基于 CompCert 内存模型的智能合约中间语言的可信编译

许 颖[1,2], 张亚丰[1,3], 许晶航[1,3], 康跃馨[1,3], 夏 清[1,3], 袁 峰[4], 左 春[5], 李玉成[1,3]

[1]中国科学院软件研究所并行软件与计算科学实验室 北京 中国 100190
[2]中国科学院大学计算机科学与技术学院 北京 中国 100049
[3]中国科学院软件研究所基础软件与系统重点实验室; 计算机科学国家重点实验室 北京 中国 100190
[4]广州软件应用技术研究院 广州 中国 511458
[5]中科软科技股份有限公司 北京 中国 100190

摘要 智能合约是区块链技术的重要组成部分之一, 具有不可篡改、自动执行等特点, 为去中心化应用提供了丰富的编程基础。近年来相关安全漏洞事件频发, 使得智能合约的安全性研究逐渐成为热点。其中, 智能合约编译器的误编译问题会令源代码编译产生不符合开发者原本预期的目标代码, 导致部署在区块链上的代码存在安全隐患, 然而现有工作较少考虑到这一问题。因此, 首先从避免合约安全漏洞的原则出发, 设计一种非图灵完备的智能合约领域专用语言 isCL。作为可信编译的源语言, 它支持复合数据类型与内置函数, 以便于开发人员编写合约; 然后设计可信编译器 i2c 的整体架构, 实现以 C 语言子集 Clight 语言为目标语言的完整翻译过程; 再针对基于 CompCert 内存模型的智能合约中间语言的翻译阶段, 定义消除复合类型、出参入参合并、生成 Clight 三个翻译阶段的相关语法、语义, 并给出语义保持性的证明思路; 最后通过服装供应链分账应用实例和 Solidity、Rust 的编译漏洞案例来分别说明 isCL 语言的实用性与智能合约可信编译的有效性。本文工作为智能合约的可信编译提供了研究思路, 有利于促进智能合约开发的安全性研究, 为实现更加安全可信的区块链应用提供有力支撑。

关键词 区块链; 智能合约语言; 领域专用语言; 形式化验证; 经过验证的编译器
中图法分类号 TP314 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2026.01.03

Trusted Compilation of Smart Contract Intermediate Language Based on the CompCert Memory Model

XU Ying[1,2], ZHANG Yafeng[1,3], XU Jinghang[1,3], KANG Yuexin [1,3], XIA Qing[1,3], YUAN Feng[4], ZUO Chun[5], LI Yucheng[1,3]

[1] Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
[2] School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China
[3] Key Laboratory of Systems Software and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
[4] Institute of Software Application Technology, Guangzhou, Guangzhou 511458, China
[5] Sinosoft Co., Ltd., Beijing 100190, China

Abstract Smart contracts are one of the critical components of blockchain technology, characterized by their immutability and automatic execution, providing a rich programming foundation for decentralized applications. In recent years, the frequent occurrence of security vulnerabilities has made the security research of smart contracts a hot topic. Among these issues, the problem of miscompilation by smart contract compilers can result in the generation of target code that does not meet the original expectations of developers, leading to security risks in the code deployed on the blockchain. However, existing works has seldom considered this issue. To tackle this problem, we first designed a non-Turing-complete domain-specific language for smart contracts, named isCL, based on the principle of avoiding contract security vulnerabilities. To serve as a trusted source language for compilation, isCL supports composite data types and built-in functions to facilitate developers in writing smart contracts. Following this, we designed the overall architecture of a trusted compiler, i2c, which implements a complete translation process into the Clight language, a subset of the C language, as the target language. Furthermore, focusing on the translation phase of the smart contract intermediate

通讯作者: 康跃馨, Email: yuexin@iscas.ac.cn。
本课题得到国家重点研发计划(No. 2022YFB2702202)资助。
收稿日期: 2024-02-04; 修改日期: 2024-07-24; 定稿日期: 2025-12-05

language based on the CompCert memory model, we define the relevant syntax and semantics for three translation stages: eliminating composite types, merging input and output parameters, and generating Clight, then provided the proof idea for semantic preservation in these stages. Finally, to demonstrate the practicality of the isCL language and the effectiveness of trusted compilation for smart contracts, we presented a case study of a clothing supply chain ledger application and examined compilation vulnerability cases in Solidity and Rust. This work provides a research approach for the trustworthy compilation of smart contracts, which is beneficial to the security research of smart contract development. It offers strong support for achieving more secure and trustworthy blockchain applications.

Key words blockchain; smart contract; formal verification; theorem proving; verified compiler

1 引言

自 2009 年中本聪提出比特币[1]以来, 其底层区块链技术逐渐成为产学研界的关注热点。区块链作为一种基于块链式数据结构的分布式账本技术, 是结合智能合约、共识协议、密码学算法、点对点网络等技术的创新应用, 拥有去中心化、公开透明和数据不可篡改的特性[2]。其中, 智能合约是部署并运行在区块链系统上的程序代码, 能够在无需依赖第三方信任的情况下, 自动、可信地执行业务逻辑。智能合约的概念最早在 1995 年由 Nick Szabo[3]提出, 2013 年首次被 Vitalik Buterin 引入以太坊平台[4]上。智能合约的编程灵活性使得开发者可以根据业务需求自定义合约规则, 搭建各种类型的区块链应用。目前, 智能合约被广泛用于金融交易、供应链管理、版权保护等应用场景中, 推动着区块链的生态发展。

智能合约安全性是区块链安全体系中至关重要的一环。智能合约出现漏洞可能会导致巨大的经济损失, 从而对区块链系统的信任体系构成严重威胁。据厦门慢雾科技有限公司统计[5], 截至 2023 年 10 月, 区块链安全问题已经导致数百亿美元的经济损失。其中针对智能合约漏洞的攻击事件数量排名第一。2016 年攻击者利用 The DAO 合约的可重入漏洞改变函数执行顺序, 盗走当时价值约 6000 万美元的以太币[6]; 2017 年 Parity Wallet 多重签名合约的漏洞导致超过 300 万美元的损失[7]; 2018 年, 美链 BEC 合约的整数溢出漏洞造成约 64 亿元人民币损失[8]。智能合约安全保障技术影响着区块链生态系统的健康发展, 已成为一个日益重要的研究方向。

编译正确性是智能合约安全性中一个不容忽视的安全问题。合约开发者通常使用智能合约领域特定的编程语言或者通用编程语言编写智能合约的源代码, 例如在以太坊上使用 Solidity[9]、Vyper[10], 在 EOS[11]上使用 Rust、C++, 等等。与传统程序的编译过程类似, 智能合约源代码需要经过编译器的语法分析、语义检查、优化等一系列步骤, 最后生成可以部署到区块链上的代码。但编译器难免会出现错误(即误编译), 使得源代码编译产生不符合开发者的原本预期的目标代码。Solidity 编译器的开发团队维护了一个编译器安全漏洞列表, 记录了 2016 年以来的 24 个不同类型的安全漏洞[12], 每项包括了漏洞的描述、严重程度等。例如在 2020 年报道了一个严重程度为中等的漏洞, 描述了在动态数组缩小长度时元素没有正确地清零, 导致重新扩展长度以后这个数组里存在遗留的旧数据。Park 等[13]在对以太坊 2.0 的智能合约进行形式化验证时发现了 Vyper 编译器的一些问题, 例如当源程序中函数的返回值类型是小于 16 位的字节数组时, 编译过程没有实现足够的零填充, 导致编译出来的字节码不符合 ABI 规范, 函数无法正常使用。对于一般程序, 编译器出错导致的问题还可以在代码后续版本里修复, 但智能合约具有部署后不易更改、公开透明的特性, 因此部署前就应该尽可能确保合约代码没有缺陷, 否则攻击者可能会利用漏洞从而造成严重的经济损失。例如, 在 2023 年 7 月, Vyper 编译器的几个版本错误地设计了重入锁, 结果生成的字节码无法有效阻止重入攻击, 导致 Curve 的部分稳定币池损失约 7000 万美元[14]。

现有研究往往使用静态分析、符号执行、模糊测试等方法检查合约代码的逻辑漏洞[15], 例如溢出问题、可重入漏洞等, 或者检查合约是否满足用户定义的形式化规范, 却较少考虑到智能合约的编译正确性。原因可能在于智能合约是相对新颖的领域, 其误编译的问题还没有得到足够关注, 其次是研究资源有限, 检查编译过程的正确性比起处理合约逻辑问题更加费时费力。然而, 误编译不仅会导致部署到区块链上的代码在逻辑上存在安全隐患, 还可能影响源代码级别的合约安全性研究, 导致通过验证的合约仍然编译出不可靠的代码, 而目标代码级别的研究又并不考虑源代码的原始语义, 无法保证目标代码的语义与用户预期一致。因此智能合约的误编译问题同样亟待解决。

形式化方法中的定理证明技术是一种有效解决程序误编译问题的方案。定理证明技术使用数学逻辑语言和相关证明序列描述并验证计算机系统或软件, 确保程序一定满足规范。CompCert 编译器[16]在编译领域受到过广泛认可, 它正是使用了定理证明

技术证明源代码和目标代码的语义保持一致, 从而将 C 语言的一个重要子集 Clight 语言可信地编译为汇编代码。后来 Yang 等[17]在使用随机差分测试检验 C 编译器时, 花费了 6 个 CPU 年测试 CompCert, 却没有发现在其他编译器上发现的在中间表示上的转换错误, 有力地证明了 CompCert 在正确性上的优势。然而, 已有的可信编译工作实现了部分通用程序语言[16, 18-19]或领域特定语言[20-21]的编译证明。这些语言与智能合约的使用场景不符, 缺少便于合约开发使用的数据结构, 也无法接入合约容器, 所以不能直接用于合约开发。同时, 由于语法复杂度与证明的工作量紧密相关, 而目前的智能合约语言语法通常比较复杂, 因此也不适合直接用作证明的源语言。

