计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(6):269−277 [doi: 10.15888/j.cnki.csa.009124]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

同步数据流语言输入结构体的可信翻译①

刘莛杨[1], 吴　锡[1], 杨　斐[2], 侯荣彬[2], 马　权[2], 王汝桥[2], 梁根华[2]

[1](成都信息工程大学 计算机学院, 成都 610225)
[2](中国核动力研究设计院 核反应堆系统设计技术重点实验室, 成都 610213)
通信作者: 吴　锡, E-mail: wuxi@cuit.edu.cn

摘　要: 为最大程度地减少同步数据流语言编译过程中由编译器引入的错误, 需要利用形式化方法自动生成代码, 保证编译器产生的代码能够应用于核能仪控系统. 本研究使用定理证明工具 Coq, 对同步数据流语言 Lustre 到 Clight 的主节点输入结构翻译阶段涉及的语法、语义及翻译算法进行了形式化定义, 并完成翻译算法的形式化证明. 研究表明这种经过形式化的编译器能够生成与源代码行为一致的可信目标代码, 同时生成的目标代码能够很好满足核能仪控系统的执行规范.

关键词: 同步数据流语言; 形式化验证; 仪控系统; 可信编译器

引用格式: 刘莛杨,吴锡,杨斐,侯荣彬,马权,王汝桥,梁根华.同步数据流语言输入结构体的可信翻译.计算机系统应用,2023,32(6):269–277. http://www.c-s-a.org.cn/1003-3254/9124.html

Trustworthy Translation of Synchronous Data-flow Language Input Structures

LIU Ting-Yang[1], WU Xi[1], YANG Fei[2], HOU Rong-Bin[2], MA Quan[2], WANG Ru-Qiao[2], LIANG Gen-Hua[2]

[1](School of Computer Science, Chengdu University of Information Technology, Chengdu 610225, China)
[2](Science and Technology on Reactor System Design Technology Laboratory, Nuclear Power Institute of China, Chengdu 610213, China)

Abstract: Formal methods are required for the automatic generation of codes to ensure that the code generated by the compiler can be applied to nuclear power instrument and control systems and thus minimize the errors introduced by the compiler during the compilation of synchronous data-flow languages. This study uses the theorem proving tool Coq to formally define the syntax, semantics, and translation algorithms involved in the translation phase of the master-node input structure of the synchronous data-flow language from Lustre to Clight and completes the formal proof of the translation algorithm. It is shown that this formalized compiler can generate credible target code that is consistent with the behavior of the source code, and meanwhile, the generated target code can well satisfy the implementation specifications of nuclear power instrument and control systems.

Key words: synchronous data-flow language; formal verification; instrument & control system; certified compiler

同步数据流语言目前在航空航天、轨道交通、核电能源等领域的安全关键系统 (safe-critical system, SCS)[1] 中得到广泛应用. 这些领域相关系统的任何一个小错误, 都将会给人类的生命财产、社会生产以及生活环境带来重大影响. 如何保证构造这些系统的基础软件 (如编译器) 的可靠性, 逐渐成为该领域的研究重点.

传统的保证编译器正确性的做法大多采用对生成代码进行大量测试或对生产过程采取严格的过程管理. 但随着软件规模的增大, 此类方法逐渐变得低效; 同时

① 基金项目: 四川省科技厅科技计划 (2020JDTD0020, 2022YFG0042); 四川科技厅重大科技专项 (2019ZDZX0007, 2022ZDZX0008)
收稿时间: 2022-11-04; 修改时间: 2022-12-10; 采用时间: 2023-01-16; csa 在线出版时间: 2023-04-23
CNKI 网络首发时间: 2023-04-24

传统检测方法无法验证错误是否由编译器本身引入. 在安全关键领域中传统方法几乎已达到了其检测能力的极限. 因此需要更严苛的方法, 即形式化的方法来保证编译器的可靠.

形式化方法基于严格的数学和机械化理论, 能够对数学理论或系统设计进行建模、归约和验证, 辅助解决和改善数学以及软件安全性方面的种种问题. 何炎祥等人对编译器领域的可信软件构造理论和方法进行了深入研究, 将可信编译器的构造方式分为两类, 分别是对编译器自身进行正确性验证的“经验证的编译器 (verified compiler)”和对编译后代码进行正确性验证的“验证编译器 (verifying compiler)”[2]. 通过将编译器的执行语义抽象为数学断言, 并以定理证明的形式直接验证编译器的源语言和目标语言语义的一致性, 能够最大限度地减少“误编译”问题产生. 对编译器进行定理证明的研究中 CompCert[3] 为杰出代表, 该编译器完成了 C 语言的子集 Clight 到汇编语言的编译和证明工作. 其编译工作被划分为多个阶段, 使用辅助证明工具 Coq[4] 对每个阶段的语义保持性进行了证明, 保证编译过程的正确性. CompCert 已达到了人们所期望的最高可信程度, 其生成代码可直接用于工业生产中, 为工业界后续制定高可靠的编译标准奠定了基础.

对编译器编译后的代码进行正确性验证的方式有携带证明 PCC (proof carrying code) 和在携带证明基础上更加通用的翻译确认 (translation validation). 相较于翻译确认, 携带证明偏向于对编译后的结果进行安全性验证, 而不是对编译器本身进行证明[5]. 在携带证明的基础上, Pnueli 等人[6] 首先提出了翻译确认的机制, 该方法通过构建一种统一的语义框架来为源语言和目标语言进行建模, 并为两个语义模型构建出特定的语义等价关系, 然后设计出一种能够验证两个语义模型等价关系的确认程序. 该程序可以通过求解证明、符号计算、类型检查或静态分析等方式来确认语义模型是否等价, 如果等价则会给出证明脚本, 否则会给出反例. 基于翻译确认思想, Pnueli 实现了两种同步数据流语言到 C 的编译器的翻译确认工作[7], 这两种编译器也包含了大约 100 条优化规则, 并且证明翻译确认同样可以保证优化后的编译器的正确性.

