软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3554−3569 [doi: 10.13328/j.cnki.jos.007350] [CSTR: 32375.14.jos.007350]

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

基于下推自动机的同步数据流语言可信编译*

于 涛[1], 王珊珊[1], 徐芊卉[1], 董晓晗[1], 胡代金[2], 罗 杰[1], 杨溢龙[2], 吕江花[1], 马殿富[1]

[1](北京航空航天大学 计算机学院, 北京 100191)

[2](北京航空航天大学 软件学院, 北京 100191)

通信作者: 罗杰, E-mail: luojie@buaa.edu.cn; 杨溢龙, E-mail: yilongyang@buaa.edu.cn;
吕江花, E-mail: jhlv@buaa.edu.cn; 马殿富, E-mail: dfma@buaa.edu.cn

摘 要: 同步数据流语言 Lustre 是安全关键系统开发中常用的开发语言, 其现存的官方代码生成器和 SCADE 的 KCG 代码生成器既没有经过形式化验证, 对用户也处于黑盒状态. 近年来, 通过证明源代码和目标代码的等价性间接证明编译器的正确性的翻译确认方法被证明是成功的. 基于下推自动机的编译方法和基于语义一致性的验证方法, 提出 Lustre 语言可信编译方法, 能够将 Lustre 语言转换为 C 语言并进行形式化验证以保证编译的正确性, 并使用 Isabelle 对翻译转换过程进行严格的正确性证明.

关键词: 同步数据流语言; 经过验证的编译器; 形式化验证; Lustre 语言

中图法分类号: TP311

中文引用格式: 于涛, 王珊珊, 徐芊卉, 董晓晗, 胡代金, 罗杰, 杨溢龙, 吕江花, 马殿富. 基于下推自动机的同步数据流语言可信编译. 软件学报, 2025, 36(8): 3554–3569. http://www.jos.org.cn/1000-9825/7350.htm

英文引用格式: Yu T, Wang SS, Xu QH, Dong XH, Hu DJ, Luo J, Yang YL, Lyu JH, Ma DF. Trusted Compilation for Synchronous Dataflow Language Based on Pushdown Automata. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3554–3569 (in Chinese). http://www.jos.org.cn/1000-9825/7350.htm

Trusted Compilation for Synchronous Dataflow Language Based on Pushdown Automata

YU Tao[1], WANG Shan-Shan[1], XU Qian-Hui[1], DONG Xiao-Han[1], HU Dai-Jin[2], LUO Jie[1], YANG Yi-Long[2], LYU Jiang-Hua[1], MA Dian-Fu[1]

[1](School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

[2](School of Software, Beihang University, Beijing 100191, China)

Abstract: The synchronous dataflow language Lustre is commonly used in the development of safety-critical systems. However, existing official code generators and the SCADE KCG code generator have not been formally verified, and their inner workings remain opaque to users. In recent years, translation validation methods that indirectly verify compiler correctness by proving the equivalence between source and target code have proven successful. This study proposes a trusted compilation method for the Lustre language, based on a pushdown automaton compilation approach and a semantic consistency verification method. The proposed method successfully implements a trusted compiler from Lustre to C and rigorously proves the translation process's correctness using Isabelle.

Key words: synchronous dataflow language; verified compiler; formal verification; Lustre language

同步数据流语言是建模语言的一种, 主要被应用于开发实时控制系统[1], 如飞控系统、车载系统等. 同步数据流语言具有数据流特性和同步特性, 因此特别适合实时控制系统的开发. 近年来出现了一些同步数据流语言, 包括 Lustre[2,3]、Esterel[4,5]、Signal[6,7]等. 其中的 Lustre 目前已经更新到 Lustre V6 版本[8].

* 基金项目: 国家重点研发计划 (2022YFB4501900)
本文由"形式化方法与应用"专题特约编辑陈明帅研究员、田聪教授、熊英飞副教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-26; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-17

SCADE 语言是一种以 Lustre 为主, 融合了 Esterel、Signal 语言特性的建模语言; SCADE suite 工具可以图形化的开发模块, 并将 SCADE 语言编译为 C 语言, 进而编译为可执行代码, 常用于在安全关键系统的建模和开发. 但是, SCADE suite 作为一个商用软件, 其编译过程对于用户不可见, 源码和目标码之间不可追溯, 因此无法对其进行深入的验证, 保证编译过程的正确性. 虽然对编译器的测试技术的研究已经发展出相当多的成果, 但测试的方法仍然无法从根本上证明正确性: 对于任意一种语言, 满足其文法规则的程序都是无限多的. 因此, 对于安全关键系统, 仍然需要引入更加可靠的方法——形式化方法进行验证.

编译器的形式化验证工作始于 1967 年, McCarthy 等人[9]证明了一个将算数表达式编译为机器语言的简单编译器的正确性, 表达式仅允许常量、变量和二进制加法. 尽管该证明面向的目标语言是一个十分简单的语言, 但其将源语言、目标语言和编译器进行形式化描述的方法对编译器的形式化验证领域有深远的影响.

可信编译器的一个重要实现是 Leroy[10]于 2009 年发布的 CompCert 编译器. 该编译器将一个 Clight (一个 C 语言的重要子集) 编译到机器码, 使用 Coq[11]对编译器进行编程并证明其正确性, CompCert 目前已经支持生成 ARM, PowerPC, RISC-V 和 x86 处理器的机器码. 如图 1 所示, 除了词法分析、语法分析和静态检查外, 整个编译过程经过 7 种中间表示, 而中间的每种变换都经过了形式化验证. 经过 Csmith 工具检查, CompCert 编译器在正确性上优于常见的 C 编译器: 它是唯一一个 Csmith 无法找到“错误代码”错误的编译器[12].

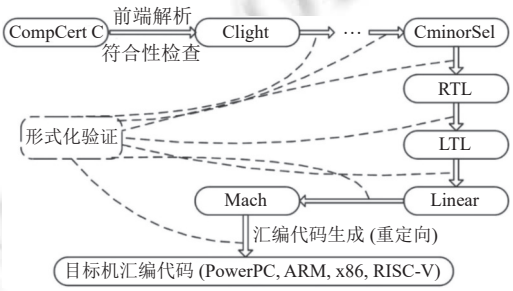

图 1 CompCert 编译器前后端架构

在同步数据流语言领域, Paulin 等人[13]于 2006 年开启了一个针对 Lustre 语言的可信编译器项目. 其团队的 Bourke 等人[14]最终于 2017 年完成了从 Lustre 语言的子集到上文提到的 CompCert 编译器的源语言 Clight 的可信编译器 Vélus. 该编译器首先将 Lustre 源程序进行规范化和调度, 形成中间表示 SN-Lustre, 并通过常规的编译程序将其翻译为 Obc 语言, 而该语言可以作为 CompCert 编译器的输入, 进而生成可靠的机器码.

L2C 编译器是清华大学 L2C 项目组于 2010 年开始的一项针对 Lustre 语言的可信编译器项目, 截至 2017 年, 该项目组已经实现了支持多时钟语言特性的 Lustre 编译器, 项目组同时维护了一个开源的 L2C 项目[15]. L2C 编译器的编译过程经历了拓扑排序、嵌套时钟消去、时态算子消去、比较运算符翻译等多个步骤, 从源代码到目标码的编译总共经历 11 种中间状态, 图 2 中用实线指代的每种变换过程都经过了 Coq 的形式化验证.

编译器验证的另外一种可选方案是翻译确认, 该方法由 Pnueli 等人[16]首先提出, 在验证同步数据流语言的翻译过程中使用了翻译确认的方法[17]. 翻译确认的方法, 不直接证明编译器的正确性, 而是证明翻译过程中的每一步转换或者优化都正确, 否则停止编译过程.

van Ngo 等人[18,19]采用翻译确认的思想实现了 Signal 到 C 语言的可信编译器. 具体来讲, 通过证明翻译前后代码的语义等价性来证明转换的正确性, 其中的证明采用定理求解器自动验证等价关系的成立.

