软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(2):341−356 [doi: 10.13328/j.cnki.jos.004542]

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

同步数据流语言可信编译器的构造*

石 刚[1,2], 王生原[1], 董 渊[1], 嵇智源[3], 甘元科[1], 张玲波[1], 张煜承[1], 王 蕾[1], 杨 斐[1]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](新疆大学 信息科学与工程学院,新疆 乌鲁木齐 830046)

[3](科技部 高技术研究发展中心,北京 100044)

通讯作者: 石刚, E-mail: shigang1022@163.com

摘 要: 同步数据流语言近年来在航空、高铁、核电等安全关键领域得到广泛应用.然而,此类语言相关开发工具本身的安全性业已成为被高度关注的安全隐患之一.借助辅助定理证明器实现常规语言编译器的构造和验证已被证明是成功的,有望最大限度地解决误编译问题.基于这种方法,开展了从同步数据流语言(Lustre 为原型)到串行命令式语言(C 为原型)的可信编译器构造的关键技术研究.其挑战性在于两类语言之间的巨大差异,源语言具有时钟同步、数据流、并发及流数据对象等特征,而目标语言则具有顺序控制流特征.同类研究中,目前尚无针对核心翻译过程的公开成果.就单一时钟的情形实现了一个经过形式化验证的完整编译过程,相关技术将应用于安全关键领域编译系统的开发.综述了这一可信编译器的研究背景、意义、总体设计框架、核心技术、现状以及进行中或后续的工作.

关键词: 同步数据流语言;经过验证的编译器;形式化验证;形式语义;定理证明

中图法分类号: TP314 **文献标识码**: A

中文引用格式: 石刚,王生原,董渊,嵇智源,甘元科,张玲波,张煜承,王蕾,杨斐.同步数据流语言可信编译器的构造.软件学报, 2014,25(2):341−356. http://www.jos.org.cn/1000-9825/4542.htm

英文引用格式: Shi G, Wang SY, Dong Y, Ji ZY, Gan YK, Zhang LB, Zhang YC, Wang L, Yang F. Construction for the trustworthy compiler of a synchronous data-flow language. Ruan Jian Xue Bao/Journal of Software, 2014,25(2):341−356 (in Chinese). http://www.jos.org.cn/1000-9825/4542.htm

Construction for the Trustworthy Compiler of a Synchronous Data-Flow Language

SHI Gang[1,2], WANG Sheng-Yuan[1], DONG Yuan[1], JI Zhi-Yuan[3], GAN Yuan-Ke[1], ZHANG Ling-Bo[1], ZHANG Yu-Cheng[1], WANG Lei[1], YANG Fei[1]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](School of Information Science and Engineering, Xinjiang University, Urumqi 830046, China)

[3](High Technology Research and Development Center, Ministry of Science and Technology, Beijing 100044, China)

Corresponding author: SHI Gang, E-mail: shigang1022@163.com

Abstract: Synchronous data-flow languages have been widely used in safety-critical industrial areas such as airplanes, high-speed railways, nuclear power plants, and so on. However, the safety of development tools themselves for this kind of languages has become one of the potential safety problems which have been highly focused on. It has proved to be successful to implement the construction and verification of a conventional language compiler using an assistant theorem prover, and it is expected to have the opportunity in solving the miscompilation problem to the utmost extent. Based on such an approach, the key technologies for a trustworthy compiler from a synchronous data-flow language (Lustre as the prototype) to a sequential imperative language (C as the prototype) have been studied. The challenge lies in the great difference between the source and target languages, where the source language has the features of clock

* 基金项目: 国家自然科学基金(61170051, 61272086, 90818019)
收稿时间: 2013-05-04; 修改时间: 2013-09-29, 2013-12-05; 定稿时间: 2013-12-17

synchrony, data-flow, concurrency, stream data object, etc., while the target language is instead with the sequential and control-flow features. Among the similar researches, there is no reported result so far for the key translation stages. Recently, this research completed a formal verification for the whole compilation stages in the case of single clock. The related technologies will be taken into the development of a safety-level compiler in the country's nuclear power system. The paper presents a survey on the trustworthy compiler with the research background, the architecture, the key technologies, the current status, and the future work.

Key words: synchronous data-flow language; verified compiler; formal verification; formal semantics; theorem proving

随着计算机控制系统在人们生活中的普及,软件自身的可靠性也越来越受到重视.在航空、高铁、核电及军事等高安全要求领域的软件系统——安全关键系统(safety-critical system,简称 SCS)[1]更是受到高度的重视.而随着软件系统的复杂度越来越高,软件系统的安全性保证也变得越来越困难.这些系统的开发,仅仅依靠过程规范、代码审核和系统测试来保证软件安全还远远不够,通常需要采用形式化验证方法来保证软件可靠性.在某些安全性严格要求的领域,不但对目标系统的开发需要形式化方法来保证,而且开发过程所需工具,如编译器等,也必须经过形式化验证.

(1) 同步模型与同步数据流语言

工业生产中的很多实时系统(如控制系统)都是响应系统,它是以环境决定的速率对环境做出连续响应的系统.此环境可以随时提供新的输入,系统通过运算产生新的输出,使得程序不断地与环境进行交互,且以并发的方式体现.

同步模型是为了适应响应系统而开发的程序运行模型.它把响应系统分成连续的原子时间片(响应周期),每个时间片又分成事件采集、逻辑运算和事件发射这 3 个阶段,从而有效地满足响应系统的要求.在同步模型中,事件的采集、逻辑运算和事件的发射必须在同一个响应周期内完成,这称为同步模型的同步假设.时间片的大小是由响应系统的环境决定的.同步假设的有效性检查,就是评估系统对环境的最大响应时间能否满足环境要求.图 1 为同步模型示意图.

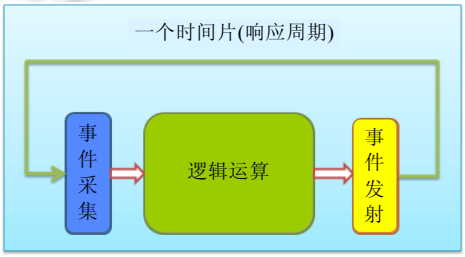


Fig.1 Synchronous model

图 1 同步模型

关于同步模型编程范型的理论工作 30 多年前已经开始[2],之后出现了许多得到广泛应用的同步语言[3−5],最著名的如 Esterel[6,7],Lustre[8,9]和 Signal[10,11].同步语言的范型主要可分为两类:命令式(imperative)和陈述式(declarative).Esterel 是命令式同步语言,侧重于描述控制流;Lustre 和 Signal 是陈述式同步语言,具有数据流特征,常称为同步数据流语言.Lustre 是一种函数式(functional)风格的语言,具有确定语义;Signal 是关系型(relational)语言,具有非确定语义.

(2) 编译器的可信

对于编译器来说,可信的具体指标就是其正确性,要保证从源程序到目标程序的翻译过程正确,即,保证源语言的特征被正确、完整地实现,能够实现语义保持性,杜绝误编译.