近年来 Ngo 等人[8] 基于翻译确认的思想, 也开展了同步数据流语言 Signal 到 C 的编译器验证工作, 包括: 针对源代码和目标代码的统一语义框架 PDS (polynomial dynamical systems), 并给出两种语义的抽象时钟等价关系, 并通过 SMT 求解器来验证等价关系的一致性, 从而保证了编译器的时钟语义一致性; 然后 Ngo 等人在一种一阶逻辑公式定义的时钟模型的基础上, 开展了保持时钟语义的翻译确认工作[9,10]; 利用同步数据流语言的求值图 (SDVGs) 对翻译前后依赖关系的保持性进行了确认[10]; 并利用 SDVGs 对求值语义的保持性进行了确认[11]. 基于以上技术, 该团队提出了一种对大规模同步数据流语言编译器验证的拓展性良好的翻译确认方案, 该方案主要侧重于对同步数据流语言时钟、数据依赖以及变量求值等方面等价性的保持.

如果翻译前后两种语言的语义等价性易于构造, 或者两者的等价关于易于定义, 那么使用翻译确认是一种非常有效的验证方式. 其易于拓展的特性, 能够保证在不改动原有编译框架的基础上增加额外内容. 但是当源语言和目标语言差异较大时, 其定义工作相对较难, 同时证明过程也更加困难; 同时由于翻译确认只会关注部分性质的保持性, 其不可避免地会出现误报, 因而在工业级的开发中更多作为基于定理证明方式的补充证明.

Coq 是目前流行的定理证明工具之一. 该工具的核心是一种名为归纳构造演算的基础理论. 通过该理论, Coq 将多类型的函数式编程语言和高阶逻辑进行结合, 拥有强大的数学理论基础. 同时 Coq 在编程推理方面也有强大的表现能力, 允许用户通过构造简单项, 执行简单证明, 并在此基础上进行拓展, 直至完成完整的证明系统[4]. Coq 还提供一体化的交互式证明、编译环境和程序抽取功能, 以便用户能够及时地编写、验证定义的定理, 并将完成验证的程序抽取为 OCaml 语言. 代码的成功抽取也意味着程序性质验证成功.

目前国内外也正逐步开展基于定理证明方法的算法与软件安全性验证工作, 文献 [12] 总结了矩阵的形式化方法并完成对矩阵的分块矩阵的运算方法进行形式化定义及验证工作; 文献 [13] 对操作系统的需求层进行形式化建模, 并提出模型需要满足的相关性质, 然后对模型进行验证; 文献 [14] 对分布式系统中常用的分布式共识算法 Basic Paxos 进行了形式化地建模验证, 证明了该算法满足共识性. 目前国外基于 Lustre 语言编译器的验证工作, 是法国 Pouzet 教授团队的 Vélus 编译器[15]. 该编译器以一种基于对象的中间语言作为桥梁, 实现了一个小型但具有 Lustre 语言全部特征的子集 MiniLS 中间语言, 并基于该语言实现了一个同步数据流语言可信编译器的原型系统. 随后基于该

原型系统完成了基于对象的中间语言 Obc 到 Clight 的翻译与验证. 国内对可信的 Lustre 编译器的研究有清华大学的 L2C 编译器[16]. 相较于 Vélus, 该编译器支持更多 Lustre 特性[17]. 该编译器在完全验证单一时钟的基础上, 实现了 Lustre 嵌套时钟特性的验证, 能够应用于工业开发中. 基于定理证明的形式化证明方式工作量较大, 整体逻辑较为复杂; 当翻译前后中间语言跨度较大时, 将翻译分为多个阶段进行, 能够最大程度地保证生成代码的可靠性和安全性, 同时也能提升编译器的可扩展性. 对于一个工业级的可信编译器来说, 良好的可扩展性也能提升工业生产的效率. 基于定理证明的形式化证明方式是目前安全关键领域相关系统最理想的证明方式.

目前多数基于模型的形式化研究往往对部分现实问题进行了简化, 如 Vélus 编译器不能完全支持 Lustre 中常用的时钟、时态算子[17]. 本文基于我国安全关键领域的实际需求, 针对工业生产中参数的转化问题进行研究. 在工业生产中不可避免地需要传递大量参数, 编译器直接生成的代码如果参数过多, 会出现代码可读性差, 难以维护的问题, 同时还可能因为参数的生成错误, 导致系统出错. 因此为了增加代码的可读性, 减少维护成本, 并为后期增加对状态机特性提供支持, 需要将相应的输入参数转化为易于维护和拓展的形式, 同时对生成的输入参数进行相应的检查. 为了保证翻译修改后代码的可靠性和正确性, 并确保生成的代码能正确应用在安全关键系统之中, 转化过程需要采用定理证明的方式进行实现. 在总体方法上, 本文采用与上述文献相同的基于模型的形式化验证思想, 以面向领域的类 Lustre 语言 Lustre*为源语言, 完成与 Clight 对接; 通过交互式辅助证明工具 Coq, 将同步数据流语言编译至嵌入式控制领域常用代码过程涉及的基础数据类型、数据结构以及翻译算法进行抽象, 并以数学断言的形式给出整个翻译阶段的形式化定义.

基于以上论述, 本文从同步数据流语言的特性出发, 总结同步数据流在输入结构体翻译阶段的问题. 同时对该翻译阶段的语法、语义以及语义环境进行形式化定义, 找出该阶段需要验证的性质. 然后结合现实应用, 定义翻译算法和验证算法正确性的思路, 最后完成翻译的证明. 整个实验过程将在 Coq 中完成, 文中将给出部分定义实例. 以上研究成果将为同步数据流语言编译器的验证工作提供理论依据.

