋值语句: $x \Leftarrow e$;

(9) 下一状态赋值语句: $x := e$;

(10) Always 语句: $alw(p)$;

(11) Next 语句: next p;

(12) 循环语句: $\text{while}(b)\,p$;

(13) 条件语句: $\text{if}(b)\text{then } p \text{ else } q$;

(14) 投影语句: $(p_1,\ldots,p_m)\,prj\,q$;

(15) 函数调用语句: $fun(e_1,\ldots,e_n)$;

(16) 外部函数调用语句: $\text{ext } fun(e_1,\ldots,e_n)$;

其中, x 为变量, e 为算数表达式, b 为布尔表达式, $p_1,\ldots,p_m,p,q$ 为 MSVL 语句.

区间长度语句 $\text{empty}, \text{skip}, len(n)$ 分别声明了当前区间长度为 0, 1, n; 并行语句 $p\|q$ 表明 p 与 q 在当前状态下同时开始执行, 并有可能在不同时间结束; 非确定选择语句 p or q 表示在当前状态下可以执行 p 或 q 中的任意一个; 顺序语句 $p;q$ 表示 p 与 q 按照顺序执行; 合取语句 p and q 表示 p 与 q 在当前状态下同时开始执行并同时结束; 等待语句 $await(b)$ 将会循环判断表达式 b 的真假, 直到 b 为真时结束循环; 立即赋值语句 $x \Leftarrow e$ 与下一状态赋值语句 $x := e$ 分别表示在当前状态和下一状态对变量进行赋值; $alw(p)$ 表示在所有状态下执行 p; next p 表示在下一状态执行 p; $\text{while}(b)\,p$, $\text{if}(b)\text{then } p \text{ else } q$ 以及函数调用语句的用法与其他高级程序设计语言相同. MSVL 不仅包含赋值语句、循环语句、条件判断语句等基本语句, 还加入了框架结构和投影结构, 为描述软硬件系统提供了更强的表达能力, 区间框架语句 $frame(x)$ 使得变量 x 的值能够在区间上自动遗传, 否则变量 x 仅在被赋值时的状态下有确定的值, 在其他状态下变量值是不确定的; 投影语句 $(p_1,\ldots,p_m)\,prj\,q$ 使得 $p_1,\ldots,p_m$ 与 q 能够并行执行, 且 $p_1,\ldots,p_m$ 顺序执行, 而 q 在另一个状态区间上执行.

1.2 MSVL 操作语义

定义 MSVL 程序 P_m 的格局为四元组 (P_m,σ_{i-1},s_i,i), 其中区间 σ_{i-1} 记录了 $< s_0,\ldots,s_{i-1} >$ 中所有状态的信息, s_i

表示当前状态, 且有 $s_i=(s_i^l,s_i^r)$, $s_i^l(x)$ 表示变量 x 在 s_i 状态下的存储位置, $s_i^r(x)$ 表示 x 在 s_i 状态下的值, i 是区间 σ_{i-1} 包含的状态个数, 亦可记作 $|\sigma_{i-1}|$. 程序起始格局为 $co_0=(P_m,\epsilon,s_0,0)$, 终止格局为 $co_f=(\text{true},\sigma,\varnothing,|\sigma|+1)$. 令 $\rightarrowtail$ 表示在相同状态内格局间的变化关系, $co \overset{*}{\rightarrowtail} co'$ 表示格局 co 在同一状态下经过多步转换为格局 co', $co \overset{+}{\rightarrowtail} co'$ 表示至少一步. 令 $\rightarrow$ 表示不同状态下格局间的变化, $\overset{*}{\rightarrow}$ 表示不同状态下格局间经过多步发生的变化, $\overset{+}{\rightarrow}$ 表示至少一步. 令 V 为变量的集合, D 为所有类型数据的集合, N_0 为非负整数的集合. 对于表达式 exp, 其格局为四元组 (exp,σ_{i-1},s_i,i). 左值表达式的求值规则为 $(la,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)$, 表示左值表达式 la 的存储位置是 $s_i^l(la)=(bl,\delta)$, 其中 bl 表示内存块索引, 是一个表示地址的整数, δ 表示块内字节偏移量. 右值表达式的求值规则为 $(ra,\sigma_{i-1},s_i,i)\Downarrow n$, 表示右值表达式 ra 的值为 n.

表 1 为 MSVL 的左值表达式求值规则, 其中 $sizeof(\tau)$ 表示类型 τ 的存储大小, $type(a)$ 表示表达式 a 的类型, $field\_offset(v_\varphi,\varphi)$ 表示结构体成员列表 φ 中名称为 v_φ 的成员变量距结构体首地址的偏移量, $ptr(bl,\delta)$ 表示指向存储位置 (bl,δ) 的指针值, 布尔值处理为整数. 规则 L1 处理变量, L2 和 L3 处理数组元素, L4 和 L5 处理结构体成员, L6 处理指针解引用.

表 1 左值表达式求值规则

序号	求值规则
L1	$(id,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}s_i^l(id)$
L2	$\dfrac{(ra,\sigma_{i-1},s_i)\Downarrow v\ (id,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)}{(id[ra],\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta+v\cdot sizeof(\tau))}$ 其中,τ为$id[ra]$的类型
L3	$\dfrac{(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1\ (ra_2,\sigma_{i-1},s_i,i)\Downarrow v_2\ (id,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)}{(id[ra_1][ra_2],\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta+(v_1\cdot n+v_2)\cdot sizeof(\tau))}$ 其中,n表示数组id的列数,τ为$id[ra_1][ra_2]$的类型
L4	$\dfrac{(la,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)\quad type(la)=\text{struct}\ id\{\varphi\}field\_offset(v_\varphi,\varphi)=\delta'}{(la.v_\varphi,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta+\delta')}$
L5	$\dfrac{(pt,\sigma_{i-1},s_i,i)\Downarrow ptr(bl,\delta)\quad type(pt)=\text{struct}\ id\{\varphi\}*field\_offset(v_\varphi,\varphi)=\delta'}{(pt\rightarrow v_\varphi,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta+\delta')}$
L6	$\dfrac{(pt,\sigma_{i-1},s_i,i)\Downarrow ptr(bl,\delta)}{(*pt,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)}$

命题 1. 对于任意给定的表达式 la, 使用左值表达式求值规则求出的值 (bl,δ) 是唯一的.

证明: 反证法, 若 la 的左值不唯一, 有两个不同的存储位置 (bl,δ) 和 (bl',δ'), 对 la 进行赋值, 则有:

(1) (bl,δ) 和 (bl',δ') 都发生变化, 表明 la 同时影响两个存储位置, 与左值表达式定义不符;

(2) (bl,δ) 和 (bl',δ') 都不发生变化, 表明 la 不影响两个存储位置, 与左值表达式定义不符;

(3) (bl,δ) 和 (bl',δ') 其中一个发生变化另一个不发生变化, 表明 la 只有一个存储位置, 与假设不符.

表 2 为 MSVL 的右值表达式求值规则, 其中 c 表示常量, x 表示 V 中的变量, v,v_1,v_2 表示 D 中的值, $(\tau)v_1$ 为 $cast(v_1, type(v_1,\tau))$ 的简写, 表示将 v_1 从它原始类型 $type(v_1)$ 强制转换为期望的类型 τ. 规则 R1 处理常量, R2 处理可以作为左值表达式的表达式, R3 处理取地址表达式, R4 处理强制类型转换, R5 和 R6 处理数组元素, R7 处理结构体成员, R8 和 R9 处理算数运算, R10 和 R11 处理 if 语句求值, R12 处理前一状态 ($\ominus$) 操作, R13 处理参数列表求值.

表 2　右值表达式求值规则

序号	求值规则
R1	$(c,\sigma_{i-1},s_i,i)\Downarrow c$
R2	$\dfrac{(la,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}s_i^l(x)}{(la,\sigma_{i-1},s_i,i)\Downarrow s_i^r(x)}$
R3	$\dfrac{(la,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)}{(\&la,\sigma_{i-1},s_i,i)\Downarrow ptr(bl,\delta)}$
R4	$\dfrac{(ra,\sigma_{i-1},s_i,i)\Downarrow v_1}{((\tau)ra,\sigma_{i-1},s_i,i)\Downarrow v}$ 其中, $v=(\tau)v_1$
R5	$\dfrac{(ra,\sigma_{i-1},s_i,i)\Downarrow v\quad(id,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)}{(id[ra],\sigma_{i-1},s_i,i)\Downarrow ptr(bl,\delta+v\cdot sizeof(\tau))}$ 其中, τ 为 $id[ra]$ 的类型
R6	$\dfrac{(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1\,(ra_2,\sigma_{i-1},s_i,i)\Downarrow v_2\,(id,\sigma_{i-1},s_i,i)\overset{l}{\Rightarrow}(bl,\delta)}{(id[ra_1][ra_2],\sigma_{i-1},s_i,i)\Downarrow ptr(bl,\delta+(v_1\cdot n+v_2)\cdot sizeof(\tau))}$ 其中, n 表示数组 id 的列数, τ 为 $id[ra_1][ra_2]$ 的类型
R7	$\dfrac{type(la)=\text{struct}\ id\{\varphi\}\quad field\_offset(v_\varphi,\varphi)=\delta'}{(la.v_\varphi,\sigma_{i-1},s_i,i)\Downarrow ptr(bl,\delta+\delta')}$
R8	$\dfrac{(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1}{(mop_1\ ra_1,\sigma_{i-1},s_i,i)\Downarrow v}$ 其中, $v=mop_1\ v_1$
R9	$\dfrac{(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1\quad(ra_2,\sigma_{i-1},s_i,i)\Downarrow v_2}{(ra_1\ mop_2\ ra_2,\sigma_{i-1},s_i,i)\Downarrow v}$ 其中, $v=v_1\ mop_2\ v_2$
R10	$\dfrac{(b,\sigma_{i-1},s_i,i)\Downarrow\text{true}\quad(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1}{(\text{if}\,(b)\,\text{then}\ ra_1\ \text{else}\ ra_2,\sigma_{i-1},s_i,i)\Downarrow v_1}$
R11	$\dfrac{(b,\sigma_{i-1},s_i,i)\Downarrow\text{false}\quad(ra_2,\sigma_{i-1},s_i,i)\Downarrow v_2}{(\text{if}\,(b)\,\text{then}\ ra_1\ \text{else}\ ra_2,\sigma_{i-1},s_i,i)\Downarrow v_2}$
R12	$\dfrac{(ra,\sigma_{i-m-1},s_{i-m},i-m)\Downarrow v}{(\ominus^m ra,\sigma_{i-1},s_i,i)\Downarrow v}$ 其中, $m\leqslant i$
R13	$\dfrac{(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1,\ldots,(ra_k,\sigma_{i-1},s_i,i)\Downarrow v_k}{((ra_1,\ldots,ra_k),\sigma_{i-1},s_i,i)\Downarrow vs}$ 其中, $vs=(v_1,\ldots,v_k)$

表 3 为 MSVL 布尔表达式求值规则.

表 3　布尔表达式求值规则

序号	求值规则
B1	$(\text{true},\sigma_{i-1},s_i,i)\Downarrow\text{true}$
B2	$(\text{false},\sigma_{i-1},s_i,i)\Downarrow\text{false}$
B3	$\dfrac{(ra_1,\sigma_{i-1},s_i,i)\Downarrow v_1\quad(ra_2,\sigma_{i-1},s_i,i)\Downarrow v_2}{(ra_1\ mrop\ ra_2,\sigma_{i-1},s_i,i)\Downarrow v},v=\begin{cases}\text{true} & \text{如果}v_1\ mrop\ v_2\text{成立}\\ \text{false} & \text{否则}\end{cases}$
B4	$\dfrac{(b,\sigma_{i-1},s_i,i)\Downarrow\text{true}}{(\neg b,\sigma_{i-1},s_i,i)\Downarrow\text{false}}\quad\dfrac{(b,\sigma_{i-1},s_i,i)\Downarrow\text{false}}{(\neg b,\sigma_{i-1},s_i,i)\Downarrow\text{true}}$
B5	$\dfrac{(b_1,\sigma_{i-1},s_i,i)\Downarrow v_1\quad(b_2,\sigma_{i-1},s_i,i)\Downarrow v_2}{(b_1\wedge b_2,\sigma_{i-1},s_i,i)\Downarrow v},v=\begin{cases}\text{true} & \text{如果}v_1=\text{true}\text{且}v_2=\text{true}\\ \text{false} & \text{否则}\end{cases}$
B6	$\dfrac{(b_1,\sigma_{i-1},s_i,i)\Downarrow v_1\quad(b_2,\sigma_{i-1},s_i,i)\Downarrow v_2}{(b_1\vee b_2,\sigma_{i-1},s_i,i)\Downarrow v},v=\begin{cases}\text{true} & \text{如果}v_1=\text{true}\text{或}v_2=\text{true}\\ \text{false} & \text{否则}\end{cases}$

表 4 为 MSVL 语句的等价规则, 其中 ms 为 MSVL 语句. SKIP 处理 skip 语句. UASS 处理下一状态赋值语句. AND 处理合取语句. NEXT 处理下一状态语句, $\bigcirc ms$ 表示下一状态执行 ms, more 表示当前区间未结束与 ¬empty 等价. $alw(ms)$ 可以表示为 $\Box ms$. CHOP 处理顺序语句并对其进行简化. IF 将条件语句转化成等价程序. WHL 将循环语句转化成等价的条件语句. PAR 处理了并行语句. 表 5 为 MSVL 真值的语义等价规则, 其中 P_m 为 MSVL 语句.

表 4 语句等价规则

序号	等价规则
SKIP	$\text{skip} \equiv \bigcirc \text{empty}$
UASS	如果$(la,\sigma_{i-1},s_i,i) \overset{l}{\Rightarrow} s_i^l(x)$,且$(ra,\sigma_{i-1},s_i,i) \Downarrow n$,那么$la := ra \equiv \bigcirc(x \Leftarrow n \wedge \text{empty})$
AND	$ms_1 \text{ and } ms_2 \equiv \wedge\{ms_1, ms_2\}$
NEXT	$\text{next } ms \equiv \wedge\{\bigcirc ms, \text{more}\}$
ALW	(1) $\wedge\{\Box ms, \text{empty}\} \equiv \wedge\{ms, \text{empty}\}$ (2) $\wedge\{\Box ms, \text{more}\} \equiv \wedge\{ms, \bigcirc\Box ms\}$
CHOP	(1) $\wedge\{w, ms_1\}; ms_2 \equiv \wedge\{w, ms_1; ms_2\}$ (2) $\bigcirc ms_1; ms_2 \equiv \bigcirc(ms_1; ms_2)$ (3) $\text{empty}; ms_2 \equiv ms_2$ (4) $\Box\text{more}; ms_2 \equiv \Box\text{more}$
IF	$\text{if}(b)\text{then}\{ms_1\}\text{else}\{ms_2\} \equiv (b \wedge ms_1) \vee (\neg b \wedge ms_2)$
WHL	$\text{while}(b)\{ms\} \equiv \text{if}(b)\text{then}\{ms \wedge \text{more}; \text{while}(b)\{ms\}\}\text{else}\{\text{empty}\}$
PAR	$ms_1 \parallel ms_2 \equiv \vee\{\wedge\{ms_1;\ \text{true}, ms_2\}, \wedge\{ms_1, ms_2;\ \text{true}\}, \wedge\{ms_1, ms_2\}\}$

表 5 真值的语义等价规则

序号	等价规则
F1	$\wedge\{\text{false}, P_m\} \equiv \text{false}$
F2	$\vee\{P_m, \text{false}\} \equiv P_m$
F3	$\wedge\{P_m, \neg P_m\} \equiv \text{false}$
T1	$\wedge\{\text{true}, P_m\} \equiv P_m$
T2	$\vee\{P_m, \text{true}\} \equiv \text{true}$
T3	$\vee\{P_m, \neg P_m\} \equiv \text{true}$

MSVL 赋值语句规则如下.

● MIN1 如果 $\exists j, 1 \leqslant j \leqslant n, (la_j,\sigma_{i-1},s_i,i) \overset{l}{\Rightarrow} s_i^l(x_j)$ 且 $(ra_j,\sigma_{i-1},s_i,i) \Downarrow v_j$, 那么 $(\wedge\{P_m, \wedge_{k=1}^{n}\{la_k \Leftarrow ra_k\}\},\sigma_{i-1},s_i,i) \rightarrowtail (\wedge\{P_m, \wedge_{k=1,k\neq j}^{n}\{la_k[v_j/x_j]\}\},\sigma_{i-1},(s_i^l, s_i^r[v_j/x_j]),i)$.

变量 x_j 的存储位置 la_j 求左值得到 $s_i^l(x_j)$, 变量 x_j 的值 ra_j 求值得到 v_j . 将立即赋值语句 $la_j \Leftarrow ra_j$ 从该格局中删除并将 s_i 状态下的变量 x_j 的值设置为 $v_j(1 \leqslant k \leqslant n且k \neq j)$.

● MIN2 如果 $(\ominus x,\sigma_{i-1},s_i,i) \Downarrow v$, 并且在程序 P_m 中没有语句 $la \Leftarrow ra$, 其中, $(la,\sigma_{i-1},s_i,i) \overset{l}{\Rightarrow} s_i^l(x)$, 那么 $(P_m,\sigma_{i-1},s_i,i) \rightarrowtail (P_m,\sigma_{i-1},(s_i^l, s_i^r[v/x]),i)$. 如果当前状态下未对变量 x 进行赋值, 则其值与上一状态保持一致.

当前状态下所有变量都被设置后, 程序简化为 $\bigcirc ms$ 或者 empty. 其求值规则如下.