由于安全关键系统的建模语言商用代码生成器存在不可追溯、不可验证、不可扩展的问题, 本文的研究目标是在符合 DO-178C 开发规范的环境中, 将同步数据流语言 Lustre 编译为 C 语言目标码, 并保证编译过程中的源码可追溯性. 为实现这一目标, 本文提出了一种基于下推自动机的编译方法, 该方法基于上下文无关文法, 与所编译的语言无关, 具有良好的可扩展性.

为保证编译过程的正确性和可靠性, 本文进一步提出了一种基于语义确认的形式化验证方法. 该方法能对编

译过程提供严格的验证机制, 确保源码与目标码之间的可追溯性, 达到对编译系统形式化验证的要求. 该验证方法不仅适用于当前的编译任务, 还为未来在其他语言环境中的应用奠定了基础.

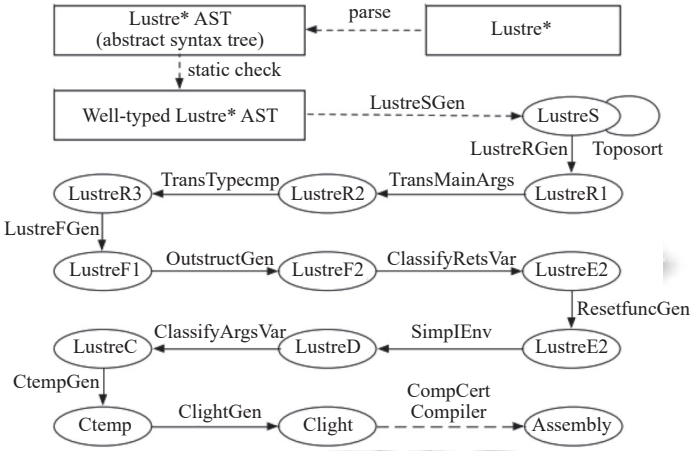

图 2 L2C 编译器翻译过程

1 源语言特性

Lustre 语言是同步数据流语言的一个重要实例, 也是 SCADE 语言的主要组成部分. Lustre 语言遵循同步数据流的基本要求: 满足同步假设、具有数据流特性、满足并行性. 除此之外, Lustre 遵循单时钟模型, 即系统中存在一个基础时钟, 在每一个周期之内都恒定为真, 而其他的时钟都严格是基础时钟的子时钟, 只在基础时钟为真的时刻才可能为真.

Lustre 语言是以赋值语句和表达式为核心的语言, 不存在显式的控制语句, 一个 Lustre 程序主要由以下几个部分构成.

(1) Program (程序): 描述了程序的整体框架.

(2) Package & Model (包和模型): 描述了包和模型的定义以及导入, 提供了某种封装性的特性, 类似 C 语言中的类.

(3) Node & Function (节点和函数): 节点和函数是 Lustre 中运行的基本单位, 其地位相当与其他语言当中的函数. 基础的 Lustre 的节点和函数定义由一条节点定义语句、一系列变量声明语句和一系列赋值语句和断言语句组成. Node 和 Function 的区别在于 Node 是有状态的, 依赖之前周期的值, 而 Function 是无状态的, 只依赖当前周期的输入.

(4) 声明语句: Lustre 的声明语句主要集中出现在节点定义之后、节点中的赋值语句之前. 声明的变量主要有 bool、int、real 这 3 种基础类型和枚举类型、用户自定义的结构体.

(5) 赋值语句和表达式: 赋值语句由左值和右值组成, 左值为已经声明的变量, 右值为表达式. 表达式中包含各种基本的运算符、Lustre 中特有的高阶运算符和时态算子. 此外, Lustre 也支持函数调用作为表达式的一部分.

下面是 Lustre 较为重要的特性, 包括时态算子、时钟算子等.

(1) 时态算子: 时态算子是一类依赖于先前周期信息的运算符, 包括 pre、→和 fby. 其中 pre 运算符是单目运算符, 获取被操作流上一个周期的值; arrow 运算符是双目运算符, 在第 1 个周期内获取左侧流的值, 在此后的周期获取右侧流的值; fby 运算符是双目运算符, 是 pre 和→的结合 (A fby B 等价于 A →pre(B)), 在第 1 个周期内获取左侧流的值, 在此后的每个周期内获取右侧流在上一个周期的值.

(2) 时钟算子: 时钟算子是一类改变流的时钟的运算符, 包括 when 运算符、current 运算符和 merge 运算符. 前面提到的例子中, 几乎所有的流在每个周期内都有值, 这个周期就是基础时钟周期. 而 Lustre 允许通过 when 运

算符为流指定不同的周期, 例如 A when ck 表示创建一个新的流, 其值为 A, 时钟周期为 ck, A when ck 只有在 ck 为真时才有值. 可以认为 A when ck 是对 A 的一个采样. current 是一个单目运算符, 作用是将一个非基础时钟周期的流填补为一个基础时钟周期的值. merge 运算符是 current 的一个扩展, 作用是将多个时钟周期互补的流组合为一个基础时钟周期的流.

(3) if 运算符: Lustre 中的 if 和其他语言中 if 条件语句相近的语法, 但其实是作为一个运算符出现在表达式当中的, 类似于 C 语言中的三目条件运算符 (?:). 例如, if A then B else C 表示若在该周期流 A 为 true, 则取 B 的值, 否则取 C 的值.

(4) 静态参数节点: 带有静态参数的节点 (或函数) 是一类未完成的函数, 需要给出静态参数的值从而实例化才能使用, 支持的静态参数类型包括类型、常量和节点. 使用类型作为静态参数类似于 C++中的函数模板, 使用节点或函数作为静态参数类似于 C++中的函数指针作为参数. 需要注意的是, 所有的静态参数需要在编译阶段就完全已知, 而不能在运行时确定.

(5) 数组迭代器: 数组迭代器是 Lustre V6 版本的新特性, 可以用迭代器来操作数组, 提供了一种 (受限制的) 高阶编程概念. 数组迭代器也可以认为是 Lustre 内置的静态参数节点, Lustre V6 中内置有 fill、red、fillred、map、boolred 这 5 种数组迭代器. 以 red 为例, red<<op, size>>(init, array) 接受两个静态参数和两个实参, 静态参数 op 表示要对数组进行的操作, 静态参数 size 表示数组的大小, 输入 init 表示初始值, 输入 array 表示需要操作的数组. 抽象地说, red 迭代器将数组映射为标量; fill 迭代器将标量映射为数组; fillred 是 fill 和 red 结合, 将数组和标量映射为数组和标量; map 将数组映射为数组; boolred 是针对 bool 类型的 red, 将 bool 数组映射为 bool 标量.

2 编译器总体框架

图 3 是可信编译系统的编译系统架构图, 以代码生成模块为核心, 主要有 3 个数据输入和 1 个数据输出.