1 同步数据流语言

同步数据流语言源于工业生产中使用的响应式系统, 这些系统需要不断地与环境交互, 然后将从环境中采集的实时数据作为输入, 完成相应计算并做出对应的输出. 传统 C 语言在系统规模扩大后不再能胜任响应式系统的开发要求, 为此人们开展了同步泛型的相关研究, 并在 30 多年的研究成果, 研究出多个得到广泛应用的同步语言, 如 Esterel[18], Lustre[19,20], Signal[21,22]. 其中 Lustre 与 Signal 侧重于描写语言的数据流特征, 因此也被称为同步数据流语言.

同步数据流语言重点在于同步与数据流特性, 其中同步指语言遵循的基本原则同步假设: 系统从当前周期的数据采集任务开始到下一周期的数据采集任务之前, 必须完整的完成一次数据采集, 数据逻辑运算并得出相应输出; 数据流特性指程序中所有变量都是一个无限长的流式数据对象, 该对象中包含了变量的取值和当前周期下的时钟, 每个时钟周期中变量能否取值将由变量的时钟决定, 除明确指定外, 每个变量的默认时钟为所属节点的时钟.

在 Lustre 语言中, 主节点将作为与外界进行数据交流的窗口, 执行同步假设中数据采集任务. 并且主节点承担着调用其他节点执行逻辑运算的任务, 并输出相应结果. 基于以上原因主节点的默认时钟恒为真, 即每个周期主节点都需要从外界得到相应输入. 但不同的系统为 Lustre 程序提供输入的方式不尽相同, 且 Lustre 语言的输入参数结构与嵌入式 C 语言有较大差异. 在实际工业生产中, 采用一种合理的翻译方式来解决数据流与输入结构是一个难点.

目前 Lustre 语言已被广泛应用于安全关键领域的嵌入式软件中, 此类软件使用图形编辑器来描述图形控制算法, 并将图形控制算法以 Lustre 语言的形式输出, 如 SCADE Suit 和核能高级工程师站 NASPES (nuclear advanced software platform of engineer station software). 其中 NASPES 是由中国核动力研究设计院开发的基于模型的控制算法开发环境, 是核能高级仪控软件平台 NASPIC (nuclear advanced software platform instrument and control) 的重要组成部分, 主要应用于核电站嵌入式软件控制领域.

NASPES 集成了图形化控制算法设计、可信代码生成、控制算法调试和仿真功能. 如图 1 仪控算法开发过程所示, 核电仪控工程师首先用图形化的方式在

NASPES 的组态绘图软件 NASLAND 中对核电站控制算法进行建模, 然后使用代码生成器将图形算法翻译为 C 程序. 在实施阶段, 生成的 C 程序会被上传到 NASPES 中编译为可执行代码, 最终在 ARM48 中运行. 为提升核仪控工程师的开发效率, NASLAND 中包含了 100 多个核能领域仪控设备的基础算子模块, 除基础的与、或、非门外还包括集成多个算子的边沿脉冲发生器模块、报警模块等. 目前已设计 10 余个基于基础算子模块的核仪控算法组态.

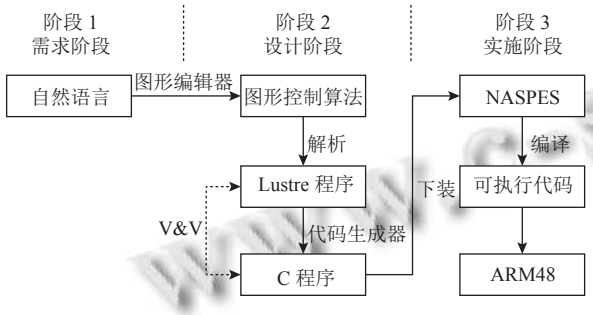

图 1 仪控算法开发过程

早期 NASPES 将图形控制算法翻译为 C 程序主要使用 SCADE 的安全代码生成器 KCG, 该生成器采用了大规模测试并遵守了严格的验证和确认过程 (V&V), 以消除错误来保证翻译的正确性, 但其依旧无法保证解决“误编译”问题. 为保证代码可靠, 有必要在代码生成器的开发过程中找到一种更严格的解决“误编译”的方法.

本文将给出一种 Lustre 语言主节点输入结构的翻译方式, 使其符合命令式语言特征, 并给出该翻译方式翻译前后语义保持性的证明方法, 保证翻译结果的可靠性, 避免“误编译”现象的发生.

2 形式化的语法和语义定义

在翻译和证明工作中, 语法和语义的定义是基础, 也是重点. 由于 Coq 无法直接验证语法和语义的正确性, 当多个问题叠加在一起时, 证明需要考虑额外的内容. 因此, 本文参考 CompCert 的设计思路, 将翻译分为多个阶段进行, 保证每个阶段解决单一问题. 如图 2 所示, 本文将整个翻译过程划分为 Lustre*、Lustre* AST、LustreR1、LustreR2 等多个中间阶段. 每个阶段根据功能不同, 设计了独立的语法和语义, 不同阶段需要通过具体的翻译通道 (如 TransMainArgs) 进行过渡. 如果翻译前后不同阶段的语法规则类似, 这些中间阶段可基于某一阶段的基础语法, 根据功能增加操作语义.

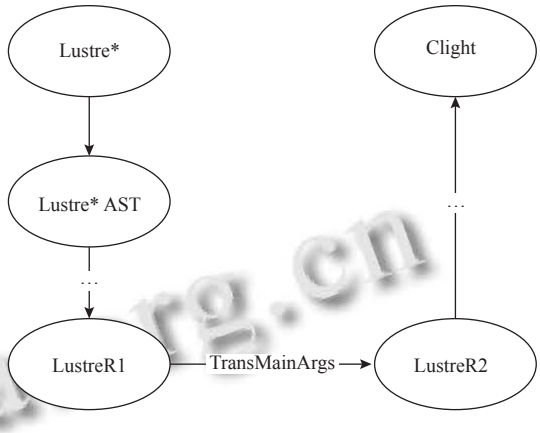

图 2 翻译过程

本文将主节点输入结构的翻译工作进行提取, 于 TransMainArgs 阶段执行. 翻译前后中间阶段 LustreR1 和 LustreR2 的语义以 LustreR 阶段语义作为基础, 并根据工作内容对语义适当调整.