为解决智能合约程序的误编译问题, 本文首先设计一种非图灵完备的智能合约语言 isCL, 语法上支持复合数据类型及其内置函数, 便于合约开发者编写代码逻辑。以 isCL 语言为源语言、以 CompCert 可信编译器提出的 C 语言子集 Clight 语言为目标语言, 实现可信编译器 i2c。为简化相关的验证过程, 以 Clight 语言语义中更加抽象的大步语义作为语义定义基础[22], 由于已完成证明的阶段进行了构造复合数据类型环境、合并函数参数等工作, 翻译阶段前后程序、函数的语法发生改变, 复合数据类型表达式和函数调用的语义定义也有所区别, 因此介绍这些不同之处, 以及如何使用策略证明语义保持性。最后分别给出智能合约应用实例和误编译案例, 以说明 isCL 语言以及可信编译的实用性和有效性。

本文第 2 节介绍国内外关于可信编译、智能合约的形式化验证等方面工作; 第 3 节介绍本文提出的智能合约语言 isCL 的语法; 第 4 节介绍 i2c 编译器的总体框架; 第 5 节介绍已证明阶段的相关语法、语义定义及语义保持性的证明思路; 第 6 节介绍 isCL 语言的智能合约应用实例和 Solidity、Rust 的误编译案例; 第 7 节说明本文工作的现状和未来计划; 第 8 节给出本文的工作总结。

2 相关工作

一些研究工作从类型系统、适用领域等不同角度出发, 提出了新的智能合约语言及其验证工具[15], 旨在增强智能合约的安全性。Move 语言[23]最早是 Meta 为 Diem 区块链开发的合约语言, 采用静态类型、函数式编程, 基于借用语义、线性资源类型等概念设计, 其验证器可以将 Move 程序及程序规范编译成 Boogie 代码, 从而进行形式化验证。Schrans 等[24]提出的 Flint 语言也类似地受到线性类型理论的启发, 通过类型状态限制和资产类型特性等增强安全性, 编译器最终产生 EVM 字节码。神荼团队研发的 DeepSEA 语言[25]采用函数式编程, 合约可以编译部署到 Shentu Chain、Ethereum 等平台, 其编译器能够生成输入程序的字节码与一个可用于形式化证明程序性质的模型, 以及字节码行为与模型之间的匹配证明。Bernardo 等[26]提出了 Tezos 平台的 Michelson 语言, 这是一种类似 Forth 语言的基于栈的解释型语言, 图灵完备, Michelson 语言的验证框架 Mi-Cho-Coq 包括一个由 Coq 实现的解释器和最弱前置条件计算函数, 用于验证合约的功能正确性。也有一些工作关注中间语言, Sergey 等[27]开发的 Scilla 中间语言基于元理论与有限状态机模型, 能够验证智能合约的时态性与安全性, Bernardo 等在 Michelson 语言之后提出的 Albert 中间语言[28]对栈进行了抽象, Albert 的编译器用 Coq 实现, 以 Michelson 为目标语言。Santos Reis 等[29]则进一步提出了 Tezla 中间语言, 它由 Michelson 代码反编译得到, 便于应用已有的静态分析工具。除此以外, 还出现了越来越多的领域特定语言(domain-specific language, DSL), 特别是面向金融领域的 DSL, 例如 Malowe[30]、Findel[31]和 Daml[32]等。它们被设计为非图灵完备语言以简化代码行为、减少安全漏洞, 为针对性地实现业务逻辑提供了便利。然而这些工作尚未考虑误编译的问题。

本文与智能合约语言的形式化语义领域密切相关, 这些工作有的应用了诸如 K 框架的通用形式化工具以便自动化验证, 有的则使用定理证明工具进行更加针对性的建模。从不同语言层次来看, 在高级语言和中间语言方面, Jiao 等[33]使用 K 框架定义了 Solidity 语言比较完整的形式化语义。Yang 等[34]实现了 Solidity 的一个较大子集的形式化语法和语义, 使得 Solidity 程序能够逐行翻译到 Coq 中进行形式化验证。Annenkov 等[35]提出了一个基于 Coq 的智能合约验证框架 ConCert, 定义了一种 ACORN 语言核心子集的语法和语义, 结合深、浅嵌入的方式, 将 ACORN 合约程序自动转换成 Coq 程序, 并且用元编程工具 MetaCoq 证明了嵌入过程的可靠性, 从而更安全地验证合约功能与时态属性。Han 等[36]在 Isabelle/ HOL 定理证明器中给出了以太坊中间语言 Yul 的类型系统与小步操作语义的形式化定义, 可用于模拟合约执行和相关的定理证明。在低级语言层面, Hirai[37]在 Lem 框架中定义 EVM, 并用 Isabelle/ HOL 证明了合约的一些安全属性, Grishchenko 等[38]给出了 EVM 字节码的完整小步语义并在 F*框架中进行形式化, 也定义了一些合约安全属性, Hildenbrandt

等[39]则使用 K 框架实现了一个可执行的 EVM 字节码形式化规范 KEVM。

上述这些研究的验证目标主要是合约的安全与功能属性, 例如关注可重入或者整数溢出问题, 以及是否满足了用户编写在代码里的形式化规范, 未能验证编译过程的正确性。

形式化验证编译器的一种方法是翻译确认, 最早由 Pnueli 等[40]提出。翻译确认侧重于证明翻译前后代码的等价性质, 以确保编译器能够正确地编译出结果, 而无需验证编译器本身。比如, Clément 等[41]针对仿射形式的 Halide 规范, 给出了一种独立于调度语言的翻译确认算法。而在智能合约领域, Bhargavan 等[42]考虑到可以通过关系推理的方法, 验证 Solidity 代码翻译得到的 F*代码和对应的 EVM 字节码经过反编译得到的 F*代码的等价性, 但没有更进一步的工作。Krijnen 等[43]针对 Cardano 区块链的 Plutus 语言, 实现了编译器里大多数翻译遍的关系验证。这种方法比较简单, 便于集成到编译流程中, 可扩展性比较好, 但只能用于部分性质的验证。

另一种方法是形式化地证明编译器本身的正确性, 使得输入任意一个有效的源程序, 编译器都能产生一个语义等价的目标程序, 从而实现可信的编译过程。一个具有代表性的工作是 Leroy 等[16]提出的经过验证的 C 编译器 CompCert, CompCert 的实现为后来的可信编译研究提供了重要的成功经验。以 CompCert 项目为基础, Song 等[44]使用轻量级验证技术支持 CompCert 的多语言链接, Koenig 等[45]扩展了正确性定理, 使之能够直接描述编译后的程序组件在相互交互时的行为特征, Pouzet 团队开发的 Vélus 编译器[20]和王生原团队开发的 L2C 编译器[21]将同步数据流语言 Lustre 翻译到 Clight 语言[46], 验证其主要翻译阶段的语义保持性, 再与 CompCert 编译器对接, 以实现 Lustre 语言的可信编译。除此以外, Klein 等[18]给出了类 Java 语言 Jinja 的形式化语义以及一系列安全性证明。Kumar 等[19]则在 CakeML 项目里使用 HOL4 对 Standard ML 的一个子集的编译流程进行数学证明。比起翻译确认技术, 这种方法能够更加全面地验证编译器, 可信度高。本文受到 CompCert 等相关工作经验的启发, 对智能合约的编译过程进行形式化, 给源语言到目标语言的每个中间语言定义语义, 并确保逐个翻译阶段的语义保持不变, 从而为智能合约的可信编译提供新的研究思路。

3 isCL 语言

智能合约语言的语法设计缺陷可能会给合约的正确执行带来安全隐患, 一些研究讨论了由语言语法引起的漏洞[47-48], 根据这些研究, 表 1 总结了常见的合约漏洞类型。

表 1 常见的合约漏洞类型与 isCL 语言的相应设计

Table 1 Common contract vulnerability types and corresponding design in isCL language

合约漏洞类型	相应设计
默认变量类型	严格类型定义
变量未初始化	变量默认初始值
同名变量误用	禁止变量重名
未处理异常	异常发生程序终止
重入漏洞	禁止函数递归调用
调用栈溢出	禁止函数递归调用

如表 1 所述, 为了保证安全性, isCL 通过制定更加严格的编程语法来降低合约出现漏洞的风险。

首先, 默认类型漏洞指的是在 Solidity 的某些版本中声明变量时, 默认了变量类型是持久化存储的数据类型, 这会导致不必要地占用合约存储空间。isCL 语言有着严格的类型定义, 要求明确定义变量的类型。例如, 由 isCL 语言编写的智能合约在经过编译器处理时, 语法分析阶段检查变量是否指明类型, 而静态类型检查阶段则会检查语句里变量类型是否匹配, 如图 1 所示, 在翻译赋值语句的过程中, 首先翻译左表达式和右表达式, 推断并替换右表达式的值为 None 时的选项类型, 再判断左右表达式的类型是否相等, 如果类型相等, 返回翻译以后的赋

```
1:  Fixpoint trans_stmt (s: stmtP): res stmtW :=
2:    match s with
3:    | SassignP e1 e2 l =>
4:      do e1 <- trans_lexpr e1 l;
5:      do e2 <- trans_expr e2 l;
6:      (* 推断并替换右表达式为None时的类型 *)
7:      do e2 <- infer_and_replace_none e2 (typeofW e1) l;
8:      match is_equal_type (typeofW e1) (typeofW e2) with
9:      | true => OK (SassignW e1 e2)
10:       | false =>
11:         match e1 with
12:         | EvarW id _ =>
13:             match Pos.eqb id OUT with
14:             (* 返回的表达式类型不正确 *)
15:             | true => Error (error_msg l ("……"))
16:             (* 赋值语句中的表达式类型不一致 *)
17:             | false => Error (error_msg l ("……"))
18:             end
19:         (* 赋值语句中的表达式类型不一致 *)
20:         | _ => Error (error_msg l ("……"))
21:         end
22:     end
23:   ……
24:   end
```