为了保障编译器实现的正确性,传统的方法是通过大量的测试.例如,GCC(4.7 版)的 torture 测试集包含 2 853 个 C 源程序用例[12],商用的 Plum Hall Standard Validation Suite for C 有 29 424 个用例[13],Lustre V6[14]的开源软件包中含有 140 个左右的源程序用例,等等.还有一些 Bug-hunting 工具(如 Csmith[15])可能产生更多或更独特的源程序用例.

然而和其他软件一样,通过测试只能发现错误,并不能保证编译器是正确的.Scade[16]工具的代码生成器KCG 或许是获得民用航空软件生产许可的第一个商用编译器,其设计开发过程(很严格的 V&V 过程)符合民航电子系统的国际标准 DO-178B,并成功应用于空客(airbus)A340 和 A380 的设计中.尽管如此,这并不足以说明Scade 的编译器不存在"误编译".事实上,业界已经发现 Scade KCG 的某些翻译漏洞.

为增加编译器的安全和可信程度,仅通过测试和严格的过程管理都是不够的,人们很自然地会想到形式化验证的途径.在 CC(common criteria)安全评估标准[17]中,将可信性分为 7 个级别(EAL1~EAL7),级别越高,形式化规范和验证的程度越高.航空无线电委员会(RTCA)近期也已推出民航电子系统的国际标准 DO-178C,与DO-178B 相比,增加的内容包括了对形式化规范和验证的要求[18].

人们在几十年前就开展了编译器形式化验证的工作:McCarthy 等人在 1967 年手工证明了一个简单编译器(从算术表达式翻译到栈式机目标代码)的正确性[19];随后,Milner 等人在 1972 年给出了相应的机械化证明[20];Dave 于 2003 年的综述列举了从 1967 年~2003 年的大部分相关工作[21],包含从针对简单语言的单遍编译器到较成熟的代码优化遍等形形色色的工作.近年来,随着技术的不断进步,已经可以验证较为复杂的编译器.例如,Leroy 等人开发的验证过的 C 编译器 CompCert[22],Chlipala 给出一个从非纯函数式的语言到汇编语言的一个经过验证的编译器[23],Klein 等人验证了一个从 Java 核心子集到 Java 虚拟机的编译器[24].

CompCert 编译器[22]是经过验证的可信编译器的杰出代表.该编译器将 C 的一个重要子集 Clight 翻译为PowerPC 汇编代码(目前也支持 IA32 后端),使其可以直接用于范围广泛的嵌入式应用开发.该编译器包含了许多分析与优化,所生成的代码可与 gcc-0 产生的代码匹敌.该编译器将编译过程划分为多个阶段,每个阶段的翻译正确性(语义保持性)都借助辅助定理证明器 Coq[25,26]进行了证明,且这些证明可由独立的证明检查器检查,这使得 CompCert 的中间层转换过程达到了我们所能期望的最高可信程度[27].最近,Yang 等人在关于 Csmith 的研究工作[15]中对主流的 C 编译器进行测试,共报告了 325 个 bugs,其中包括 Intel CC,GCC 和 LLVM 等.在所比较的 11 种开源或商用的 C 编译器中,CompCert 表现较为突出,在 6 个 CPU 年中,其中间转换过程没有发现 bugs.

编译器验证的另外一种可选方案是翻译确认(translation validation).翻译确认(translation validation)是Pnueli 等人首先提出来的[28].他们采用翻译确认的方法来验证同步数据流语言的翻译(或编译)过程[28,29],所给示范例子的源语言是 Signal 特征的多时钟同步数据流语言,目标语言是 C.翻译确认的方法不是直接验证翻译程序,而是用统一的语义框架为某一翻译过程的源和目标代码建模,两个模型之间定义一种求精(refining)等价关系,设计一种可自动证明二者等价性的分析程序(成功时可给出证明脚本,不成功时给出反例),再给出一种独立的证明检查器(proof checker),可最后确认翻译的正确性.

Ngo 等人基于翻译确认的思想开展了同步数据流语言 Signal 到 C 的编译器验证工作[30],其最新的进展是:对源和目标代码用统一语义框架 PDS(polynomial dynamical system)建模,给出一种源和目标之间的抽象时钟等价关系,从而证明编译器可保持时钟语义的一致性[30~32].求精等价关系的证明采用 SMT 求解器[33]自动完成.

对于翻译(或编译)程序的验证来说,翻译确认有时是一个很好的选择[22],这种方法适用于翻译前后语义保持性的确认过程较容易构造,并且比翻译程序的验证更简单的情形.我们认为,翻译确认方法的最大优点是不放弃现有编译器,比如,基于一些大型编译器也可以有类似的工作[34].这种方法的可扩性(scalability)较好.

在 L2C 项目中,我们选择了对编译器本身进行验证.当源语言和目标语言的语义定义达到认可的程度时,原理上可以保证源程序的一般性质都可以保持到目标程序.与上述翻译确认的做法相比,这是一种彻底的做法;而基于翻译确认的方法往往只是关注部分性质的保持性(当然,也可以逐步逼近一般性质).然而,这项工作相当艰巨,正如前述 Pouzet 等人的项目,是一项长期的工作.然而,我们的考虑是:1) 实际中有值得这样做的需求;2) 如果在面向验证的编译器结构上下功夫,加强证明过程的局部化和模块化研究,可减轻需要扩展和维护时的负担;3) 在某些扩展和维护工作量过大的翻译阶段,还可以采用翻译确认的方法作为补充,比如 CompCert 项目,也有个别阶段采用了翻译确认方案;4) CompCert 项目的成功,告诉我们这一选择的重要价值和可行性.

受 CompCert 项目[22]的启发,Paulin 等人于 2006 年启动了一个有关同步数据流语言编译器验证的长线项目[35],源语言接近 Scade 的 Lustre 语言且具有 Lucid Synchrone[36]的特征.该项目的工作目标与本课题组的 L2C

项目相似.就我们所了解,该项目的工作进展目前仍集中于前端,完成了类型检查和时钟演算相关过程的验证,后续又在开展因果性分析程序的验证工作[37].Biernacki 等人在文献[38,39]中描述了该项目的一些相关工作.他们将 Lustre 语言的一个较早版本翻译至 Java 和 C 代码,采用了一种基于对象的中间语言.然而,拟采用的语义模型以及翻译过程的正确性(即语义保持性)验证的工作尚未见诸报道.

本文第 1 节给出可信编译器的总体设计框架.第 2 节介绍翻译过程正确性相关的核心内容.第 3 节是关于现状及后续工作计划.第 4 节是本文的一个简短总结.

1 L2C 的可信编译器的总体设计框架

L2C 项目和 CompCert 项目一样,源语言、目标语言和各阶段中间语言的语法、语义、翻译过程的定义以及正确性证明都在交互式辅助定理证明工具 Coq[25,26]中实现.在 Coq 中,程序、性质和证明可用同一种语言来描述. Coq 系统用 OCaml 实现,完全开源,其核心很小且常年稳定,被广泛应用于工业界和研究领域.