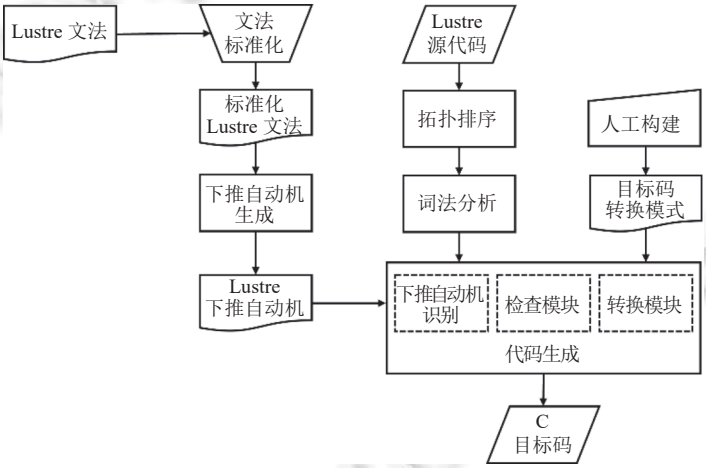

图 3 编译系统架构图

本文将 Lustre V6 的文法人工进行标准化, 得到标准化 Lustre 文法, 将其输入下推自动机生成模块, 下推自动机生成模块将上下文无关文法转换为下推自动机, 得到 Lustre 文法的下推自动机文件. 该文件在构建后即持久化, 不会在每次编译过程都重复这一过程.

目标码转换模式由人工分析 Lustre V6 和 C 语言语义手工构建, 目标码模式是消除掉语境对目标码序列的影响而获得的目标码序列的一般化表示形式, 将下推自动机识别过程和转换动作结合起来.

Lustre 源代码首先经过拓扑排序正确处理赋值语句之间的顺序, 随后经过词法分析得到词法单元流, 最终转换生成 C 目标码. 在生成代码时, 先生成当前层级代码, 然后递归生成子文法单元代码.

本文的研究对象是 Lustre 语言的 V6 版本, 参考资料主要为 Lustre V6 官网公开的官方文档和文法文档. 其中文法文档给出了 Lustre V6 完整的上下文无关文法, 文法形式如图 4 所示. 为了能够使用该文法, 首先需要对它进行标准化, 主要工作有 3 点: 消除化简 EBNF 范式、消除文法左递归、格式化符号.

Ebnf group *NodesRules*

⟨*TypedLv6IdsList*⟩ ::= ⟨*TypedLv6Ids*⟩ { **;** ⟨*TypedLv6Ids*⟩ }
⟨*TypedLv6Ids*⟩ ::= ⟨*Lv6Id*⟩ { **,** ⟨*Lv6Id*⟩ } **:** ⟨*Type*⟩
⟨*TypedValuedLv6Ids*⟩ ::= ⟨*TypedValuedLv6Id*⟩ { **;** ⟨*TypedValuedLv6Id*⟩ }
⟨*TypedValuedLv6Id*⟩ ::= ⟨*Lv6Id*⟩ (**:** ⟨*Type*⟩ | **,** ⟨*Lv6Id*⟩ { **,** ⟨*Lv6Id*⟩ } **:** ⟨*Type*⟩ | **:** ⟨*Type*⟩ **=** ⟨*Expression*⟩)
⟨*NodeDecl*⟩ ::= ⟨*LocalNode*⟩
⟨*LocalNode*⟩ ::= **node** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*Params*⟩ **returns** ⟨*Params*⟩ [**;**] ⟨*LocalDecls*⟩ ⟨*Body*⟩ (**.** | [**;**])
| **function** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*Params*⟩ **returns** ⟨*Params*⟩ [**;**] ⟨*LocalDecls*⟩ ⟨*Body*⟩ (**.** | [**;**])
| **node** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*NodeProfileOpt*⟩ **=** ⟨*EffectiveNode*⟩ [**;**]
| **function** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*NodeProfileOpt*⟩ **=** ⟨*EffectiveNode*⟩ [**;**]
| **unsafe node** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*Params*⟩ **returns** ⟨*Params*⟩ [**;**] ⟨*LocalDecls*⟩ ⟨*Body*⟩ (**.** | [**;**])
| **unsafe function** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*Params*⟩ **returns** ⟨*Params*⟩ [**;**] ⟨*LocalDecls*⟩ ⟨*Body*⟩ (**.** | [**;**])
| **unsafe node** ⟨*Lv6Id*⟩ ⟨*StaticParams*⟩ ⟨*NodeProfileOpt*⟩ **=** ⟨*EffectiveNode*⟩ [**;**]

图 4 Lustre V6 语法规则

扩展巴科斯-瑙尔范式 (extended Backus-Naur form, EBNF) 是一种主要用于描述上下文无关语法的元语法符号表示法. 相较于常见的产生式表示方法, EBNF 范式定义了许多助记符. 助记符能简化产生式, 便于书写、阅读和理解, 但是在识别算法中支持助记符会极大增加开发的难度和工作量. 考虑到扩展性, 其他语言的上下文无关文法表示并不一定采用 EBNF 范式表达, 即使同为 EBNF 范式, 表达方式也可能略有不同 (Lustre V6 官方文档中的定义方式就和 ISO/IEC 14977 中定义的 EBNF 范式不同). 权衡之下, 本文选择手动将 EBNF 范式中的助记符化简, 得到没有助记符的上下文无关文法版本. 并且为了防止在之后下推自动机识别的过程中产生无限循环, 需要消除左递归. 最后, 为了便于识别程序区分产生式中的终结符和非终结符, 将终结符书写为小写字母开头的字符串, 将非终结符书写为大写字母开头的字符串, 如表 1 所示.

表 1 标准化前后的文法产生式

Lustre 产生式	标准化后的Lustre 产生式
TypedLv6IdsList::= *TypedLv6Ids* {; *TypedLv6Ids*}	*TypedLv6IdsList* → *TypedLv6Ids TypedLv6IdsList*1 *TypedLv6IdsList*1 → ; *TypedLv6Ids TypedLv6IdsList*1 → $
TypedLv6Ids::= *Lv6Id* {, *Lv6Id*} : *Type*	*TypedLv6Ids* → *Lv6Id TypedLv6Ids*1 : *Type* *TypedLv6Ids*1 → , *Lv6Id TypedLv6Ids*1 → $

文法标准化后, 可以输入到下推自动机生成模块自动生成下推自动机. 上文提到, 上下文无关文法和下推自动机的表达能力是等价的, 每个上下文无关文法都对应一个下推自动机. 进一步的, 由于上下文无关文法的并仍然是上下文无关文法, 所以下推自动机的并仍然是下推自动机. 所以, 由下推自动机生成模块生成的下推自动机可以识别 Lustre V6 程序, 并且由多个上下文无关文法生成的下推自动机也可以合并为一个下推自动机. 因此, 既可以将整个语言的上下文无关文法一次性转化成下推自动机, 也可以分别转换文法单元再合并为一个下推自动机. 生成的下推自动机如表 2 所示, 其中箭头“→”左侧表示当前状态, 箭头“→”右侧由可能到达的状态列表组成, 列表中的每个元素为一个三元组{Q, a, PUSH/POP e}, 代表在读入字符 a 时, 到达 Q 状态, 并在栈中压入 (PUSH) 或弹出 (POP) 元素 e.