图 1 静态类型检查阶段的部分语句翻译函数

Figure 1 A partial statement translation function for the static type checking phase

值语句, 否则判断左表达式的类型, 针对它是否是变量表达式和是否具有函数返回值的特殊标识 OUT 来分别给出错误信息。

变量未初始化会使得变量访问未知的内容, 直接访问变量内容的操作造成程序出现意外结果, 例如 Solidity 的未初始化 Storage 指针问题[49]可能使得开发者定义未初始化的 Storage 指针, 从而错误地修改合约里的存储内容。当一个变量没有初始值时, isCL 语言将根据其类型强制赋予一个默认初始值并给出警告。

同名变量误用指的是全局和局部变量重名, 这可能导致开发者使用与原本预期不符的变量, 造成程序逻辑错误。因此, isCL 语言不允许全局变量与局部变量重名。

未处理异常指的是 Solidity 的一些低级别调用函数在出现错误时不会抛出异常而是给出表示结果的返回值, 例如 Send 函数在转账失败时返回布尔值 False, 如果开发者不能正确地处理返回值, 会增加合约中的潜在风险。而 isCL 语言在程序出现错误的情况下, 将直接终止程序, 回滚合约状态。

重入漏洞、调用栈溢出的问题则与函数递归调用的特性有关, isCL 的语法禁止函数递归和函数循环递归, 但支持可迭代数据结构进行遍历, 这样既能确保合约的可终止性又能提供便利的数据结构操作方法。

以安全性为出发点, 像许多智能合约语言那样[30-32], isCL 语言设计为非图灵完备语言。非图灵完备语言具有比较简单的编程语法, 这样既能降低合约漏洞, 也便于证明编译过程的正确性。尽管这一定程度上限制了开发人员在解决问题时的选择, 但在大部分场景下, 开发人员并不需要函数的递归调用功能。例如, Jansen 等[50]通过分析以太坊上已经部署的智能合约发现, 只有 3.6%的合约使用了函数递归调用功能。isCL 语言还是一种静态语言, 这有助于提前发现程序中潜在的类型错误和安全问题。附录给出了 isCL 语言的语法定义。具体来说, 一个程序由结构体块、上下文变量块、状态变量块、全局常量块以及函数块构成。结构体块用于注册新的结构体类型, 例如交易发起者的账户地址, 状态变量块用于声明在区块链上持久化状态值的变量, 全局常量块定义了程序的全局常量, 函数块定义了程序的函数。状态变量是智能合约语言特有的语法特点, 它在编译过程中直接翻译为全局变量。目前 isCL 语言除了提供了常用的运算符和操作语句, 还特别支持复合数据类型 String、Option 等, 以及它们的内置函数, 以满足智能合约业务场景需要, 例如, indexOf 方法可以得到 String 类型数据中指定字符串第一次出现的索引, get 方法可以从 Option、List 或 Map 类型的数据中获取元素。

综上所述, isCL 语言具有以下主要的语法特性:

(1) 静态检查: 如前所述, isCL 语言的设计依赖严格的类型系统, 能够配合编译器前端的类型检查进行报错反馈, 检查出绝大部分错误。

(2) 易用性: isCL 语言的语法考虑到合约开发的常用操作, 针对数据结构的运算引入了内置函数, 简化了智能合约逻辑的表达, 也提升了智能合约的可读性。

(3) 严格语义: 每一层的抽象语法树(Abstract syntax tree, AST)都将会赋予明确语义, 这样既明确了任意程序的执行效果, 又能用于后续证明翻译过程的语义保持性。

4 i2c 编译器的总体框架

i2c 编译器以 isCL 语言作为源语言, Clight 语言作为目标语言, 接入 CompCert 以形成可信编译链, 最后能够编译得到 X86 等多种架构的可执行文件。

i2c 编译器的总体框架如图 2 所示, 其中, 各个翻译阶段的主要工作分别是:

(1) 词法语法分析: 对源程序进行词法和语法分析, 得到 AST。

(2) 预处理 AST: 对 AST 进行预处理, 包括禁止对常量赋值、语法糖转换、循环时数据结构的不变性检查等。

(3) 静态类型检查: 对尚未标注类型的表达式进行类型推导、检查和标注, 包括但不限于检查每个表达式的类型是否符合类型匹配的规则。

(4) 规范化 AST: 生成类型良好、具有规范形式的 AST。

(5) 区分入参出参: 区分函数的输入参数和输出参数。

(6) 找出复合类型: 找出程序中出现过的所有复合数据类型, 包括嵌套在复合数据类型中的类型。

(7) 生成内置函数: 生成复合数据类型所需要的内置函数。

(8) 替换内置函数: 将需要调用复合数据类型内置函数的语句, 翻译为对上一阶段生成的内置函数的调用。

(9) 消除复合类型: 将复合数据类型翻译为结构体类型。

(10) 出参入参合并: 将函数的输出参数合并到输入参数。

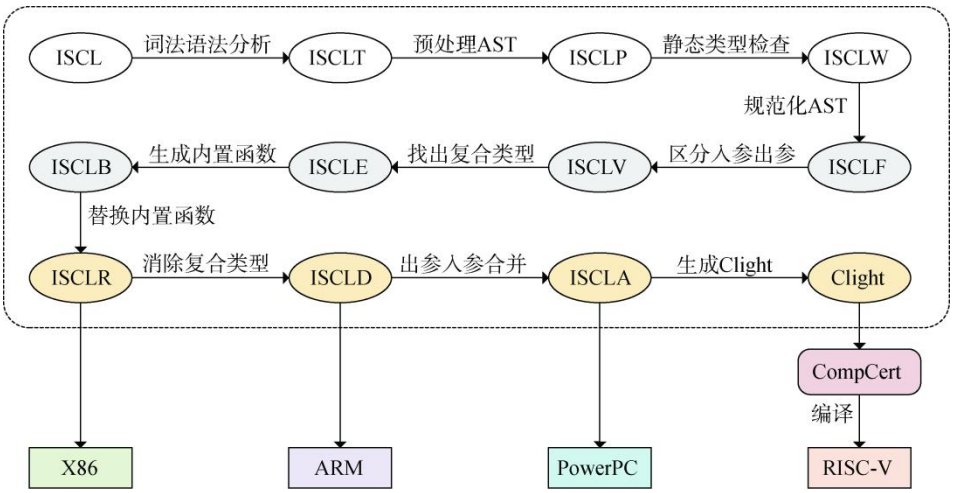

图 2　i2c 编译框架

Figure 2　Compilation framework of i2c

(11) 生成 Clight: 生成 Clight 的 AST, 将节点类型完全翻译为 Clight 语言的类型, 并将内存相关语句翻译为对 malloc 等函数的调用。

(1)~(3)是编译常见的前期阶段, (4)是为之后的翻译证明做准备。isCL 语言与 Clight 语言在语法上的两个重要区别是函数参数和复合数据类型。Clight 语言只支持单返回值, 然而为了进行标识函数是否正常退出等工作, 需要生成多个函数返回值, 因此需要将输出参数合并到输入参数, (5)、(10)用于参数合并; Clight 语言只支持结构体类型, 因此需要将前述的复合数据类型翻译成结构体类型, 并且在翻译过程中生成自定义的内置函数, (6)~(9)用于复合数据类型和内置函数的生成。(11)是翻译到 Clight 语言的过渡阶段。

虚线所表示的前 4 个翻译阶段暂时不作为形式化验证的主要任务, 因为词法、语法分析的过程往往验证困难, 这些阶段又不易出错, 因而 ISCL~ISCLW 中间语言无需语义定义。从 L2C 项目[21]的实践经验得知逆序证明能够降低反复修改证明的可能性, 因此本文同样从后向前地进行证明。

图 2 所示的(5)~(8)阶段以基于对象的方式进行语义操作, 允许编译器以高度抽象的方式处理数据, 简化类型检查和语义规则, (9)~(11)阶段则是基于 CompCert 内存模型实现语义, 从而最终翻译到 Clight 语言并对接 CompCert 编译器。目前已经完成证明的阶段是基于 CompCert 内存模型的中间语言翻译阶段。

下面用图 3 所示的示例程序来简要解释各个阶段的工作。程序定义了一个名为 Cargo 的结构体, 用于在区块链上记录货物信息。结构体中包含 3 个属性: name 表示货物名称, quantity 表示货物数量, isDelivered 表示货物是否交付。所有货物记录为一个 List[Cargo]类型的状态变量 *Cargos*, markDelivered 函数用于标记货物状态为已交付。该程序经过翻译得到的部分 Clight 程序如图 4 所示, 接下来以此说明各个阶段的翻译工作。

```
1:  /* 货物名称, 货物数量, 货物是否交付 */
2:  struct Cargo{
2:      name: String,
3:      quantity: Int,
4:      isDelivered: Bool
5:  };
6:
7:  /* 存储货物列表 */
8:  storage cargos: List[Cargo];
9:
10: /**
11: * 将指定货物标记为已交付
12: * @param name 要标记为已交付的货物名称
13: */
14: func markDelivered(name: String) = {
15:     for (c in cargos){
16:         if(name == c.name){
17:             c = c.isDelivered.set(true);
18:             return;
19:         }
20:     }
21: }
```