2.1 抽象语法树的定义

定义翻译函数前, 需要对翻译中间过程的抽象语法树进行定义. 在 Coq 语法树将以构造子和构造内容的形式进行定义. 基于该定义, Coq 能够对源程序的语法结构进行静态检查, 保证语法结构的正确, 并将源码内容按定义方式进行提取.

在构造抽象语法树的过程中, 为保证语法树在各阶段中的通用性, 以及专注于对翻译问题的研究, 语法树的设计采用自顶而下的方式, 将其划分为 4 个层次, 分别是程序层, 节点层, 等式层以及表达式层. 各个翻译阶段程序层和节点层的定义框架大致相同, 只是语句层和表达式层定义会根据阶段做出相应调整.

程序层作为语法定义的最顶层, 主要用于在各个翻译阶段之间传递源程序携带的关键信息. Lustre 程序被划分为类型定义块, 全局常量块以及节点块, 不同阶段程序体会携带额外的内容, 但框架整体变化不大. 其抽象语法如下所示:

$$\begin{aligned}&\langle program\rangle ::=\langle decls\rangle\\&\quad\langle decls\rangle ::=\langle type\_decl\rangle\\&\qquad\qquad|\langle const\_decl\rangle\\&\qquad\qquad|\langle node\_decl\rangle\end{aligned}\tag{1}$$

节点层用于确定程序具体逻辑. 节点层中包含该程序的节点块以及相关信息. 节点块中存放了一个由

一系列节点组成的列表, 其中主节点作为程序最初的入口函数, 其他节点作为功能承担者被主节点或除自己以外的其他节点调用. 单个 Lustre 的节点由节点名输入参数, 输出参数, 局部变量定义, 语句块等主要部分和辅助证明的参数 (如节点类型) 等构成, 节点层的定义如下:

$$\begin{aligned}
\langle node_decl\rangle &::= \langle funcType\rangle\ IDENT\ (\langle decls\rangle)\\
&\quad\ \text{returns}\ (\langle decls\rangle)\ \langle body\rangle\\
\langle funcType\rangle &::= node|function\\
\langle decls\rangle &::= [\langle var_decl\rangle\ ;\ \langle var_decl\rangle]\\
\langle var_decl\rangle &::= IDENT\ \{,\ IDENT\ \}:\\
&\quad\ \langle kind\rangle\ [when\ \langle clock_expr\rangle]\\
\langle clock_expr\rangle &::= IDENT\\
&\quad\ |not\ IDENT\\
&\quad\ |not\ (IDENT)\\
&\quad\ |IDENT\ (IDENT)\\
\langle body\rangle &::= [var\langle decls\rangle]\ let\ \langle equations\rangle\ tel\ [';']
\end{aligned} \tag{2}$$

语句层的定义是对节点语句块的语句进行抽象得到, 根据阶段的不同, 语句结构也不尽相同, 但大多语句可以统一表示为: 右值表达式列表赋值给左值表达式列表的形式. 在 TransMainArgs 阶段, 语句的定义方式如下:

$$\begin{aligned}
\langle equations\rangle &::= \langle equation\rangle\ \langle equation\rangle\\
\langle equation\rangle &::= \langle lhs\rangle\ =\langle expr\rangle\\
\langle lhs\rangle &::= \langle lhs_id\rangle\ ,\ \{\langle lhs_id\rangle\}\\
\langle lhs_id\rangle &::= IDENT
\end{aligned} \tag{3}$$

与语句层类似, 表达式层的内容也会根据翻译阶段的不同而有所差异. 在本文工作阶段, 表达式主要分为两类, 一类是无法继续细分的表达式如常量表达式, 变量表达式, 输入参数表达式等; 另一类是一些基本运算操作, 结构体求成员变量表达式等. 此处仅展示常用表达式定义方式:

$$\begin{aligned}
\langle expr\rangle &::= \langle atom_expr\rangle\\
&\quad\ |\ \langle expr_list\rangle\\
&\quad\ |\ \langle temp_expr\rangle\\
&\quad\ |\ \langle unop\rangle\ \langle expr\rangle\\
&\quad\ |\ \langle expr\rangle\ \langle binop\rangle\ \langle expr\rangle\\
&\quad\ |\ \langle nary\rangle\ \langle expr\rangle\\
&\quad\ |\ \text{if}\ \langle expr\rangle\ \text{then}\ \langle expr\rangle\ \text{else}\ \langle expr\rangle\\
&\quad\ |\ \langle struct_expr\rangle\\
\langle expr_list\rangle &::= (\langle expr\rangle\ \{\ ,\langle expr\rangle\ \})
\end{aligned} \tag{4}$$

2.2 语义环境的定义

由于 Lustre 语言的数据流特性, 其语义环境的定义需要考虑周期的相关操作. 在定义过程中, 语义环境被分为全局环境和局部环境两项. 全局环境用于保存整个程序的一些关键信息, 包括所有节点的信息以及全局常量相关信息. 局部环境用于存放节点的内容信息, 局部环境分为 3 部分, 分别是用于存储节点局部信息的本地环境 *le*, 用于存放历史信息的时态环境 *te* 和用于统筹历史信息和节点调用信息的顶层环境 *e*.