为了方便,同时考虑到今后从 C 代码到可执行目标代码的可信翻译需求,L2C 项目可信编译器的目标语言 C 直接借用了 CompCert 编译器中 Clight 的语法和语义定义[40].

L2C 项目的可信编译器结构几经改变才趋于稳定,目前的总体结构如图 2 所示.目前所定义的源语言原型 Lustre*是 Lustre 语言的一个核心子集,综合参考了 Scade 工具的 Lustre 语言版本和 Lustre V6[41],能够体现同步数据流语言的主要特征.为了方便,本文以附录形式给出了当前 Lustre*语言的语法定义(附录 A)以及 Lustre*程序的一个样例(附录 B),仅供读者必要时参考.

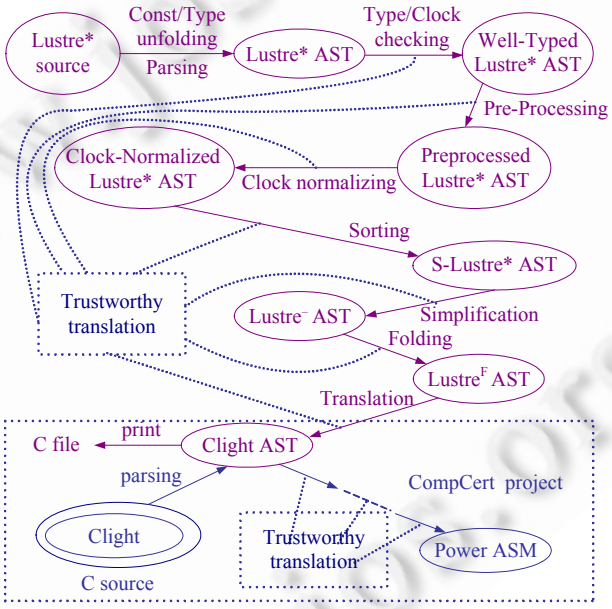


Fig.2 Architecture of the trustworthy compiler L2C

图 2 L2C 项目的可信编译器总体结构

在项目开展初期,编译器仅包含 4 个语言层次:Lustre* Source,Well-Typed Lustre* AST,Lustre⁻ AST 以及 Clight AST.然后,为方便语义定义,我们将串行化的任务提前,而不是在最后生成 Clight AST 时进行;这样,引入了 S-Lustre* AST;该中间语言及其后续中间语言均具有串行语义.然后,为确保翻译至 Lustre⁻ AST 后仍可保持原有因果性,增加了一个针对时态运算(pre,FBY)的预处理过程,引入了 Preprocessed Lustre* AST.接着,由于 Lustre⁻与 Clight 之间仍有很大语义差距,所以引入中间语言 Lustre^F AST,以便消去时态算子.当项目进行了相当长的一个阶段后,实践表明,若考虑多时钟/时钟层次,这些语言层次翻译过程的语义保持性的证明相当困难.同时也考虑

到今后扩展的需求,所以引入中间语言 Clock-Normalized Lustre* AST.

图 2 中编译器各翻译步骤的工作分别是:

(A) Lustre*源程序,经过一个语法解析过程(其间穿插常量/类型展开或求值的工作),被翻译至 Lustre* AST.

(B) 对 Lustre* AST 进行类型检查(含普通类型检查、因果性检查、时钟检查以及初始化检查)的过程,得到 Well-Typed Lustre* AST.

(C) Well-Typed Lustre* AST 经过预处理后得到 Preprocessed Lustre* AST(其中,pre,FBY 等时态算子的操作数简化为变量或常量).

(D) Preprocessed Lustre* AST 经时钟的归一化处理,消去 when 取样算子,得到只包含默认时钟的 Clock-Normalized Lustre* AST.

(E) 经过串行化过程之后,所得到的 S-Lustre* AST 是 Clock-Normalized Lustre* AST 的一个拓扑排序.

(F) S-Lustre* AST 经进一步简化得到 Lustre⁻ AST(类似三地址码).

(G) 消去时态运算后,从 Lustre⁻ AST 得到 Lustre^F AST.

(H) 最后,将 Lustre^F AST 转换至 CompCert 项目所定义的 C 语言子集 Clight 的抽象语法树 Clight AST.

在 L2C 可信编译器总体结构中,除了第(A)步以外,第(B)步~第(H)步都需要形式化证明各阶段源语言到目标语言的语义保持性.目前,全部完成的步骤是第(E)步~第(H)步.

2 翻译过程的正确性

在图 2 所示的可信编译器总体结构中,除了语法解析过程(穿插常量/类型展开或求值)以外,其余各翻译步骤都需要形式化证明各阶段源语言到目标语言的语义保持性.

如图 3 所示,S 语言程序 P 被翻译至 T 语言程序 $T(P)$.S_S 是 S 语言的语义函数,S_T 是 T 语言的语义函数.从语言 S 至语言 T 的语义保持性是指:

$$\forall P.(sound(P)\Rightarrow sound(T(P))\wedge S_S(P)\approx S_T(T(P))) \quad \text{(性质 A)}$$

这里,$sound(P)$和 $sound(T(P))$意味着可以正常得到 $S_S(P)$和 $S_T(T(P))$.$\approx$是单向模拟等价关系.$S_S(P)\approx S_T(T(P))$意味着:P 的所有环境变量在 $T(P)$都有匹配对象,且 $S_S(P)$对环境的改变可以通过 $S_T(T(P))$对相匹配环境的改变进行模拟.

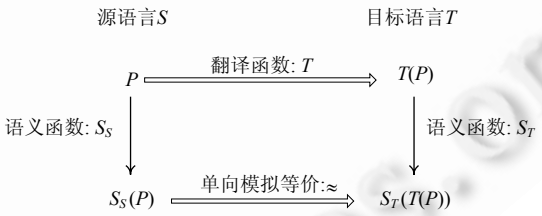


Fig.3 Semantics preserving

图 3 语义保持性

对于合法的 Lustre*程序,求值结果总是确定的(对每一层语言的语义都要证明这个性质),这意味着语义计算的确定性(并发语义的情形也只是求值过程的并发,而不同求值过程的结果是确定的).在这种特殊情形下,我们证明上述单向模拟关系就可以了.

以下从具有代表性的几个侧面来展开说明.

2.1 串行语义

串行化之后的中间语言,包括 S-Lustre*,Lustre⁻,Lustre^F 以及 Clight 等,均具有串行语义.Clight 的语法和语义同 CompCert 编译器中的定义[40].S-Lustre*和 Lustre⁻的语义框架一致,只是因语法成分不同而导致了语义规则的

差异.Lustre^F 由于已消除时态算子,因而语义环境比起 Lustre$^-$有了很大的简化,拉近了与 Clight 语义的差距.