图 3　isCL 示例程序

Figure 3　A isCL sample program

首先, 前 3 个阶段从 isCL 程序中生成 AST 并进行必要的预处理和检查。此后的翻译阶段基本上都是在遍历 AST 的基础上记录信息, 或者修改 AST 的节点、结构以生成新的 AST。

规范化 AST 阶段进行核心翻译的预备工作, 生成类型良好的 AST。包括但不限于简化表达式的定义、针对每个本地变量生成初始化语句、将状态变量直接翻译为全局变量。例如, 图 4 第 18 行是状态变量翻译后得到的全局变量 *Cargos*。

```
1:  typedef struct {
2:      int _size;
3:      signed char *_data;
4:  } _string;
5:
6:  typedef struct {
7:      _string name;
8:      int quantity;
9:      _Bool isDelivered;
10: } _Cargo;
11:
12: typedef struct {
13:     int _capacity;
14:     int _size;
15:     _Cargo *_data;
16: } _list_Cargo;
17:
18: _list_Cargo cargos;
19:
20: void _init_string(_string *_a1) {
21:   (*_a1)._size = 0;
22: }
23:
24: void _init_Cargo(_Cargo *_a1) {
25:     _init_string(&(*_a1).name);
26:     (*_a1).quantity = 0;
27:     (*_a1).isDelivered = 0;
28: }
29:
30: void _equals_string(_Bool *_a1, _string *_a2, _string *_a3) {
31:     (*_a1) = 0;
32:     if (! ((*_a2)._size == (*_a3)._size)) {
33:         return;
34:     }
35:     $5 = 0;
36:     for (; 1; $5 = $5 + 1) {
37:         if (! ($5 < (*_a2)._size)) {
38:             break;
39:         }
40:         if (! ((*_a2)._data[$5] == (*_a3)._data[$5])) {
41:             return;
42:         }
43:     }
44:     (*_a1) = 1;
45: }
46:
47: ……
48:
49: void markDelivered(_string *name) {
50:     int _i;
51:     _Bool _f_tmp1;
52:     _i = 0;
53:     for (; 1; _i = _i + 1) {
54:         if (! (_i < cargos._size)) {
55:             break;
56:         }
57:         _equals_string(&_f_tmp1, name, &cargos._data[_i].name);
58:         if(_f_tmp1) {
59:             cargos._data[_i].isDelivered = 1;
60:             return;
61:         }
62:     }
63: }
```

图 4　示例程序翻译而来的部分 Clight 程序

Figure 4　Part of Clight program translated from the sample program

区分入参出参阶段中, 如果一个函数中的所有语句里出现的表达式是该函数的输入参数和输出参数, 则在 AST 中标记它们, 例如图 4 第 57 行的 *name* 被标记为当前函数的输入参数, 为之后输出参数合并到输入参数做准备。

找出复合类型阶段遍历整个 AST 找出复合数据类型, 例如 String、Cargo 和 List[Cargo]。

生成内置函数阶段针对上一阶段找出的所有复合数据类型, 生成事先确定的内置函数。图 4 列举了部分内置函数, 包括_init_string、_init_Cargo 和_equals_string, 它们分别用于初始化 String、Cargo 类型的变量, 以及判断两个字符串是否相等。

替换内置函数阶段将需要调用内置函数的语句翻译为对上一阶段生成的内置函数的调用语句。例如第 57 行需要调用_equals_string 函数, 在此阶段以前的 AST 上, 它被定义为需要调用_equals_string 函数的语句, 这类语句还不是函数调用语句, 在语法上不包含_equals_string 函数的具体执行(但其语义与_equals_string 函数执行的语义完全等价), 在此阶段以后, 它被翻译为函数调用语句, 所调用的函数就是上一阶段生成的_equals_string 函数, 语法上自然可以展开为_equals_string 函数里事先确定的每一条语句。

消除复合类型阶段将所有数据类型一一对应地翻译为 Clight 语言的数据类型, 特别是将复合数据类型翻译为结构体类型。例如, String 类型翻译得到_string 结构体类型, 结构体的成员包括记录字符串大小的整数类型变量 *_size*, 以及记录字符串实际内容的字符指针类型变量 *_data*。除此以外, 这一阶段还会翻译语句中存在的输入参数和输出参数。对于输入参数, 根据类型是否是复合数据类型决定是否翻译为解引用形式, 对于输出参数则全部翻译为解引用形式, 这使得函数内部可以修改相应变量的值。例如_equals_string 函数中输入参数具有复合数据类型 String, 因此这个阶段将该函数语句中的输入参数翻译为(*_*a2*)、(*_*a3*), 输出参数 *_a1* 翻译为(*_*a1*)。

出参入参合并阶段把函数的输出参数合并到输入参数。

生成 Clight 阶段将各种类型翻译至完全符合 Clight 定义的类型, 还将内存相关语句翻译为对 malloc 函数等的调用, 以符合语法并完成翻译。

5　翻译证明

本节针对已完成证明的3个翻译阶段, 即消除复合类型、出参入参合并和生成Clight阶段, 介绍基于CompCert内存模型的中间语言翻译阶段的语法定义、语义环境和语义规则, 特别是翻译前后的主要变化, 并介绍语义保持性定理, 最后给出语义保持性的证明思路。由于目标语言是Clight

语言, 许多细节可以参考Clight语言相关语法及其大步语义定义[22]。

5.1 语法定义、语义环境

从语法定义的角度来说, ISCLR 与 ISCLD 的主要区别是程序语法结构和语义环境。

ISCLR 的程序语法结构如下:

$$\begin{gathered}\mathrm{prog_R} ::= (\mathrm{pmain},\mathrm{ptypes_R},\mathrm{pdefs_R})\\ \mathrm{pmain} ::= \mathrm{id}\\ \mathrm{ptypes_R} ::= \mathrm{type}*\\ \mathrm{pdefs_R} ::= (\mathrm{id} \rightarrow \mathrm{globdef_R})*\\ \mathrm{globdef_R} ::= \mathrm{func_R} \mid \mathrm{type}\end{gathered}$$

pmain 指明了 main 函数的 id, ptypes 记录了程序中找到的所有复合数据类型。$\mathrm{pdefs_R}$ 是一个列表, 记录了全局变量 id 与变量类型的对应关系, 以及函数 id 与函数定义 $\mathrm{func_R}$ 的对应关系。

ISCLD 的程序语法结构如下:

$$\begin{gathered}\mathrm{prog_D} ::= (\mathrm{pmain},\mathrm{public},\mathrm{ptypes_D},\mathrm{pdefs_D},\mathrm{cenv})\\ \mathrm{public} ::= \mathrm{id}*\\ \mathrm{ptypes_D} ::= \mathrm{composite\_definition}*\\ \mathrm{pdefs_D} ::= (\mathrm{id} \rightarrow \mathrm{globdef_D})*\\ \mathrm{globdef_D} ::= \mathrm{func_D} \mid \mathrm{type}\\ \mathrm{cenv}:: = (\mathrm{id} \rightarrow \mathrm{composite})*\end{gathered}$$

ISCLD 的程序语法结构与 Clight 语言一致, public 记录了公共函数和全局变量的 id。$\mathrm{ptypes_D}$ 是结构体类型或枚举类型的定义列表, cenv 是根据 $\mathrm{ptypes_D}$ 构造出来的环境(下称复合环境), 用于记录到复合数据类型定义 composite 的映射。composite 表示 Clight 语言中结构体或联合体, 指明当前类型是结构体还是联合体、其所拥有成员的具体信息、占用空间大小等, 内容比 composite_definition 丰富。消除复合类型阶段就是要根据自定义的 5 种复合数据类型翻译构造出相应的复合环境。

ISCLD 的程序语法结构与 ISCLA 十分类似, 区别在于函数语法结构:

$$\begin{gathered}\mathrm{func_D} ::= (\mathrm{params_D},\mathrm{return_D},\mathrm{vars_D},\mathrm{temps_D},\mathrm{body_D})\\ \mathrm{func_A} ::= (\mathrm{params_A},\mathrm{vars_A},\mathrm{temps_A},\mathrm{body_A})\end{gathered}$$

即经过出参入参合并阶段, 输入参数 $\mathrm{params_D}$ 和输出参数 $\mathrm{return_D}$ 合并为输入参数 $\mathrm{params_A}$。

在语义环境方面, ISCLR 的全局环境 ge、本地环境 e 和临时环境 le 分别定义为:

$$\begin{gathered}\mathrm{ge_R} ::= \mathrm{genv_R}\\ \mathrm{genv_R} ::= (\mathrm{globalenv}\ \mathrm{func_R}\ \mathrm{type})\\ \mathrm{e}:: = \mathrm{id} \rightarrow (\mathrm{block},\mathrm{type})\\ \mathrm{le}:: = \mathrm{id} \rightarrow \mathrm{val}\end{gathered}$$

globalenv 可以看作是构造全局环境 genv 的一种方法。全局环境 $\mathrm{ge_R}$ 将函数与全局变量的 id 映射到其内存位置, 再将内存位置进一步映射到定义。本地环境 e 将 id 映射到数据的内存位置和类型, 临时环境 le 将 id 直接映射到值。

ISCLD、ISCLA 的语义环境与 Clight 定义完全一致, 以 ISCLD 为例, 与上述 ISCLR 环境的差别主要在于:

$$\begin{gathered}\mathrm{ge_D} ::= (\mathrm{genv_D},\mathrm{cenv})\\ \mathrm{genv_D} ::= (\mathrm{Genv.t}\ \mathrm{func_D}\ \mathrm{type})\end{gathered}$$