表 2 下推自动机

文法产生式	对应的下推自动机
TypedLv6IdsList::= *TypedLv6Ids* {; *TypedLv6Ids*} *TypedLv6Ids*::= *Lv6Id* {, *Lv6Id*} : *Type*	Start → [{0 Q0, *ε*, PUSH Δ}] Q0 → [{1 Q_loop, *ε*, PUSH *TypedLv6IdsList* }] Q_loop → [{2 Q_loop, ;, POP ;}, {3 Q_loop, , , POP , }, {4 Q_loop, :, POP :}, {5 End, Δ, POP Δ}, {6 Q_*TypedLv6IdsList*, *ε*, POP *TypedLv6IdsList*}, {9 Q_*TypedLv6IdsList*1, ;, POP *TypedLv6IdsList*1}, {12 Q_loop, *ε*, POP *TypedLv6IdsList*1}, {13 Q_*TypedLv6Ids*, *ε*, POP *TypedLv6Ids*}, {18 Q_*TypedLv6Ids*1, , , POP *TypedLv6Ids*1}, {21 Q_loop, *ε*, POP *TypedLv6Ids*1}] Q_*TypedLv6IdsList* → [{7 Q_*TypedLv6IdsList*1, *ε*, PUSH *TypedLv6IdsList*1}] Q_*TypedLv6IdsList*1 -> [{8 Q_loop, *ε*, PUSH *TypedLv6Ids*}, {10 Q_*TypedLv6IdsList*11, *ε*, PUSH *TypedLv6IdsList*1}] Q_*TypedLv6IdsList*11 → [{11 Q_loop, *ε*, PUSH *TypedLv6Ids*}] Q_*TypedLv6Ids* → [{14 Q_*TypedLv6Ids*1, *ε*, PUSH Type}] Q_*TypedLv6Ids*1 → [{15 Q_*TypedLv6Ids*2, *ε*, PUSH :}, {19 Q_*TypedLv6Ids*11, *ε*, PUSH *TypedLv6Ids*1}] Q_*TypedLv6Ids*2 → [{16 Q_*TypedLv6Ids*3, *ε*, PUSH *TypedLv6Ids*1}] Q_*TypedLv6Ids*3 → [{17 Q_loop, *ε*, PUSH *Lv6Id*}] Q_*TypedLv6Ids*11 → [{20 Q_loop, *ε*, PUSH *Lv6Id*}]

3 核心转换步骤

本节将以提到的 Lustre 语言的主要特性为线索来解释在转换过程中的关键节点是如何处理的.

3.1 同步数据流的并发性

Lustre 的语句并行执行, 与书写顺序无关. 然而, 语句之间的数据依赖关系是客观存在的, 例如 (1) a=1; (2) b=a; 两条语句需要按 (1)、(2) 的顺序执行. 因此, 需要对 Lustre 源程序按照数据依赖关系进行拓扑排序, 使得生成的 C 语言目标程序能够按照正确的顺序串行执行. 未经拓扑排序的源程序直接编译生成的目标代码在执行时可能产生错误或未定义的结果.

通过识别程序中语句间的依赖关系: 即一条赋值语句的右值是另一条赋值语句左值的情况, 并根据识别到的语句间的依赖关系对语句执行拓扑排序, 使得目标程序能够正确串行执行.

拓扑排序算法首先遍历所有的赋值语句, 保存赋值语句的源代码, 左值变量和右值中的变量列表和数量到 assign 结构体中. 在一条赋值语句中, 左值对右值中所有的变量构成依赖关系, 但 pre 和 fby 运算符修饰的变量除外: pre 和 fby 运算符依赖流上一个周期的值, 在上一个周期就已经计算完毕.

在遍历完程序后, 遍历 assigns 数组, 找到依赖变量数为 0 的赋值语句, 将其取出并保存到结果数组中, 并将该赋值语句的左值从所有赋值语句的依赖中去除. 以此类推, 直到取出所有的赋值语句. 这就完成了对赋值语句的拓扑排序.

3.2 目标码转换模式

目标码转换模式表示了一段 Lustre 语法单元会被转换为何种 C 语言的代码. 例如, Lustre 的 if 表达式语法单元“if < Expression1 > then < Expression2 > else < Expression3 >”可能被转换为“Expression1 > ? < Expression2 > : < Expression3 >”或者“if(< Expression1 >){temp = < Expression2 >} else {temp = < Expression3 >}”; 其中尖括号包裹的非终结符表示该文法单元在本转换步骤中不转换, 需要后续被递归地转换.

目标码转换模式由人工构建, 首先需要在充分理解 Lustre 语义的基础上找到对应的 C 语言表示; 此后需要通过大量的测试明确语法单元的具体特性和相互关系, 从而修正 C 目标码模式, 得到最终的目标码转换模式. 下面将详细说明几个语法单元的目标码转换模式构建的过程.

3.2.1 时态算子

时态算子 pre、→ 和 fby 是 Lustre 中独有的特性, 它们运算结果依赖于过去周期的信息, 因此无法简单地转换为某些 C 语言的运算符.

时态算子依赖于过去周期信息的特性主要需要在目标语言中实现 3 个功能.

(1) 生成变量保存过去周期的信息;

(2) 调用运算符时读取该变量的值;

(3) 调用结束后更新该变量的值.

对于时态运算符之间, 以及时态运算符和普通运算符之间的组合和嵌套的问题, 可以利用 Lustre 的特性进行解决. 对于 Lustre 中的每个运算符, 在运算后将生成一个新的匿名流. 例如 x+1 将生成一个时钟周期与 x 相同, 而值为 x+1 的新的匿名流. 因此, pre (x+1) 需要存储 x+1 的值; 而 pre pre x 实际上是对 pre x 这个流再做 pre 运算, 即取 pre x 的上一个周期的值. 由此, 不需要考虑存储多个周期前的值的问题.

表 3 是时态算子目标码模式, 以 function 结尾的函数为辅助函数, 在编译器内以模板的形式存在, 当一个类型的运算符被调用时, 就生成对应类型的辅助函数. ctx 是上下文 (context) 的缩写, 表示运算符依赖的之前周期的信息, 如算法 1 所示. 其中 result_ctx_type 为程序上下文信息的集合, 其中包含了所有在源程序中需要用到的上下文信息. pre_get_function 表示获取上一周期该<Expression> 的值, 借助变量 ctx (内存空间是运行时申请的). pre_set_function 计算当前周期<Expression> 的值, 并将其赋给 ctx; arrow_function 首先判断当前是否处于初始周期, 若是则返回<Expression1> 表达式的值, 否则, 返回<Expression2> 表达式的值.

表 3 时态算子目标码模式

产生式左部	产生式右部	目标码模式
Expression	pre <Expression>	temp = pre_get_function(ctx); pre_set_function(<Expression>, ctx);
	<Expression1> → <Expression2>	temp = arrow_function(<Expression1>, <Expression2>, ctx);
	<Expression1> fby <Expression2>	temp = arrow_function(<Expression1>, pre_get_function(ctx), ctx); pre_set_function(<Expression2>, ctx);

算法 1. ctx 结构体代码 (目标 C 代码).

```
typedef struct {
   int _memory;
} pre_int_ctx_type;
typedef struct {
   int test;
   pre_int_ctx_type pre_int_ctx_tab[1];
} result_ctx_type;
```

算法 2 为最终生成的 pre 运算符的 ctx 结构体和辅助函数的目标代码. pre_int_ctx_type 是 pre 运算符针对 int 型的上下文信息, 因此其_memory 成员保存了 int 型变量. 在辅助函数 pre_int_get 和 pre_int_set 中, 分别对 ctx 中的_memory 进行了存取. 若程序中有不止一个对 int 流取 pre 的操作, 则该辅助函数可以复用.

算法 2. 辅助函数代码实例 (目标 C 代码).

```
int pre_int_get(pre_int_ctx_type* ctx){
   return ctx → _memory;
}
void pre_int_set(int s1, pre_int_ctx_type* ctx){
   ctx → _memory = s1;
}
```

3.2.2 时钟算子

时钟算子 when、current 和 merge 是控制流的时钟的运算符. 其中 when 生成一个非基础周期的流, current 生成一个基础周期的流, merge 将多个非基础周期的流合并为一个基础周期的流. 这里需要指出, Lustre 中的运算符会生成一个新的匿名流, 而非改变输入的流. 例如 A when ck 将生成一个值取决于 A, 而时钟周期取决于 ck 的匿

名流, 而非将 A 的流改变为 ck.

表 4 为 3 个时钟算子的目标码模式. 其中 when 运算符的目标码为第 1 个表达式的值, 因为 when 运算符在表达式中不改变目标码, 仅告知编译器流的时钟即可. current 运算符在时钟为 false 时返回上一个有效的值, 因此依赖于先前周期的信息, 所以目标码和时态算子一样, 使用辅助函数和上下文结构体的方式实现. merge 运算符将多个互补的流融合为一个流, 因此每个周期都有一个有效值, 不依赖于先前周期.