下面主要对 S-Lustre*或 Lustre$^-$语义框架(简称串行 Lustre 语义)的核心部分进行半形式化的解释.

Lustre 串行语义与常规顺序语言的语义的最大差别,就是需要处理好时钟以及时态算子,这些都体现在语义环境和语义规则的定义中.

首先介绍语义环境.由于要考虑时钟和时态算子,加之与 node 的调用纠结起来,使得 Lustre 串行语义环境比起常规顺序语言的语义环境要复杂许多.语义环境由全局和局部环境组成.由于不含全局量,全局环境(global environment)*ge* 可简单地定义为

$$ge ::= id \rightarrow fd$$

可以通过 node 的 *id* 在全局环境 *ge* 中查找 node 的具体定义.局部环境 *e* 的定义比较复杂:

- $e ::= (te, e^*)$;
- $te ::= le^*$;
- $le ::= id \rightarrow (v, ck, \tau)$.

这里,我们定义了一个 3 层的局部环境.底层环境 *le* 与传统语言语义的局部环境类似,刻画了 node 在某个时钟周期的局部环境,它将局部变量的 *id* 映射到当前时钟下的取值 *v* 以及时钟 *ck* 和类型τ.由于在之前的翻译阶段增加了时钟归一化处理,*ck* 一定是指全局时钟,因而可省略.中间层的 *te* 刻画一种时态环境,是 *le* 的一个列表,该列表的第 *n* 个元素表示第 *n* 个时钟周期的 *le*.由于每个 node 在任何时钟周期都被执行,所以在每个时钟周期开始的时刻,所有 node 的 *te* 的长度均相等.一个 node 实例的顶层环境 *e* 是一个复杂的树状结构,其数据域 *te* 维护了该 node 实例中局部变量、输入和输出参量的历史取值,其每个子树本身又是新 node 实例的顶层环境,第 *n* 个子树代表该 node 中第 *n* 个调用的子 node 实例的顶层环境.

图 4 刻画了 node A 调用 node B 两次、调用 node C 一次,而 node C 调用 node B 一次的情形下,node A 实例所形成的局部环境.

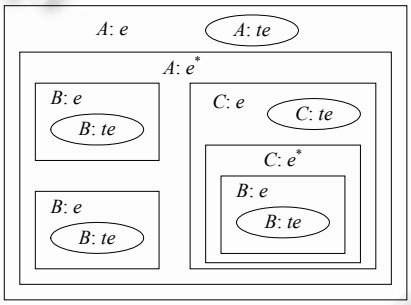


Fig.4 An example of node local environment

图 4 Node 局部环境示意

限于篇幅,本文仅以 if 表达式和 fby**表达式为例说明串行 Lustre 表达式求值的语义规则.虽然 S-Lustre*和 Lustre$^-$的语义在框架上是一致的,但由于 Lustre$^-$的普通表达式中不含 node 调用,所以二者的 if 表达式语义规则大不相同.而二者的 fby 表达式语法上相同,所以 fby 表达式的语义规则很相似.

S-Lustre*中 if 表达式的语义规则:

$$\frac{ge \vdash (e, ord, c, n) \Rightarrow (e_1, ord_1, VtrueL :: nil) \quad ge \vdash (e_1, ord_1, ex_1, n) \Rightarrow (e_2, ord_2, vl_1) \quad ge \vdash (e_2, ord_2, ex_2, n) \Rightarrow (e_3, ord_3, vl_2)}{ge \vdash (e, ord, \text{if } c \text{ then } ex_1 \text{ else } ex_2, n) \Rightarrow (e_3, ord_3, vl_1)} \quad \text{(IF-True)}$$

** fby 是 FBY 的特殊情况:*a* fby *b* 相当于 *FBY*(*b*,1,*a*).目前,L2C 项目已完成的部分没有支持 FBY.fby 与 pre 之间的关系:*a* fby *b* 相当于 $a \rightarrow pre(b)$.自 S-Lustre*之后的中间语言中,仅包含 fby 和 arrow(→)两个时态算子,不含 pre.

S-Lustre*的并发特征,if 表达式的子表达式 c,ex_1,ex_2 是并发执行的,这不同于命令式语言的 if 控制语句.

$$\frac{ge\vdash(e,ord,c,n)\Rightarrow(e_1,ord_1,VfalseL::nil)\quad ge\vdash(e_1,ord_1,ex_1,n)\Rightarrow(e_2,ord_2,vl_1)\quad ge\vdash(e_2,ord_2,ex_2,n)\Rightarrow(e_3,ord_3,vl_2)}{ge\vdash(e,ord,\text{if } c \text{ then } ex_1 \text{ else } ex_2,n)\Rightarrow(e_3,ord_3,vl_2)}\quad\text{(IF-False)}$$

其中,VtrueL 和 VfalseL 是布尔语义常值;形式断言(judgement):

$$ge\vdash(e,ord,expr,n)\Rightarrow(e',ord',vl).$$

其含义为:在全局环境 ge 中,某个 node 实例中的表达式 $expr$(可以是表达式列表)于第 n 个时钟周期在局部环境 e 下进行求值,求值结果为 vl(值的列表),同时,环境 e 被修改为 e',所执行的子节点次序编号由 ord 变为 ord'.再详细一些解释,这里的 e 是当前执行点的环境,即,第 n 个周期下 $expr$ 求值之前的树形局部环境;而 e'则是当前执行点之后的环境,即,第 n 个周期下 $expr$ 求值之后的树形局部环境.次序编号 ord 对应于将被调用的子 node 实例的环境,而 ord'则对应于该表达式求值之后将被调用的子 node 实例的环境.换句话说,次序编号用于标记树形局部环境中子环境的位置.

在 Lustre⁻的语义中,对于表达式求值,形式断言可简化为

$$ge,te\vdash(expr,n)\Rightarrow v.$$

其含义为:在全局环境 ge 及某个 node 实例的时态环境 te 中,表达式 $expr$(单个表达式,而非表达式列表)于第 n 个时钟周期进行求值,求值结果为 v(单个值,而不是值的列表).局部环境以及所执行的子节点次序编号均不发生改变.实际上,总有$|te|=n$.这里重复使用 n,主要考虑到方便性.

在 Lustre⁻中,if 表达式的语义规则如下:

$$\frac{ge,te\vdash(c,n)\Rightarrow VtrueL\quad ge,te\vdash(a_1,n)\Rightarrow v_1\quad ge,te\vdash(a_2,n)\Rightarrow v_2}{ge\vdash(\text{if } c \text{ then } a_1 \text{ else } a_2,n)\Rightarrow v_1}\quad\text{(IF-True)}$$

$$\frac{ge,te\vdash(c,n)\Rightarrow VfalseL\quad ge,te\vdash(a_1,n)\Rightarrow v_1\quad ge,te\vdash(a_2,n)\Rightarrow v_2}{ge\vdash(\text{if } c \text{ then } a_1 \text{ else } a_2,n)\Rightarrow v_2}\quad\text{(IF-False)}$$

其中,a_1,a_2 为原子表达式.