全局环境中增加的复合环境 cenv 就是上述程序语法结构中的 cenv。

上述的语法区别将自然而然地反映在各个中间语言的语义规则里, 接下来围绕它们展开证明。

5.2 语义保持性定理

这里给出总体的程序翻译的语义保持性定理。

定理 1. 程序翻译的语义保持性.

$$\begin{gathered}\forall \mathrm{prog1},\mathrm{prog2},\mathrm{main},\mathrm{m}.\\ \mathrm{trans\_prog}(\mathrm{prog1}) = \mathrm{prog2} \wedge\\ \mathrm{exec\_prog\_before}(\mathrm{prog1},\mathrm{main},\mathrm{m}) \rightarrow\\ \mathrm{exec\_prog\_after}(\mathrm{prog2},(\mathrm{trans\_fundef}(\mathrm{main})),\mathrm{m}).\end{gathered}$$

其中, trans_prog 是程序翻译函数, trans_fundef 是函数翻译函数, exec_prog_before、exec_prog_after 分别是前后两种中间语言的语义求值函数。定理表示, 如果是从同种内存状态 m 出发, 一个函数 main 在程序 prog1 的上下文中执行, 与在经过翻译后的程序 prog2 的上下文中执行, 对于语义环境与内存状态的改变一致。

要证明这一语义保持性, 就需要证明函数调用翻译以及语句翻译的语义保持性。这二者是互归纳定义的, 因此需要互归纳的证明。这里以 ISCLR 翻译到 ISCLD 的消除复合类型阶段为例, 简洁起见, 仅给出语句翻译的语义保持性定义:

定理 2. 语句翻译的语义保持性.

$$\begin{gathered}\forall \mathrm{e1},\mathrm{e2},\mathrm{le},\mathrm{le'},\mathrm{m},\mathrm{m'},\mathrm{s1},\mathrm{s2},\mathrm{t}.\\ \mathrm{ge1} \vdash \mathrm{ISCLR.exec\_stmt}(\mathrm{e1},\mathrm{le},\mathrm{m},\mathrm{s1},\mathrm{t},\mathrm{le'},\mathrm{m'},\mathrm{out1}) \wedge\\ \mathrm{trans\_stmt}(\mathrm{s1}) = \mathrm{s2} \wedge \mathrm{trans\_out}(\mathrm{out1}) = \mathrm{out2} \wedge\\ \mathrm{env\_match}(\mathrm{e1},\mathrm{e2}) \wedge \mathrm{env\_none}(\mathrm{e1},\mathrm{e2}) \rightarrow\\ \mathrm{ge2} \vdash \mathrm{ISCLD.exec\_stmt}(\mathrm{e2},\mathrm{le},\mathrm{m},\mathrm{s2},\mathrm{t},\mathrm{le'},\mathrm{m'},\mathrm{out2}).\end{gathered}$$

其中, trans_stmt 是语句翻译函数, t 代表程序的事件轨迹。env_match 和 env_none 共同定义了本地环境的匹配关系, 前者表示若根据 id 在 e1 找到数据的内存位置与类型, 则在 e2 能找到同样的内存位置与对应类型, 后者表示 e1 中不存在的映射, 在 e2 中同样不存在。该引理表示, 如果在全局环境 ge1 和本地环境 e1 下, 使用 ISCLR 语义求值规则执行语句 s1, 临时环境由 le 变为 le', 内存状态由 m 变为 m', 得到执行结果 out1, 并且本地环境 e1 与 e2 匹配, 则在 ge2 和 e2 环境下, 在 ISCLD 的语义求值规则下执行语句 s2, 临时环境、内存状态发生了相同的改变, 得到相匹配的执行结果 out2。

函数调用翻译的语义保持性是类似的。得证以后, 就能直接在程序翻译的语义保持性定理的证明上应用。更进一步地, 由于函数和语句定义中包含表达式, 自然也要证明表达式翻译的语义保持性, 像这样层层递进, 最终实现证明。

5.3 语义规则

从语义规则的角度来看, ISCLR 与 ISCLD 的主要区别是复合数据类型表达式与结构体类型表达式的语义规则不同, ISCLD 与 ISCLA 的主要区别是函数调用语句语义规则不同。

ISCLR 的复合数据类型表达式的语义求值规则与 ISCLD 结构体类型表达式的语义求值规则相对应。二者的不同体现在如何计算成员的偏移。以 String 类型为例, String 类型具有记录字符串大小和记录字符串实际内容的变量所对应的成员, 分别用 SIZE、DATA 表示, ISCLR 上的语义规则如(1)所示。

$$\frac{\begin{array}{c}ge,e \vdash (expr,String) \Rightarrow (blk,ofs)\\ field\_offset_R(i,fields) = Delta\\ fields = (SIZE,Tint)::(DATA,Tpointer\ Tchar)::nil\end{array}}{ge,e \vdash (expr.i,ty) \Rightarrow (blk,ofs + delta)} \quad (1)$$

规则表示, ISCLR 想要得到表达式 expr 的成员 i 的内存位置与偏移量, 则需要首先在全局环境 ge 和本地环境 e 中求值表达式 expr, 得到它的内存位置 blk 与偏移量 ofs, 接着根据成员 i 的标识从类型内部结构 fields 中找到偏移量 delta, 从而得到成员的准确位置。具体地说, 如果成员 i 是 SIZE, delta 为 0 字节, 如果成员 i 是 DATA, 则 delta 为 8 字节(经过指针类型对齐计算得到)。相应地, 成员的类型 ty 是整数类型或者字符指针类型。而 ISCLD 上的语义规则如(2)所示。

$$\frac{\begin{array}{c}ge,e \vdash (expr,(Struct\ id\ att)) \Rightarrow (blk,ofs)\\ ge \vdash id \Rightarrow composite\\ field\_offset_D(i,composite) = delta\end{array}}{ge,e \vdash (expr.i,type) \Rightarrow (blk,ofs + delta)} \quad (2)$$

二者不同之处在于, expr 的类型变为结构体类型, id 是结构体的标识, 根据 id 可以从全局环境 ge 中找到对应的 composite, 进而根据 composite 的内容计算出偏移量 delta。

ISCLD 与 ISCLA 的函数调用语义规则相对应。ISCLD 的函数调用语义规则如(3)所示。规则表示, 首先在全局环境 ge 中, 根据 id 取得对应的函数定义 fundef, 再根据环境与内存状态, 得到输入参数表达式列表 al 的求值结果 vargs, 输出参数表达式列表 rl 求值结果 vrets, 并以所有求值结果作为函数参数对应的值执行函数, 执行以后, 内存状态由 m 变为 m'。

$$\frac{\begin{array}{c}ge \vdash id \Rightarrow fundef\\ ge,e \vdash (m,al) \Rightarrow vargs\\ ge,e \vdash (m,rl) \Rightarrow vrets\\ ge \vdash (m,fundef,vargs +\!\!+ vrets) = m'\end{array}}{ge,e \vdash (m,eval\_scall_D(id,al,rl)) \Rightarrow m'} \quad (3)$$

函数调用语义规则在 ISCLA 里如(4)所示。

$$\frac{\begin{array}{c}ge \vdash id \Rightarrow fundef\\ ge,e \vdash (m,al) \Rightarrow vargs\\ ge \vdash (m,fundef,vargs) = m'\end{array}}{ge,e \vdash (m,eval\_Scall_A(a,al)) \Rightarrow m'} \quad (4)$$

可以看到, 将输出参数全部合并到输入参数以后, 输入参数列表是 al。类似的, 规则得到函数定义以后, 以 al 的求值结果作为参数对应的值, 执行函数并改变内存状态。简化表达的规则里没有写出函数的具体类型, 实际上经过合并, 函数类型中的输出参数类型变为 void 类型。

5.4 消除复合类型阶段的证明

在消除复合类型阶段里, 复合类型翻译为结构体类型, 因此需要证明复合类型与结构体类型表达式的成员访问语义规则的语义保持, 通过应用复合类型转换引理和成员偏移等价引理来实现。

复合类型转换引理的含义是, 对于任意类型, 如果翻译以前存在于程序的$ptypes_R$里, 说明这是复合类型, 因此翻译以后存在与之对应的composite, 其所记录的成员和该类型本身应当拥有的成员是相同的。

引理 1. 复合类型转换引理.

$$\begin{array}{l}\forall ty,defs.\ composite\_type\ ty \wedge\\ In(trans\_composite\_definition\ ty)\ defs \rightarrow\\ su,m,a.\\ In(Composite(trans\_type\_to\_ident\ ty)\ su\ m\ a)\ defs \wedge\\ su = Struct \wedge a = noattr \wedge m\\ = (composite\_members\ ty).\end{array}$$

证明思路: 首先对假设和类型ty进行分解, 在前提中展开trans_composite_definition函数的定义。通过分解情况分析找到复合类型定义, 定义里的类型是结构体, 无额外属性, 并且结构体类型拥有ty所对应的成员, 从而证明复合类型定义的存在性, 得证。

成员偏移等价引理要证明的是翻译前后计算成员对齐和大小的结果等价。翻译以前, 由于ISCLR里并不存在复合数据类型的全局环境, 计算类型对齐、大小的函数是直接根据该类型的构造进行计算的, 例如String类型的对齐确定为8字节。翻译以后, ISCLD里的全局环境里存在着类型所对应的composite, 其成员co_alignof记录了String类型的大小, 具体构造方式可以参见Clight语言类型表达式定义相关的模块[51]。由于计算大小函数依赖于计算对齐函数, 这里以alignof函数为例给出等价性定理以

及证明。引理3定义了ISCLR的alignof函数和ISCLD的Ctypes.align函数的等价性, 引理4定义了ISCLR的alignof_fields函数和ISCLD的align_composite函数。它们在Coq中应当以互归纳的形式描述和证明:

引理 2.align_of 函数与 Ctypes.align_of 函数的等价性.