● TR1 $(\bigcirc ms,\sigma_{i-1},s_i,i) \rightarrow (ms,\sigma_i,s_{i+1},i+1)$

● TR2 $(\text{empty},\sigma_{i-1},s_i,i) \rightarrow (\text{true},\sigma_i,\varnothing,i+1)$

MSVL 函数调用可分为外部调用和内部调用[17]. 执行外部调用时, 程序 P_m 的前后时序状态区间可认为未发生变化, 执行内部调用时, 程序 P_m 正常记录时序状态区间的变化. MSVL 函数调用求值规则如下.

(1) 内部调用, 对于 function $f(\tau_1 x_1,\ldots,\tau_k x_k,\tau\ RVal)\{mdcl;ms\}$, 其中 $mdcl$ 是函数内部变量定义语句. 求值规则如下.

● FUN $f(ra_1,\ldots,ra_k,RVal) \equiv ((\wedge_{j=1}^{k}\tau_j x_j \Leftarrow ra_j \wedge mdcl);ms;)$

$$\bigcirc(\text{ext } mfree(x_1,\ldots,x_k,RVal,mdcl) \wedge \text{empty})$$

其中, $mfree(x_1,\ldots,x_k,RVal,mdcl)$ 是一个用于释放变量和内存的外部函数.

(2) 外部函数调用, 仅关注执行前后的信息. 分为用户定义函数和外部函数两种.

用户定义函数 ext $f(ra_1,\ldots,ra_k,RVal)$ 类似于白盒调用, 其规则如下.

● EXT1 如果 $\sigma' =< s_0',\ldots,s_n' >$, $s_0' = s_i$ 且 $((\wedge_{j=1}^{k}\tau_j x_j \Leftarrow ra_j \wedge mdcl);ms,\epsilon,s_0',0) \overset{+}{\rightarrow} (\text{true},\sigma',\varnothing,n+1)$, 那么, $(\wedge\{\bigcirc P_m,$

ext $f(ra_1,\ldots,ra_k,RVal)\},\sigma_{i-1},s_i,i)\rightarrow(P_m,\sigma_i,s_{i+1},i+1)$ 且 $s_{i+1}=s'_n$.

外部函数 ext $g(ra_1,\ldots,ra_k)$ 类似于黑盒调用, 其规则如下.

● EXT2 如果对每个 $1\leqslant j\leqslant k$, $(ra_j,\sigma_{i-1},s_i,i)\Downarrow v_j$, 且 $<s_i>$ 是 $g(v_1,\ldots,v_k)$ 的模型, 那么 $(\wedge\{\bigcirc P_m,\text{ext } g(ra_1,\ldots,ra_k)\},\sigma_{i-1},s_i,i)\rightarrow(P_m,\sigma_i,s_{i+1},i+1)$, $s_{i+1}=s_i$. 这表示 $g(ra_1,\ldots,ra_k)$ 在一个单独的状态 s_i 上执行, 将其插入主区间 σ_i.

2 Solidity 语法与操作语义

本节首先回顾 Solidity 语法, 随后对其操作语义进行了定义, 其中对 Solidity 的形式化定义侧重于动态语义, 为 Solidity 语言定义了状态与语义元素, 在全局环境上与局部环境上定义了相关操作, 以大步语义的形式制定了 Solidity 的求值规则, 分为左值表达式、右值表达式以及语句的求值规则, 并根据 Solidity 语法, 给出了表达式与语句的操作语义.

2.1 Solidity 语法

Solidity 的语法接近于 JavaScript, 相比于高级程序设计语言, 缺少多线程以及高并发等特性. Solidity 的表达式归纳定义如下.

le ::= $id\,|\,le[e]\,|\,le.id$

e ::= $constant\,|\,id\,|\,le\,[e]\,|\,le.e\,|\,e_1\,(e_2^*)\,|\,e^*\,|\,unop\ e\,|\,e_1\ binop\ e_2\,|\,le\ assign\ e$

$unop$ ::= $++\,|\,--\,|\,!\,|\,\sim$ $\quad bop_1$ ::= $+\,|\,-\,|\,*\,|\,/\,|\,\%$

bop_2 ::= $<<\,|\,>>\,|\,\&\,|\,\,|\,|\,\wedge$ $\quad bop_3$ ::= $===\,|\,!=$

bop_4 ::= $=>\,|\,>=\,|\,<\,|\,<=$ $\quad bop_5$ ::= $\&\&\,|\,|\,|$

binop ::= $bop_1\,|\,bop_2\,|\,bop_3\,|\,bop_4\,|\,bop_5$

assign ::= $==\,|\,+=\,|\,-=\,|\,*=\,|\,/=$

le 表示左值表达式, 可以通过寻址操作获取其内存地址, id 为变量名称; e 为表达式, 包括常量 $constant$, 变量 id, 数组元素, 结构体成员等, $e_1\,(e_2^*)$ 表示函数调用, e^* 表示元组; $unop$ 表示单目运算, 包括自增 ($++$), 自减 ($--$), 逻辑非 (!) 和位运算符 ($\sim$); $binop$ 表示双目运算, 包括算术运算符记为 bop_1, 位运算符记为 bop_2, 关系运算符记为 bop_3 和 bop_4, 以及逻辑运算符 (&&,||), 记为 bop_5; $assign$ 表示赋值表达式.

Solidity 程序的相关定义如下:

$limit$ ::= public | private | internal

$location$ ::= memory | storage

$type$ ::= uint8 | uint16 | uint32 | uint64 | uint128 | uint256 | address | bool

| mapping ($type_1$ => $type_2$) | id_t | $type$[]

$StructDef$ ::= struct id {($type$ id)*}

$VarDef$ ::= $type$ $location$? id

$FunDef$ ::= function id ($VarDef^*$) $limit$? $FunReturn$? $stmt$

$FunReturn$ ::= returns (($type$ id)*)

$Block$ ::= $StructDef\,|\,VarDef\,|\,FunDef$

P_s ::= contract id_c {$Block$}

其中 $limit$ 表示可见性关键词 public (任何用户或者合约都能调用和访问), private (只能在其所在的合约中调用和访问), internal (子合约可以访问父合约中定义的 internal 函数); $location$ 表示数据内存类型 memory (存储于函数内部) 和 storage (存储于函数外部); $type$ 表示 Solidity 中的数据类型, 包含: uint8–uint256, 地址类型 address (表示区块链中的合约地址), 布尔类型 bool, 映射类型 mapping, 自定义类型 id_t 以及数组. $StructDef$ 表示结构体定义, $VarDef$ 表示变量定义, $FunDef$ 表示函数定义, $FunReturn$ 表示函数返回定义. P_s 表示 Solidity 程序, 由关键字

contract、合约名称与合约体 *Block* 组成, 合约体 *Block* 包含结构体定义、变量定义与函数定义.

Solidity 基本语句定义为 *stmt* , 包含语句块 $stmt^*$, 表达式语句 *e*; , 变量声明语句 *VarDef* , 返回语句 return, 条件判断语句 if-else, 循环语句, 转向语句 throw. 语法定义如下:

$$stmt ::= \{stmt^*\}|e;|VarDef|\text{return } e;|\text{return;}|\text{if}\,(e)\,stmt_1\,(\text{else } stmt_2)?$$

$$|\text{for}\,(stmt_1; e; stmt_2)\,stmt|\text{while}\,(e)\,stmt|\text{throw;}$$

2.2 语义定义策略

本节在定义 Solidity 操作语义时, 首先借鉴了文献 [20] 中 Solidity 语义定义的基本形式, 其中给出的大步语义形式的 Solidity 求值规则与核心动态语义为本文提供了基础结构, 并且对于以太坊状态和 Solidity 状态的形式化定义也为本文定义语义元素提供了思路. 之后参考了文献 [21] 中对于 Solidity 内存模型的研究, 其中给出的对 Solidity 内存模型以及函数调用的形式化定义, 为本文定义语义元素中的内存模块提供了理论基础, 也为函数调用的操作语义提供了值得借鉴的步骤. 然后参考文献 [18] 中证明 C 语言与 MSVL 操作语义等价性的方法, 确定了本文的等价性证明思路. 针对将来 Solidity 快速迭代引入的新版本特性, 可根据语句变化完善现有的操作语义, 进而将本文的等价性研究工作扩展到 Solidity 高阶版本.

本节所定义的 Solidity 操作语义覆盖了 Solidity 的主要语法, 并给出了更详尽的语义元素以及操作函数定义, 相较于文献 [20] 的 Solidity 操作语义更加完善. 本文采用大步语义的形式定义 Solidity 的操作语义, 这一方面是由于文献 [18] 均采用大步语义的形式, 保持形式的一致有利于之后等价性的证明以及研究的扩展; 另一方面大步语义在描述高级程序设计语言的操作语义上更有优势. 小步语义通常用于描述机器级语言的语义, 更注重于描述程序每走一步的规则, 解释执行语句的每个步骤如何进行, 即将父结构不断拆分为子结构, 最终递归得到结果; 而大步语义则更关注程序的开始与结束的状态变化, 描述如何得到语句执行终止的最终状态, 预设了语句可终止, 所给出的操作语义更加高效, 易于调试和扩展[22].

大步语义的求值规则有如下基本形式[23]:

$$\rho \vdash e \Rightarrow v \tag{1}$$

$$s_1 \vdash e \Rightarrow s_2 \tag{2}$$

其中, ρ 为环境, 公式 (1) 表示在环境 ρ 中表达式 *e* 的值为 *v* . s_1 与 s_2 为状态, 公式 (2) 表示在状态 s_1 下执行表达式 *e* 会得到新状态 s_2 . 基于此本文定义了 Solidity 的求值规则.

语义的推导规则通常有如下形式:

$$\frac{P_1 P_2 \ldots P_n}{C} \tag{3}$$

其中, $P_i (1 \leqslant i \leqslant n)$ 为假设或者前提, *C* 表示结论, 若假设或前提 P_i 均成立, 则结论 *C* 一定成立, 若没有假设或前提, 结论依然成立, 那么结论即为公理.

2.3 内存模型

Solidity 与其他语言不同, 使用 memory、storage、callback 和 stack 这 4 个不同的数据位置, 其中 stack 只有在为字节码赋予语义时才有研究意义, 而 calldata 位置的作用可以忽略, 因此本文中仅关注前两项 memory 和 storage 数据位置.

(1) memory: 存储函数内部的所有本地数据, 即局部变量. memory 数据在函数执行完成后被删除.

(2) storage: 存储函数外部的数据, 即状态变量. 在 storage 中的变量将永久保存在区块链上. storage 是一组存储槽, 每个槽长 32 字节, 可通过 256 位地址寻址.

(3) callback: 存储函数的传入执行数据, 不可修改.

(4) stack: 用于加载变量和存储 EVM 生成的中间值.

Solidity 中状态变量可以是机器整数、地址、数组、结构体或者映射类型. 状态变量为 storage 类型, 永久存储在区块链中. 函数中的局部变量默认为 memory 类型, 也可显示声明为 storage 类型. 此外, 全局变量在 storage 中

的存储位置由数据类型决定, 其规则如下:

(1) 值类型仅使用它们所需的字节, 若当前插槽的剩余空间不足以存放一个值类型, 则将其存放在下一个存储插槽.

(2) 结构体与数组类型总是放在一个存储插槽的开始, 结构体与数组中的各元素按规则紧密存储.

(3) 映射与动态数组不可预知大小, 因此仅占用 32 字节, 其包含的元素存储的位置由 keccak-256 哈希函数计算确定. 映射所占用的插槽实际上并未使用, 但仍需要. 动态数组所占用的插槽存放数组中元素的数量.

可知基本类型仅使用存储它们所需的字节, 结构体和数组等总是占用一整个新插槽, 因此 storage 地址由插槽索引值 sl 以及插槽内偏移量 δ 决定.

2.4 语义元素

本文定义 Solidity 状态 $\mu =< a,\sigma,M >$ 为一个三元组, 包含当前合约在区块链中的地址 address, 记为 a, 以太坊网络状态 σ 和内存 memory 状态 M, memory 用于保存当前正在执行的函数的局部变量. 执行智能合约代码会改变以太坊网络的状态, 从形式化的角度来看, 以太坊网络状态 σ 可由智能合约地址 a 映射得到, 定义网络状态 $\sigma =< b,p,S >$ 为一个三元组, 包含合约余额 b, 即账户所持有的货币数量, 合约程序 p, 以及存储 storage 的状态 S, storage 用于保存合约的状态变量.

表 6 为 Solidity 语义元素的定义, 其中左值 lv 表示变量在 storage 中或者 memory 中的地址, sl 为 storage 插槽索引, storage 地址 $addr$ 是一个由存储插槽索引与偏移量 δ 组成的序偶对. 求值环境包括全局环境 g 与局部环境 l. fd 为函数定义, 全局环境 g 将程序全局变量和函数名映射到存储插槽索引, 若为函数, 则可以从索引映射到函数定义局部环境. 局部环境 l 将局部变量映射到 memory 地址. 内存状态 M 从 memory 地址映射到其内容, 存储状态 S 从 storage 插槽索引映射到其内容. τ 表示表达式所对应的类型.

表 6 语义元素

语义元素	定义	说明
左值 lv	$lv ::= loc \mid addr$	loc表示局部变量在memory中的地址, $addr$ 表示全局变量在storage中的地址
storage地址 $addr$	$addr ::= (sl,\delta)$	storage插槽索引与偏移量组成的序偶对
全局环境 g	$g ::= (id \to sl)\times(sl \to fd)$	从全局变量映射到storage插槽索引, 并从插槽索引映射到函数定义
函数局部环境 l	$l ::= id \to loc$	从局部变量映射到memory地址, 函数局部环境 l 用于保存当前函数内部的局部变量
内存memory状态	$M ::= loc \to v$	从memory地址映射到内容
存储storage状态	$S ::= sl \to v$	从storage插槽索引映射到内容
类型 τ	τ_e	表达式 e 的类型
结构体成员变量	λ_{id}	结构体 id 的成员列表

表 7 给出了在存储 storage 状态、全局环境和局部环境上的操作. 在 storage 状态上有 *store* 、 *load* 、 *delete* 、 *alloc* 这 4 个基本操作, 函数 *store* 在 storage 地址 $addr$ 处存储类型为 τ 的值 v. 函数 *load* 在 storage 地址 $addr$ 处根据内存长度加载其对应的内容. 函数 *alloc* 能够在 storage 地址 $addr$ 处根据类型 τ 分配相应的存储长度. Solidity 的 storage 永久存储在区块链之上, 区块链作为一种公有资源, 为避免滥用, 鼓励使用者主动对空间进行回收, 释放空间时会返还 gas, 但是 Solidity 中的 *delete* 与其他语言有所不同, 并非释放空间, 而是对变量进行初始化.

在全局环境上和局部环境上均有函数 *getAddr* 和函数 *getSize* , *getAddr* 可以根据全局环境或局部环境以及变量名称从 storage 或者 memory 中获取其映射的内容. 函数 *getSize* 可根据全局环境或局部环境以及当前变量的类型, 获取其所需的内存长度.

表 8 给出了部分函数定义, 左值函数用于对地址的求值, 在左值表达式求值时给出具体应用, 其中, 函数 $field\_offset(id,\varphi)$ 根据结构成员列表 φ 进行查询, 得到名称为 id 的成员变量的偏移量. 函数 $LVs\_struct(s\_id, addr, id)$ 首先根据函数 $field\_offset(id,\varphi)$ 得到成员变量 id 在 storage 中的偏移量, 再结合结构体的首地址 $addr$ 得到成员变量在 storage 中的实际位置. 函数 $LVs\_arr(le,addr,i)$ 首先根据数组类型 τ 与索引值计算元组相对于数组首

地址的偏移量, 再结合数组的首地址 $addr$, 得到成员变量在 storage 中的实际位置. 右值函数中给出了单目运算与双目运算的函数. 布尔函数中根据变量的类型与值, 计算真值.

表 7 操作函数定义

语义元素	定义	说明
storage操作	$store(S,addr,\tau,v)=S'$	在地址addr处存储类型为 τ 的数据v后, 更新storage状态
	$load(S,addr,sizeof(\tau))=v$	得到存储在地址addr到 $(addr+sizeof(\tau)-1)$ 之间的值
	$delete(S,addr)=S'$	solidity在0.6.12及之后的版本中, 使用delete删除一个元素, 并不是真的删除, 只是把这个值变成初始值, 即非零字节设置为零
	$alloc(S,id,sizeof(\tau))=(S',addr)$	storage由插槽slot组成, storage根据变量名称及数据类型决定为其分配相应的空间
全局环境上的操作	$getAddr(g,id)=addr$	计算全局变量 id 在storage中的存储位置
	$getsize(g,\tau_e)$	根据表达式类型得到所需的内存长度
局部环境上的操作	$getAddr(l,id)=addr$	计算局部变量 id 在memory中的存储位置
	$getsize(l,\tau_e)$	根据变量类型得到所需的内存长度

表 8 相关函数定义

语义元素	定义	说明
左值函数	$field\_offset(id,\varphi)$	返回结构体成员列表 φ 中名称为 id 的成员变量在storage中的偏移量
	$LVs\_struct(s\_id,addr,id)=addr+field\_offset(id,\varphi_{s_{id}})$	根据结构体首地址及成员变量名称 id 得到成员变量在storage中的存储位置
	$LVs\_arr(le,addr,i)=addr+i\cdot getSize(g,\tau_{le})$	根据数组、数组首地址以及数组下标 i , 得到数组第 i 项在storage中的存储位置
右值函数	$V\_unop(unop,v)=unop\ v$	根据值 v 和单目运算符 $unop$, 计算单目运算 $unop\ v$ 的值
	$V\_binop(binop,v_1,v_2)=v_1\ binop\ v_2$	根据值 v_1,v_2 和双目运算符 $binop$, 计算单目运算 $v_1\ binop\ v_2$ 的值
布尔函数	$is\_true(v,\tau_v)$	若数据 v 不等于0, 求值结果为 true
	$is\_false(v,\tau_v)$	若数据 v 不等于0, 求值结果为 false

2.5 求值规则

本文定义 Solidity 求值规则时, 将语法元素和 Solidity 状态 μ 与执行结果联系起来, Solidity 的操作语义采用如下形式的求值规则进行定义.