在 Lustre⁻中,fby 表达式的语义规则如下:

$$\frac{n=1\quad ge,te\vdash(a_1,n)\Rightarrow v_1\quad ge,te\vdash(a_2,n)\Rightarrow v_2}{ge,te\vdash(a_1 \text{ fby } a_2,n)\Rightarrow v_1}\quad\text{(fby-1)}$$

$$\frac{n>1\quad ge,te'\vdash(a_1,n)\Rightarrow v_2\quad ge,te\vdash(a_2,n)\Rightarrow v_2'}{ge,te::te'\vdash(a_1 \text{ fby } a_2,n)\Rightarrow v_2}\quad\text{(fby-2)}$$

在 S-Lustre*中,fby 表达式的语义与 Lustre⁻中 fby 表达式的语义本质上没有什么差异,只是语义规则采用了不同的形式断言,使得二者的语义规则形式上有所不同.在 S-Lustre*中,fby 表达式的语义规则:

$$\frac{n=1\quad ge\vdash(e,ord,a_1,n)\Rightarrow(e,ord,vl_1)\quad ge\vdash(e,ord,a_2,n)\Rightarrow(e,ord,vl_2)}{ge\vdash(e,ord,a_1 \text{ fby } a_2,n)\Rightarrow(e,ord,vl_1)}\quad\text{(fby-1)}$$

$$\frac{n>1\quad ge\vdash((te',se'),ord,a_2,n-1)\Rightarrow((te',se'),ord,vl_2)\quad ge\vdash(((teh::te'),se),ord,a_2,n)\Rightarrow(((teh::te'),se),ord,vl_2')}{ge\vdash(((teh::te'),se),ord,a_1 \text{ fby } a_2,n)\Rightarrow(((teh::te'),se),ord,vl_2)}\quad\text{(fby-2)}$$

S-Lustre*与 Lustre⁻中都可以包含 call 表达式,所不同的是:在 S-Lustre*表达式中,call 表达式可以作为子表达式;而在 Lustre⁻中,call 表达式一定是一个单独的表达式,不会出现在其他表达式中.因为这种差异,我们在定义上述 if 和 fby 表达式的语义规则时,针对 S-Lustre*与 Lustre⁻使用了不同的形式断言.其他非 call 表达式的情形

也是类似的.然而对于 call 表达式,在 Lustre⁻中使用了与 S-Lustre*中相似的形式断言.由于 call 表达式的语义规则比较繁琐,本文不对此进行叙述.下面我们通过等式(equation)的语义规则,也可以体会到二者使用了相似的形式断言.

在 S-Lustre*中,等式的语义规则如下:

$$\frac{ge\vdash(((teh::te'),se),ord,expr,n)\Rightarrow(((teh_1::te');se');ord';vl) \quad localenvL\_setvars(teh_1;lhs;vl)=teh'}{ge\vdash(((teh::te'),se),ord,lhs=expr,n)\Rightarrow(((teh'::te'),se'),ord')}$$

其中,使用了形式断言:

$$ge\vdash(e,ord,eqs,n)\Rightarrow(e',ord').$$

其含义为:在全局环境 *ge* 和局部环境 *e* 中,等式列表 *eqs* 于第 *n* 个时钟周期进行计算,计算结果使环境 *e* 被改变为 *e*′,所执行的子节点次序编号由 *ord* 变为 *ord*′.

在这一规则中,使用了语义函数 *localenvL_setvars*(*le*,*lhs*,*vl*),它用值列表 *vl* 重置 *lhs* 中所有变量的状态,并返回新的(底层)局部环境.

在 Lustre⁻中,等式的语义规则使用同样的形式断言:

$$\frac{ge,(teh::te')\vdash(expr,n)\Rightarrow v \quad localenvL\_setvars(teh,lh::nil,v::nil)=teh'}{ge\vdash(((teh::te'),se),ord,lh=expr,n)\Rightarrow(((teh'::te'),se),ord)}$$

这个语义规则只适用于普通等式 *lh*=*expr*,而不适用于 *expr* 为 call 表达式的情形,所以,*se* 和 *ord* 均未发生改变.语义函数 *localenvL_setvars*(*le*,*lhs*,*vl*)的含义不变.

对于等式列表(可以包含 *expr* 为 call 表达式的等式),S-Lustre*和 Lustre⁻中使用了相同的语义规则:

$$\frac{}{ge\vdash(e,ord,nil,n)\Rightarrow(e,ord)} \quad \text{(equation-1)}$$

$$\frac{ge\vdash(e,ord,eq,n)\Rightarrow(e_1,ord_1) \quad ge\vdash(e_1,ord_1,eql,n)\Rightarrow(e_2,ord_2)}{ge\vdash(e,ord,(eq::eql),n)\Rightarrow(e_2,ord_2)} \quad \text{(equation-2)}$$

在等式列表语义的基础上,可进一步定义 node/function 的语义,这里不再详述.一个程序 prog 是由多个 node/function 组成的,如果指定某个 node 是主 node,那么我们可以定义整个程序的语义.由于 Lustre 程序处理的是流数据对象,所以其语义本质上在无穷流上执行,因此很适合通过 CoInductive 的方式来处理.然而出于技术上的考虑,我们采用的是用 Inductive 的方式来模拟这种效果.采用这种方法,定义整个程序的语义规则为

$$\frac{n>\max n}{ge\vdash(e,vargss,prog,n,\max n)\Rightarrow vretss} \quad \text{(Inductive-1)}$$

$$\frac{n\leqslant\max n \quad ge\vdash(e,prog.main,hd(vargs),n)\Rightarrow(e',hd(vrets)) \quad ge\vdash(e',tl(vargss),prog,n+1,\max n)\Rightarrow tl(vretss)}{ge\vdash(e,vargss,prog,n,\max n)\Rightarrow vretss} \quad \text{(Inductive-2)}$$

其中,*vargss* 和 *vretss* 分别是从第 *n* 个时钟周期开始的 input 和 output 流.当 *n*>max *n* 时,*vargss* 和 *vretss* 实际上不被计算,假设其取值为任意未定义值的流.*hd*(*s*)表示取流 *s* 的首元素,*tl*(*s*)用于获取流 *s* 去掉首元素后的流.形式断言:

$$ge\vdash(e,fd,vargs,n)\Rightarrow(e',vrets).$$

其含义为:在全局环境 *ge* 中,由 *fd* 定义的 node/function 于第 *n* 个时钟周期进行计算,实参值列表为 *vargs*,返回值列表为 *vargs*,计算结果使环境 *e* 被改变为 *e*′.形式断言:

$$ge\vdash(e,input,prog,n,\max n)\Rightarrow output.$$

其含义为:在全局环境 *ge* 中,程序 *prog* 于第 *n* 个时钟周期开始逐个周期反复执行,直至第 max *n* 个时钟周期结

束,主 node 的初始局部环境为 *e*,*input* 和 *output* 分别始于第 *n* 个周期的输入值列表和输出值列表的流(始于第 max *n*+1 个时钟周期的输入值列表和输出值列表均取未定义值).