∀ty. alignof(ty)
= Ctypes.alignof(cenv,(trans_type ty)).

引理 3.alignof_fields 函数与 align_composite 函数的等价性.

∀fld. alignof_fields(fld)
= Ctypes.align_composite(cenv,(trans_member fld)).

证明思路: 首先, 需要知道复合环境是如何构造的。由于构造环境函数 build_composite_env 的返回类型是泛型数据类型 res, 存在成功或者失败出错两种情况。经过翻译的程序应用 inversion 策略进行反推, 展开程序的具体结构, 然后使用 eq_refl 和 case 策略, 讨论两种情况。失败情况容易得证, 成功时可以得到复合环境和构造函数的关系, 说明复合环境是 build_composite_env 函数在执行成功情况下的返回值。接着, 在类型 ty 上归纳证明。通过静态类型检查阶段, 得知所有复合数据类型都是 isCL 语言支持的五种类型之一, 从而对每个类型进行讨论。由于任意一个类型都会翻译成 composite 并形成一个列表, 复合环境由该列表构造而成, 从而满足 Ctypes 提供的 build_composite_env_charact 定理。应用定理可得, 任意一个复合数据类型的具体构造与复合环境的关系满足 Ctypes 的一致性定理, 利用一致性定理得到类型对齐与根据成员进行计算得到的对齐结果一致。得证。

这一阶段除了需要证明消除复合数据类型的正确性以外, 还需要证明将输入、输出参数表达式翻译为解引用类型的正确性。由于在翻译以前, 表达式的语义求值规则直接定义为当前表达式的解引用类型的语义求值规则, 因此二者语义执行规则自然一致。

5.5 出参入参合并阶段的证明

出参入参合并阶段里, 所有输出参数合并到输入参数列表。翻译以前, 函数调用语句进行语义执行时, 需要分别求值输入参数和输出参数, 而翻译以后则直接求值输入参数。因此对于任何一个函数, 函数的所有语句执行完成以后环境、内存改变与参数合并前的环境、内存改变是一致的。这个证明过程中用到的引理包括但不限于程序复合环境匹配引理、表达式拼接的语义一致性引理等。

引理 4. 程序复合环境匹配.

∀prog1,prog2.
trans_prog(prog1) = prog2 →
prog_comp_env prog1 = prog_comp_env prog2.

证明思路: 首先, 分情况讨论环境构造结果的策略, 同样在目标上展开构建复合环境函数成功和失败的两种结果。在成功情况下, 将prog2反演成trans_prog(prog1), 使目标变为prog_comp_env prog1 = a, 其中a就是成功构建出来的复合环境, 然后泛化成功构建环境的前提, 在前提上重写构造成功的等式, 从而得证a和(prog_comp_env prog1)相等, 得证。在失败情况下, 由于前提是prog1成功翻译为prog2, 直接使用discriminate策略得证。

引理 5.表达式拼接的语义一致性.

∀args,rets,vargs,vrets.
eval_exprlist(args,vargs) ∧ eval_exprlist(rets,vrets)
→ eval_exprlist(args ++ rets,vargs ++ vrets).

该引理要证明的是翻译以前的输入和输出参数的语义求值结果, 与翻译以后输入参数的语义求值结果一致。也就是对于同一个语义求值函数, 表达式分别求值与表达式拼接后再求值得到结果相同。因此可以通过在前提条件上应用induction策略, 分为列表为空和列表不为空两种情况, 列表为空的情况易证, 列表不为空的情况下, 则将待证明目标的列表分为头部元素和尾部列表, 再应用归纳假设实现证明。

5.6 生成 Clight 阶段的证明

生成Clight阶段的证明比较简单。除了一些微小的不同, ISCLA的语法、语义定义已经十分接近Clight定义, 只需要从语义保持性定理出发证明即可。

6 评估

本节给出一个多方分账合约的实际应用实例, 以说明isCL语言在智能合约常见业务领域的实用性, 再对Solidity、Rust的编译器安全漏洞进行案例分析, 以说明i2c编译器所实现的可信编译流程的有效性。

6.1 智能合约应用实例

分账合约是由多方签订的对销售额进行分成的合约, 供应链上下游的风险共担是分账的典型应用场景之一。

以服装供应链为例, 如图 5 所示, 供应链包括设计师、原料供应商、成衣制造商、零售商等多个参与方。由于各参与方之间存在信息壁垒, 供应链上游企业无法及时拿到供应链下游可信的销售数据, 导致无法按需生产。例如, 某型号产品畅销时, 上游企业可能由于原料备货不足无法加量生产。除此以外, 为了达到风险共担、利益均享的目的, 各参与方之间

也希望根据事先约定的某种利益分配规则对销售金额进行分账。而建立上下游参与方的联盟链, 能够有效解决销售信息未能及时共享的问题, 建立分账合约则能够满足参与方之间按约定分账的需求。

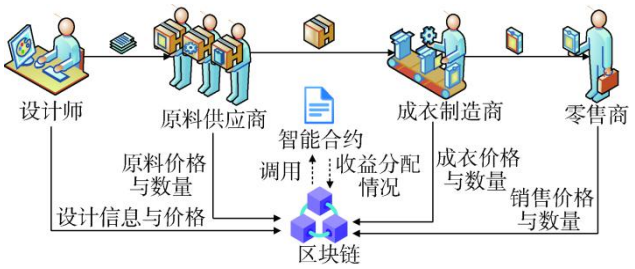

图 5　服装供应链示意图

Figure 5　Diagram of clothing supply chain

分账合约的运行流程可简化为如下两个步骤。首先, 各参与方达成关于分账比例的约定并写入智能合约。然后供应链下游零售商将销售额作为输入参数调用智能合约, 合约自动计算各方分账结果, 并写入区块链。

合约里用于按比例分成的核心函数如图6所示。实际应用场景中, 还可以根据签约时选择的分账模板进行分账、将分账结果持久化到区块链账户上。

```
/**
 * 按固定比例分账
 * @param totalSales 总销售额
 * @param remainder 存储剩余销售额的帐户名称
 * @param ratiosMap 各帐户分账的固定比例
 * @return 各帐户分得的销售额
 */
func splitFixedRatio(totalSales: Double, remainder: String,
                     ratiosMap: Map[String, Double]) : Map[String, Double] = {
    distributions: Map[String, Double];
    remain : Int = totalSales;
    account: String;
    ratio: Double;
    for (account, ratio in ratiosMap){
        amount: Double = (totalSales * ratio);
        distributions = distributions + (account -> amount);
        remain = remain - amount;
    }
    distributions = distributions + (remainder -> remain);
    return distributions;
}
```

图 6　分账合约中的一个函数

Figure 6　A function in revenue sharing contract

6.2　Solidity 案例分析

Solidity 是以太坊生态系统中最流行的智能合约语言, 它的编译器 solc 将 Solidity 源代码编译成 EVM 字节码, 其安全漏洞案例记录了一个关于 Storage 变量被错误地覆盖的问题。

如图 7 所示, 变量 a 和 b 的类型都是 32 位无符号整数, 因此它们紧凑地存储在 Storage 的同一个 256 位槽里。运行 check 函数时, 变量 a 和 x 的加法运算操作产生了溢出, 由于 0.1.6 到 0.4.3 版本的 solc 没有对溢出结果的高位进行适当的清除操作, 溢出的 1 比特被错误地写入相邻位置的变量 b 的内存空间, 导致变量 b 的值被改写为 1。

```
contract StorageOverWrite {
    //a与b相邻存储
    uint32 a = 0xFFFFFFFF;
    uint32 b = 0;
    function check() returns (uint32) {
        var x = 1;
        //a的值增加1并强制转换为uint32类型，产生溢出
        a = uint32(a + x);
        // 结果影响了b的值
        return b;
    }
}
```

图 7　导致 Storage 变量覆盖问题的 Solidity 合约

Figure 7　A Solidity contract leading to storage variable ovewrite issue

图 8 是使用 isCL 语言实现了类似行为的程序代码。前文所述的已经过形式化验证的编译流程里, 变量 a 和 x 的加法溢出始终不会改变包括变量 b 在内的任何其他变量的值, 这是因为在编译流程所基于的形式化定义的 CompCert 内存模型中, 变量 a 具有和其他变量具有不同的内存块标识符, 对一个变量的写入只影响其内存块标识符所对应的内存块内容, 不同的内存块互相之间不会受到影响。

```
storage a: Int = 2147483647;
storage b: Int = 0;
function test() = {
    x: Int = 1;
    a = a + x;
    return b;
}
```

图 8　模拟图 7 合约的 isCL 程序

Figure 8　isCL program emulating contract in Figure 7

内存模型使用映射类型定义内存结构, 以内存块标识符作为键, 以对应的内存块内容作为值。而映射类型的形式化定义里存在这样一条公理, 描述了一个键值对的更新并不影响其他键值对的内容。其中, i、j 表示不同的键, x 表示更新的值, m 表示映射集合。

公理 1. get 操作在未更新键值上的一致性.

$$\forall i,j,x,m.$$

$$i \neq j \;->\; get\ i\ (set\ j\ x\ m) \;=\; get\ i\ m.$$

因为 i2c 和 CompCert 的已证明阶段均基于 CompCert 内存模型, 且语义保持性已经得到证明, 所以不会出现 EVM 内存设计导致的相邻位置变量被错误改写这样的问题。

6.3 Rust 案例分析

Rust 是一种用途广泛的编程语言, 具有高性能、内存安全等特性, 这使其成为高效可靠地编写智能合约的选择之一。然而, Rust 官方编译器 rustc 在开发过程中难免也存在漏洞。