$p,g,l\vdash e,\mu\Rightarrow out$ 右值表达式求值

$p,g,l\vdash le,\mu\overset{l}{\Rightarrow} out$ 左值表达式求值

$p,g,l,f\vdash stmt,\mu\Rightarrow out,\mu'$ 语句求值

$out:=Normal\mid Return\mid Return\ v\mid Fail\mid v\mid lv\mid vs$ 求值结果

求值规则将合约程序 p 与全局环境 g 及局部环境 l 联系起来, 在当前状态 μ 下进行求值, 左值表达式的求值结果为内存位置 lv , 右值表达式的求值结果为值 v 或 vs , 本文使用 $\overset{l}{\Rightarrow}$ 与 $\Rightarrow$ 区分左值表达式求值与右值表达式求值. out 表示求值结果, 表达式求值与语句求值均会得到 out 中的一类结果, out 包含: $Normal$ 表示语句正常结束, $Return$ 表示在遇到 return 语句时中断控制流并跳出函数体且无返回值, $Return\ v$ 表示 return 语句含有返回值, $Fail$ 表示语句执行失败, v 表示右值, lv 表示左值, vs 表示值列表.

本文关注 Solidity 本身的语法语义, 因此认为 Solidity 状态 μ 的改变仅由其存储状态 S 或内存状态 M 的改变而决定, 然而 MSVL 中没有相应的内存结构可以区分, 因此在定义 Solidity 语句的操作语义时, 无需刻意区分 Solidity 状态 μ 具体是因存储状态 S 或内存状态 M 的改变而发生改变的.

2.6 左值表达式求值规则

Solidity 中的左值表达式定义为:

$$le ::= id \mid le[e] \mid le.id$$

其中, le 为左值表达式, id 为变量名称, e 为表达式, $le[e]$ 在 Solidity 中既可以表示对数组元素的引用, 也可以表示对映射类型的引用, 当表示对数组元素的引用时, 对 le 求值得到数组首元素的地址, 对 e 求值得到该元素在数组中的索引值, 对 $le[e]$ 进行左值表达式求值得到当前元素的地址. 当表示对映射类型的引用时, 对 le 求值得到映射类型的地址, 对 e 求值得到映射类型中的 key 的值, 对 $le[e]$ 进行左值表达式求值得到根据 key 值映射到键值对的地址.

全局变量的左值表达式操作语义如下:

$$\frac{getAddr(g,id)=addr}{p,g,l \vdash id \overset{l}{\Rightarrow} addr} \tag{E-1}$$

规则 (E-1) 表示在全局环境中根据变量名称, 使用函数 $getAddr(g,id)=addr$ 可以直接获取变量在 storage 中的存储位置.

$$\frac{\tau_{le}=\textbf{array} \quad p,g,l \vdash le,\mu \overset{l}{\Rightarrow} (sl,0) \quad p,g,l \vdash e,\mu \Rightarrow v \quad addr=(sl,0)+v\cdot getSize(g,\tau_{le})}{p,g,l \vdash le[e],\mu \overset{l}{\Rightarrow} addr} \tag{E-2}$$

规则 (E-2) 表示了获取数组元素存储位置的操作语义, 首先根据 τ_{le} 判断当前 $le[e]$ 的类型是否为数组, 确认当前表达式类型为数组类型后, 先对 le 求左值, 得到数组的首地址在 storage 中的位置. 根据 Solidity 的内存模型可知, 数组类型总是开启一个新的插槽, 因此得到首地址为 $(sl,0)$, 然后对 e 求右值, 得到元素在数组中的索引值, 最终得到元素的实际位置为首地址加上元素的偏移量.

$$\frac{\tau_{le}=\textbf{mapping} \quad p,g,l \vdash le,\mu \overset{l}{\Rightarrow} (sl,0) \quad p,g,l \vdash e,\mu \Rightarrow v \quad keccak256(h(v)\cdot(sl,0))=addr}{p,g,l \vdash le[e],\mu \overset{l}{\Rightarrow} addr} \tag{E-3}$$

规则 (E-3) 表示了映射类型的左值操作语义, 首先根据 τ_{le} 判断类型是否为映射类型, 确认当前表达式类型为映射类型后, 先对 le 求左值, 得到映射类型的首地址在 storage 中的位置. 根据 Solidity 的内存模型可知, 映射类型总是开启一个新的插槽, 因此得到首地址为 $(sl,0)$, 然后对 e 求右值, 得到映射类型的 key 值 v, 其中 $keccak256(h(v)\cdot addr)$ 为哈希计算函数, 得到该键值在 storage 中的地址 $addr$.

$$\frac{p,g,l \vdash le,\mu \overset{l}{\Rightarrow} (sl,0) \quad LVs\_struct(lv,id)=addr=(sl,0)+field\_offset(id,\varphi_{s_{id}})}{p,g,l \vdash le.id,\mu \overset{l}{\Rightarrow} addr} \tag{E-4}$$

规则 (E-4) 描述了获取结构体成员变量存储位置的操作语义. le 求左值, 得到结构体在 storage 中的首地址. 根据 Solidity 的内存模型可知, 结构体总是开启一个新的插槽, 因此得到首地址为 $(sl,0)$. 再根据函数 $LVs\_struct(lv,id)$ 得到成员变量在 storage 中的偏移量.

2.7 右值表达式求值规则

Solidity 中的右值表达式定义如下:

$$e ::= constant \mid id \mid le[e] \mid le.e \mid e_1(e_2^*) \mid e^* \mid unop\ e \mid e_1\ binop\ e_2 \mid le\ assign\ e$$

右值表达式操作语义如下:

$$\frac{p,g,l \vdash le \overset{l}{\Rightarrow} addr \quad load(S,addr,getSize(g,\tau_{le}))=v}{p,g,l \vdash le,\mu \Rightarrow v} \tag{E-5}$$

规则 (E-5) 描述了既能够成为左值表达式又能够成为右值表达式的表达式, 如数组类型、结构体类型以及变量名称, 对这些表达式进行右值表达式求值操作语义, 先获取表达式作为左值表达式时求值所得的地址, 再使用函

数 $load(S,addr,getSize(g,\tau_{le}))$ 从存储中加载其右值.

$$\frac{\tau_{le}=\textbf{array}\quad p,g,l\vdash le,\mu\overset{l}{\Rightarrow}addr_1\quad p,g,l\vdash e,\mu\Rightarrow v_1 \\ LVs\_arr(addr_1,v_1)=addr_2\quad v=load(S,addr_2,getSize(g,\tau_{le}))}{p,g,l\vdash le[e],\mu\Rightarrow v} \tag{E-6}$$

规则 (E-6) 描述了数组元素获取相应值的操作语义. 首先判断当前 $le[e]$ 是否为数组类型的引用, 参考左值表达式求值规则 (E-2), 得到数组元素的存储地址, 根据地址与函数 $load(S,addr,getSize(g,\tau_{le}))$ 得到该数组元素在 storage 中的值.

$$\frac{\tau_{le}=\textbf{mapping}\quad p,g,l\vdash le,\mu\overset{l}{\Rightarrow}(sl,0)\quad p,g,l\vdash e,\mu\Rightarrow v \\ keccak256(h(v)\cdot(sl,0))=addr\quad load(S,addr,getSize(g,\tau_e))=v}{p,g,l\vdash le[e],\mu\Rightarrow v} \tag{E-7}$$

规则 (E-7) 描述了映射类型获取相应右值的操作语义. 确认了当前表达式 $le[e]$ 为映射类型后, 根据左值表达式操作语义规则 (E-3), 得到映射类型的存储地址, 再根据函数获取相应的右值.

$$\frac{p,g,l\vdash le,\mu\overset{l}{\Rightarrow}addr_1\quad LVs\_struct(addr_1,id)=addr_2=addr_1+field\_offset(id,\varphi_{s_{id}}) \\ v=load(S,addr_2,getSize(g,\tau_{id}))}{p,g,l\vdash le,id,\mu\Rightarrow v} \tag{E-8}$$

规则 (E-8) 描述了获取结构体体成员变量值的操作语义. 结构体成员列表 λ 存放结构体成员变量与其偏移量的映射, 先根据函数 $field\_offset(id,\varphi_{s_{id}})$ 从 λ 中得到在 storage 中的偏移量, 再通过函数 $LVs\_struct(addr_1,id)$ 获取结构体成员变量的地址, 最终通过函数 $load(S,addr,getsize(g,\tau_{id}))$ 得到在 storage 中对应的值.

$$\frac{p,g,l\vdash le,\mu\overset{l}{\Rightarrow}addr\quad p,g,l\vdash e,\mu\Rightarrow v\quad store(S,addr,\tau_v,v)=S'}{p,g,l\vdash le=e,\mu\Rightarrow Normal} \tag{E-9}$$

规则 (E-9) 描述了赋值表达式的操作语义. 首先通过左值表达式求值规则得到 *le* 的地址, 再通过右值表达式求值规则得到 *e* 的实际值, 最后通过函数 $store(S,addr,\tau_v,v)$ 将类型为 τ_v 的值 v 存放在 storage 中的地址 *addr* 处.

$$\frac{p,g,l\vdash e,\mu\Rightarrow v_1\quad unop::=++\,|\,--\quad V\_unop(unop,v_1)=v}{p,g,l\vdash unop\ e,\mu\Rightarrow v} \tag{E-10}$$

规则 (E-10) 描述了单目运算表达式的操作语义, 由于自增自减操作的语义有所不同, 因此首先判断单目运算符的类型, 之后根据单目运算函数 $V\_unop(unop,v_1)=v$ 得到单目运算的结果, $v=unop\ v_1$.

$$\frac{p,g,l\vdash e,\mu\Rightarrow v_1\quad p,g,l\vdash e,\mu\overset{l}{\Rightarrow}addr \\ unop::=++\quad v=v_1+1\quad store(S,addr,\tau_v,v)=S'}{p,g,l\vdash unop\ e,\mu\Rightarrow v,\mu'} \tag{E-11}$$

规则 (E-11) 描述了自增操作的操作语义, 与其他单目运算不同, 自增操作是对本身的变量进行重新赋值, 赋值为原本的值加 1. 自减操作与之同理.

$$\frac{p,g,l\vdash e_1,\mu\Rightarrow v_1\quad p,g,l\vdash e_2,\mu\Rightarrow v_2\quad V\_binop(binop,v_1,v_2)=v}{p,g,l\vdash e_1\ binop\ e_2,\mu\Rightarrow v} \tag{E-12}$$

规则 (E-12) 描述了双目运算表达式的操作语义. 根据右值表达式求值规则得到 e_1 与 e_2 的值, 通过函数 $V\_binop(binop,e_1,e_2)=v$ 得到双目运算的结果, $v=v_1\ binop\ v_2$.

$$\frac{p,g,l\vdash e_1,\mu\Rightarrow v_1\ldots p,g,l\vdash e_k,\mu\Rightarrow v_k\quad vs=(v_1,\ldots,v_k)}{p,g,l\vdash e^*,\mu\Rightarrow vs} \tag{E-13}$$

规则 (E-13) 描述了参数列表的操作语义. 其中 $e^*=(e_1,\ldots,e_k)$, 对参数列表中每一个参数进行右值表达式求值, 得到 $(v_1,\ldots,v_k)$, 且 $vs=(v_1,\ldots,v_k)$.

2.8 语句求值规则

Solidity 基本语句定义为 *stmt* , 包含空语句, 返回语句, 表达式语句, 条件判断语句, 顺序执行语句, 循环语句以及函数调用语句等. 本节给出 Solidity 语句的操作语义

空语句表示在当前状态下不执行任何操作, 即任何变量的地址与值都未发生变化, 因此执行前后状态无变化, 其操作语义如下:

$$p,g,l,f \vdash ;,\mu \Rightarrow Normal,\mu \tag{S-1}$$

return 语句表示在当前状态下退出函数, 不执行函数内部剩余的语句, 因此执行前后状态发生变化. return 语句有 3 种情况: 无返回值 (S-2), 返回一个值 (S-3), 返回值列表 (S-4). return 语句的操作语义如下:

$$p,g,l,f \vdash \text{return};,\mu \Rightarrow Return,\mu' \tag{S-2}$$

$$\frac{p,g,l \vdash e,\mu \Rightarrow v}{p,g,l,f \vdash \text{return } e,\mu \Rightarrow Return\ v,\mu'} \tag{S-3}$$

$$\frac{p,g,l \vdash e^*,\mu \Rightarrow vs}{p,g,l,f \vdash \text{return } e^*,\mu \Rightarrow Return\ vs,\mu'} \tag{S-4}$$

条件语句根据条件判断表达式 e 的真值选择执行哪一部分的语句, 因此执行前后状态发生变化, if 语句的操作语义如下:

$$\frac{p,g,l \vdash e,\mu \Rightarrow v \quad is\_true(v,\tau_v) \quad p,g,l,f \vdash stmt_1,\mu \Rightarrow out,\mu'}{p,g,l,f \vdash \text{if}(e)\ stmt_1\ (\text{else } stmt_2),\mu \Rightarrow out,\mu'} \tag{S-5}$$

$$\frac{p,g,l \vdash e,\mu \Rightarrow v \quad is\_false(v,\tau_v) \quad p,g,l,f \vdash stmt_2,\mu \Rightarrow out,\mu'}{p,g,l,f \vdash \text{if}(e)\ stmt_1\ (\text{else } stmt_2),\mu \Rightarrow out,\mu'} \tag{S-6}$$

首先对条件判断表达式进行求值, 其中 $is\_true(v,\tau)$ 与 $is\_false(v,\tau)$ 函数用于描述判定条件 e 的求值结果. 当求值结果为 true 时, 执行第 1 个分支的语句 $stmt_1$, 得到语句 $stmt_1$ 的执行结果 out, 条件语句的执行结果与语句 $stmt_1$ 的结果一致, 参考规则 (S-5). 当判定条件求值结果为 false 时, 条件语句的执行结果与语句 $stmt_2$ 的结果一致, 参考规则 (S-6).

顺序执行语句的操作语义如下:

$$\frac{p,g,l,f \vdash stmt_1,\mu \Rightarrow Normal,\mu_1 \quad p,g,l,f \vdash stmt_2,\mu_1 \Rightarrow out,\mu_2}{p,g,l,f \vdash stmt_1;stmt_2,\mu \Rightarrow out,\mu_2} \tag{S-7}$$

$$\frac{p,g,l,f \vdash stmt_1,\mu \Rightarrow Fail|Return,\mu'}{p,g,l,f \vdash stmt_1;stmt_2,\mu \Rightarrow Fail|Return,\mu'} \tag{S-8}$$

当第 1 个语句 $stmt_1$ 正常执行结束时, 顺序执行语句的执行结果与 $stmt_2$ 的执行结果一致, 参考规则 (S-7). 当语句 $stmt_1$ 执行失败或者包含 return 语句时, $stmt_2$ 不再执行, 顺序执行语句的执行结果与 $stmt_1$ 一致, 参考规则 (S-8).

循环语句有两类, 其语法定义如下:

(1) While(e) $stmt$

(2) for($stmt_1$; e; $stmt_2$) $stmt$

for 循环语句的操作语义:

$$\frac{p,g,l,f \vdash stmt_1,\mu \Rightarrow Normal,\mu_1 \quad p,g,l,f \vdash \text{for}(;e;stmt_2)\ stmt,\mu_1 \Rightarrow out,\mu_2}{p,g,l,f \vdash \text{for}(stmt_1;e;stmt_2)\ stmt,\mu \Rightarrow out,\mu_2} \tag{S-9}$$

$$\frac{p,g,l \vdash e,\mu \Rightarrow v \quad is\_false(v,\tau_v)}{p,g,l,f \vdash \text{for}(;e;stmt_2)\ stmt,\mu \Rightarrow out,u} \tag{S-10}$$

$$\frac{\begin{array}{c}p,g,l \vdash e,\mu \Rightarrow v \quad is\_true(v,\tau_v) \quad p,g,l,f \vdash stmt,\mu \Rightarrow Normal,\mu_1 \\ p,g,l \vdash stmt_2,\mu_1 \Rightarrow out,u_2 \quad \text{for}(;e;stmt_2)\ stmt,\mu_2 \Rightarrow out,\mu_3\end{array}}{p,g,l,f \vdash \text{for}(;e;stmt_2)\ stmt,\mu \Rightarrow out,\mu_3} \tag{S-11}$$

$$\frac{p,g,l \vdash e,\mu \Rightarrow v \quad is\_true(v,\tau_v) \quad p,g,l,f \vdash stmt,\mu \Rightarrow out,\mu_1 \quad out = Return|Fail}{p,g,l,f \vdash \text{for}(;e;stmt_2)\ stmt,\mu \Rightarrow out,\mu_1} \tag{S-12}$$

for 循环语句首先执行初始化语句 $stmt_1$, 再执行循环内容, 因此可将其拆分为 $stmt_1;\text{for}(;e;stmt_2)\ stmt$, 参考规则 (S-9), 根据终止条件给出 3 个循环规则. 循环判断条件 e_1 求值为 false 时, 循环结束, 参考规则 (S-10). 循环条件 e_1 值为 true 时, 先执行循环体内语句 $stmt$, 正常执行完毕后, 再执行 e_2, 得到新的状态, 从新的状态出发, 继续进入循环, 参考规则 (S-11). 若循环体内语句执行结果为 $Return$ 或 $Fail$, 表明函数在此处返回或者出现异常, 循环立刻

结束, 参考规则 (S-12).

while 循环函数的操作语义:

$$\frac{p,g,l \vdash e,\mu \Rightarrow v \quad is\_false(v,\tau_v)}{p,g,l,f \vdash \text{while}(e)\ stmt,\mu \Rightarrow out,\mu} \tag{S-13}$$

$$\frac{\begin{array}{c}p,g,l \vdash e,\mu \Rightarrow v \quad is\_true(v,\tau_v) \quad p,g,l,f \vdash stmt,\mu \quad \Rightarrow Normal,\mu_1 \\ p,g,l,f \vdash \text{while}(e)\ stmt,\mu_1 \Rightarrow out,\mu_2\end{array}}{p,g,l,f \vdash \text{while}(e)\ stmt,\mu \Rightarrow out,\mu_2} \tag{S-14}$$

$$\frac{p,g,l \vdash e,\mu \Rightarrow v \quad is\_true(v,\tau_v) \quad p,g,l,f \vdash stmt,\mu \Rightarrow out,\mu_1 \quad out = Return|Fail}{p,g,l,f \vdash \text{while}(e)\ stmt,\mu \Rightarrow out,\mu_1} \tag{S-15}$$

while 循环语句其实质与 for 循环类似, 只是缺少了初始化语句. 因此给出 3 个不同的规则, 根据终止条件的评估结果使用相应的规则. 当循环条件 e 的求值规则 v 为 false 时, while 循环结束, 参考规则 (S-13). 当循环条件 e 的求值规则 v 为 true 时, 执行循环内的语句 *stmt* , 并产生语句执行结果, 当 *stmt* 正常执行结束, 得到新的状态 μ_2 , 并继续执行循环语句, 得到最终状态 μ_3 , 参考规则 (S-14). 当 *stmt* 为 return 语句或引发异常时, 循环立即终止, 状态不再发生改变, 参考规则 (S-15).