本地环境 *le* 存储了每个节点在每个周期的局部变量相关信息, 时态环境 *te* 将会存储每个节点自程序运行到当前周期所有的本地环境. 在 Lustre 中, 每个节点无论是否调用均会得到执行, 因此时态环境理论上是一个无限增加的流序列, 且周期相同的每个节点的序列长度相等, 每一项的序号对应了节点信息当时所处的周期. 由于节点调用的存在, 除主节点外节点之间可相互调用, 因此顶层环境 *e* 以树型结构的方式来存储节点调用的信息. 顶层环境有两层环境, 一个是当前节点的时态环境 *te*, 一个是存储当前节点所调用的子节点内容的子环境 *e**.

图 3 中描述了一个包含 A、B、C、D 这 4 个节点的语义环境, 其中各个节点的历史环境 *te* 存放了节点在不同时刻的本地环境. A 节点作为主节点调用了 B 节点和 C 节点, 其子环境 *e**中将存放 B 节点和 C 节点的局部环境. 同时 D 节点的局部环境也将作为 C 节点的子环境, 和 B、C 节点共同组成 A 节点的子环境.

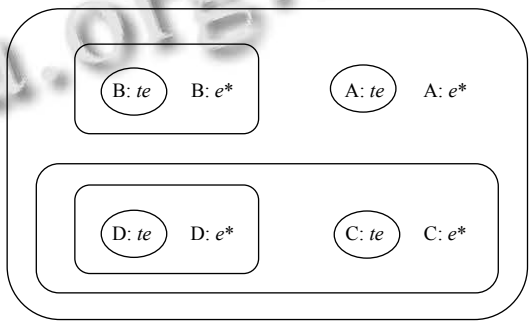

图 3 语义环境定义

2.3 操作语义定义

操作语义是对程序执行步骤的抽象, 将输入参数涉及的节点的操作语义进行形式化描述, 作为后续翻译过程的正确性的证明的基础. 一段能完整地能执行功能的代码, 在执行初时需要为所有变量分配对应的内存空间, 使变量能保存数据并基于得到的数据执行

计算逻辑. 此处使用第 2.2 节定义的语义环境来模拟实际分配的内存空间, 这样给变量分配内存将被转化为给变量分配环境空间.

基于以上思想, 节点的执行过程可以被形式化地定义为: 在节点执行初期, 为所有变量分配环境空间; 由于仅有输入参数能得到对应的值, 仅需要为输入参数对应的空间赋予对应的数值; 然后检查环境中的变量是否有重复. 完成以上步骤后, 执行节点定义的逻辑语句, 为所使用的变量以及返回值更新对应的数值. 最后检查是否存在类型不匹配问题和外部调用, 完成以上步骤即完成一次节点的执行过程.

$$\frac{\begin{array}{c} alloc\_variables(empty\_locenv, allvarsof(fd)) = te \\ locenv\_setvar(te,\ fd.args,\ vas) = te1 \\ ids\_norepet(fd) \\ gc \vdash (te1,\ fd,\ eh,\ se,\ eval\_stmt(fd.stmt)) \Rightarrow \\ (te2,\ eh1,\ se') \\ locenv\_getvars(te2, rets) = vrs \\ has\_types(vrs,\ rets) \\ external\_kind(fd.kind) = false \end{array}}{eval\_node(eh, se, fd, vas) = eh1,\ se',\ vrs} \tag{5}$$

3 输入结构的翻译与可信证明

3.1 输入结构翻译

TransMainArgs 阶段的工作重点在于如何将主节点复杂的输入参数以满足命令式语言规范的格式进行处理, 以及如何对节点中涉及到输入参数的变量进行更替. 在 C 语言中, 内容独立且可自定义的结构体能够很好地表达主节点复杂的输入参数, 其他程序也能以较为通用的方式获取并处理主节点的输入参数. 因此, 该阶段的翻译工作将着重于结构体的构造以及变量的替换方法.

在翻译前, 首先需要解决输入结构体类型定义问题. 在 Coq 中, 能被提取的内容需要符合抽象语法树定义的类型语法结构, 即必须为生成的输入参数结构体定义其标识符和结构体的成员变量类型. 为此, 如算法 1 所示, 首先对主节点的内容进行提取, 将主节点名作为结构体的类型名 nid, 并将原输入参数列表经过整理变换作为结构体的成员变量 ids, 然后根据 Coq 中的结构体语法定义, 生成Tstructnd.id[a.id;a.ty;b.id;b.ty⋯]形式的结构体类型 sty, 将其作为输入结构的类型, 并存放于全局类型定义模块中, 以便生成嵌入式 C 代码时进行声明.

代码 1.

```
Definition trans_body (fd: ident*node): node :=
    let b := snd fd in
    let nid := fst fd in
    let ids := map fst (nd_args b) in
    let sty := mkstruct fd in
    let mkv := trans_v ids (Svar INSTRUCT sty) in
    let s := trans_stmt mkv (nd_stmt b) in
    (mknode Node ((INSTRUCT, sty)::nil) (nd_rets b) (nd_flags b)
(nd_svars b) (nd_vars b) s b.(nd_sid) b.(nd_fld)).
```

然后需要构造新的结构体变量 mkv, 通过更新算法将新生成的输入结构体变量用于对涉及输入参数的语句进行修改. trans_stmt 函数将对主节点的语句块内容进行遍历, 当发现节点 b 中的节点语句块 nd_stmb 中有原输入参数变量时, 使用结构体变量 mkv 对原始变量进行修改.