一个由用户提供的问题[52]表明, 1.56 等版本的 rustc 在 release 模式下编译图 9 所示的代码时发生错误。误编译问题在实践中比较普遍, 触发编译错误的代码也并不复杂。buggy 函数包含一些与具体业务逻辑相关的代码, 然而在输入数组[-1, 1]的情况下, 正确的程序输出结果是 2, 但实际输出却是 1。

```
1:  /**
2:   * 计算给定数组中符号变化的次数
3:   * @param arr 输入数组
4:   * @return 符号变化的次数
5:   */
6:  pub fn countSignChanges(arr: Vec<i32>) -> i32 {
7:      let mut prev = 0;
8:      let mut cnt = 0;
9:      for d in arr {
10:         if d > 0 {
11:             if prev < 0 { cnt += 1; } else { cnt = 1; }
12:         } else {
13:             if prev > 0 { cnt += 1; } else { cnt = 1; }
14:         }
15:         prev = d;
16:     }
17:     cnt
18: }
19:
20: fn main() {
21:     let v = vec![-1,1];
22:     let ans = countSignChanges(v);
23:     // The right answer is 2, but the output is 1 in release build.
24:     println!("{}", ans);
25: }
```

图 9 导致编译优化问题的 Rust 代码

Figure 9 Rust code leading to compilation optimization issue

造成这一问题的根本原因在于, rustc 以 LLVM 作为后端, 而 LLVM 在分析该段代码的循环结构时, 错误地识别了不变量, 过度优化原本存在的循环, 因而产生与源代码语义不同、不存在循环的目标代码。由于 rustc 的编译链没有经过形式化验证, 难以避免这类编译漏洞。

与之相比, i2c 的编译链采用可信编译技术, 将针对每个翻译阶段进行形式化验证, 编译链里 CompCert 的代码优化过程也同样经过了证明[53], 因此确保了 isCL 程序能够编译生成的语义相同的目标代码。

7 现状及未来工作

目前, 如图 2 所示的 i2c 编译器的翻译工作已经完成, 基于 CompCert 内存模型的中间语言翻译阶段的证明工作已经完成。使用 OCamllex 和 OCamlyacc 工具进行词法、语法分析, 使用交互式定理证明工具 Coq 实现各个阶段的语法语义定义、翻译和证明。

表 2 给出了目前已完成的工作量统计, 其中语法翻译工作已经完成, 因此占比最多, 在未来的工作中, 我们将在 i2c 编译器的基础上继续从后向前地对基于对象的智能合约中间语言的翻译阶段进行正确性证明, 因此工作量的增加将会集中在语义定义和证明上, 特别是语义证明。另外还将对接基于许可链 RepChain[54]开发的合约容器。并且在此基础上, 实现更多功能特性, 例如更多的内置函数、必要的外部函数调用和 Gas 统计等。

表 2 目前已完成的工作量

Table 2 The current amount of work completed

类型	行数(比例)
词法、语法分析	454(8.17%)
语法定义	724(13.04%)
语法翻译	2672(48.11%)
语义定义	667(12.01%)
基于 CompCert 内存模型的中间语言翻译阶段的语义等价性证明	942(16.96%)
其他(例如驱动程序)	95(17.1%)
总计	5554(100%)

8 结论

本文通过提出了非图灵完备的智能合约语言 isCL 及其可信编译器 i2c 的实现, 将可信编译技术有效扩展到智能合约领域。isCL 语言通过简洁的语法设计避免合约漏洞, 它具有合约应用场景下的常见数据类型与操作, 还支持复合数据类型及其内置函数。主要给出了消除复合类型、出参合并到入参这两个翻译阶段的程序、函数语法定义, 语义规则的区别以及语义保持性的证明思路, 详细说明了构造复合数据类型环境的关键证明, 还评估了 isCL 语言语法的实用性和智能合约可信编译的有效性, 最后介绍目前为止的工作量并展望未来工作。随着整个翻译流程的证明的完善, 本文提出的 isCL 语言将能够可信地编译成可执行代码, 从而使开发者可以信赖合约编译过程, 专注于合约业务逻辑的开发, 有利于促进更加便捷、可靠的智能合约开发, 从而推动区块链信任体系的进一步发展。

参考文献

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash sytem. https://bitcoin.org/bitcoin.pdf. Oct. 2018.

[2] Yuan Y, Wang F Y. Blockchain: The state of the art and future trends[J]. *Acta Automatica Sinica*, 2016, 42(4): 481-494.
(袁勇, 王飞跃. 区块链技术发展现状与展望[J]. *自动化学报*, 2016, 42(4): 481-494.)

[3] Szabo N. Formalizing and securing relationships on public networks[J]. *First Monday*, 1997, 2(9): 1-22.

[4] Buterin V. Ethereum whitepaper[Z]. https://ethereum.org/en/whitepaper.May. 2023.

[5] Slowmist. Slowmist Hacked - Slowmist Zone[Z]. https://hacked.slowmist.io/statistics/ c=all&d=all. Nov. 2023.

[6] Mehar M I, Shier C L, Giambattista A, et al. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack[J]. *Journal of Cases on Information Technology*, 2019, 21(1): 19-32.

[7] Breidenbach L, Daian P, Juels A, et al. An In-Depth Look at the Parity Multisig Bug[Z]. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug. Jul. 2021.

[8] Etherscan. Token BeautyChain[Z]. https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d. Apr. 2018.

[9] Solidity — Solidity 0.8.22 documentation[Z]. https://docs.soliditylang.org/en/v0.8.22. Dec. 2023.

[10] Vyper — Vyper documentation[Z]. https://docs.vyperlang.org/en/stable/toctree.html. Oct. 2023.

[11] Xu B, Luthra D, Cole Z, et al. EOS: An architectural, performance, and economic analysis[J]. *Retrieved June*, 2018, 11(2019): 41.

[12] List of Known Bugs — Solidity 0.8.22 documentation[Z]. https://docs.soliditylang.org/en/v0.8.22/bugs.html#known-bugs. Mar. 2023.

[13] Park D, Zhang Y, Rosu G. End-to-End Formal Verification ofEthereum 2.0 Deposit Smart Contract[C]. *Computer Aided Verification*, 2020: 151-164.

[14] Neptune mutual. Vyper Language Zero Day Exploits[Z]. https://neptunemutual.com/blog/vyper-language-zero-day-exploits/. Aug. 2023.

[15] Tolmach P, Li Y, Lin S W, et al. A Survey of Smart Contract Formal Specification and Verification[J]. *ACM Computing Surveys*, 2021, 54(7): 1-38.

[16] Leroy X. The CompCert verified compiler, commented Coq development, Version 3.13[Z]. https://compcert.org/doc. Jul. 2023.

[17] Yang X J, Chen Y, Eide E, et al. Finding and Understanding Bugs in C Compilers[C]. *The 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011: 283-294.

[18] Klein G, Nipkow T. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler[J]. *ACM Transactions on Programming Languages and Systems*, 2006, 28(4): 619-695.

[19] Kumar R, Myreen M O, Norrish M, et al. CakeML: A Verified Implementation of ML[J]. *ACM SIGPLAN Notices*, 2014, 49(1): 179-191.

[20] Bourke T, Brun L, Dagand P É, et al. A Formally Verified Compiler for Lustre[C]. *The 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017: 586-601.

[21] Shi G, Wang S Y, Dong Y, et al. Construction for the trustworthy compiler of a synchronous data-flow language[J]. *Journal of Software*, 2014, 25(2): 341-356.
(石刚, 王生原, 董渊, 等. 同步数据流语言可信编译器的构造[J]. *软件学报*, 2014, 25(2): 341-356.)

[22] Module ClightBigstep[Z]. https://certikos.github.io/compcertmc/doc/ClightBigstep.html. Feb. 2017.

[23] Dill D, Grieskamp W, Park J, et al. Fast and Reliable Formal Verification of Smart Contracts with the Move Prover[C]. *Tools and Algorithms for the Construction and Analysis of Systems*, 2022: 183-200.

[24] Schrans F, Eisenbach S, Drossopoulou S. Writing Safe Smart Contracts in Flint[C]. *The 2nd International Conference on the Art, Science, and Engineering of Programming*, 2018: 218-219.

[25] Shentu Foundation. DeepSEA[Z]. https://github.com/shentufoundation/deepsea. Mar. 2021.

[26] Bernardo B, Cauderlier R, Hu Z L, et al. Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts[C]. *Formal Methods. FM 2019 International Workshops*, 2019: 368-379.

[27] Sergey I, Nagaraj V, Johannsen J, et al. Safer smart contract programming with scilla[J]. *Proceedings of the ACM on Programming Languages*, 2019, 3(OOPSLA): 1-30.

[28] Bernardo B, Cauderlier R, Pesin B, et al. Albert, an Intermediate Smart-Contract Language for the Tezos Blockchain[C]. *Financial Cryptography and Data Security*, 2020: 584-598.

[29] Santos Reis J, Crocker P, de Sousa S M. Tezla, an Intermediate Representation for Static Analysis of Michelson Smart Contracts[EB/OL]. 2020: arXiv: 2005.11839. https://arxiv.org/abs/2005.11839.

[30] Lamela Seijas P, Nemish A, Smith D, et al. Marlowe: Implementing and Analysing Financial Contracts on Blockchain[C]. *Financial Cryptography and Data Security*, 2020: 496-511.

[31] Biryukov A, Khovratovich D, Tikhomirov S. Findel: Secure Derivative Contracts for Ethereum[C]. *Financial Cryptography and Data Security*, 2017: 453-467.

[32] Digital Asset. Daml documentation, Version 2.6.0[Z]. https://docs.daml.com. 2023.

[33] Jiao J, Kan S L, Lin S W, et al. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity[C]. *2020 IEEE Symposium on Security and Privacy*, 2020: 1695-1712.