函数调用语句可分为两类: 内部调用和外部调用. 内部调用是指同一合约函数之间的调用, 外部调用是指不同合约函数之间的调用.

此处需引入新的语义元素:

- $\mathbb{C}$: 区块链中定义的所有智能合约标识符的集合.
- $\mathbb{G}$: 从智能合约标识符映射到各自全局变量的函数.
- $fundef(g,id_c,id_f)$ 函数有 3 个参数: 合约名称和需要调用的函数名称, 若函数 id_f 存在, 则 *fundef* 函数返回其函数定义, 记为 $func$, 否则返回 $\varnothing$.

内部函数调用的操作语义:

$$\frac{\begin{array}{c}p,g,l \vdash e^*,\mu \Rightarrow vs,\mu_1 \quad fundef(g,id_c,id_f) = func \text{ and } func \neq \varnothing \\ p,g,l,f \vdash func(vs),\mu_1 \Rightarrow v_{re},\mu_2\end{array}}{p,g,l,f \vdash id_f(e^*),\mu \Rightarrow v_{re},\mu_2} \tag{S-16}$$

$$\frac{\begin{array}{c}p,g,l \vdash e^*,\mu \Rightarrow vs,\mu_1 \quad fundef(g,id_c,id_f) = \varnothing \\ p,g,l,f \vdash id_f(vs),\mu_1 \Rightarrow Fail,\mu_2\end{array}}{p,g,l,f \vdash id_f(e^*),\mu \Rightarrow Fail,\mu_2} \tag{S-17}$$

内部函数调用时, 首先对函数的参数列表进行求值, 之后检查函数在该智能合约中是否存在定义, 若存在, 执行函数代码, 函数分为有返回值和无返回值两类, 在此不过多赘述. 若函数在该合约中不存在定义, 则执行结果为 $Fail$.

外部函数调用的形式为: $e_1.e_2(e_3^*)$, 其中 e_1 表示外部合约, e_2 表示需调用的外部函数. 首先需要从区块链中获取外部合约的信息, 判断需调用的函数是否存在, 若存在, 则执行函数, 否则调用失败, 此时 Solidity 会自动调用其回退函数 fallback, 回退函数涉及到区块链操作, 在 MSVL 中没有相对应的描述, 因此这里使用 $Fail$ 表示外部函数调用失败.

外部函数调用的操作语义:

$$\frac{\begin{array}{c}p,g,l \vdash e_1,\mu \Rightarrow id_c,\mu_1 \quad p,g,l \vdash e_2,\mu_1 \Rightarrow id_f,\mu_2 \quad p,g,l \vdash e_3^*,\mu_2 \Rightarrow vs,\mu_3 \\ id_c \notin C \quad p,g,l,f \vdash id_c.id_f(vs),\mu_3 \Rightarrow Fail,\mu_4\end{array}}{p,g,l,f \vdash e_1.e_2(e_3^*),\mu \Rightarrow Fail,\mu_4} \tag{S-18}$$

$$\frac{\begin{array}{c}p,g,l \vdash e_1,\mu \Rightarrow id_c,\mu_1 \quad p,g,l \vdash e_2,\mu_1 \Rightarrow id_f,\mu_2 \quad p,g,l \vdash e_3^*,\mu_2 \Rightarrow vs,\mu_3 \\ id_c \in \mathbb{C} \quad \mathbb{G}(id_c) = g_c \quad fundef(g_c,id_c,id_f) = func \text{ and } func \neq \varnothing \\ p,g,l,f \vdash id_c.func(vs),\mu_3 \Rightarrow v_{re},\mu_4\end{array}}{p,g,l,f \vdash e_1.e_2(e_3^*),\mu \Rightarrow v_{re},\mu_4} \tag{S-19}$$

$$\frac{\begin{array}{c}p,g,l\vdash e_1,\mu\Rightarrow id_c,\mu_1\quad p,g,l\vdash e_2,\mu_1\Rightarrow id_f,\mu_2\quad p,g,l\vdash e_3^*,\mu_2\Rightarrow vs,\mu_3\\ id_c\in\mathbb{C}\quad \mathbb{G}(id_c)=g_c\quad fundef(g_c,id_c,id_f)=\varnothing\\ p,g,l,f\vdash id_c.id_f(vs),\mu_3\Rightarrow Fail,\mu_4\end{array}}{p,g,l,f\vdash e_1.e_2(e_3^*),\mu\Rightarrow Fail,\mu_4}\tag{S-20}$$

其中, 使用 $\mathbb{C}$ 集合判断外部调用的智能合约是否存在于区块链之上, 若不存在, 调用失败, 参考规则 (S-18), 若存在, 通过 $\mathbb{G}$ 函数获取该合约的全局环境. 之后通过 *fundef* 函数判断外部调用的函数是否被定义, 若已定义, 则获取其函数定义, 执行函数并得到其返回值, 参考规则 (S-19). 若不存在其函数定义, 则调用失败, 执行结果为 $Fail$, 参考规则 (S-20).

当内部函数或外部函数成功获取到函数定义时, $func(vs)$ 为函数定义, v_{re} 为函数 $func$ 的返回值, 函数内部语句执行结果 *out* 、返回值类型与返回值之间的关系如下:

$$Normal|Return,void\ \#\ \varnothing\quad Return\ v|Return\ vs,\tau\ \#\ v_{re}\text{当}\tau\neq void\text{时}$$

对于内部函数调用 $func(vs)$ 有:

$$\frac{\begin{array}{c}func=[\tau|void]\,id_f(par)\{stmt\}\quad alloc\_mem(M,lpar)=(M_1,loc)\\ store\_mem(M_1,loc,l,par,vs)=M_2\quad p,g,l,f\vdash stmt,\mu_2\Rightarrow out,\mu_3\quad out,\tau\ \#\ v_{re}\end{array}}{p,g,l,f\vdash func(vs),\mu\Rightarrow v_{re},free\_mem(\mu_3,loc)}\tag{S-21}$$

其中, par 表示函数的形参, $alloc\_mem(M,l,par)$ 为形参分配其在内存 memory 中所需的内存, 并返回新的内存状态 M_1 和分配的内存位置 loc. 函数 $store\_mem(M_1,loc,l,par,vs)$ 使用传入的参数值 vs 为形参 par 进行赋值. $free\_mem(\mu_3,loc)$ 表示在退出函数时, 当前 Solidity 状态为 μ_3, 且在退出时会释放当前函数为保存形参所分配的内存.

对于外部函数调用 $id_c.func(vs)$ 有:

$$\frac{func=extern\,[\tau|void]\,id_f(par)\{stmt\}\quad v_{re}=id_f(par)}{p,g,l,f\vdash id_c.func(vs),\mu\Rightarrow v_{re},\mu}\tag{S-22}$$

其中, $v_{re}=id_f(par)$ 获取外部函数调用 $id_f(par)$ 返回的结果, 外部函数执行时并不会对当前合约的状态造成影响, 因此外部函数调用执行前后状态 μ 未发生改变.

3 Solidity 与 MSVL 操作语义等价性证明

SOL2M 转换器[12,13]的功能是将 Solidity 程序转换为功能等价的 MSVL 程序, 为保证转换前后的程序在操作语义上等价, 本文定义了 Solidity 语言子集的操作语义, 并在本节中给出 Solidity 子集操作语义与 MSVL 操作语义之间的等价性证明.

3.1 Solidity 到 MSVL 的转换规则

在过去的研究中, 通过 JavaCC 对 Solidity 进行了词法和语法分析, 实现了 Solidity 到 MSVL 的自动转换工具 SOL2M 转换器, 完成了对 Solidity 程序的自动化建模, 并通过实例进行了 Solidity 程序的形式化验证, 说明了方法的可行性. 本节主要介绍 SOL2M 转换器的转换规则, 为下文的等价性证明奠定基础.

表 9 为 Solidity 到 MVSL 的基本类型转换规则, 将 Solildity 类型 τ_s 与 MSVL 类型 τ_m 一一对应, 是后续内存注入函数 α 定义及 α 等价性证明的基础. 其中使用形式化方法的抽象解释, 简化了字节和地址类型的变量声明, 其长度限定为 1, 因此均转换为 MSVL 的 char 类型. 此外, 由于 MSVL 中没有映射类型, 因此使用结构体对其进行抽象描述, 其中一个成员变量对应映射类型中的键, 另一个对应映射类型中的值. 每一组 mapping 键值对在 MSVL 中都有一个相对应的结构体实例表示, 整个 mapping 类型由结构体数组抽象表示.表 10 为 Solidity 到 MSVL 的核心转换规则, block_s 表示 Solidity 程序块, 转换后的 MSVL 程序块使用 block_m 表示.

表 9　Solidity 到 MSVL 的基本类型转换规则

Solidity数据类型	MSVL数据类型
int, int8, int16, …, int256	int
uint, uint8, uint16, …, uint256	int

表 9 Solidity 到 MSVL 的基本类型转换规则 (续)

Solidity数据类型	MSVL数据类型
bytes, bytes1, bytes2, …, bytes32	char
bool	bool
address	char

表 10 Solidity 到 MSVL 的核心转换规则

类型	Solidity程序	MSVL程序
声明语句	uint public a=0; uint[] public arr;	int a and a<== 0 and skip; int arr[MAX] and skip;

结构体	struct id{
	$block_s$
	}
		struct id{
		$block_m$
		}

函数	function vote(uint id) public{
	uint a= id;
	}
		function vote (int id){
		frame(a) and (
		int a and a<== id and skip
		)
		};

转向语句	throw; $block_s$;	skip;

	function fun() {
	return 0;
	}
		function fun(int return_value) {
		frame(return_flag)(
		int return_flag <== 0 and skip;
		return_flag := 1 and return_value:= 0)
		};

条件语句	if(expression){
	$block_s$
	}else{
	$block_s$
	}
		if(expression) then {
		$block_m$
		}else{
		$block_m$
		}

循环语句	for(statedef;e1;e2){
	$block_s$
	}
		while(e1) {
		$block_m$
		e2;
		}

含算术运算符的表达式	e1[+, -, *, /, %]e2 e1[++, --]	e1[+, -, *, /, %]e2 e1:= e1[+, -]1
含赋值运算符的表达式	e1=e2 e1[+, -, *, /, %]=e2	e1:= e2 e1:= e1[+, -, *, /, %]e2
含逻辑运算符的表达式	e1&&e2 e1‖e2	e1 AND e2 e1 OR e2

3.2 引理定义

本文在第 2 节定义了 Solidity 状态 $\mu =< a,\sigma,M >$, 其中网络状态 $\sigma =< b,p,S >$. S 表示 storage 存储状态, storage 用于保存合约的状态变量, M 表示 memory 内存状态, memory 用于保存合约函数内的局部变量. 本文重点关注 Solidity 语言本身的语法操作, 因此可以认为 Solidity 状态 μ 的改变仅由其存储状态 S 和内存状态 M 决定, 而 MSVL 中无论是状态变量还是局部变量, 均存放在同一内存中, 没有相应的结构可以区分 storage 与 memory. 因此本文从语义逻辑的角度出发, 将 MSVL 内存分为两部分, 记为 $mstorage$ 与 $mmemory$, 分别对应 Solidity 中的 storage 与 memory. 这两部分对应的 MSVL 内存状态分别记为 $s[s]$ 与 $s[m]$, 当其分别与 Solidity 的存储状态 S 和内存状态 M 等价时, 认为 Solidity 状态 μ 与 MSVL 的内存状态 s 等价时, 即状态等价. 在本节证明过程不对其分别证明.

本文使用函数 α 表示内存注入[24], 定义为一个从 Solidity 的 storage 地址 $addr$ 或者 memory 地址 loc 到 MSVL 的存储位置 (bl,δ_m) 的单射函数, 表明 Solidity 程序中的 $addr$ 或者 loc 对应 MSVL 程序中的块索引 bl 和偏移 δ_m.

基于内存注入函数 α, Solidity 中的值 v_s 和 MSVL 中的值 v_m 的等价关系记为 $\alpha \vdash v_s \sim v_m$, 定义如下:

- V1 $\alpha \vdash c \sim c$, 其中 $v_s = v_m = c$.
- V2 $\alpha \vdash ptr(addr,i_s)\ ptr(bl,i_m)$, 其中 $v_s = ptr(addr,i_s)$, $v_m = ptr(bl,i_m)$, 当且仅当存在 $\delta_m \in N_0$ 满足 $\alpha(addr) = (bl,\delta_m)$ 且 $i_s/getSize(g,\tau_s) = (i_m-\delta_m)/sizeof(\tau_m)$.
- V3 $\alpha \vdash map(v_{s1}) \sim map(v_{m1})$, 当且仅当 $v_{s1} = v_{m1}$ 且 $map(v_{s1}) = map(v_{m1})$, 其中 map 为描述映射关系的函数, 在 Solidity 中对于任意映射类型的变量 mp 有 $map(v_{s1}) = v_s = mp[v_{s1}]$, 在 MSVL 中对于与 mp 相应的结构体数组 ma 有 $map(v_{m1}) = v_m = ma[k].right$ 且 $ma[k].left = v_{m1}$.

其中, V1 表示 Solidity 与 MSVL 中常量 c 的等价. V2 表示 Solidity 的地址 i_s 与 MSVL 的地址 $ptr(bl,i_m)$ 等价. V3 描述了 Solidity 与 MSVL 映射类型的等价.

引理 1. 对于给定的 α, 任意的 $addr,bl,i_m,j \in \mathbb{Z}$, 如果 $\alpha \vdash ptr(addr,i_s) \sim ptr(bl,i_m)$, 那么 $\alpha \vdash ptr(addr,i_s + j*sizeof(\tau_s)) \sim ptr(bl,i_m + j*sizeof(\tau_m))$.

证明:

(1) $\alpha \vdash ptr(addr,i_s) \sim ptr(bl,i_m)$ 已知

(2) $\Rightarrow \alpha(addr) = (bl,\delta_m) \wedge i_s/sizeof(\tau_s) = (i_m-\delta_m)/sizeof(\tau_m)$ V2

(3) $\Rightarrow \alpha(addr) = (bl,\delta_m) \wedge i_s/sizeof(\tau_s) + j = (i_m-\delta_m)/sizeof(\tau_m) + j$ (2)

(4) $\Rightarrow \alpha(addr) = (bl,\delta_m) \wedge (i_s + j*sizeof(\tau_s))/sizeof(\tau_s) = (i_m + j*sizeof(\tau_m) - \delta_m)/sizeof(\tau_m)$ (3)

(5) $\Rightarrow \alpha \vdash ptr(addr,i_s + j*sizeof(\tau_s)) \sim ptr(bl,i_m + j*sizeof(\tau_m))$ V2

值得注意的是, $\text{addr} + i_s + j*sizeof(\tau_s)$ 与 $bl + i_m + j*sizeof(\tau_m)$ 上限均为计算机内存大小.

定义 1 (状态等价). 对于给定的内存注入 α, Solidity 与 MSVL 的内存状态等价, 记为 $\alpha \vdash \mu \sim s$, 当且仅当以下条件成立:

对 Solidity 程序中任意的状态变量 $x_s \in Dom(g)$, MSVL 程序中对于变量 $x_m \in Dom(s)$, 且 $addr,bl,i_m \in \mathbb{Z}$, 以及 $v_s,v_m \in \mathbb{D}$. 如果在 Solidity 程序中有 $p,g,l \vdash x_s,\mu \overset{l}{\Rightarrow} addr$ 且 $p,g,l \vdash x_s,\mu \Rightarrow v_s$. 在 MSVL 程序中有 $s^l(x_m) = (bl,i_m)$ 且 $s^r(x_m) = v_m$, 那么有 $\alpha \vdash addr \sim ptr(bl,i_m)$ 且 $\alpha \vdash v_s \sim v_m$. 局部变量分析过程与全局类似不再赘述..

定义 1 描述了 Solidity 程序与 MSVL 程序之间的状态等价关系, 其实质是, 当 Solidity 程序中的任意变量与 MSVL 程序中相对应变量的存储位置和值都等价, 则 Solidity 程序和 MSVL 程序状态等价.