3.2 语义等价的标准

在操作语义中, 两条命令等价的公式如下:

$$c_1 \sim c_2,\ \text{iff}\ \forall \sigma,\ \sigma' \in \Sigma.\ \langle c_1, \sigma\rangle \to \sigma' \Leftrightarrow \langle c_2, \sigma\rangle \to \sigma' \tag{6}$$

式 (6) 表示, 对于任意状态σ和σ', 当状态为σ时, 执行命令c_1得到的新状态与在状态σ下执行c_2得到的新状态相同且均为σ'时, 那么可以认为命令c_1和c_2等价. 我们将该定理拓展到编译器中, 则是确保目标语言保留了源语言所有可观测的行为 (包括执行, 终止, 分歧, 甚至是错误行为) 一致. 但事实上, 当两种语言语义有较大差距或者编译器本身对部分错误内容进行优化后, 该定义几乎无法满足. 因此需要采用一种较弱的, 但能保证语义正确执行的方案: 保证源语言和目标语言拥有安全且一致的行为, 则证明两者在语义上等价.

从形式化的角度看, 证明系统真正需要检查的是编译后的代码是否满足了某些功能规范: 即我们需要将编译前后的行为以数学断言的方式定义, 并证明这些行为是一致的. 这样从程序外部看, 程序执行得到的结果并没有发生变化, 同时也保证了编译器的安全性, 从而保证了程序的等价. 翻译过程中任意两个阶段的中间语言定义为 S 和 T (T 由 S 经过一次或多次翻译得到), 语义保持性的公式可描述为:

$$\forall P\ prop(P) \Rightarrow prop(\tau(P)) \wedge S_S(P) \approx S_T(\tau(P)) \tag{7}$$

其中, τ为一个或多个阶段组成翻译函数, 即 S 阶段的程序体 P 可经过翻译函数τ, 翻译至后续阶段的中间语言 T 的程序体$\tau(P)$; $prop(P)$ 表示程序 P 需要满足的一些性质 (如左右值不相交, 内存隔离等性质), 而每个阶

段$prop(P)$的性质会根据工作内容不同有所差异. S_S和S_T分别代指在 S 阶段和 T 阶段的执行语义, ≈表示一种单向等价关系, $S_s(P) \approx S_T(\tau(P))$意为在 S 阶段的程序体 P 对应的所有环境变量在翻译后的$\tau(P)$中都能找到匹配对象, 且 S 阶段的执行语义对环境的更改, 都可由 T 阶段的执行语义对相应的环境的改变进行等价模拟.

3.3 语义等价性的证明

基于上述过程中对整个翻译阶段内容的形式化描述, 可以通过 Coq 自带的定理证明策略, 对翻译过程的语义等价性进行证明. 语义等价性的描述本质上是对翻译可能引起的情况以性质的方式进行描述. 证明即是使用机械化的逻辑, 证明这些性质以消除由翻译引起的误解. 证明的实现则是对定义的性质进行归纳: 将较大而复杂的性质拆分为小且简单性质进行证明, 当复杂性质的所有子性质均得到证明时, 则复杂性质得到证明. 由于语义等价性标准的确立, 可以根据具体的翻译内容将语义等价命题拆分为多个部分进行证明, 此处将描述一些对等价性证明较为重要的命题证明内容.

3.3.1 翻译前后局部环境的匹配证明

通过对执行语义的描述可以看到, 环境是对内存空间的模拟. 因此对输出结果一致性的保证在一定程度上可以通过对环境空间一致性的保证体现. 在 Trans-MainArgs 阶段翻译前后局部环境的差别主要源于输入参数的改变, 因此系统为所有变量分配的环境空间可能由此出现不同. 通过 Coq 对环境能否匹配的逻辑推论过程进行形式化描述, 并通过已知条件和证明策略对该过程进行证明, 从而保证环境的一致性.

从环境的角度来看翻译前后的程序的执行过程, 首先翻译前后的程序均需要为程序中所有变量基于其类型分配环境空间. 由于翻译生成了新的结构体变量, 在分配环境的过程中, 翻译后的程序本身需要检查结构体类型以及变量是否存在, 然后依据结构体变量中成员变量数量和类型分配内存空间. 此时新结构体类型由于是通过将输出参数列表组合而成, 根据 C 语言内存分配的定义, 结构体内容的内存分配还需要满足对齐规则且大小与原参数大小一致. 满足以上规则即可保证翻译前后分配的环境空间 (内存空间) 是一致的.

基于以上推论, 局部环境匹配的推理过程可以被形式化地描述为代码 2 所示的形式化引理: 对于任意翻译前的环境 $e1$, 为程序的所有变量分配环境空间后得到新的环境空间, 此新空间中不能存在翻译后的结构体变量 INSTRUCT, 同时需要保证每一个翻译前的输入参数变量类型大小均小于等于 4 B. 那么如果存在一个环境空间 $e2$, 为该空间所有变量分配环境空间后, 每个变量都能在 $e1$ 中找到对应的变量; 且环境 $e2$ 中存在一个特殊的结构体变量, 该结构体变量被分配的空间大小和 $e1$ 中所有输入参数的空间大小相等; 同时 $e1$ 的每一个输入参数均能在结构体中找到对应的变量. 满足以上过程的推断即可保证编译前后的局部环境保持了一致性.

代码 2.

```
Lemma alloc_variables_exists:
  forall f e1, alloc_variables empty_locenv (allvarsof f) e1 ->
  forall fid, ~ In INSTRUCT (allidsof f ++ predidof f) ->
  ids_norepet f ->
  sizeof_fld (fieldlist_of (nd_args f)) <= Int.max_signed ->
  (forall id ty, In (id, ty) (nd_args f) -> 0 < sizeof ty) ->
  exists e2, alloc_variables empty_locenv (allvarsof (trans_body
(fid,f))) e2
    /\ locenv_match (fid,f) e1 e2
    /\ exists m, e2 ! INSTRUCT = Some (m,mkstruct (fid,f))
    /\ Z.of_nat (length m) = sizeof (mkstruct (fid,f))
    /\ (forall id ty, In (id, ty) (nd_args f) ->
          exists mv : mvl, e1 ! id = Some (mv,ty)
            /\ Z.of_nat (length mv) = sizeof ty).
```