以上我们描述了 S-Lustre*和 Lustre$^-$语义定义的主要思想.对于 Lustre^{F}来说,由于已消去时态算子(fby,arrow 等),所以其语义定义中可以不考虑时态环境(如图 4 所示的 *te*).此外,其语义类似于 Lustre$^-$,这里不再赘述.

2.2 语义保持性举例

我们仅给出 S-Lustre*到 Lustre$^-$的语义保持性定理及证明概要,以此为例说明语义保持性的基本含义,其余层次之间的证明框架大致相似.这是一种单向的语义保持,之前已经说过,由于 Lustre 语义是确定的,所以足以达到我们证明翻译正确性的目标.

定理 1. *trans_programS_correct*(从 S-Lustre*到 Lustre$^-$的语义保持性):

$$\forall progS, progM, eS, eM, mainS, mainM, varges, vretss, \max n.$$
$$initial\_stateS(progS, eS, mainS) \wedge initial\_stateM(progM, eM, mainM) \wedge trans(progS) =$$
$$progM \wedge geS \vdash iter\_exec\_programS(progS, mainS, es, 1, \max n, vargss, vretss) \rightarrow$$
$$geM \vdash iter\_exec\_programM(progM, mainM, eM, 1, \max n, vargss, vretss).$$

其中,S-Lustre*程序 *progS* 经过 *trans* 翻译函数后得到 Lustre$^-$程序 *progM*.*initial_stateS* 将 *progS* 的执行环境 *eS* 初始化为空环境,将主节点置为 *mainS*;*initial_stateM* 将 *progM* 的执行环境 *eM* 初始化为空环境,将主节点置为 *mainM*.*iter_exec_programS* 的定义对应第 2.1 节的 Iuductive-1 和 Iuductive-2.*iter_exec_programM* 也类似.语义保持性的含义:如果 *progS* 在环境 *eS* 下的执行对输入实参的流 *vargss* 返回值的流 *vretss*,那么 *progM* 在环境 *eM* 下的执行对输入实参的流 *vargss* 同样是返回值的流 *vretss*;而 *eS* 和 *eM* 从初始化为空环境后自然匹配,然后从第 1 个周期到之后任何一个周期皆保持匹配关系.

为便于归纳证明,我们将定理 *trans_programS_correct* 推广到更一般的定理:

定理 2. *trans_programS_correct_general*(从 S-Lustre*到 Lustre$^-$语义保持性的一般情形):

$$\forall progS, progM, eS, eM, mainS, mainM, vargss, vretss, pid, n, \max n.$$
$$trans(progS) = progM \wedge trans\_node(mainS, pid) = mainM \wedge envL\_match(eS, eM) \wedge isPID(pid) \wedge$$
$$geS \vdash iter\_exec\_programS(progS, mainS, eS, n, \max n, vargss, vretss) \rightarrow$$
$$geM \vdash iter\_exec\_programM(progM, mainM, eM, n, \max n, vargss, vretss).$$

其中,*trans_node* 为主节点翻译函数,*envL_match* 是环境的匹配关系.*iter_exec_programS* 和 *iter_exec_programM* 分别是源和目标程序语义执行关系式.

从定理 *trans_programS_correct_general* 可知,我们实际是需要证明:同样的输入/输出在 LustreS 中满足执行关系式 *iter_exec_programS*,那么在 LustreM 中仍然满足执行关系式 *iter_exec_programM*,而执行可以理解为一个周期一周期主节点函数的串行执行.我们可以将程序执行的等级性转化为求节点函数的环境匹配的问题,即证明:

引理. *trans_nodeS_all_correct*(从 S-Lustre*节点到 Lustre$^-$节点的语义保持性):

$$\forall eS, eS', eM, nodeS, nodeM, pid, vargs, vrets, n.$$
$$ge \vdash eval\_nodeS(eS, eS', nodeS, vargs, vrets, n) \wedge trans\_node(nodeS, pid) =$$
$$nodeM \wedge envL\_match(eS, eM) \wedge isPID(pid) \rightarrow$$
$$\exists eM'.(ge \vdash eval\_nodeM(tge, eM, eM', nodeM, vargs, vrets, n) \wedge envL\_match(eS', eM')).$$

其中,*Trans_node* 是节点翻译函数,*eval_nodeS* 和 *eval_nodeM* 分别是源和目标节点的语义求值函数.

引理 *trans_nodeS_all_correct* 是整个语义保持性证明的核心,其证明也比较复杂.由于 *eval_nodeS* 是互归纳定义的,因此需要用到互归纳证明.

在引理 *trans_nodeS_all_correct* 的证明中,需要在不同阶段不断分析修改其表现形式.比如,我们需要用到从 S-Lustre*表达式转换到 Lustre$^-$等式列表、从 S-Lustre*表达式列表转换到 Lustre$^-$等式列表、从 S-Lustre*等式转换到 Lustre$^-$等式列表、从 S-Lustre*等式列表转换到 Lustre$^-$等式列表等一系列转换的语义保持性.

这些语义保持性都可以描述成:翻译前后的程序单元执行时,对于语义环境的改变是相一致的.不同类别的程序单元,根据其翻译前后的特征,其语义保持性的表述形式会有所不同.比如,对于表达式翻译前后的语义保持性,我们需要分别就原子表达式、一元表达式、二元表达式、条件表达式、fby 表达式以及 call 表达式等相应的语义规则分别加以证明.

诸如原子表达式、fby 表达式的情形比较简单,由于前端的预处理步骤(图 2 中的 Pre-Processing),使得 S-Lustre*和 Lustre$^-$中的 fby 表达式形式上没有差别,它们的操作数都只可能是原子表达式.此类表达式翻译后不会引入新的中间变量,也不会引入新的等式列表.

条件表达式就要复杂一些,要引入新的中间变量和新增等式列表.

下面我们仅以条件表达式为例,来简单说明其语义保持性的证明.

先看 S-Lustre*中条件表达式 if *ex* then ex_1 else ex_2 的翻译过程:将 *ex*,ex_1 和 ex_2 分别递归翻译到 Lustre$^-$中的等式列表 *eqs*,eqs_1 和 eqs_2.设 *ex*,ex_1 和 ex_2 的结果分别对应到 *eqs*,eqs_1 和 eqs_2 中某等式的左端 *id*,id_1 和 id_2,则 if *ex* then ex_1 else ex_2 对应到 Lustre$^-$中就对应于表达式 if *id* then id_1 else id_2.然而,除了 *id*,id_1 和 id_2 以及 *ex*,ex_1 和 ex_2 中的变量外,*eqs*,eqs_1 和 eqs_2 中还有许多其他的中间变量.

首先,我们要证明翻译前后程序中 *id* 的相关性质:翻译前环境中的 *id* 会保留至翻译后,并且新添的 *id* 不会与原环境中 *id* 相交.如果翻译前后环境中的 *id* 之间满足这一性质,并且同一名称的 *id* 在两个环境中的取值相等,则称翻译前后的环境是匹配的.