表 4 时钟算子目标码模式

产生式左部	产生式右部	目标码模式
Expression	<Expression> when <Expression>	<Expression>
	current <Expression>	current_function(<Expression>, <Expression>, ctx)
	merge <*Lv6Id*> <Mergecaselist>	switch(<*Lv6Id*>) {< Mergecaselist>}

以 merge 运算符为例, <Mergecaselist> 在 merge 产生式的转换中不做处理, 交由<Mergecaselist> 产生式转换, 其产生式和目标码模式如表 5 所示, 本质上是填补了 C 语言 switch 语句中 case 的部分.

表 5 merge 运算符目标码模式

产生式左部	产生式右部	目标码模式
Mergecaselist	<MergeCase> <Mergecaselist>	<MergeCase> <Mergecaselist>
	<MergeCase>	<MergeCase>
	empty	—
MergeCase	(*Lv6Id* → <Expression>)	case *Lv6Id*: temp = <Expression>; break;
	(true → <Expression>)	case true: temp = <Expression>; break;
	(false → <Expression>)	case false: temp = <Expression>; break;

3.3 C 代码生成

代码生成的过程大致可分为 3 个步骤: (1) 通过下推自动机识别算法对程序进行识别的同时生成 Lustre 程序的目标码; (2) 生成 Lustre 程序所有节点和辅助函数、结构体的目标码; (3) 生成 C 目标代码主循环.

在目标码转换模式和下推自动机的基础上, 将目标码转换模式拆解为具体的转换动作, 即每个下推自动机识别过程中的状态转移动作需要执行的转换动作. 随后将转换动作嵌入到下推自动机识别算法当中, 在识别的同时就完成了 Lustre 源程序的转换.

表 6 以伪代码形式展示了 Lustre 程序与 C 程序的对应关系, Lustre 程序主要由 node 组成, 每一个 node 对应于 C 程序中的一个函数. 由于 Lustre 的输入参数与输出参数可能不止一个值, 因此在 C 程序中使用输入和输出结构体来描述各个参数的类型. *<expr>* 表达式由算术和逻辑表达式, 条件表达式, 时态表达式, 节点调用组成; 分别对应于 C 语言中的算术和逻辑表达式, 条件表达式, 条件表达式, 函数调用.

表 6 Lustre 程序及对应的 C 程序

Lustre程序	Lustre程序对应的C程序

node function(para1, para2:paraType)
returns(retPara1, retPara2:retType)
let
  <*expr*1>
  <*expr*2>
  <*expr*3>
tel
	typedef struct{
	  paraType para1;
	  paraType para2;
	}function_in;
	typedef struct{
	  retType retPara1;
	  retType retPara2;
	}function_out;
	void function(function_in *in, function_out *out){
	  <*expr*1>
	  <*expr*2>
	  <*expr*3>
	}

4 转换的正确性证明

转换的形式化验证系统如图 5 所示, 系统的输入为编译前后的 Lustre 源程序和 C 目标程序, 输出为源程序和目标程序语义一致性的证明序列, 以此说明编译系统编译的正确性. 在编译系统生成目标程序之后, 会给出 Lustre 源程序的文法单元序列以及目标程序的目标码模式序列; 同时可以获得源程序的操作语义以及目标程序的操作语义. 随后通过操作语义构建出形式为前置条件的证明前提和形式为后置条件的证明目标. 最后, 在公理系统中根据公理系统推导规则进行证明, 其中的证明序列主要针对源程序中每一条文法单元和对应目标码序列的语义一致性, 而这部分可以分别手工证明每一条孤立的文法单元和对应目标码序列. 将手工的证明结果作为引理, 就可以自动化地证明所需证明目标.

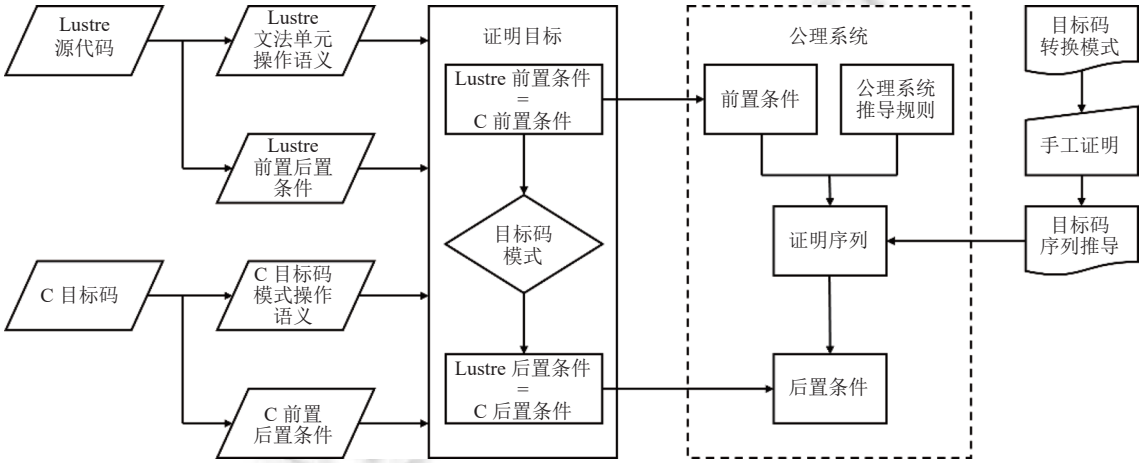

图 5 形式化验证系统架构图

证明目标的组成主要分为 3 个部分, 首先是假定 Lustre 源程序和 C 目标程序的输入相等, 即源程序中的输入变量和目标程序中的输入变量一一对应相等. 在此基础上, 需要证明的目标就是源程序的输出变量和目标程序中的输出变量一一对应相等, 而输出变量分别由源程序和目标程序根据输入变量计算得来.

4.1 文法单元和目标码模式的等价性证明

根据 Lustre 和 C 文法单元的操作语义的形式化表示, 构造证明序列证明单个的 Lustre 文法单元和对应 C 目标码序列的操作语义的等价性. 以赋值语句<*Equation*>为例, 证明过程如下. 首先将证明目标分解为数个命题, 包括 Lustre 和 C 的赋值语句定义命题, Lustre 和 C 的前置条件相等命题, Lustre 和 C 后置条件相等命题. 其中前 3 个命题为证明前提, 第 4 个命题为证明目标. 在本文形式化验证过程中所使用的符号集如附录 A 所示.

在表 7 中, 命题 $P1$、$P2$ 和 $P3$ 是证明的前提, $P4$ 是证明的目标. 在表 8 中最后的推导结果 $S8: Lv6Id = left$ 即需要证明的目标. 因此, *Equation* 文法在编译前后的语义具有一致性. 证明过程中, 公式 $S1$、$S2$ 和 $S3$ 分别由前提得来, $S4$ 由 $S1$、$S2$、和 $S3$ 推导得来, 后续的公式 $S5$–$S8$ 也类似.