该引理的证明过程则是不断地使用 Coq 提供的证明策略, 将推论拆分为多个子性质, 并构造为新的引理进行证明. 如证明翻译后的为所有变量分配空间得到的环境依旧为 $e2$ 时, 需要先保证翻译前的所有变量能正确分配空间得到 $e2$. 而所有变量的空间分配可以拆分为对输入变量、输出变量和局部变量各自的空间分配. 由于 3 种变量在翻译前未进行修改, 可以直接对三者通过内存和环境的匹配予以证明. 对于参与到翻译过程中的变量的相关推论也是相同思路. 在翻译后, 此时由于不能保证翻译后的输入结构体和输出、局部变量的隔离性, 不能简单地拆分. 因此需要通过引入新的引理, 将为输入变量和局部变量的环境分配转换到在总体环境下单独执行, 得到新的环境空间. 这样为所有翻译后的变量分配环境空间将转化为: 在已为输入变量和局部变量分配空间后的环境下, 为输入结构体分配空间, 以得到同样的环境 $e2$, 通过此方法推论得以证明并将转化为的证明的前提条件.

基于以上思想, 复杂的推论将被逐步拆分为多个简单的, 基于翻译后输入结构体性质的假设, 最后对输入结构体在环境中的存在性进行证明, 从而保证了分配参数后的局部环境在翻译前后的一致性.

根据执行语义来看, 除了分配参数后环境的一致性以外, 环境的匹配还包括输入参数绑定后的环境一致性问题, 以及翻译前后执行语义的一致性问题. 除此之外包括程序初始化的一致性等, 此类性质在推论的定义思路上与本问题类似, 因此不做过多赘述.

3.3.2 程序结果的一致性证明

通过对第 3.3.1 节提及的翻译过程中出现的不一致的语法的语义性质的定义与证明, 能够最终证明程序体在翻译前后语义的一致性, 即翻译前后输入、输入结果以及环境的变化的等价. 基于第 3.3.1 节推论的形式化思路, 对程序的执行过程和本阶段的工作内容进行抽象, 程序执行的推论形式化描述如代码 3 所示: 对翻译前的程序体 prog1 来说, 为主节点初始化后可得到对应的常量环境和自定义环境 *e*. 其中翻译前的程序体, 在为其输入对应的输入数据流 vass 后, 执行程序的逻辑运算得到对应的输出数据流 vrss. 由于该阶段的工作是输入参数结构的修改, 不涉及内容修改, 因此输入数据流对应的值与初始化后环境空间中存储的输入参数的值匹配. 而对于翻译后的程序体 prog2 来说, 在保证该程序体的主节点初始化后得到的环境与翻译前相同, 且两个程序的返回值列表相等的基础上, 如果为翻译后结构体提供与翻译前的程序相同的输入数据流, 在翻译后的程序执行逻辑运算后, 其得到的输出数据流与翻译前的程序经计算得到的输出数据流的结果一定是一致的, 即翻译行为并没有改变原程序体的语义, 从而保持了语义的一致性. 其形式化定义如代码 3.

代码 3.

```
Theorem trans_program_correct:
  forall gc e main1 vass mass vrss maxn,
  Lenv.initial_state prog1 gc init_node main1 e->
  exec_prog prog1 gc (eval_node true) main1 e 1 maxn vass vrss ->
  vargs_matchss (nd_args (snd main1)) vass mass ->
  exists main2, Lenv.initial_state1 prog2 gc init_node main2 e
    /\ nd_rets (snd main2) = nd_rets (snd main1)
    /\ nd_kind (snd main2) = Node
    /\ exec_prog1 prog2 gc (eval_node true) main2 e 1 maxn mass vrss.
```

对该过程的证明将通过 Coq 的证明策略, 以分类讨论的形式, 分解为多个覆盖范围更小的性质, 而这些性质推论多数已提前证明完毕, 如第 3.3.1 节介绍的局部环境匹配性质. 但在证明过程中可以发现, 该定理是一个整体性的推论, 需要全面考虑所有可能出现的情况, 因此还需要提出更多引理, 如当输入参数不存在时候的环境的匹配的证明. 然后将这些已证明的子定理应用于整个证明过程的假设中, 将待证明的假设逐步转化为前提, 应用于不同情况, 即可完成整个程序正确性定理的证明.

4 实验

实验总共由 3 部分组成: 首先是构造主节点输入结构的翻译程序, 使用翻译程序对近 300 个 Lustre 源程序测试用例进行测试, 保证翻译过程的基本正确性; 然后再通过交互式定理证明器 Coq 对翻译过程的语义进行保持性证明, 保证翻译过程的语义一致性; 最后使用 NASLAND 的基础算子和组态算法对翻译程序进行验证. 在对翻译过程进行语义一致性证明时可能会修改翻译过程的代码, 因此前两个实验会不断交替进行, 直到所有语义证明完成.

在 Coq 中, 如果翻译过程符合 Coq 的编码规范, 并且证明过程没有发生错误, Coq 可以通过自身的提取机制, 将翻译算法提取到 Ocaml 程序中, 最后将 Ocaml 编译为可执行的可信代码编译器.

在测试阶段, 首先将单个核仪控基础算子构造为独立的算法组态, 并利用可信代码编译器生成所有算法组态的 C 程序. 然后通过人工检查的方式, 检查代码的正确性, 并与 KCG 生成的 C 程序进行人工对比. 最后将由可信代码编译器以及 KCG 生成的 C 程序分别通过 NASPES 编译为可执行代码, 下装到核电站嵌入式设备中, 对比验证功能的正确性. 最终结果显示, 由可信代码编译器生成的 C 程序正确实现了 NASLAND 中 100 多个基础算子以及 10 余个核仪控算法组态的功能.

实验表明通过形式化的可信代码翻译程序能够生成与源代码行为一致的目标代码, 并保证目标代码可信.