定义 2 (左值表达式等价). 对于给定的内存注入 α, Solidity 与 MSVL 的左值表达式 e 与 a 等价, 记为 $\alpha \vdash e \sim_l a$, 当且仅当以下条件成立:

对于任意的 $\mu,s,addr,bl,i_m,\sigma$, 如果 $\alpha \vdash \mu \sim s$, 并且在 Solidity 程序中有 $p,g,l \vdash e,\mu \overset{l}{\Rightarrow} addr$, 在 MSVL 程序中有 $(a,\sigma,s,|\sigma|+1) \overset{l}{\Rightarrow} (bl,i_m)$, 那么有 $\alpha \vdash addr \sim ptr(bl,i_m)$.

定义 2 描述了 Solidity 与 MSVL 的左值表达式的等价定义. 其实质是对于 Solidity 中任意的左值表达式 e 和 MSVL 中相对应的左值表达式 a, 其所表示的存储位置等价, 则说明表达式表示相同的值, 因此左值表达式等价.

定义 3 (右值表达式等价). 对于给定的内存注入 α, Solidity 与 MSVL 的右值表达式 e 与 a 等价, 记为 $\alpha \vdash e \sim_r a$, 当且仅当以下条件成立:

对于任意的 μ,s,v_s,v_m,σ, 如果 $\alpha \vdash \mu \sim s$, 并且在 Solidity 程序中有 $p,g,l \vdash e,\mu \Rightarrow v_s$, 在 MSVL 程序中有 $(a,\sigma,s,|\sigma|+1) \Downarrow v_m$, 那么有 $\alpha \vdash v_s \sim v_m$.

定义 3 描述了 Solidity 与 MSVL 的右值表达式的等价定义. 对于 Solidity 中任意的右值表达式 e 和 MSVL 中相对应的右值表达式 a, 其所表示的值等价, 则说明右值表达式等价. 右值表达式等价实际上就是状态等价条件下的表达式等价.

定义 4 (表达式等价). 对于给定的内存注入 α, Solidity 与 MSVL 的表达式 e 与 a 等价, 记为 $\alpha \vdash e \sim_e a$, 当且仅当 e 和 a 均为左值表达式时, 有 $\alpha \vdash e \sim_l a$, 或者, 当 e 和 a 均为右值表达式时, 有 $\alpha \vdash e \sim_r a$.

定义 5 (Solidity 语句等价). 从状态 $u =< a, \sigma, M >$ (其中 $\sigma =< b, p, S >$) 开始执行语句 $stmt$ 等价于执行 $stmt'$, 记为 $(stmt, \mu) \cong (stmt', \mu')$, 当且仅当从 Solidity 状态 u 出发执行 $stmt$ 和 $stmt'$ 分别得到两个区间 $\mu_\sigma = (\mu_1, \mu_2, \ldots)$ 和 $\mu'_\sigma = (\mu'_1, \mu'_2, \ldots)$, 且这两个区间满足, 对所有的 $i \geqslant 0$ 有 $\mu_i = \mu'_i$.

定义 6 (语句等价). 对于给定的内存注入 α, Solidity 语句 $stmt$ 与 MSVL 语句 ms 等价, 记为 $\alpha \vdash stmt\ ms$, 当且仅当以下条件成立:

对于任意的 μ, s_i, μ', out, 如果有 $\alpha \vdash \mu \sim s_i$, 并且在 Solidity 中有 $p, g, l, f \vdash stmt, \mu \Rightarrow out, \mu'$, 那么 MSVL 中一定存在某个 σ 满足 $(ms, \sigma_{i-1}, s_i, i) \xrightarrow{*} (true, \sigma, \varnothing, |\sigma|+1)$, 且 $\alpha \vdash \mu' \sim s_{|\sigma|}$.

定义 6 描述了 Solidity 语句与 MSVL 语句的等价定义, 其含义为如果 Solidity 语句与 MSVL 语句在执行前的状态等价, 那么执行后的状态也等价.

3.3 表达式等价性证明

定理 1. 通过 SOL2M 转换器, 将 Solidity 表达式 e 转换为 MSVL 表达式 a, 记为 $a = TranExp(e)$. 对于给定的内存注入 α, 任意的 $\mu \in S, s \in M$, 如果 $\alpha \vdash \mu \sim s$, 则 $\alpha \vdash e \sim_e a$.

证明: 对 e 的结构使用结构归纳法对定理进行证明. 归纳奠基:

1) 对于常数 c, 显然成立.

2) 对于变量 id_s, $id_m = TranExp(id_s)$, 其中 id_s 和 id_m 均为左值表达式.

(1) $\alpha \vdash \mu \sim s$ 已知条件

(2) $\Rightarrow$ $\forall \delta_m, addr, i_m, \mu. \left(p, g, l \vdash id_s, \mu \overset{l}{\Rightarrow} addr\right) \wedge s^l(id_m) = (bl, \delta_m) \rightarrow$

$\alpha \vdash load(S, addr, sizeof(\tau_s)) \sim ptr(bl, i_m)$ E-1, 定义1, (1)

(3) $(id_m, \sigma, s, |\sigma|+1) \overset{l}{\Rightarrow} s^l(id_m)$ L1

(4) $\Rightarrow$ $\forall \delta_m, addr, i_m, \mu. \left(p, g, l \vdash id_s, \mu \overset{l}{\Rightarrow} addr\right) \wedge (id_m, \sigma, s, |\sigma|+1) \overset{l}{\Rightarrow} (bl, \delta_m)$

$\rightarrow \alpha \vdash load(S, addr, sizeof(\tau_s)) \sim ptr(bl, i_m)$ (1, 2, 3)

(5) $\Leftrightarrow$ $\alpha \vdash id_s \sim_l id_m$ 定义2, (4)

(6) $\Rightarrow$ $\alpha \vdash id_s \sim_e id_m$ 定义4, (5)

3) 对于类型为 τ 的一维数组元素, Solidity 中表示为 $le[e]$, 其中 le 为 id, MSVL 中表示为 $id_m[ra]$, 其中 $id_m[ra] = TranExp(le[e])$, $ra = TranExp(e)$, id_m、le、$id_m[ra]$ 和 $le[e]$ 均为左值表达式, e 和 ra 均为右值表达式. E-2 为反向使用, 因为数组元素的位置具有唯一性, 由首地址与偏移量决定, 所以可进行逆向推理.

(1) $\alpha \vdash le \sim_e id_m \wedge \alpha \vdash e \sim_e ra$ 归纳假设

(2) $\alpha \vdash \mu \sim s$ 已知条件

(3) $p, g, l \vdash le[e], \mu \overset{l}{\Rightarrow} addr + i_s$ 假设

(4) $\Rightarrow$ $p, g, l \vdash le, \mu \overset{l}{\Rightarrow} addr \wedge p, g, l \vdash e, \mu \Rightarrow v_s \wedge (i_s = v_s \cdot sizeof(\tau_{s_{le}}))$ E-2, (3)

(5) $(id_m[ra], \sigma, s, |\sigma|+1) \overset{l}{\Rightarrow} (bl, i_m)$ 假设

(6) $\Rightarrow$ $(id_m, \sigma, s, |\sigma|+1) \overset{l}{\Rightarrow} (bl, 0) \wedge (ra, \sigma, s, |\sigma|+1) \Downarrow v_m \wedge i_m = v_m \cdot sizeof(\tau_m)$ L2, (5)

(7) $(id_m,\sigma,s,|\sigma|+1)\overset{l}{\Rightarrow}(bl,0)\wedge p,g,l\vdash le,\mu\overset{l}{\Rightarrow}addr\wedge(ra,\sigma,s,|\sigma|+1)\Downarrow v_m\wedge p,g,l\vdash e,\mu\Rightarrow v_s$ (4, 6)

(8) $\Rightarrow\ \alpha\vdash load(S,addr,sizeof(\tau_s))\sim ptr(bl,0)\wedge\alpha\vdash v_s\sim v_m$ 定 义2, 3, 4, (1, 2, 7)

(9) $\Rightarrow\ \alpha\vdash load(S,addr,sizeof(\tau_s))\sim ptr(bl,0)\wedge v_s=v_m$ v_s与v_m均为整型数值, V1, (8)

(10) $\Rightarrow\ \alpha\vdash load(S,addr+v_s\cdot sizeof(\tau_s),sizeof(\tau_s))\sim ptr(bl,v_m\cdot sizeof(\tau_m))$ 引理1, (9)

(11) $\Rightarrow\ \alpha\vdash load(S,addr+i_s,sizeof(\tau_s))\sim ptr(bl,i_m)$ (10)

(12) $\Leftrightarrow\ \alpha\vdash le[e]\sim_l id_m[ra]$ 定义2, E-2, R5, 假设, (11)

(13) $\Leftrightarrow\ \alpha\vdash le[e]\sim_e id_m[ra]$ 定义4, (12)

二维数组同理可证, Solidity 中表示为 $le[e_1]$, 其中 le 为 $id[e_2]$.

4) 对于结构体成员变量, Solidity 中表示为 $le.id_s$, MSVL 中表示为$la.id_m$, 且$la.id_m=TranExp(le.id_s)$, $la=TranExp(le)$, le、la、$le.id_s$ 和 $la.id_m$ 为左值表达式, id_m 和 id_s 均为右值表达式. Solidity 程序中结构体 $Struct_s$ 成员列表 λ 中的第 k 个成员, 转换为 MSVL 程序中结构体 $Struct_m$ 成员列表 λ 中的第 k 个成员. 假设结构体成员列表中第 i 个成员的类型为 τ_i.$field\_offset(id,\varphi)$ 返回结构体成员列表 φ 中成员变量 id 在内存中的偏移量, 为了简化书写, 用 δ 表示 $\delta=field\_offset(id,\varphi)=sizeof(\tau_1)+\ldots+sizeof(\tau_{k-1})$. E-4 为反向使用, 因为结构体成员变量的位置具有唯一性, 由首地址与偏移量决定, 所以可进行逆向推理.

(1) $\alpha\vdash le\sim_e la\wedge\alpha\vdash id_s\sim_e id_m$ 归纳假设

(2) $\alpha\vdash\mu\sim s$ 已知条件

(3) $p,g,l\vdash le.id_s,\mu\overset{l}{\Rightarrow}addr+i_s$ 定义2, 假设

(4) $\Rightarrow\ p,g,l\vdash le,\mu\overset{l}{\Rightarrow}addr\wedge i_s=\delta$ E-4, (3)

(5) $(la.id_m,\sigma,s,|\sigma|+1)\overset{l}{\Rightarrow}(bl,i_m)$ 假设

(6) $\Rightarrow\ (la,\sigma,s,|\sigma|+1)\overset{l}{\Rightarrow}(bl,j_m)\wedge(i_m=j_m+\delta)$ L4, (5)

(7) $\Rightarrow\ \alpha\vdash load(S,addr,sizeof(\tau_s))\sim ptr(bl,j_m)$ 定义2, 4, (1, 2, 4, 6)

(8) $\Rightarrow\ \alpha\vdash load(S,addr+\delta,sizeof(\tau_s))\sim ptr(bl,j_m+\delta)$ 引理1, (4, 6, 7)

(9) $\Rightarrow\ \alpha\vdash load(S,i_s,sizeof(\tau_s))\sim ptr(bl,i_m)$ (8)

(10) $\Leftrightarrow\alpha\vdash le.id_s\sim_l la.id_m$ 定义2, (9)

(11) $\Rightarrow\ \alpha\vdash le.id_s\sim_e la.id_m$ 定义4, (10)

5) 对于类型为 τ 的一维数组元素, Solidity 中表示为$le[e]$, 其中le 为id, MSVL 中表示为$id_m[ra]$, 其中 $id_m[ra]=TranExp(le[e])$, $ra=TranExp(e)$, id_m 和 le 均为左值表达式, $id_m[ra]$、$le[e]$、e 和 ra 均为右值表达式. E-2 与 E-6 为反向使用, 因为数组元素的位置具有唯一性, 由首地址与偏移量决定, 所以可进行逆向推理.

(1) $\alpha\vdash le[e]\sim_l id_m[ra]\wedge\alpha\vdash e\sim_e ra$ 归纳假设

(2) $\alpha\vdash\mu\sim s$ 已知条件

(3) $p,g,l\vdash le[e],\mu\Rightarrow v_s$ 定义3, 假设

(4) $\Rightarrow\ p,g,l\vdash le[e],\mu\overset{l}{\Rightarrow}addr+i_s\wedge v_s=load(S,addr+i_s,sizeof(\tau_s))$ E-2, E-6, (3)

(5) $(id_m[ra],\sigma,s,|\sigma|+1)\Downarrow \mathrm{v_m}$ 假设

(6) $\Rightarrow\ (id_m,\sigma,s,|\sigma|+1)\overset{l}{\Rightarrow}(bl,i_m)\wedge v_m=ptr(bl,i_m)$ L2, R5, (5)

(7) $\Rightarrow\ \alpha\vdash load(S,addr+i_s,\ sizeof(\tau_s))\sim ptr(bl,i_m)$ 定义2, (1, 2, 4, 6)

(8) $\Leftrightarrow\alpha\vdash v_s\sim v_m$ 定义1, (4, 6, 7)

(9) $\Rightarrow\ \alpha\vdash le[e]\sim_r id_m[ra]$ 定义3, (8)

(10) $\Leftrightarrow \alpha \vdash le[e] \sim_e id_m[ra]$ 定义4, (9)

二维数组同理可证.

6) 对于结构体成员变量, Solidity 中表示为 $le.id_s$, MSVL 中表示为 $la.id_m$, 且 $la.id_m = TranExp(le.id_s)$, $la = TranExp(le)$, le 和 la 为左值表达式, $le.id_s$、$la.id_m$、id_m 和 id_s 均为右值表达式.E-4 与 E-8 为反向使用, 因为结构体成员变量的位置具有唯一性, 由首地址与偏移量决定, 所以可进行逆向推理.

(1) $\alpha \vdash le.id_s \sim_l la.id_m \wedge \alpha \vdash id_s \sim_e id_m$ 归纳假设

(2) $\alpha \vdash \mu \sim s$ 已知条件

(3) $p,g,l \vdash le.id_s, \mu \Rightarrow v_s$ 定义3, 假设

(4) $\Rightarrow (p,g,l \vdash le, \mu \overset{l}{\Rightarrow} addr + i_s) \wedge (v_s = load(S, addr + i_s, sizeof(\tau_s)))$ E-4, E-8, (3)

(5) $(la.id_m, \sigma, s, |\sigma|+1) \Downarrow v_m$ 假设

(6) $\Rightarrow$ $(la, \sigma, s, |\sigma|+1) \overset{l}{\Rightarrow} (bl, i_m) \wedge v_m = ptr(bl, i_m)$ L4, R7, (5)

(7) $\Rightarrow$ $\alpha \vdash load(S, addr + i_s, sizeof(\tau_s)) \sim ptr(bl, i_m)$ 定义2, (1, 2, 4, 6)

(8) $\Leftrightarrow \alpha \vdash v_s \sim v_m$ 定义1, (4, 6, 7)

(9) $\Rightarrow$ $\alpha \vdash le.id_s \sim_r la.id_m$ 定义3, 假设, (8)

(10) $\Rightarrow$ $\alpha \vdash le.id_s \sim_e la.id_m$ 定义4, (9)

7) 对于 Solidity 中除自增自减表达式外的其他单目运算 $unope$, MSVL 中有 $mop_1 ra = TranExp(unope)$, 其中 $ra = TranExp(e)$. E-10 为反向使用, 因为单目运算可逆向操作, 所以可进行逆向推理.

(1) $\alpha \vdash e \sim_e ra$ 归纳假设

(2) $\alpha \vdash \mu \sim s$ 已知条件

(3) $p,g,l \vdash unope, \mu \Rightarrow v_s \wedge (mop_1\ ra, \sigma, s, |\sigma|+1) \Downarrow v_m$ 假设

(4) $\Rightarrow$ $p,g,l \vdash e, \mu \Rightarrow v_{s1} \wedge v_s = unop\ v_{s1} \wedge (ra, \sigma, s, |\sigma|+1) \Downarrow v_{m1} \wedge v_m = mop_1\ v_{m1}$ E-10, R8, (3)

(5) $\Rightarrow$ $\alpha \vdash v_{s1} \sim v_{m1} \wedge v_s = unop\ v_{s1} \wedge v_m = mop_1\ v_{m1}$ 定义2, 3, 4, (1, 2, 4)

(6) $\Rightarrow$ $v_{s1} = v_{m1} \wedge v_s = unop\ v_{s1} \wedge v_m = mop_1\ v_{m1}$ (5)

(7) $\Rightarrow$ $v_s = v_m$ (6)

(8) $\Rightarrow$ $\alpha \vdash v_s \sim v_m$ V1, (7)

(9) $\Leftrightarrow \alpha \vdash unop\ e \sim_r mop_1\ ra$ 定义3, (8)

(10) $\Rightarrow$ $\alpha \vdash unop\ e \sim_e mop_1\ ra$ 定义4, (9)

8) 对于 Solidity 中的双目运算 $e_1\ binop\ e_2$, MSVL 中有 $ra_1\ mop_2\ ra_2 = TranExp(e_1\ binop\ e_2)$, 其中 $ra_1 = TranExp(e_1)$ 且 $ra_2 = TranExp(e_2)$. E-12 为反向使用, 因为双目运算可逆向操作, 所以可进行逆向推理.