表 7 赋值语句文法证明的命题

命题	说明
$P1: (\sigma[expr] \to \sigma[m]) \Rightarrow \sigma[Equation] \to \sigma[Lv6Id/m])$	Lustre赋值语句语义
$P2: (\sigma_c[expr_c] \to \sigma_c[m_c]) \Rightarrow \sigma_c[Equation_c] \to \sigma_c[left/m_c])$	C赋值语句语义
$P3: \sigma[expr] = \sigma_c[expr_c]$	输入 (前置条件) 相等
$P4: Lv6Id = left$	输出 (后置条件) 相等, 证明目标

图 6 是 *Equation* 文法在 Isabelle 中的证明代码序列, 主要分为 3 个部分: 对 Lustre 中 *Equation* 文法的形式化定义, 对 *Equation* 文法的 C 目标码模式的形式化定义, 对上述两个定义的等价性证明. 由于 Isabelle 使用的是

函数式语言, 其形式化表示与上文表格中的语义定义在形式上有所不同. *equation_lustre* 以函数式的形式定义了 Lustre 中赋值语句的语义, 该函数接受两个表达式, 并返回第 2 个表达式的值. *Equation_c* 也同理. *lemma equivalence_prioerty_equation* 是一个引理, 一个引理需要被证明, 也可以在被证明后用于其他引理的证明. 在本文中表示为证明目标: 先前定义的两个函数是等价的, 具体是指, 假设两个函数的输入是相等的, 那么他们的返回值也相等.

表 8 赋值文法证明过程

公式	证据
$S1:\sigma[expr]=\sigma_c[expr_c]$	$P3$
$S2:\sigma[expr]\rightarrow\sigma[m]$	$P1$
$S3:\sigma_c[expr_c]\rightarrow\sigma[m_c]$	$P2$
$S4:m=m_c$	$S1,S2,S3$
$S5:\sigma[Equation]\rightarrow\sigma[Lv6Id/m])$	$P1$
$S6:\sigma_c[Equation_c]\rightarrow\sigma_c[left/m_c])$	$P2$
$S7:\sigma_c[Equation_c]\rightarrow\sigma_c[left/m])$	$S4,S6$
$S8:Lv6Id=left$	$S5,S7$

```
(* equation *)
(* Lustre Program *)
definition equation_lustre :: "'a signal ⇒ 'a signal ⇒ 'a signal" where
  "equation_lustre expression1 expression2 ≡
    (λt. expression2 t)"

(* Corresponding C Function *)
definition equation_c :: "'a signal ⇒ 'a signal ⇒ 'a signal" where
  "equation_c expression1 expression2 ≡
    (λt. expression2 t)"

(* Property for Equivalence *)
lemma equivalence_property_equation:
  assumes preconditions: "∀t. expression2_l t = expression2_c t"
  shows "∀t. equation_lustre expression1_l expression2_l t =
    equation_c expression1_c expression2_c t"
  unfolding equation_lustre_def equation_c_def
  using preconditions by auto
```

图 6 *Equation* 文法 Isabelle 证明序列

图 7 为 Isabelle 中开始证明前, 分别运行两条证明语句之后的证明目标. 第 1 个部分尚未开始证明, 证明目标即 lemma 中的证明目标 (shows 之后的部分). 第 2 部分为运行了第 1 条证明语句 *unfolding equation_lustre_defequation_c_def* 后的结果, 该语句展开了上文对 *equation_lustre* 和 *Equation_c* 的定义并自动化简, 由此证明目标变为了 $\forall t.expression2\_lt = expression2\_ct$, 而这正好和证明的前提相同. 因此第 2 条语句 *using preconditions by auto* 表示自动应用证明前提, 这就完成了对该引理的证明, 也即 *Equation* 的文法单元和目标码序列是等价的. 通过这种方式, 可以独立地证明每条文法和对应目标码模式的语义等价性.

```
proof (prove)
goal (1 subgoal):
 1. ∀t. equation_lustre expression1_l expression2_l t = equation_c expression1_c expression2_c t

proof (prove)
goal (1 subgoal):
 1. ∀t. expression2_l t = expression2_c t

theorem equivalence_property_equation:
 ∀t. ?expression2_l t = ?expression2_c t ⟹
 ∀t. equation_lustre ?expression1_l ?expression2_l t = equation_c ?expression1_c ?expression2_c t
```

图 7 证明过程中的证明目标变化

4.2 源程序和目标程序等价性证明

通过下推自动机识别能得到编译前后的源程序和目标程序的文法单元, 从而生成对应的程序整体的语义, 即

在 Isabelle/HOL 中以 Iris 语言建模的形式化表示. 程序整体的语义由各个语法单元的语义组成, 而每条语法单元的语义的等价性在第 4.1 节当中得到了证明. 因此, 可以利用语法单元语义一致的结论证明程序的语义一致性.

图 8 展示了一个包含了加法和 pre 运算符的 Lustre 程序和对应 C 程序一致性的证明序列. 与第 4.1 节中的证明过程类似, definition *usepre_lustre* 和 definition *usepre_c* 分别是 Lustre 语言和 C 语言代码的定义, 其中 *add_lustre*、*add_c*、*pre_lustre*、*pre_c* 已经分别定义且证明了等价性. 所以, 在等价性引理 lemma *equivalence_property_usepre* 的证明中, 只需要分别对它们进行展开并进行化简, 就可以证明它们之间的一致性.

```
definition usepre_lustre :: "int_signal ⇒ int_signal" where
  "usepre_lustre expression_l  ≡ add_lustre (pre_lustre expression_l)"

definition usepre_c :: "int_signal ⇒ pre_int_ctx_type ⇒ int_signal" where
" usepre_c expression_c ctx ≡add_c( pre_c expression_c ctx)"

lemma equivalence_property_usepre:
  assumes preconditions: "∀t. expression_l t = expression_c t"
  shows "∀t. pre_lustre expression_l t = pre_c expression_c
     (previous_value = (if t = 0 then 0 else expression_c (t - 1))) t"
  unfolding usepre_lustre_def usepre_c_def
  unfolding pre_lustre_def pre_c_def
  unfolding add_lustre_def add_c_def
  using preconditions by auto
```

图 8　程序语义一致性证明序列

上文的证明体现出, 在已经定义了子文法的语义和证明了子文法语义的一致性的基础上, 对于程序一致性的证明在 Isabelle 中可以通过如下的方法证明.

(1) 定义 Lustre 源程序和 C 目标程序的语义, Lustre 程序和 C 程序的语义由其子文法递归给出;

(2) 定义证明前提和证明目标, 其中证明前提为 Lustre 和 C 程序的前置条件相等, 证明目标为 Lustre 和 C 程序的后置条件相等;

(3) 将 (1) 中定义的 Lustre 和 C 程序语义使用 *unfolding* 命令以文法单元为单位依次展开并化简;

(4) 使用 *using preconditions by auto* 命令应用证明前提 (即前置条件相等) 并进行自动推理并得到证明结论. Isabelle 的自动推理能力有限, 但经过 (3) 的展开, 其形式已经可以自动推导出结论.

5　实验与分析

本节展示了对提出的基于下推自动机的 Lustre 语言可信编译器进行系统测试的过程和结果, 测试在指定的实验环境 (处理器为 Intel Core i5-13600KF CPU @ 3.50 GHz, 内存为 32 GB, 操作系统为 Windows 11) 下进行.

5.1　代码生成测试结果与分析

代码生成测试主要进行功能测试, 即测试代码生成的结果是否正确, 测试需要对编译器实现的所有 Lustre 文法进行测试. Lustre V6 版本共有 73 个文法单元, 本文实现的编译器共支持其中的 72 个. 在 72 个文法单元中, 程序结构定义文法有 49 条, 核心的赋值语句和表达式文法有 23 条. 其中程序结构定义文法在每次测试中都会涉及, 只需单独测试其中不常用的部分, 赋值语句和表达式文法为测试的重点.

针对 72 个文法单元和对应的 218 个产生式, 本文共构建了 150 个以上的基本测试用例, 旨在测试单条文法的正确性. Lustre 中核心的表达式文法的测试用例数如后文表 9 所示, 其中运算符组合测试是在一个表达式中使用多个运算符, 测试运算符之间的优先级结合性等关系是否正确.

对生成的 C 代码进行静态分析, 检查是否存在语法错误, 并将本文编译器生成的代码和官方代码生成器生成的代码的运行结果进行比较, 观察运行结果是否符合预期. 经过测试, 本文实现的代码生成模块在涉及的所有文法单元中均能正确生成目标代码.