这样,我们的目标成为:if *ex* then ex_1 else ex_2 在 S-Lustre*语义中对环境的改变,能够匹配依次执行 *eqs*,eqs_1,eqs_2 和 if *id* then id_1 else id_2 在 Lustre$^-$语义中对环境的改变,并且 if *ex* then ex_1 else ex_2 求值的结果与 if *id* then id_1 else id_2 的求值结果相同.

具体到语义规则,我们仅以 If-False 为例,If-True 的情形类似.

证明思路是:设任意可以匹配的 S-Lustre*环境 *e* 和 Lustre$^-$环境 *eM*.

假设在 S-Lustre*中,在环境 *e* 和实例编号 *ord* 下执行 if *ex* then ex_1 else ex_2 的结果为 vl_2 以及 *ord* 改变为 ord_3.可以根据两个规则 If-True 或 If-False 得出这个结果,我们假设是 If-False.那么,根据 S-Lustre*中 If-False 规则,存在 e_1,e_2,e_3 以及 ord_1,ord_2,ord_3,满足:1) 在环境 *e* 下执行 *ex* 的结果为 *Vfalse*(为了简洁,我们忽略周期编号 *n* 以及列表形式的值 *Vfalse*::*nil*,下同),环境改变为 e_1,代表当前节点实例号的 *ord* 改变为 ord_1;2) 环境 e_1 下执行 ex_1 的结果为 vl_1,环境改变为 e_2,*ord* 由 ord_1 变为 ord_2;3) 环境 e_2 下执行 ex_2 的结果 vl_2,环境改变为 e_3,*ord* 由 ord_2 变为 ord_3.

然后我们证明:1) 在 *eM* 环境下执行等式列表 *eqs* 后,环境变为可与 e_1 匹配的 eM_1,eM_1 中 *id* 的取值为 *Vfalse*,*ord* 改变为 ord_1;2) 在 eM_1 环境下执行等式列表 eqs_1 后,环境变为可与 e_2 匹配的 eM_2,eM_2 中 id_1 的取值为 vl_1,*ord* 由 ord_1 变为 ord_2;3) 在 eM_2 环境下执行等式列表 eqs_2 后,环境变为可与 e_3 匹配的 eM_3,eM_3 中 id_2 的取值为 vl_2,*ord* 由 ord_2 变为 ord_3.这个过程会递归用到第 2.1 节中的 equations 规则以及其他规则.

最后,我们可由第 2.1 节中关于 Lustre$^-$的 If-False 规则得出结论:在环境 *eM* 和实例编号 *ord* 下,依次执行 *eqs*,eqs_1,eqs_2 和 if *id* then id_1 else id_2 的结果使环境变为 eM_3,*ord* 改变为 ord_3.

2.3 并行语义到串行语义的转换之拓扑排序

如图 2 所示,在 L2C 项目的设计方案中,串行化(sequentializing)之前的各语言层次采用并发语义,其后的各语言层次采用顺序语义.串行化的正确性意味着语义过渡是合理的.

执行串行化算法的结果就是使程序的每个 node/function 内部的等式确定一个满足因果关系的计算序列(即,前面的求值不依赖于后面的变量).串行化正确性证明的目标是:(A) 先要证明串行化前后不会造成等式或者节点的缺失,前后代码相互之间是一种重新排列的关系;(B) 串行化前的程序根据并发语义得到的求值结果,与串行化后的程序根据顺序语义得到的求值结果是相同的.

串行化前的程序 *P* 满足性质 *sound*(*P*),其中包括了并发语义求值结果的确定性.对于后者的证明,需要验证所有交叉执行(interleaving)结果的一致性.在现阶段的 L2C 项目实现中,并发粒度是等式,即,串行化算法中的拓

扑排序是以单个等式为单位.

虽然以单个等式为单位的并发粒度可以满足合作企业目前的实际需求,但在 L2C 项目的总体规划中,将采用更细的并发粒度(以单个基本操作为单位),这会在理论上更加合理.当然,工作量会加大不少.

2.4 时态消去

时态消去对应图 2 中的 folding 过程,它将 Lustre$^-$ AST 翻译至 Lustre^F AST.Lustre$^-$和 Lustre^{F}的语法相近,只是后者没有了时态算子(arrow 和 fby).在对 arrow 和 fby 进行翻译时,需要引进两种变量:一种是自定义变量,作为缓存变量存在用来保存上个周期的值;另一种是标志变量,区别为是否第 1 个周期.

arrow 的翻译相对简单,我们将 $lh=a_1\rightarrow a_2$ 翻译为 lh=if *flagid* then a_2 else a_1,其中,*flagid* 为标志变量.对于 fby 的翻译,则需要保存上一周期的值,我们将 $lh=a_1$ fby a_2 翻译为两个等式:

lhs=if *flagid* then *preatom* else a_1

…

preatom=a_2

其中,第 2 个等式 *preatom*=a_2 比较特殊,将它放在所有其他等式的后面,这样做才能保证语义的等价性,其作用主要是用来保存上一个周期的值.这样的翻译给证明带来了困难,翻译不再是线性到线性的对应翻译,而是线性的翻译到两段隔开的程序,因此在证明中需要额外的定义来描述分割的程序.

需要证明的从 Lustre$^-$到 Lustre^{F}翻译的语义保持性与之前 S-Lustre*到 Lustre$^-$的语义保持性定理类似,此处不再赘述.

2.5 翻译到C代码

图 2 中的 translation 过程将 Lustre^F AST 翻译到 Clight AST,后者可输出为 C 代码.

从 Lustre^F 到 Clight 的翻译难度不大,但语义保持性证明较为困难,工作量也较大.为了可以直接使用 CompCert 项目中的 Clight 语义定义,需要在执行方式、特别是存储模式方面进行很好的适应.

2.6 时钟归一化

时钟归一化(对应图 2 中的 Clock-Normalizing 过程)可以消去程序中的 when 算子,使得从 Clock-Normalized Lustre*开始往后的中间语言程序中只含单一时钟,即,默认的全局时钟.这使得后续的翻译阶段有所简化,特别是使之后各个阶段的翻译正确性证明大大降低了难度.

从任意一个时钟转化到默认全局时钟的基本思路是:

首先,把每个子表达式 *expr* 的时钟用一个基本时钟的数据流 *cc* 表示出来.为了方便,称 *cc* 为 *expr* 的时钟流.举个例子,如果 *expr_a* 的时钟为 when (*c* when *d*),那么这个数据流的时钟流为 *cc_a*=if *d* then *c* else false.