[34] Yang Z, Lei H. Lolisa: Formal syntax and semantics for a subset of the solidity programming language in mathematical tool coq[J]. *Mathematical Problems in Engineering*, 2020, 2020(1): 6191537.

[35] Annenkov D, Nielsen J B, Spitters B. ConCert: A Smart Contract Certification Framework in Coq[C]. *The 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020: 215-228.

[36] Han N, Li X M, Zhang Q Y, et al. Executable semantics of ethereum intermediate language[J]. *Journal of Software*, 2021, 32(6): 1717-1732.
(韩宁, 李希萌, 张倩颖, 等. 以太坊中间语言的可执行语义[J]. *软件学报*, 2021, 32(6): 1717-1732.)

[37] Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers[C]. *Financial Cryptography and Data Security*, 2017: 520-535.

[38] Grishchenko I, Maffei M, Schneidewind C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts[C]. *Principles of Security and Trust*, 2018: 243-269.

[39] Hildenbrandt E, Saxena M, Rodrigues N, et al. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine[C]. *2018 IEEE 31st Computer Security Foundations Symposium*, 2018: 204-217.

[40] Pnueli A, Siegel M, Singerman E. Translation Validation[M]. Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 1998: 151-166.

[41] Clément B, Cohen A. End-to-end translation validation for the halide language[J]. *Proceedings of the ACM on Programming Languages*, 2022, 6(OOPSLA1): 1-30.

[42] Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal Verification of Smart Contracts: Short Paper[C]. *The 2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016: 91-96.

[43] Krijnen J O G, Chakravarty M M T, Keller G, et al. Translation Certification forSmart Contracts[C]. *Functional and Logic Programming*, 2022: 94-111.

[44] Song Y, Cho M, Kim D, et al. CompCertM: CompCert with C-assembly linking and lightweight modular verification[J]. *Proceedings of the ACM on Programming Languages*, 2019, 4(POPL): 1-31.

[45] Koenig J, Shao Z. CompCertO: Compiling Certified Open C Components[C]. *The 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021: 1095-1109.

[46] Kang Y X, Gan Y K, Wang S Y. Comparison of two trustworthy compilers Vélus and L2C for synchronous languages[J]. *Journal of Software*, 2019, 30(7): 2003-2017.
(康跃馨, 甘元科, 王生原. 同步数据流语言可信编译器 Vélus 与 L2C 的比较[J]. *软件学报*, 2019, 30(7): 2003-2017.)

[47] Ni Y D, Zhang C, Yin T T. A survey of smart contract vulnerability research[J]. *Journal of Cyber Security*, 2020, 5(3): 78-99.
(倪远东, 张超, 殷婷婷. 智能合约安全漏洞研究综述[J]. *信息安全学报*, 2020, 5(3): 78-99.)

[48] Praitheeshan P, Pan L, Yu J S, et al. Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey[EB/OL]. 2019: arXiv: 1908.08605. https://arxiv.org/abs/1908.08605.

[49] Slowmist. SlowMist: A Brief Analysis of the Security Risk of Uninitialized Storage Pointers in Solidity (Released in 2018)[Z]. https://slowmist.medium.com/slowmist-a-brief-analysis-of-the-security-risk-of-uninitialized-storage-pointers-in-solidity-4dde4855e254. May. 2023.

[50] Jansen M, Hdhili F, Gouiaa R, et al. Do Smart Contract Languages Need to Be Turing Complete? [C]. *Blockchain and Applications*, 2020: 19-26.

[51] Module Ctypes[Z]. https://compcert.org/doc/html/compcert.cfrontend.Ctypes.html. Nov. 2022.

[52] Wrong optimization[Z]. https://github.com/rust-lang/rust/issues/98568. Jun. 2022.

[53] Monniaux D, Six C. Formally verified loop-invariant code motion and assorted optimizations[J]. *ACM Transactions on Embedded Computing Systems*, 2023, 22(1): 1-27.

[54] Li C X, Chen S, Zheng L S, et al. RepChain—A permissioned blockchain toolkit implemented by reactive programming[J]. *Journal of Software*, 2019, 30(6): 1670-1680.
(李春晓, 陈胜, 郑龙帅, 等. 响应式许可链基础组件——RepChain[J]. *软件学报*, 2019, 30(6): 1670-1680.)

附录 isCL 语言的语法定义(EBNF)

⟨Program⟩::={⟨StructDef⟩}{⟨StorDef⟩}{⟨ConstDef⟩}{⟨FuncDef⟩}
⟨StructDef⟩::="struct"⟨IDENT⟩"{"⟨VarDecl⟩{","⟨VarDecl⟩}"}"
⟨VarDecl⟩ ::= ⟨IDENT⟩":"⟨Type⟩
⟨Type⟩::="Bool"|"Int"|"Double"|"String"|"Option""["⟨Type⟩"]"
|"List""["⟨Type⟩"]"
|"Map""["⟨Type⟩","⟨Type⟩"]"|⟨IDENT⟩
⟨StorDef⟩::="storage"⟨VarDecl⟩" = "⟨Const⟩
⟨ConstDef⟩::="const"⟨VarDecl⟩" = "⟨Const⟩
⟨Const⟩::="true"|"false"|⟨INT⟩|⟨DOUBLE⟩|⟨STRING⟩|"none"
|"some""("⟨Const⟩")"
|"("⟨Const⟩{","⟨Const⟩}")"
|"("⟨Const⟩"->"⟨Const⟩{","⟨Const⟩"->"⟨Const⟩}")"
|⟨IDENT⟩"{"⟨Const⟩{","⟨Const⟩}"}"
⟨FuncDef⟩::="func"⟨IDENT⟩"("⟨ParaDecl⟩")"[":"⟨Type⟩{","⟨Type⟩}]
" = ""{"{⟨Stmt⟩}"}"
⟨ParaDecl⟩::=[⟨VarDecl⟩{","⟨VarDecl⟩}]
⟨Stmt⟩::=⟨ConstDef⟩
|⟨VarDecl⟩[" = "⟨Expr⟩]
|⟨IDENT⟩" = "⟨Expr⟩
|"if""("⟨Expr⟩")""{"{⟨Stmt⟩}"}"
{"else if""{"{⟨Stmt⟩}"}"}
["else""{"{⟨Stmt⟩}"}"]
|"for""("⟨IDENT⟩[","⟨IDENT⟩]"in"⟨Expr⟩")"
"{"{⟨Stmt⟩}"}"
|"require""("⟨Expr⟩")"
|⟨CustomFunc⟩|"return"[⟨Expr⟩]
⟨Expr⟩::=⟨Const⟩|⟨Unop⟩⟨Expr⟩|⟨Expr⟩⟨Binop⟩⟨Expr⟩|"("⟨Expr⟩")"
|"some""("⟨Expr⟩")"|"("[⟨Expr⟩{","⟨Expr⟩}]")"
|"("[⟨Expr⟩"->"⟨Expr⟩{","⟨Expr⟩"->"⟨Expr⟩}]")"
|⟨IDENT⟩"{"⟨Expr⟩,{ ","⟨Expr⟩}"}"|⟨Expr⟩"."⟨IDENT⟩
|⟨Expr⟩"."⟨BuiltInFunc⟩"("[⟨Expr⟩{","⟨Expr⟩}]")"
|⟨CustomFunc⟩
⟨CustomFunc⟩::=⟨IDENT⟩"("{⟨Expr⟩{","⟨Expr⟩}}")"
⟨BuiltInFunc⟩::="indexOf"|"replace"|"substring"|"length"|"isEmpty"
|"get"|"set"|"add"
⟨Unop⟩::="!"|" − "
⟨Binop⟩::=" + "|" − "|" * "|"/"|"%"|"&&"|"||"|"=="|"!="|" < "|" > "|
|"<="|">="

许颖 于 2021 年在武汉大学计算机科学与技术专业获得学士学位。现在中国科学院大学计算机科学与技术专业攻读硕士学位。主要研究方向为区块链和形式化验证等。Email: xuying2022@iscas.ac.cn

张亚丰 于 2017 年在北京邮电大学信息与通信工程专业获得硕士学位。现任中国科学院软件研究所并行软件与计算科学实验室工程师。主要研究方向为区块链。Email: yafeng@iscas.ac.cn

许晶航 于 2020 年在吉林大学计算机科学与技术专业获得硕士学位。现任中国科学院软件研究所并行软件与计算科学实验室工程师。主要研究方向为区块链、自然语言处理和机器学习等。Email: jinghang@iscas.ac.cn

康跃馨 于 2019 年在清华大学计算机科学与技术专业获得硕士学位。现在中国科学院大学计算机科学与技术专业攻读博士学位。主要研究方向为智能合约、编译器和形式化验证。Email: yuexin@iscas.ac.cn

夏清 于 2022 年在中国科学院大学计算机应用技术专业获得博士学位。现任中国科学院软件研究所并行软件与计算科学实验室特别研究助理。主要研究方向为区块链链上攻击检测、智能合约安全。Email: xiaqing2018@iscas.ac.cn

袁峰 于 2006 年在中国科学院大学软件工程专业获得博士学位。现任广州软件应用技术研究院研究员。主要研究方向为物联网、大数据和人工智能。Email: yf@gz.iscas.ac.cn

左春 于 1988 年在海军工程学院电子计算机专业获得硕士学位。现任中科软科技股份有限公司研究员。主要研究方向为软件工程。Email: zuochun@sinosoft.com.cn

李玉成 于 2007 年在北京大学公共管理专业获得硕士学位。现任中国科学院软件研究所并行软件与计算科学实验室研究员。主要研究方向为并行软件与计算科学。Email: yucheng@iscas.ac.cn