5.2　形式化验证测试结果与分析

形式化验证测试需要针对 Lustre 文法单元给出对应的语义一致性证明. 首先需要对单个运算符的语义一致性

给出证明, 如图 9 是对 merge 运算符的语义一致性证明, 本节共对表达式文法单元中的 15 种运算符给出了语义一致性证明.

表 9 表达式文法测试用例个数统计

表达式文法单元	测试用例个数
<constant>	3
布尔运算符	2
算数运算符	3
比较运算符	3
int <Expression>	2
real <Expression>	2
pre <Expression>	3
current <Expression>	4
<Expression> fby <Expression>	3
<Expression> → <Expression>	3
if <Expression> then <Expression> else <Expression>	3
# (<Expression>)	2
nor (<Expression>)	2
merge <*Lv6Id*> <Mergecaselist>	4
<CallByPosExpression>	3
<CallByNameExpression>	4
[<ExpressionList>]	2
<Expression> ^ <Expression>	2
<Expression> | <Expression>	2
<Expression>[<Expression>]	2
<Expression> . <*Lv6Id*>	2
(<ExpressionList>)	2
运算符组合测试	20
总计	78

```
(* converse to if - else if - ... - else  *)
definition merge_lustre :: "string signal ⇒ 'a signal ⇒ 'a signal ⇒ 'a signal" where
  "merge_lustre a b c ≡ (λt. if a t = ''Pile'' then b t else if a t = ''Face'' then c t else undefined)"

definition merge_c :: "string signal ⇒ 'a signal ⇒ 'a signal ⇒ 'a signal" where
  "merge_c a b c ≡ (λt. if a t = ''Pile'' then b t else if a t = ''Face'' then c t else undefined)"

lemma equivalence_property_merge:
  assumes preconditions: "∀t. a_l t = a_c t ∧ b_l t = b_c t ∧ c_l t = c_c t"
  shows "∀t. merge_lustre a_l b_l c_l t = merge_c a_c b_c c_c t"
  unfolding merge_lustre_def merge_c_def
  using preconditions by auto

end
```

图 9 merge 运算符一致性证明

根据单个运算符文法的语义一致性证明, 可以由此进行程序的一致性证明. 本节对包含了上述 15 种文法的 10 个 Lustre 程序进行了验证测试, 平均证明长度 (含定义) 为 15 行. 由 Isabelle 对语义一致性证明序列进行自动推理并得到程序语义一致证明结论, 从而说明该形式化验证方法的有效性.

5.3 系统运行耗时结果分析

表 10 给出编译系统各个阶段的平均运行时间和生成的目标代码运行 1 000 万周期的时间, 其中下推自动机生成仅需执行一次, 在每次编译时不需要执行. 在系统建模的实际生产中, 通常不会频繁进行编译, 因此对于编译的时间效率并不敏感.

表 10 编译各阶段耗时与代码运行时间

编译阶段	运行时间 (ms)
下推自动机生成	30.100
词法分析 (每100行平均)	18.200
代码生成 (每100行平均)	80.010
目标代码运行时间 (1 000万周期)	120.000

6 局限性与扩展性讨论

Lustre 语言到 C 语言的目标码转换模式和转换动作支持 Lustre V6 语言中 99% 的文法单元和语言特性, 且涵盖了 SCADE 中所有和 Lustre 相关的语言特性. 利用下推自动机生成、下推自动机识别和代码生成的算法, 以及 Lustre 到 C 的目标码转换模式, 实现了由 Lustre 语言到 C 语言的编译系统. 编译系统支持类型检查和 Lustre 语言特有的始终检查、拓扑排序功能. 该编译器符合 DO-178C 开发规范, 且主要算法和模块独立于 Lustre 语言和 C 语言, 具有较好的可扩展性. 该方法已经应用于 SCADE 语言的编译器开发中, 目前已经完成了对时钟算子、时态算子、数组与结构体算子、算术逻辑运算、函数调用、状态机和条件块等相关文法的代码生成, 并对某一大型飞机的刹车系统和主飞行显示器系统 (primary flight display, PFD) 进行案例研究, 最终转换得到的 C 目标码与原系统 SCADE 模型的运行结果一致, 充分验证了其在 SCADE 语言上的扩展性.

该编译系统并未涉及编译优化相关内容, 而编译优化在编译器当中是重要的一部分. 未来将主要从这几个方面进行优化: 1) 消除无用的代码, 采用数据流分析, 找出没有影响输出的变量或表达式并删除; 2) 常量传播, 如果某些输入或中间变量的值在编译时已经是常量, 运用值分析技术进行替换, 减少运行时的计算开销; 3) 循环优化, 实施循环不变代码提升, 将循环中不变的部分移到循环外, 减少循环次数, 并结合循环展开技术减少循环控制开销.

7 总结与未来工作

本文针对安全关键系统开发过程中编译的可靠性问题, 研究了基于下推自动机的编译方法和基于语义一致性的验证方法. 提出了 Lustre 语言可信编译方法, 实现了基于下推自动机的 Lustre 语言到 C 语言的可信编译, 并使用 Isabelle 对编译方法进行了验证. 本文的研究成果如下.

(1) 下推自动机的生成算法: 该算法能够自动将符合一定格式的上下文无关文法转换为对应的下推自动机, 该算法分 4 步得到了完整的由开始状态到中间状态最后到终止状态的下推自动机. 且由该算法生成的下推自动机具有扩展性, 可以独立地生成下推自动机再进行合并.

(2) 下推自动机的识别算法: 该算法独立于下推自动机的具体结构, 将下推自动机作为输入的一部分. 该算法接受下推自动机生成算法产生的下推自动机和源程序经过词法分析得到的词法单元流作为输入, 模拟下推自动机的运行, 对源程序进行语法分析和语义分析.

(3) 基于下推自动机的识别算法的代码生成方法: 该方法通过将代码的编译过程抽象为源文法单元和目标码转换模式, 将目标码转换模式编写为转换动作嵌入下推自动机识别过程当中, 从而在下推自动机识别的过程中完成源语言到目标语言的编译过程.

(4) 由 Lustre 语言到 C 语言的编译系统: 该编译系统支持类型检查和 Lustre 语言特有的始终检查、拓扑排序功能. 该系统符合 DO-178C 开发规范, 且主要算法和模块独立于 Lustre 语言和 C 语言, 具有较好的可扩展性.

(5) 基于语义一致性的形式化验证方法: 该方法将编译过程的正确性转化为编译前后程序的语义一致性. 而编译前后程序的一致性又可规约为编译前后程序所包含的文法单元的一致性, 由此可以利用 Isabelle/HOL 的自动推导能力自动地证明程序的一致性.

本文实现的基于下推自动机的编译方法在理论上有良好的可扩展性和语言无关性, 未来可以在编译器扩展性上进行进一步的研究和实践, 或将该方法扩展应用到其他语言的编译当中.

References:

[1] Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, De Simone R. The synchronous languages 12 years later. Proc. of the IEEE, 2003, 91(1): 64–83. [doi: 10.1109/JPROC.2002.805826]

[2] Caspi P, Pilaud D, Halbwachs N, Plaice JA. Lustre: A declarative language for real-time programming. In: Proc. of the 14th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. Munich: ACM, 1987. 178–188. [doi: 10.1145/41625.41641]

[3] Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language Lustre. Proc. of the IEEE, 1991, 79(9): 1305–1320. [doi: 10.1109/5.97300]

[4] 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]

[5] Berry G. The foundations of Esterel. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner. Cambridge: The MIT Press, 2000. 425–454.