(1) $\alpha \vdash e_1 \sim_e ra_1 \wedge \alpha \vdash e_2 \sim_e ra_2$ 归纳假设

(2) $\alpha \vdash \mu \sim s$ 已知条件

(3) $p,g,l \vdash e_1\ binop\ e_2, \mu \Rightarrow v_s \wedge (ra_1\ mop_2\ ra_2, \sigma, s, |\sigma|+1) \Downarrow v_m$ 假设

(4) $\Rightarrow$ $p,g,l \vdash e_1, \mu \Rightarrow v_{s1} \wedge p,g,l \vdash e_2, \mu \Rightarrow v_{s2} \wedge (ra_1, \sigma, s, |\sigma|+1) \Downarrow v_{m1} \wedge$
$(ra_2, \sigma, s, |\sigma|+1) \Downarrow v_{m2} \wedge v_s = v_{s1}\ binop\ v_{s2} \wedge v_m = v_{m1}\ mop_2\ v_{m2}$ E-12, R9, (3)

(5) $\Rightarrow$ $\alpha \vdash v_{s1} \sim v_{m1} \wedge \alpha \vdash v_{s2} \sim v_{m2} \wedge v_s = v_{s1}\ binop\ v_{s2} \wedge v_m = v_{m1}\ mop_2\ v_{m2}$ 定义2, 3, 4, (1, 2, 4)

(6) $\Rightarrow$ $v_{s1} = v_{m1} \wedge v_{s2} = v_{m2} \wedge v_s = v_{s1}\ binop\ v_{s2} \wedge v_m = v_{m1}\ mop_2\ v_{m2}$ (5)

(7) $\Rightarrow$ $v_s = v_m$ V1, (6)

(8) $\Rightarrow\ \alpha \vdash v_s \sim v_m$ (7)

(9) $\Leftrightarrow \alpha \vdash e_1\ binop\ e_2 \sim_r ra_1\ mop_2\ ra_2$ 定义3, (8)

(10) $\Rightarrow\ \alpha \vdash e_1\ binop\ e_2 \sim_e ra_1\ mop_2\ ra_2$ 定义4, (9)

9) 对于 Solidity 中的参数列表 $e^* = (e_1,\ldots,e_k)$, MSVL 中记为 $ra^* = (ra_1,\ldots,ra_k) = TranExp(e^*)$. 其中 e_i 与 $ra_i(1 \leqslant i \leqslant k)$ 均为右值表达式. E-13 为反向使用, 因为参数组成列表为可逆操作, 所以可进行逆向推理.

(1) $\alpha \vdash e_i \sim ra_i(1 \leqslant i \leqslant k)$ 归纳假设

(2) $\alpha \vdash \mu \sim s$ 已知条件

(3) $p,g,l \vdash (e_1,\ldots,e_k),\mu \Rightarrow v_s \wedge ((ra_1,\ldots,ra_k),\sigma,s,|\sigma|+1) \Downarrow v_m$ 假设

(4) $\Rightarrow\ p,g,l \vdash e_i,\mu \Rightarrow v_{si} \wedge v_s = (v_{s1},\ldots,v_{sk}) \wedge$
$(ra_i,\sigma,s,|\sigma|+1) \Downarrow v_{mi} \wedge v_m = (v_{m1},\ldots,v_{mk})(1 \leqslant i \leqslant k)$ E-13, R13, (3)

(5) $\Rightarrow\ v_{si} = v_{mi}(1 \leqslant i \leqslant k)$ (1, 2, 4)

(6) $\Rightarrow\ (v_{s1},\ldots,v_{sk}) = (v_{m1},\ldots,v_{mk})$ (5)

(7) $\Rightarrow\ \alpha \vdash (v_{s1},\ldots,v_{sk}) \sim (v_{m1},\ldots,v_{mk})$ V1, (6)

(8) $\Leftrightarrow \alpha \vdash (e_1,\ldots,e_k) \sim_r (ra_1,\ldots,ra_k)$ 定义3, (7)

(9) $\Rightarrow\ \alpha \vdash (e_1,\ldots,e_k) \sim_e (ra_1,\ldots,ra_k)$ 即 $\alpha \vdash e^* \sim_e ra^*$ 定义4, (8)

10) 对于映射类型 $le[e]$, MSVL 中没有映射类型, 因此使用结构体抽象描述键值对, 用结构体数组 $id[ra]$ 抽象描述映射类型, 满足 $id[ra].left$ 与 e 求右值相等, 并且有 $id[ra].right = TranExp(le[e])$. 其中, $le[e]$ 与 $id[ra].right$ 均为右值表达式. SOL2M 转换器为映射类型在 MSVL 中定义了函数 $mapping\_map$, 其功能是遍历数组找到成员变量 $left$ 值与参数 l 值相同的结构体, 即满足 $id[ra].left = l$, 且在 MSVL 中有 $l = TranExp(e)$. 其中 $le[e]$ 与 $id[ra].right$ 均为右值表达式. E-7 为反向使用, 因为数组元素的位置具有唯一性, 由首地址与偏移量决定, 所以可进行逆向推理.

(1) $\alpha \vdash e \sim_e l$ 归纳假设

(2) $\alpha \vdash \mu \sim s$ 已知条件

(3) $p,g,l \vdash le[e],\mu \Rightarrow v_s \wedge (id[ra].right,\sigma,s,|\sigma|+1) \Downarrow v_m$ 假设

(4) $\Rightarrow\ p,g,l \vdash e \Rightarrow v_{s1} \wedge v_s = map(v_{s1})$ E-7, (3)

(5) $\Rightarrow\ (id_m[ra].left,\sigma,s,|\sigma|+1) \Downarrow v_{m1} \wedge (l,\sigma,s,|\sigma|+1) \Downarrow v_{m2} \wedge v_m = map(v_{m1})$ L2, R7, (4)

(6) $\Rightarrow\ v_{s1} = v_{m2} \wedge v_{m1} = v_{m2}$ V1, (1, 2, 4, 5)

(7) $\Rightarrow\ v_{s1} = v_{m1}$ (6)

(8) $\Rightarrow\ \alpha \vdash map(v_{s1}) \sim map(v_{m1}) \wedge v_s = map(v_{s1}) \wedge v_m = map(v_{m1})$ V3, (4, 5, 7)

(9) $\Rightarrow\ \alpha \vdash v_s \sim v_m$ (8)

(10) $\Leftrightarrow \alpha \vdash le[e] \sim_r id[ra].right$ 定义3, (9)

(11) $\Rightarrow\ \alpha \vdash le[e] \sim_e id[ra].right$ 定义4, (10)

自增自减表达式和赋值表达式将在下一节中作为赋值语句给出赋值语句等价性证明.

3.4 语句等价性证明

定理 2. 通过 SOL2M 转换器, 将 Solidity 语句 $stmt$ 转换为 MSVL 语句 ms , 记为 $ms = TranStmt(stmt)$. 对于任意给定的内存注入 α , 任意的 $\mu \in S, s_i \in M$, 如果 $\alpha \vdash \mu \sim s_i$, 那么有 $\alpha \vdash stmt \sim_s ms$.

证明: 本文采用规则归纳法对定理进行证明, 设 P 为性质.

$$P(\text{stmt},\mu,\mu',\text{out}) \Leftrightarrow \left(\alpha \vdash \mu \sim s_i \Rightarrow (ms,\sigma_{i-1},s_i,i) \xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge \alpha \vdash \mu' \sim s_{|\sigma|}\right) \quad \text{(TER)}$$

施规则归纳于命令去证明

$$p,g,l,f \vdash stmt,\mu \Rightarrow \text{out},\mu' \Rightarrow P(stmt,\mu,\mu',out)$$

以下是具体证明过程. 归纳奠基:

1) 规则 (S-1) 处理空语句';', 结论显然成立.

2) 规则 (S-5) 和 (S-6) 处理条件语句"$\text{if}(e)\ stmt_1 \text{ else } stmt_2$ ".$TranStmt(\text{if}(e)\ stmt_1 \text{ else } stmt_2)$ = "$\text{if}(b)$ then $\{ms_1\}$ else $\{ms_2\}$" , 其中$b = TranExp(e)$, $ms_1 = TranStmt(stmt_1)$, $ms_2 = TranStmt(stmt_2)$.

对于规则 (S-5), 其中 b 求值为 $true$, 其证明如下:

(1) $\alpha \vdash e \sim_e b$ 定理1

(2) $\alpha \vdash \mu \sim s_i$ 已知条件

(3) $p,g,l \vdash e,\mu \Rightarrow \text{true} \wedge p,g,l,f \vdash stmt_1,\mu \Rightarrow out',\mu' \wedge P(stmt_1,\mu,\mu',out)$ 归纳假设

(4) $\Leftrightarrow$ $(ms_1,\sigma_{i-1},s_i,i) \xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge \alpha \vdash \mu' \sim s_{|\sigma|}$ TER, (2, 3)

(5) $\Rightarrow$ $(b,\sigma_{i-1},s_i,i) \Downarrow \text{true}$ 定义3, 4, (1, 2, 3)

(6) $\Rightarrow$ $(\text{if } (b) \text{ then } \{ms_1\} \text{ else } \{ms_2\},\sigma_{i-1},s_i,i)$

$\rightarrowtail ((b \wedge ms_1) \vee (\neg b \wedge ms_2),\sigma_{i-1},s_i,i)$ IF

$\rightarrowtail (ms_1,\sigma_{i-1},s_i,i)$ B4, T1, F1, F2, (5)

$\xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge (\alpha \vdash \mu' \sim s_{|\sigma|})$ (4)

(7) $\Leftrightarrow$ $P(\text{if}(e)\ stmt_1 \text{ else } stmt_2,\mu,\mu',out)$ TER, (6)

对于规则 (S-6), 其中 b 求值为 false, 其证明如下:

(1) $\alpha \vdash e \sim_e b$ 定理1

(2) $\alpha \vdash \mu \sim s_i$ 已知条件

(3) $p,g,l \vdash e,\mu \Rightarrow \text{false} \wedge p,g,l,f \vdash stmt_2,\mu \Rightarrow out,\mu' \wedge P(stmt_2,\mu,\mu',out)$ 归纳假设

(4) $\Leftrightarrow$ $(ms_2,\sigma_{i-1},s_i,i) \xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge \alpha \vdash \mu' \sim s_{|\sigma|}$ TER, (2, 3)

(5) $\Rightarrow$ $(b,\sigma_{i-1},s_i,i) \Downarrow \text{false}$ 定义3, 4, (1, 2, 3)

(6) $\Rightarrow$ $(\text{if}(b)\text{then}\{ms_1\}\text{else}\{ms_2\},\sigma_{i-1},s_i,i)$

$\rightarrowtail ((b \wedge ms_1) \vee (\neg b \wedge ms_2),\sigma_{i-1},s_i,i)$ IF

$\rightarrowtail (ms_2,\sigma_{i-1},s_i,i)$ B4, T1, F1, F2, (5)

$\xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge (\alpha \vdash \mu' \sim s_{|\sigma|})$ (4)

(7) $\Leftrightarrow$ $P(\text{if}(e)\ stmt_1 \text{ else } stmt_2,\mu,\mu',out)$ TER, (6)

3) 规则 (S-2)、(S-3) 与 (S-4) 处理 return 返回语句, 对于带有返回值的"return e"语句, SOL2M 转换器转换时引入变量 $RVal$ 存储返回值, 以及变量 $rflag$ 标志语句是否具有返回值.

4) 规则 (S-7) 与 (S-8) 是处理顺序语句"$stmt_1;stmt_2$ ", $TranStmt(stmt_1;stmt_2)$ = "$ms_1;ms_2$" . 其中ms_1 = $TranStmt(stmt_1)$, $ms_2 = TranStmt(stmt_2)$. 对于规则 (S-7), 其中 stmt_1 正常执行结束, 不存在 return 语句或是执行失败, 其证明如下:

(1) $\alpha \vdash \mu \sim s_i$ 已知条件

(2) $p,g,l,f \vdash stmt_1,\mu \Rightarrow Normal,\mu_1 \wedge P(stmt_1,\mu,\mu_1,Normal) \wedge$

$p,g,l,f \vdash stmt_2,\mu_1 \Rightarrow out,\mu_2 \wedge P(stmt_2,\mu_1,\mu_2,out)$ 归纳假设

(3) $\Rightarrow$ $(ms_1,\sigma_{i-1},s_i,i) \xrightarrow{*} (\text{true},\sigma_j,\varnothing,j+1) \wedge \alpha \vdash \mu_1 \sim s_j \wedge$

$(ms_2,\sigma_{j-1},s_j,j) \xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge \alpha \vdash \mu_2 \sim s_{|\sigma|}$ TER, (1, 2)

(4) $\Rightarrow$ $(ms_1;ms_2,\sigma_{i-1},s_i,i)$

$\xrightarrow{*} (empty;ms_2,\sigma_{j-1},s_j,j)$ (3)

$\rightarrowtail (ms_2,\sigma_{j-1},s_j,j)$ CHOP

$\xrightarrow{*} (\text{true},\sigma,\varnothing,|\sigma|+1) \wedge \alpha \vdash \mu_2 \sim s_{|\sigma|}$ (3)

(5) $\Leftrightarrow$ $P(stmt_1;stmt_2,\mu,\mu_2,out)$ TER, (4)

对于规则 (S-8), $stmt_1$ 为 return 语句或是执行失败时, 其证明同理可得.

5) 对于赋值语句“$le = e$”, 规则 (E-7) 给出了赋值表达式“$le = e$”的操作语义, 赋值语句的操作语义与其一致. $TranStmt(le = e) =$ “$la := ra$”, 其中 $la = TranExp(le)$, $ra = TranExp(e)$.

(1) $\alpha \vdash le \sim_e la$ 定理1

(2) $\alpha \vdash e \sim_e ra$ 定理1

(3) $\alpha \vdash \mu \sim s_i$ 已知条件

(4) $p,g,l \vdash le \overset{l}{\Rightarrow} addr \wedge (la,\sigma_{i-1},s_i,i) \overset{l}{\Rightarrow} (bl,j_m)$ 假设

(5) $\Rightarrow$ $\alpha \vdash load(S,addr,sizeof(\tau_s)) \sim ptr(bl,j_m)$ 定义2, 4, (1, 3, 4)

(6) $p,g,l \vdash e \Rightarrow v_s \wedge (ra,\sigma_{i-1},s_i,i) \Downarrow v_m$ 假设

(7) $\Rightarrow$ $\alpha \vdash v_s \sim v_m$ 定义3, 4, (2, 3, 6)

在 MSVL 程序中有:

(8) $s_i^l(x_m) = (bl,j_m)$ 假设

(9) $(la := ra,\sigma_{i-1},s_i,i)$

$\rightarrowtail (\bigcirc(x_m \Leftarrow v_m \wedge \text{empty}),\sigma_{i-1},s_i,i)$ UASS, (8)

$\rightarrow ((x_m \Leftarrow v_m \wedge \text{empty}),\sigma_i,s_{i+1},i+1)$ TR1

$\rightarrowtail (\text{empty},\sigma_i,(s_{i+1}^l,s_{i+1}^r[v_m/x_m]),i+1)$ MIN1

(10) $\Rightarrow$ $s_{i+1}^l(x_m) = (bl,j_m) \wedge s_{i+1}^r(x_m) = v_m$ (9)

在 Solidity 程序中有:

(11) $(\alpha \vdash \mu \sim s_i) \wedge s_i^l(x_m) = (bl,j_m) \wedge \alpha \vdash load(S,addr,sizeof(\tau_s)) \sim ptr(bl,j_m)$ (3, 5, 8)

(12) $\Rightarrow$ $p,g,l \vdash x_s,\mu \overset{l}{\Rightarrow} addr$ 定义1

(13) $store(S,addr,\tau,v_s) = S'$ 假设

(14) $\Rightarrow$ $p,g,l \vdash x_s,\mu' \overset{l}{\Rightarrow} addr \wedge load(S',addr,sizeof(\tau_s)) = v_s$ E-9, (12, 13)

(15) $\Rightarrow$ $p,g,l \vdash x_s,\mu' \overset{l}{\Rightarrow} addr \wedge p,g,l \vdash x_s,\mu' \Rightarrow v_s$ E-5, (14)

Solidity 与 MSVL 中的其他变量的位置和值均未发生改变, 因此

(16) $\Rightarrow$ $(la := ra,\sigma_{i-1},s_i,i) \xrightarrow{*} (\text{true},\sigma_{i+1},\varnothing,i+2)$ TR2, (9)

(17) $\Rightarrow$ $\alpha \vdash \mu' \sim s_{i+1}$ (3, 5, 7, 10, 15)

(18) $\Leftrightarrow$ $P(le = e;,\mu,\mu',Normal)$ TER, (16, 17)

6) 对于 Solidity 单目运算中的自增操作“$++e;$”, 规则 (E-11) 处理自增表达式, 语句同理, 在 Solidity 中有:

(1) $p,g,l \vdash e,\mu \overset{l}{\Rightarrow} addr \wedge p,g,l \vdash e,\mu \Rightarrow v$ E-1, E-9

(2) $store(S,addr,sizeof(\tau_s),v+1)=S_1$ 假设

(3) $\Rightarrow$ $p,g,l \vdash ++e;,\mu \Rightarrow Normal,\mu_1$ E-11, (1, 2)

(4) $\Rightarrow$ $p,g,l \vdash e,\mu_1 \overset{l}{\Rightarrow} addr \wedge load(S_1,addr,sizeof(\tau_s))=v+1$ E-5, (3)

(5) $\Rightarrow$ $\mu_\sigma=(\mu,\mu_1)$ 定义5, (4)

对于 Solidity 赋值运算“$e=e+1$”, 规则 (E-9) 处理赋值表达式, 语句同理, 在 Solidity 中有:

(6) $p,g,l \vdash e,\mu \overset{l}{\Rightarrow} addr \wedge p,g,l \vdash e,\mu \Rightarrow v$ E-1, E-9

(7) $store(S,addr,sizeof(\tau_s),v+1)=S_1'$ 假设

(8) $\Rightarrow$ $p,g,l \vdash e=e+1;,\mu \Rightarrow Normal,\mu_1'$ E-9, (6, 7)

(9) $\Rightarrow$ $p,g,l \vdash e,\mu_1' \overset{l}{\Rightarrow} addr \wedge load(S_1',addr,sizeof(\tau_s))=v+1$ E-5, (8)

(10) $\Rightarrow$ $\mu_\sigma'=(\mu,\mu_1')$ 定义5, (9)

自增语句和赋值语句操作均从 Solidity 状态 μ 出发, 得到 μ_1 与 μ_1', Solidity 中只有变量 e 的值发生改变, 其他变量的位置和值均未改变, 因此有 $\mu_1=\mu_1'$.