5 结论

为提升编译器的可靠性, 并保证可信编译器生成的可信代码能够应用于安全关键领域的核仪控系统中. 本文运用形式化的方法对同步数据流语言 Lustre*到 C 的编译器的输入参数的翻译工作进行了形式化描述及验证. 该阶段的工作均借助辅助定理证明助手 Coq 完成.

(1) 通过对 Lustre 的语法解析, 定义了 Lustre 和中间语言 LustreR 的抽象语法树, 在 Coq 中形式化地描述的了该抽象语法树.

(2) 根据 Lustre 的特性, 定义出该语言的语义环境, 并基于语义环境形式化地演绎了节点的执行语义.

(3) 针对该阶段的主要工作内容, 形式化地定义主

节点输入参数的翻译方法, 将复杂的输入参数以结构体的方式翻译, 以满足核电仪控系统的使用要求.

(4) 定义翻译前后语义等价的判断标准, 并基于该标准完成对翻译前后执行语义一致性的形式化验证.

(5) 本研究实现了 Lustre 语言的输入参数从同步数据流模式到命令式语言的翻译, 测试通过了 100 多个核能领域仪控设备基础算子以及 10 余个核仪控算法组态, 其生成的可信代码应用于核仪控设备上. 可以为后续其他同步特征的命令式语言翻译提供参考.

参考文献

1 Knight JC. Safety critical systems: Challenges and directions. Proceedings of the 24th International Conference on Software Engineering. Orlando: Association for Computing Machinery, 2002. 547–550. [doi: 10.1145/581339.581406]

2 何炎祥, 吴伟, 刘陶, 等. 可信编译理论及其核心实现技术: 研究综述. 计算机科学与探索, 2011, 5(1): 1–22. [doi: 10.3778/j.issn.1673-9418.2011.01.001]

3 Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009, 52(7): 107–115. [doi: 10.1145/1538788.1538814]

4 Bertot Y, Castéran P. Interactive theorem proving and program development: Coq'Art: The Calculus of Inductive Constructions. Springer Science & Business Media, 2013.

5 Necula GC, Lee P. The design and implementation of a certifying compiler. ACM SIGPLAN Notices, 1998, 39(4): 612–625, [doi: 10.1145/989393.989454]

6 Pnueli A, Siegel M, Singerman E. Translation validation. Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lisbon: Springer, 1998. 151–166, [doi: 10.1007/bfb0054170]

7 Pnneli A, Shtriehman O, Siegel M. Translation validation for synchronous languages. Proceedings of the 25th International Colloquium on Automata, Languages, and Programming. Aalborg: Springer, 1998. 235–246. [doi: 10.1007/bfb0055057]

8 Ngo VC, Talpin JP, Gautier T, *et al*. Formal verification of compiler transformations on polychronous equations. Proceedings of the 9th International Conference on Integrated Formal Methods. Pisa: Springer, 2012. 113–127, [doi: 10.1007/978-3-642-30729-4_9]

9 Ngo VC, Talpin JP, Gautier T, *et al*. Translation validation for clock transformations in a synchronous compiler. Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering. London: Springer, 2015. 171–185.

10 Ngo VC, Talpin JP, Gautier T, *et al*. Formal verification of synchronous data-flow program transformations toward certified compilers. Frontiers of Computer Science, 2013, 7(5): 598–616. [doi: 10.1007/s11704-013-3910-8]

11 Ngo VC, Talpin JP, Gautier T. Translation validation for synchronous data-flow specification in the SIGNAL compiler. Proceedings of the 35th International Conference on Formal Techniques for Distributed Objects, Components, and Systems. Grenoble: Springer, 2015. 66–80, [doi: 10.1007/978-3-319-19195-9_5]

12 麻莹莹, 马振威, 陈钢. 基于 Coq 的分块矩阵运算的形式化. 软件学报, 2021, 32(6): 1882–1909. [doi: 10.13328/j.cnki.jos.006255]

13 姜菁菁, 乔磊, 杨孟飞, 等. 基于 Coq 的操作系统任务管理需求层建模及验证. 软件学报, 2020, 31(8): 2375–2387. [doi: 10.13328/j.cnki.jos.005961]

14 李亚男, 邓玉欣, 刘静. 基于 Coq 的 Paxos 形式化建模与验证. 软件学报, 2020, 31(8): 2362–2374. [doi: 10.13328/j.cnki.jos.005960]

15 Bourke T, Brun L, Dagand PÉ, *et al*. A formally verified compiler for Lustre. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Orlando: Association for Computing Machinery, 2017. 586–601.

16 尚书, 甘元科, 石刚, 等. 可信编译器 L2C 的核心翻译步骤及其设计与实现. 软件学报, 2017, 28(5): 1233–1246. [doi: 10.13328/j.cnki.jos.005213]

17 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. 软件学报, 2019, 30(7): 2003–2017. [doi: 10.13328/j.cnki.jos.005755]

18 Berry G, Gonthier G. The ESTEREL synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 1992, 19(2): 87–152. [doi: 10.1016/0167-6423(92)90005-V]

19 Pilaud D, Halbwachs N, Plaice JA. LUSTRE: A declarative language for real-time programming. Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Munich: ACM, 1987. 178–188.

20 Halbwachs N, Caspi P, Raymond P, *et al*. The synchronous data flow programming language LUSTRE. Proceedings of the IEEE, 1991, 79(9): 1305–1320. [doi: 10.1109/5.97300]

21 LeGuernic P, Gautier T, Le Borgne M, *et al*. Programming real-time applications with SIGNAL. Proceedings of the IEEE, 1991, 79(9): 1321–1336. [doi: 10.1109/5.97301]

22 Le Guernic P, Talpin JP, Le Lann JC. Polychrony for system design. Journal of Circuits, Systems, and Computers, 2003, 12(3): 261–303. [doi: 10.1142/s0218126603000763]

(校对责编: 牛欣悦)