[6] Le Guernic P, Gautier T, Le Borgne M, Le Maire C. Programming real-time applications with SIGNAL. Proc. of the IEEE, 1991, 79(9): 1321–1336. [doi: 10.1109/5.97301]

[7] 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]

[8] Lustre V6 home. http://www-verimag.imag.fr/Lustre-V6.html?lang=en

[9] McCarthy J, Painter J. Correctness of a compiler for arithmetic expressions. In: Mathematical Aspects of Computer Science 19: Proc of Symposia in Applied Mathematics. American Mathematical Society, 1967. 33−41.

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

[11] CompCert. http://compcert.inria.fr/

[12] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 283–294. [doi: 10.1145/1993498.1993532]

[13] Paulin C, Pouzet M. Certified compilation of SCADE/Lustre. 2006. https://www.lri.fr/~paulin/lustreincoq.pdf

[14] Bourke T, Brun L, Dagand PÉ, Leroy X, Pouzet M, Rieg L. A formally verified compiler for Lustre. ACM SIGPLAN Notices, 2017, 52(6): 586–601. [doi: 10.1145/3140587.3062358]

[15] Shang S, Gan YK, Shi G, Wang SY, Dong Y. Key translations of the trustworthy compiler L2C and its design and implementation. Ruan Jian Xue Bao/Journal of Software, 2017, 28(5): 1233–1246 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5213.htm [doi: 10.13328/j.cnki.jos.005213]

[16] Pnueli A, Siegel M, Singerman E. Translation validation. In: Proc. of the 4th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Lisbon: Springer, 1998. 151–166. [doi: 10.1007/BFb0054170]

[17] Pnueli A, Shtrichman O, Siegel M. Translation validation for synchronous languages. In: Proc. of the 25th Int'l Colloquium. Aalborg: Springer, 1998. 235–246. [doi: 10.1007/BFb0055057]

[18] van Ngo C, Talpin JP, Gautier T, Le Guernic P, Besnard L. Formal verification of compiler transformations on polychromous equations. In: Latella D, Treharne H, eds. Proc. of the 2012 Int'l Conf. on Integrated Formal Methods, 2012. 113–127.

[19] van Ngo C, Talpin JP, Gautier T, Le Guernic P. Translation validation for clock transformations in a synchronous compiler. In: Proc. of the 18th Int'l Conf. on Fundamental Approaches to Software Engineering, 2015. 171–185. [doi: 10.1007/978-3-662-46675-9_12]

附中文参考文献:

[15] 尚书, 甘元科, 石刚, 王生原, 董渊. 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]

附录 A. 形式化验证过程中所使用的符号集

在本文形式化验证过程中所使用的公理系统其符号集如表 A1 所示.

表 A1 符号集

类型	符号	意义
语义符号	σ	程序取值集合, 表示当前的程序运行状态, 每个周期都会有这样一个取值集合
	$\sigma[c]$	表示在状态 σ 下执行程序c
	σ^{-i}	表示在 i 周期前的程序状态
	$\sigma^n[c]$	表示在状态 σ^n 下执行程序 c, 每个周期都有一个状态集合
	$\sigma[x/v]$	定义一个新状态, 程序变量 x 在该状态下的值为 v
	$\sigma[(x:t)/v]$	定义一个新状态, 程序变量 x 在该状态下的值为 v, 类型为 t
	$\sigma[c]\to\sigma_1$	程序在状态 σ 下运行 c 后到达终止状态 σ_1
	$\sigma\to cur$	表示将 σ 设为当前周期;
	$\sigma[c]\to\sigma_1[c1]$	程序在状态 σ 下运行 c 后得到的结果为状态 σ_1, 且下一步要执行的程序为 $c1$
逻辑联结词	$\neg$	一元联结词, 非
	$\to$	二元联结词, 蕴含
	$\vee$	二元联结词, 或
	$\wedge$	二元联结词, 且
运算符	=	一元运算符, 赋值
	<	二元运算符, 小于
	>	二元运算符, 大于

附录 B. Lustre 操作语义

由于篇幅限制, 以下给出 Lustre 部分文法的操作语义表示. 表 B1 是 Lustre 表达式单元文法 (时态和时钟运算符部分) 的操作语义. 时态运算符 pre 用于获取表达式在上个周期的值, 假设表达式 *expr*1 在上个周期的语境 σ^{-1} 中的取值为 m, 则该表达式 (pre *expr*1) 在当前语境 σ 下的取值为 m. 时钟运算符 when 用于指定流的时钟周期, 若时钟表达式 *ClockExpr* 在当前语境 σ 的取值为 true, *expr*1 在 σ 的取值为 m, 则表达式 *expr* 在 σ 的取值为 m; 若时钟表达式 *ClockExpr* 在当前语境 σ 的取值为 false, 则表达式 *expr* 在 σ 的取值为 *nil*. current 运算符用于将非基础周期的流转化为基础周期的流, 若表达式 *expr*1 的时钟在当前语境 σ 的取值为 true 且 *expr*1 在 σ 的取值为 m, 则 *expr* 在 σ 的取值为 m, 若表达式 *expr*1 的时钟在当前周期内取值为 false 且 *expr* 在上个周期的语境 σ^{-1} 的取值为 m, 则 *expr* 在 σ 的取值为 m.

表 B1 Lustre 表达式文法单元语义定义 (时态和时钟运算符部分)

结构元素	文法单元	前置/后置条件
< *expr* >	pre < *expr*1 >	$\left(\sigma^{-1}[expr1]\to\sigma^{-1}[m]\right)\Rightarrow(\sigma[expr]\to\sigma[m])$
	< *expr*1 > arrow < *expr*2 >	第1周期, $(\sigma[expr1]\to\sigma[m])\Rightarrow(\sigma[expr]\to\sigma[m])$ 非第1周期, $(\sigma[expr2]\to\sigma[m])\Rightarrow(\sigma[expr]\to\sigma[m])$
	< *expr*1 > fby < *expr*2 >	第1周期, $(\sigma[expr1]\to\sigma[m])\Rightarrow(\sigma[expr]\to\sigma[m])$ 非第1周期, $\left(\sigma^{-1}[expr2]\to\sigma^{-1}[m]\right)\Rightarrow(\sigma[expr]\to\sigma[m])$
	< *expr*1 > when < *ClockExpr* >	$(\sigma[ClockExpr]\to\sigma[\text{true}],\sigma[expr1]\to\sigma[m])\Rightarrow(\sigma[expr]\to\sigma[m])$ $(\sigma[ClockExpr]\to\sigma[\text{false}])\Rightarrow(\sigma[expr]\to\sigma[nil])$
	current < *expr*1 >	$(\sigma[expr1.Clock]\to\sigma[\text{false}])\Rightarrow\left(\sigma[expr]\to\sigma^{-1}[expr]\right)$ $(\sigma[expr1.Clock]\to\sigma[\text{true}],\sigma[expr1]\to\sigma[m])\Rightarrow(\sigma[expr]\to\sigma[m])$

于涛(1999－), 男, 硕士, 主要研究领域为形式化验证, 可信编译.

王珊珊(1999－), 女, 硕士, 主要研究领域为形式化验证, 可信编译.

徐芊卉(2000－), 女, 硕士生, 主要研究领域为形式化验证, 可信编译.

董晓晗(2000－), 女, 硕士生, 主要研究领域为验证编译器, 形式化方法.

胡代金(2002－), 男, 硕士生, CCF 学生会员, 主要研究领域为形式化验证, 可信编译.

罗杰(1981－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件基础理论, 知识获取与推理, 群体智能.

杨溢龙(1988－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为智能化软件工程.

吕江花(1975－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件形式化方法, 面向航空航天的安全攸关系统可信性验证.

马殿富(1960－), 男, 博士, 教授, CCF 会士, 主要研究领域为安全关键系统, 形式化方法.