(11) $\mu_l\sigma\big|=(\mu,\mu_1)\wedge\mu_l\sigma\big|'=(\mu,\mu_1')\wedge\mu_1=\mu_1'$ (5, 10)

(12) $\Rightarrow$ $(++e;,\mu)\cong(e=e+1;,\mu)$ 定义5, (11)

(13) $P(e=e+1;,\mu,\mu',Normal)$ 步骤5中已证明

(14) $\Rightarrow$ $P(++e;,\mu,\mu',Normal)$ (12, 13)

7) 规则 (S-13) 处理 while 循环语句“$\text{while}(e)\,stmt$”, 语句 $stmt$ 正常执行结束, 不存在 return 语句或是执行失败, 经过 SOL2M 转换器, MSVL 中有“$\text{while}(b)\{ms\}$”, 其中 $b=TranExp(e)$, $ms=TranExp(stmt)$.

(1) $\alpha \vdash e\sim_e b$ 定理1

(2) $\alpha \vdash \mu \sim s_i$ 已知条件

(3) $p,g,l \vdash e,\mu \Rightarrow false$ 假设

(4) $\Rightarrow$ $(b,\sigma_{i-1},s_i,i)\Downarrow false$ 定义3, 4, (1, 2, 3)

(5) $\Rightarrow$ $((\text{while}(b)\{ms\}),\sigma_{i-1},s_i,i)$

$\rightarrowtail (\text{if}(b)\text{then}\{ms\wedge\text{more};\text{while}(b)\{ms\}\}\text{else}\{\text{empty}\},\sigma_{i-1},s_i,i)$ WHL

$\rightarrowtail ((b\wedge(ms\wedge\text{more};\text{while}(b)\{ms\}))\vee(\neg b\wedge\text{empty}),\sigma_{i-1},s_i,i)$ IF

$\rightarrowtail (\text{empty},\sigma_{i-1},s_i,i)$ B4, F1, T1, F2, (4)

$\rightarrow (\text{true},\sigma_i,\varnothing,i+1)\wedge(\alpha \vdash \mu \sim s_i)$ TR2, (2)

(6) $\Leftrightarrow$ $P(\text{while}(e)\,stmt,\mu,\mu',Normal)$ TER, (5)

8) 规则 (S-14) 同样处理 while 循环语句“$\text{while}(e)\,stmt$”, 语句 $stmt$ 正常执行结束, 经过 SOL2M 转换器, MSVL 中有“$\text{while}(b)\{ms\}$”, 其中 $b=TranExp(e)$, $ms=TranExp(stmt)$.

(1) $\alpha \vdash e\sim_e b$ 定理1

(2) $\alpha \vdash \mu \sim s_i$ 已知条件

(3) $p,g,l \vdash e,\mu \Rightarrow \text{true}\wedge p,g,l,f \vdash stmt,\mu \Rightarrow Normal,\mu_1 \wedge P(stmt,\mu,\mu_1,Normal)\wedge$

$p,g,l,f \vdash \text{while}(e)\,stmt,\mu_1 \Rightarrow out,\mu_2 \wedge P(\text{while}(e)\,stmt,\mu_1,\mu_2,out)$ 假设

(4) $\Leftrightarrow$ $((ms,\sigma_{i-1},s_i,i)\overset{*}{\rightarrow}(true,\sigma_j,\varnothing,j+1)\wedge\alpha \vdash \mu_1 \sim s_j)\wedge$

$(\text{while}(b)\{ms\},\sigma_{j-1},s_j,j)\overset{*}{\rightarrow}(true,\sigma,\varnothing,|\sigma|+1)\wedge\alpha \vdash \mu_2 \sim s_{|\sigma|}$ TER, (3)

(5) $\Rightarrow$ $(b,\sigma_{i-1},s_i,i)\Downarrow true$ 定义3, 4, (1)

(6) $\Rightarrow$ $((\text{while}(b)\{ms\}),\sigma_{i-1},s_i,i)$

$\rightarrowtail (\text{if}(b)\text{then}\{ms\wedge\text{more};\text{while}(b)\{ms\}\}\text{else}\{\text{empty}\},\sigma_{i-1},s_i,i)$ WHL

$\rightarrowtail ((b\wedge(ms\wedge\text{more};\text{while}(b)\{ms\}))\vee(\neg b\wedge\text{empty}),\sigma_{i-1},s_i,i)$ IF

$\rightarrowtail (ms\wedge\text{more};\text{while}(b)\{ms\},\sigma_{i-1},s_i,i)$ B4, F1, T1, F2, (4)

$\xrightarrow{*} (\text{empty};\text{while}(b)\{ms\},\sigma_{j-1},s_j,j)$ (4)

$\rightarrowtail (\text{while}(b)\{ms\},\sigma_{j-1},s_j,j)$ CHOP

$\xrightarrow{*} (true,\sigma,\varnothing,|\sigma|+1)\wedge(\alpha\vdash\mu_2\sim s_{|\sigma|})$ (4)

(7) $\Leftrightarrow$ $P(\text{while}(e)\,stmt,\mu,\mu',Normal)$ TER, (6)

9) 规则 (S-15) 同样处理 while 循环语句“$\text{while}(e)\,stmt$”, 如果语句 $stmt$ 包含无返回值的 return 语句, 经过 SOL2M 转换器, MSVL 中有“$\text{while}(b\text{ and }rflag=0)\{ms\}$”, 其中 $b=TranExp(e)$, $ms=TranExp(stmt)$, $rflag$ 初始值为 0, 表示当前未出现 return 语句.

(1) $\alpha\vdash e\sim_e b$ 定理1

(2) $\alpha\vdash\mu\sim s_i$ 已知条件

(3) $p,g,l\vdash e,\mu\Rightarrow\text{true}\wedge p,g,l,f\vdash stmt,\mu\Rightarrow Return,\mu_1\wedge P(stmt,\mu,\mu_1,Return)$ 假设

(4) $\Leftrightarrow$ $(ms,\sigma_{i-1},s_i,i)\xrightarrow{*}(\text{true},\sigma_j,\varnothing,j+1)\wedge\alpha\vdash\mu'\sim s_j$ TER, (2, 3)

(5) $\Rightarrow$ $(b,\sigma_{i-1},s_i,i)\Downarrow\text{true}$ 定义3, 4, (1)

(6) $\Rightarrow$ $(b\wedge rflag=0,\sigma_{i-1},s_i,i)\Downarrow\text{true}$ B3, B5, (5)

(7) $\Rightarrow$ $(\text{while}(b\text{ and }rflag=0)\{ms\},\sigma_{i-1},s_i,i)$

$\overset{*}{\rightarrowtail}((b\wedge rflag=0\wedge(ms\wedge\text{more};\text{while}(b\wedge rflag=0)\{ms\})\vee$

$\neg(b\wedge rflag=0)\wedge\text{empty}),\sigma_{i-1},s_i,i)$ WHL, IF

$\rightarrowtail (ms\wedge\text{more};\text{while}(b\wedge rflag=0)\{ms\},\sigma_{i-1},s_i,i)$ B4, F1, T1, F2, (6)

$\xrightarrow{*}(\wedge\{\text{empty},rflag\Leftarrow 1\};\text{while}(bandrflag)\{ms\},\sigma_{j-1},s_j,j)$ (3, 4)

$\rightarrowtail (\text{empty};\text{while}(b\wedge rflag=0)\{ms\},\sigma_{j-1},s_j[1/rflag],j)$ MIN1

$\rightarrowtail (\text{while}(b\wedge rflag=0)\{ms\},\sigma_{j-1},s_j,j)$ CHOP

$\overset{*}{\rightarrowtail}((b\wedge rflag=0\wedge(ms\wedge\text{more};\text{while}(b\wedge rflag=0)\{ms\})\vee$

$\neg(b\wedge rflag=0)\wedge\text{empty}),\sigma_{j-1},s_j,j)$ WHL, IF

$\rightarrowtail (\text{empty},\sigma_{j-1},s_j,j)$ F1, T1, F2

$\rightarrow(\text{true},\sigma_j,\varnothing,j+1)\wedge(\alpha\vdash\mu'\sim s_j)$ (4)

(8) $\Leftrightarrow$ $P(\text{while}(e)\,stmt,\mu,\mu',Return)$ TER, (7)

同理可证得含有返回值的 return 语句.

10) 规则 (S-9) 至 (S-12) 处理 for 循环语句“$\text{for}(stmt_1;e;stmt_2)\,stmt$”. 根据规则 (S-9) 与定义 5, 执行语句 $stmt_1$ 后得到状态 μ_1, 执行“$\text{for}(;e;stmt_2)\,stmt$”后得到状态 μ_2, 显然可知 $(\text{for}(stmt_1;e;stmt_2)\,stmt)\cong(stmt_1;\text{for}(;e;stmt_2)\,stmt)$. 根据规则 (S-10) 至 (S-12) 可知 $(\text{for}(;e;stmt_2)\,stmt)\cong(\text{while}(e)\{stmt;stmt_2\})$. 因此有 $(\text{for}(stmt_1;e;stmt_2)stmt)\cong(stmt_1;\text{while}(e)\{stmt;stmt_2\})$, 则 for 循环语句的等价性证明可由步骤 9 证明.

11) 规则 (S-16) 与 (S-21) 处理内部函数调用语句“$id_f(e^*)$”, 在 MSVL 中有“$f(ra^*,RVal)=TranStmt(e_1(e_2^*))$”, 其中, $f=TranExp(id_f)$, $ra^*=TranExp(e^*)$, $e^*=(e_1,\ldots,e_k)$, $ra^*=(ra_1,\ldots,ra_k)$, 且 $e_i=TranExp(ra_i)(1\leqslant r\leqslant k)$.

(1) $\alpha \vdash e^* \sim ra^*$ 定理1

(2) $\alpha \vdash \mu \sim s_i$ 已知条件

(3) $\wedge_{i=1}^{k}(p,g,l \vdash e_i,\mu \Rightarrow v_i) \wedge vs = (v_1,\ldots,v_k) \wedge alloc\_mem(M,l,par+dcl) = (M_1,loc) \wedge$
$store\_mem(M_1,loc,l,par,vs) = M_2 \wedge p,g,l,f \vdash stmt,\mu_2 \Rightarrow Return\ v_s,\mu_3 \wedge$
$free\_mem(M_3,loc) = M_4 \wedge P(stmt,\mu_2,\mu_3,Return\ v_s)$ 归纳假设

(4) $\alpha \vdash \mu_2 \sim s_u$ 假设

(5) $\Rightarrow\ (ms,\sigma_{u-1},s_u,u) \xrightarrow{*} (\text{true},\sigma_j,\varnothing,j+1) \wedge (\alpha \vdash \mu_3 \sim s_j)$ TER, (3)

(6) $(f(ra_1,\ldots,ra_k,RVal),\sigma_{i-1},s_i,i)$
$\rightarrowtail (id(ra_1,\ldots,ra_k,RVal),\sigma_{i-1},s_i,i)$
$\rightarrowtail ((\wedge_{r=1}^{k}\tau_r y_r \wedge mdcl \Leftarrow ra_r);ms;\bigcirc(\text{ext}\ mfree(y_1,\ldots,y_k,mdcl) \wedge \text{empty}),\sigma_{i-1},s_i,i)$ FUN
$\xrightarrow{*} (ms;\bigcirc(\text{ext}\ mfree(y_1,\ldots,y_k,mdcl) \wedge \text{empty}),\sigma_{u-1},s_u,u)$ MIN1, TR1

(7) $(ms;\bigcirc(\text{ext}\ mfree(y_1,\ldots,y_k,mdcl) \wedge \text{empty}),\sigma_{u-1},s_u,u)$
$\xrightarrow{*} (RVal \Leftarrow ra \wedge \text{empty};\bigcirc(\text{ext}\ mfree(y_1,\ldots,y_k,mdcl) \wedge \text{empty}),\sigma_{j-1},s_j,j)$ (6)
$\rightarrowtail (\text{empty};\bigcirc(\text{ext}\ mfree(y_1,\ldots,y_k,mdcl) \wedge \text{empty}),\sigma_{j-1},(s_j^l,s_j^r[v_m/RVal]),j)$ MIN1
$\rightarrow (\text{ext}\ mfree(y_1,\ldots,y_k,mdcl) \wedge \text{empty},\sigma_j,s_{j+1},j+1)$ CHOP, TR1
$\rightarrow (\text{true},\sigma_{j+1},\varnothing,j+2)$ EXT2, TR2

Solidity 中, 函数内部的参数变量默认为 memory 类型, 在函数退出时自动释放内存. MSVL 中同样在函数中先给变量 $y_1,\ldots,y_k$ 分配内存块, 并在函数退出时释放内存. 变量声明的前后, Solidity 状态由 μ 变为 μ_2, MSVL 内存状态由 s_i 变为 s_u, 整个过程不影响语句的执行和其他变量的值与位置, 且对所有的 $1 \leqslant r \leqslant k$, MSVL 中有 $(ra_r,\sigma_{i-1},s_i,i) \Downarrow v_{mr}$, 由 (1) 和 (2) 可知 $\alpha \vdash v_{sr} \sim v_{mr}$, 因此有 $\alpha \vdash \mu_2 \sim s_\mu$. 由 (5) 可知, $\alpha \vdash \mu_3 \sim s_j$. 又因为 Solidity 中从 μ_3 到 μ_4 仅是将局部变量从 s_j 中移除, 而 MSVL 中从 s_j 到 s_{j+1} 也是将 $y_1,\ldots,y_k$ 和 $mcdl$ 中的变量从 s_j 中移除, 其他变量的值和位置没有发生变化, 因此有 $\alpha \vdash \mu_4 \sim s_{j+1}$.

(8) $\Rightarrow\ (f(ra_1,\ldots,ra_k,RVal),\sigma_{i-1},s_i,i) \xrightarrow{*} (\text{true},\sigma_{j+1},\varnothing,j+2) \wedge \alpha \vdash \mu_4 \sim s_{j+1}$ (5, 6)

(9) $\Leftrightarrow\ P(id_f(e^*),\mu,\mu_4,out)$ TER, (7)

同理可证, 如果函数没有返回值, 结论依然成立.

12) 规则 (S-19) 与 (S-22) 处理外部函数调用“$e_1.e_2(e_3^*)$”.MSVL 中有 $TranStmt(e_1.e_2(e_3^*))$ = “ext $f(ra^*)$”, 其中 $f = TranExp(e_2)$, $e_3^* = (e_{s1},\ldots,e_{sk})$ 且 $ra^* = (ra_1,\ldots,ra_k)$, 有 $ra_r = TranExp(e_{sr})(1 \leqslant r \leqslant k)$, 即 $TranExp(e_3^*) = ra^*$.

(1) $(\wedge\{\bigcirc\text{empty},\text{ext}\ f(ra_1,\ldots,ra_k)\},\sigma_{i-1},s_i,i)$
$\rightarrow (\text{empty},\sigma_i,s_{i+1},i+1)$ EXT2
$\rightarrow (\text{true},\sigma_{i+1},\varnothing,i+2) \wedge s_i = s_{i+1}$ TR2

(2) $\Rightarrow\ (\wedge\{\bigcirc\text{empty},\text{ext}\ f(ra_1,\ldots,ra_k)\},\sigma_{i-1},s_i,i)$
$\xrightarrow{*} (\text{true},\sigma_{i+1},\varnothing,i+2) \wedge \alpha \vdash \mu \sim s_{i+1}$ (1)

(3) $\Leftrightarrow\ P(e_1.e_2(e_3^*),\mu,\mu,out)$ TER, (2)

其中, (1) 表示 $s_i = s_{i+1}$.

定理 3. 通过 SOL2M 转换器, Solidity 程序 P_s 转换为 MSVL 程序 P_m, 那么 P_s 和 P_m 语义等价, 记作 $P_s \sim P_m$.

证明: 假设 Solidity 程序 P_s 由 k_1 个表达式和 k_2 条语句构成, 其中 k_1 和 k_2 是常数. 当通过 SOL2M 转换器将 Solidity 程序转换为 MSVL 程序时, 有 $P_m = TranPrgm(P_s)$. 其中 $a_i = TranExp(e_i)(0 \leqslant i \leqslant k_1)$ 和 $ms_j = TranStmt$

$(stmt_j)(0 \leqslant j \leqslant k_2)$. 令 S 和 s_0 为程序的初始状态, 根据定理 1 和定理 2, 对给定的 α, 如果 $\alpha \vdash S \sim s_0$, 那么对于所有的 $0 \leqslant i \leqslant k_1$ 和 $0 \leqslant j < k_2$, 有 $\alpha \vdash e_i \sim_e a_i$ 和 $\alpha \vdash stmt_j \sim ms_j$. 因此, P_s 等价于 P_m, 即 $P_s \sim P_m$.

推论 1. 通过 SOL2M 转换器, Solidity 程序 P_s 转换为 MSVL 程序 P_m, 那么 P_s 和 P_m 的执行结束状态等价.

证明: 根据定理 3 可知 P_s 和 P_m 等价, 根据定义 6 可知对于给定的内存注入 α, P_s 和 P_m 执行终止时有 $\alpha \vdash \mu' \sim s_{|\sigma'|}$, 即 P_s 和 P_m 的执行结束状态等价.

4 相关工作

目前, 国内外学者针对智能合约操作语义的研究主要分为两种, 分别是对于 EVM 操作语义的研究和对于 Solidity 操作语义的研究.

2018 年, Zakrzewski[20]提出了 Solidity 子集的形式化规范, 给出了 Solidity 的核心数据模型和函数修饰符等特殊功能的形式化规范, 对 Solidity 的形式化更注重于动态语义, 采用的是一元函数大步语义 (big-step semantics). 提供了对于 Solidity 语言特性的准确描述和一些核心结构的动态语义, 所提出的语义在 Coq 中以可执行的形式给出.

2019 年, Yang 等人[25]提出了一种定义在 Coq 中的 Solidity 子集的形式语义, 称为 Lolisa. 首先将 Solidity 程序翻译成 Lolisa, 使用词法分析器对智能合约进行分析, 生成 Solidity token 流, 并根据 Lolisa 的语法糖, 词法分析器生成相应的 Lolisa token 流. 之后将 Solidity token 流作为解析器参数, 生成智能合约的语法树. 然后将 Lolisa 的 token 流作为语法树的 token, 通过语法分析器重建 Lolisa 语法树, 并输出由 Lolisa 重写的形式智能合约. 此外 Yang 等人[26]还为 Lolisa 在 Coq 中实现了一个经过正式验证的解释器, 称为 FEther, FEther 严格遵循基于 GERM 框架的 Lolisa 的形式语法和语义, 保证了智能合约与其正式模型之间的一致性. FEther 是以太坊第一个支持 Coq 混合验证技术的证明引擎, 将符号执行与高阶逻辑定理证明相结合, 包含一组专有的自动策略, 以高度自动化的方式执行和验证 Coq 中的智能合约.

2020 年, Jiao 等人[27,28]为 Solidity 开发了一种结构化操作语义 (structural operational semantics, SOS). 在 Solidity 类型、表达式和语句的规则之下, 抽象出了 K 框架[29]中 Solidity 的可执行语义, 能够涵盖官方 Solidity 文档所指定的大部分语义. 重点在于 Solidity 存储 (storage) 和内存 (memory) 访问的语义计算上, 并在 K 框架中实现了所提出的结构化操作语义, 提供了一个可达性逻辑证明. 此外, 以 DAO 攻击的 4 种变体为实例, 模拟了其在区块链中的行为, 结果表明生成的 Solidity 语义是可执行的, 并且可以通过可执行语义检测到智能合约中的一些漏洞, 有助于验证智能合约中的安全属性.

2020 年, Velasco[21]提出了一种可以在 Maude 中实现的 Solidity 语义. 基于大步语义给出了 Solidity 的高级语义定义, 在一定程度上参考了“Executable operational semantics of Solidity”[27], 重点关注 Solidity 的内存模型, 给出了详细的语法规则. 给出一个初始规则, 作为其余所有规则的起始点, 当智能合约部署到区块链时会分配一个新地址, 同时得到一个初始状态实例 σ. 依据规则从初始状态 σ 开始执行, 之后给出了访问和更新 Solidity 存储 (storage)、表达式、基本语句、变量声明和函数调用的操作语义.

2021 年, Marmsoler 等人[30]提出了一个在 Isabelle/HOL 中可执行的 Solidity 语义, 这种形式化语义为 Solidity 程序建立了交互式程序验证环境的基础, 并允许通过符号执行来检查 Solidity 程序. 该方法首先在 Isabelle/HOL 中为 Solidity 子集提供了可执行的表示性语义, 然后提出了基于语法的模糊框架, 可以自动验证以太坊区块链的正式语义, 其次使用 Isabelle 的代码生成器从所提出的形式化语义自动生成 Solidity 求值器, 并使用 Haskell 作为代码生成器的目标平台, 为 Solidity 程序构建集成的验证和符号执行环境, 并展示了对常数折叠和内存优化两个例子的形式化验证.

2016 年, Luu 等人[31]提出了一种用于 EVM 字节码的静态分析工具, 它依赖于符号执行对智能合约的安全性漏洞进行检测. 静态分析方法的优势是为可以静态确认的合约属性实现完全自动化. Oyente 忽略了与智能合约调

用和创建相关的几个重要命令, 提供了简化的 EVM 字节码语义. Oyente 是使用 Python 进行开发的符号执行引擎, 支持大多数 EVM 操作码, 将智能合约字节码反编译成控制流图, 并执行控制流分析, 使用 Z3 作为求解器来评估其满足性. Oyente 支持检测多个常见的智能合约安全漏洞, 如交易顺序依赖漏洞、时间戳依赖、异常障碍和重入漏洞. 从区块链中收集了 19366 份智能合约作为实例, 验证了 Oyente 工具的可行性, 并对其准确率和误判率进行了统计分析. 作为 Solidity 安全性验证的开源工具, Oyente 工具增加了以太坊开发者创建安全可靠的分散式应用程序的能力, 也为其他研究人员奠定了理论基础.

2017 年, Hildenbrandt 等人[32]提出了 EVM 的第 1 个完全可执行的正式语义, 称为 KEVM, 是在 K 框架[27]中定义的正式严格的可执行语义. 抽象了区块链系统本身的一些细节, 使用所定义的语义自动生成了正式派生的 EVM 解释器, 可以在合理的执行时间内运行完整的以太坊虚拟机测试套件, 并通过了 40689 个 EVM 合规性测试套件, 展示了解释器良好的性能. 为 EVM 提出了语义优先的形式化验证方法, 使用了为 K 框架开发的可达性逻辑证明程序[33], 该证明程序以一个 K 定义和一组逻辑可达性声明作为输入来验证, 验证者在假设语义的情况下, 自动证明语言执行空间上的可达性定理. 文献 [32] 给出了两个实例的完整验证, 第 1 个实例验证了合约中算术操作的实际重要属性, 所验证的属性包含 EVM 源程序的功能正确性和 gas 复杂性, 第 2 个实例以以太坊生态系统中被广泛使用的令牌标准 ERC20 的传递函数为例, 验证其正确操作, 通过两个实例证明了该方法的可行性.

2018 年, Grishchenko 等人[34]提出了一种 EVM 字节码的完整小步语义 (small-step semantics), 该语义在 Luu 等人提出的字节码的基础上, 做了大量修改, 处理了部分缺失的指令, 例如合约调用和调用创建, 并正式定义了智能合约的一些核心安全属性, 例如调用完整性、原子性和独立于矿工控制的参数. 文献 [34] 将提出的语义在 F*中形式化, F*针对程序验证进行了优化, 并允许利用 SMT 求解器执行手动证明和自动证明. 在 F*中的形式化严格遵循了提出的小步语义, 最终通过将 F*编译为 OCaml, 使用官方以太坊测试套件验证了所提出的可执行语义. 在 F*中的形式化公开可用, 促进了 EVM 字节码静态分析技术的设计及其稳健性证明.

2018 年, Amani 等人[35]在字节码级别上扩展了 Isabelle/HOL 中现有的 EVM 形式化, 该方法以非结构化字节码为目标, 独立于高级编程语言 (如 Solidity), 一方面减少对高级工具正确性的依赖, 另一方面字节码是以太坊实际编程语言, 区块链上的所有智能合约均基于该语言. 该方法首先扩展了 Isabelle/HOL 定理证明中的 EVM 形式化, 涵盖了智能合约的正确属性, 其次给出了健全的程序逻辑, 能够在字节码级别验证智能合约, 之后提供了 Isabelle 策略, 以支持使用逻辑规则自动生成验证条件, 最后以托管协议智能合约为例, 生成了相应的 EVM 字节码, 并使用所提出的程序逻辑验证了 EVM 字节码的功能正确性, 证明了该方法的可行性和适用性.

综上所述, 目前对于智能合约语义的研究工作集中在以太坊字节码语义和 Solidity 高级语义上, Luu、Grishchenko、Hildenbrandt 和 Amani 等人重点研究的是 EVM 操作语义, 这为 Solidity 高级语言操作语义的定义提供了参考价值. Solidity 操作语义的定义与其应用的形式化方法息息相关, Jiao、Zakrzewski、Velasco、Yang 和 Marmsoler 等人对 Solidity 语言的操作语义进行了研究, 其中 Jiao、Yang 以及 Marmsoler 等人重点关注 Solidity 语言的可执行语义, 其目的是能够在 K 框架、Coq 或 Isabelle/HOL 中执行语义并对智能合约进行形式化验证.

5 总结与展望

为了更好地对智能合约进行安全验证, 之前的工作基于 MSVL 与 PPTL 对智能合约进行形式化验证[12−13], 开发了 SOL2M 转换器, 实现了对 Solidity 程序的半自动化建模, 进而使用 UMC4M 实现了 Solidity 的可重入漏洞检测. 该转换器在制定转换规则时重点关注 Solidity 与 MSVL 的词法及语法. 本文从数学角度出发, 制定基于大步语义风格的 Solidity 子集的操作语义, 并从 MSVL 和 Solidity 的操作语义出发, 使用结构归纳法以及规则归纳法证明两者操作语义的等价性, 建立形式化语言和 Solidity 之间基于语义的映射关系, 为基于形式化方法的智能合约安全验证奠定理论基础. 目前定义的操作语义只针对 Solidity 的一个子集, 囊括了主要语法但仍存在未定义的结构, 如 require 语句等, 后续将进一步扩充该子集. 同时, 随着 Solidity 的版本迭代, 需要根据新特性语句的开始与结束状态变化, 不断完善其操作语义及等价性证明. 本文研究工作为 SOL2M 转换器提供了严格的数学依据, 从操作语

义等价的角度为基于 MSVL 的智能合约形式化验证提供了理论基础, 据此可对 SOL2M 转换器进行持续的优化和扩展. 进一步, 随着 Solidity 操作语义与等价性证明的不断扩展, 将对更多版本的 Solidity 智能合约进行研究, 并将该方法应用到基于其他语言的智能合约验证.

References:

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. 2008. https://bitcoin.org/bitcoin.pdf

[2] Zhang J, Gao WZ, Zhang YC, Zheng XH, Yang LQ, Hao J, Dai XX. Blockchain based intelligent distributed electrical energy systems: Needs, concepts, approaches and vision. Acta Automatica Sinica, 2017, 43(9): 1544–1554 (in Chinese with English abstract). [doi: 10.16383/j.aas.2017.c160744]

[3] Buterin V. A next-generation smart contract and decentralized application platform. 2014. https://ethereum.org/en/whitepaper/

[4] Underwood S. Blockchain beyond bitcoin. Communications of the ACM, 2016, 59(11): 15–17. [doi: 10.1145/2994581]

[5] Delmolino K, Arnett M, Kosba A, Miller A, Shi E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: Proc. of the 2016 Int'l Workshops on Financial Cryptography and Data Security. Christ Church: Springer, 2016. 79–94. [doi: 10.1007/978-3-662-53357-4_6]

[6] Szabo N. Formalizing and securing relationships on public networks. First Monday, 1997, 2(9). [doi: 10.5210/fm.v2i9.548]

[7] Clack CD, Bakshi VA, Braine L. Smart contract templates: Foundations, design landscape and research directions. arXiv:1608.00771v3, 2017.

[8] Mehar MI, Shier CL, Giambattista A, Gong E, Fletcher G, Sanayhie R, Kim HM, Laskowski M. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. Journal of Cases on Information Technology (JCIT), 2019, 21(1): 19–32. [doi: 10.4018/JCIT.2019010102]

[9] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts (SoK). In: Proc. of the 6th Int'l Conf. on Principles of Security and Trust. Uppsala: Springer, 2017. 164–186. [doi: 10.1007/978-3-662-54455-6_8]

[10] Qian P, Liu ZG, He QM, Huang BT, Tian DZ, Wang X. Smart contract vulnerability detection technique: A survey. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 3059–3085 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6375.htm [doi: 10.13328/j.cnki.jos.006375]

[11] Wang PW, Yang HT, Meng J, Chen JC, Du XY. Formal definition for classical smart contracts and reference implementation. Ruan Jian Xue Bao/Journal of Software, 2019, 30(9): 2608–2619 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5773.htm [doi: 10.13328/j.cnki.jos.005773]

[12] Wang XB, Yang XY, Shu XF, Zhao L. Formal verification of smart contract based on MSVL. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1849–1866 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6253.htm [doi: 10.13328/j.cnki.jos.006253]

[13] Zhu YK. A method based on MSVL for modeling and verifying the smart contract [MS. Thesis]. Xi'an: Xidian University, 2020 (in Chinese with English abstract). [doi: 10.27389/d.cnki.gxadu.2020.002974]

[14] Wang M, Tian C, Zhang N, Duan ZH. Verifying full regular temporal properties of programs via dynamic program execution. IEEE Trans. on Reliability, 2019, 68(3): 1101–1116. [doi: 10.1109/TR.2018.2876333]

[15] Ma YT, Duan ZH, Wang XB, Yang XX. An interpreter for framed tempura and its application. In: Proc. of the 1st Joint IEEE/IFIP Symp. on Theoretical Aspects of Software Engineering. Shanghai: IEEE, 2007. 251–260. [doi: 10.1109/TASE.2007.10]

[16] Wang XB, Tian C, Duan ZH, Zhao L. MSVL: A typed language for temporal logic programming. Frontiers of Computer Science, 2017, 11(5): 762–785. [doi: 10.1007/s11704-016-6059-4]

[17] Zhang N, Duan ZH, Tian C. A mechanism of function calls in MSVL. Theoretical Computer Science, 2016, 654: 11–25. [doi: 10.1016/j.tcs.2016.02.037]

[18] Wang M. Verifying full regular temporal properties of programs via dynamic program execution [Ph.D. Thesis]. Xi'an: Xidian University, 2019 (in Chinese with English abstract). [doi: 10.27389/d.cnki.gxadu.2019.000059]

[19] Wang XB, Duan ZH, Zhao L. Formalizing and implementing types in MSVL. In: Proc. of the 3rd Int'l Workshop on Structured Object-oriented Formal Language and Method. Queenstown: Springer, 2013. 62–75. [doi: 10.1007/978-3-319-04915-1_5]

[20] Zakrzewski J. Towards verification of Ethereum smart contracts: A formalization of core of solidity. In: Proc. of the 10th Int'l Conf. on Verified Software: Theories, Tools, and Experiments. Oxford: Springer, 2018. 229–247. [doi: 10.1007/978-3-030-03592-1_13]

[21] Velasco PP. Providing formal semantics for solidity to allow its verification [BS. Thesis]. Madrid: Universidad Complutense de Madrid, 2020.

[22] Dagnino F. A meta-theory for big-step semantics. ACM Trans. on Computational Logic, 2022, 23(3): 20. [doi: 10.1145/3522729]

[23] Kahn G. Natural semantics. In: 4th Annual Symp. on Theoretical Aspects of Computer Science. Passau: Springer, 1987. 22–39. [doi: 10.1007/BFb0039592]

[24] Blazy S, Dargaye Z, Leroy X. Formal verification of a C compiler front-end. In: Proc. of the 14th Int'l Symp. on Formal Methods. Hamilton: Springer, 2006. 460–475 [doi: 10.1007/11813040_31]

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

[26] Yang Z, Lei H. FEther: An extensible definitional interpreter for smart-contract verifications in Coq. IEEE Access, 2019, 7: 37770–37791. [doi: 10.1109/ACCESS.2019.2905428]

[27] Jiao J, Kan SL, 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. San Francisco: IEEE, 2020. 1695–1712. [doi: 10.1109/SP40000.2020.00066]

[28] Jiao J, Lin SW, Sun J. A generalized formal semantic framework for smart contracts. In: Proc. of the 23rd Int'l Conf. on Fundamental Approaches to Software Engineering. Dublin: Springer, 2020. 75–96. [doi: 10.1007/978-3-030-45234-6_4]

[29] K semantic framework. 2024. https://kframework.org

[30] Marmsoler D, Brucker AD. A denotational semantics of solidity in Isabelle/HOL. In: Proc. of the 19th Int'l Conf. on Software Engineering and Formal Methods. Virtual Event: Springer, 2021. 403–422. [doi: 10.1007/978-3-030-92124-8_23]

[31] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 254–269. [doi: 10.1145/2976749.2978309]

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

[33] Stefănescu A, Park D, Yuwen SJ, Li YL, Roşu G. Semantics-based program verifiers for all languages. ACM SIGPLAN Notices, 2016, 51(10): 74–91. [doi: 10.1145/3022671.2984027]

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

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

附中文参考文献:

[2] 张俊, 高文忠, 张应晨, 郑心湖, 杨柳青, 郝君, 戴潇潇. 运行于区块链上的智能分布式电力能源系统: 需求、概念、方法以及展望. 自动化学报, 2017, 43(9): 1544–1554. [doi: 10.16383/j.aas.2017.c160744]

[10] 钱鹏, 刘振广, 何钦铭, 黄步添, 田端正, 王勋. 智能合约安全漏洞检测技术研究综述. 软件学报, 2022, 33(8): 3059–3085. http://www.jos.org.cn/1000-9825/6375.htm [doi: 10.13328/j.cnki.jos.006375]

[11] 王璞巍, 杨航天, 孟佶, 陈晋川, 杜小勇. 面向合同的智能合约的形式化定义及参考实现. 软件学报, 2019, 30(9): 2608–2619. http://www.jos.org.cn/1000-9825/5773.htm [doi: 10.13328/j.cnki.jos.005773]

[12] 王小兵, 杨潇钰, 舒新峰, 赵亮. 面向 MSVL 的智能合约形式化验证. 软件学报, 2021, 32(6): 1849–1866. http://www.jos.org.cn/1000-9825/6253.htm [doi: 10.13328/j.cnki.jos.006253]

[13] 朱云凯. 基于 MSVL 的智能合约建模与验证 [硕士学位论文]. 西安: 西安电子科技大学, 2020. [doi: 10.27389/d.cnki.gxadu.2020.002974]

[18] 王猛. 基于动态执行的程序时序性质验证 [博士学位论文]. 西安: 西安电子科技大学, 2019. [doi: 10.27389/d.cnki.gxadu.2019.000059]

王小兵(1979－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为形式化方法, 形式化验证, 时序逻辑.

杨潇钰(1999－), 女, 硕士, 主要研究领域为形式化方法, 形式化验证, 时序逻辑.

常家俊(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为形式化方法, 形式化验证, 时序逻辑.

赵亮(1984－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化方法, 形式化验证, 时序逻辑.

李春奕(1996－), 女, 博士生, 主要研究领域为形式化方法, 形式化验证, 时序逻辑.