其次,定义一个运算 *cur*,它将任意时钟的数据流转化到基本时钟的数据流.假设有一个数据流 *expr*,其本来的时钟流为 *cc*.将 *expr* 本身的定义扩展到任意时钟周期上.再定义一个新的数据流 *y*=if *cc* then *expr* else *pre y*.它们之间的关系示意:

expr	_	_	_	1	_	_	_	2	_	_	3	_	_	_	4 …
expr	?	?	?	1	?	?	?	2	?	?	3	?	?	?	4 …
cc	f	f	f	t	f	f	f	t	f	f	t	f	f	f	t …
y	_	_	_	1	1	1	1	2	2	2	3	3	3	3	4 …

(扩展到任意周期)

对包含时态运算的等式进行特殊的转换:

$y=x$ when c	转换为	$y=cur(x,cc)$	(*cc* 为 *x* 的时钟流,注意,这不同于 *c*)
$y=pre(x)$	转换为	$y=cur(pre(x),cc)$	
$z=y\rightarrow x$	转换为	z=if *cf* then *y* else *x*	

定义第 1 时钟周期的 *flag*,即 *cf*.*cf* 与 *cc* 的关系示意:

cc	f f f t f f f t f f f t f f f t …
cf	t t t t t t t t f f f f f f f f …

先定义一个 *cf′*=true→(*pre cf′*) and not (*pre cc*),有:

cf′ t t t t f f f f f f f f f f f f ...

然后,再由它得到 *cf*=*cur*(*cf′*,*cc*)即可.

目前已完成一个原型系统,用于验证这个时钟归一化算法.首先定义了一个具有代表性的小语言,然后定义该语言的一种并发操作语义,然后证明这一算法满足语义保持性.整个过程已在 Coq 中实现.

2.7 类型检查

图 2 所示的 Type-Checking 过程除了传统的静态语义检查以外,还包括时钟演算、因果性以及初始化等检查工作.这个过程的正确性主要是确保类型检查后的 Well-Typed Lustre* AST 满足这些语义规则所刻画的性质,同时,这些处理前后需保持所处理程序信息的完整性.

在目前的实现中,有关因果性检查的部分是在 Sequentializng 阶段完成的.有关普通类型检查和时钟演算的内容,包含正确性证明也已经完成.然而,这个阶段,包括后续的 Pre-processing,Clock Normalizing 以及 Sequentializng 等阶段的正确性证明工作还没有连贯起来.在此之前,需要先完善并发操作语义的工作.

3 现状及下一步工作

对于不包含 when,FBY 以及 array 的 Lustre*语言(见附录 A),目前已实现如图 2 总体设计所描述的完整编译过程,并且从串行化(等式粒度)开始的各阶段的翻译正确性证明均已完成.Parsing 阶段直接通过 OCaml 代码实现,其余阶段都在 Coq 工具中实现,包含各阶段的语法、语义、翻译过程以及翻译正确性证明.从翻译过程相关的 Coq 代码抽取的 OCaml 代码、Parsing 阶段的 OCaml 代码、驱动及配置相关的 OCaml 代码联编后,生成一个完整的编译程序.

在目前的工作基础上,扩充对 FBY、高阶运算以及 array 的支持即可满足安全关键系统开发的要求.相关的产品化工作正在进行中,企业方已完成一个扩展语言的编译器试用版本(扩展部分仅完成了翻译过程,验证工作正在进行),2012 年 9 月开始在非安全级系统中进行降级试用,所生成的代码由质量监督部门通过人工代码审核、与国外同类复合航空标准的生成工具(SCADE)对比及大量测试实验等方法进行对比,至今表现优秀,没有发现任何与编译过程相关的错误.

从研究角度来看,正在进行的工作还包括完善并发操作语义定义,并将 Type-Checking,Pre-processing,Clock Normalizing 以及 Sequentializng 等阶段连贯起来.相关原型系统已基本完成.

未来我们将进一步扩充 Lustre*的语言成分,完成对高阶运算、数组及 FBY 算子的验证工作,并与企业方合作完成产品的验收工作.

4 总 结

本文对正在开展的同步数据流语言 Lustre*到 C 的可信编译器项目进行了介绍.该编译器的核心翻译过程借助辅助定理证明工具 Coq,经过严格的形式化验证.已完成的部分在扩充数组、高阶运算和 FBY 时态运算之后能够满足国内安全关键领域的实际需求.相关研究工作在某些方面取得了突破.

致谢 在此,我们向对本文的工作给予支持和建议的同行表示感谢.

References:

[1] Knight JC. Safety critical systems: Challenges and directions. In: Proc. of the 24th Int'l Conf. on Software Engineering (ICSE 2002). IEEE Xplore, 2002. 547−550. [doi: 10.1145/581339.581406]

[2] Milner R. Calculi for synchrony and asynchrony. Theoretical Computer Science, 1983,25(3):267−310.

[3] Halbwachs N. Synchronous programming of reactive systems, a tutorial and commented bibliography. In: Proc. of the 10th Int'l Conf. on Computer-Aided Verification (CAV'98). LNCS 1427, Springer-Verlag, 1998. [doi: 10.1007/BFb0028726]

[4] Halbwachs N. A synchronous language at work: The story of Lustre. In: Proc. of the 2nd ACM/IEEE Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE 2005). Washington: IEEE Computer Society, 2005. [doi: 10.1109/MEMCOD.2005.1487884]

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

[6] 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]

[7] Berry G. The foundations of esterel. In: Proc. of the Proof, Language and Interaction: Essays in Honour of Robin Milner. Cambridge: The MIT Press, 2000. 425−454.

[8] Caspi P, Pilaud D, Halbwachs N, Plaice J. Lustre: A declarative language for programming synchronous systems. In: Proc. of the 14th ACM Symp. on Principles of Programming Languages (POPL'87). New York: ACM, 1987. [doi: 10.1145/41625.41641]

[9] Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous dataflow programming language LUSTRE. Proc. of the IEEE, 1991,79(9):1305−1320.

[10] 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]

[11] Le Guernic P, Talpin JP, Le Lann JC. Polychrony for system design. Journal for Circuits, Systems and Computers, 2003,12(3): 261−304. [doi: 10.1142/S0218126603000763]

[12] http://gcc.gnu.org/

[13] http://www.plumhall.com/stec.html

[14] http://www-verimag.imag.fr/Lustre-V6.html

[15] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2011). New York: ACM, 2011. 283−294. [doi: 10.1145/1993498.1993532]

[16] http://www.esterel-technologies.com/products/Scade-suite/

[17] http://www.commoncriteriaportal.org/cc/

[18] Summary of difference between DO-178B and DO-178C. http://faaconsultants.com/html/do-178c.html

[19] McCarthy J, Painter J. Correctness of a compiler for arithmetical expressions. In: Proc. of the Symposia in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science. Washington: AMS, 1967. 33−41.

[20] Milner R, Weyhrauch R. Proving compiler correctness in a mechanized logic. In: Proc. of the 7th Annual Machine Intelligence Workshop, Vol.7. Edinburgh: Edinburgh University Press, 1972. 51−72.

[21] Dave MA. Compiler verification: A bibliography. ACM SIGSOFT Software Engineering Notes, 2003,28(6):2−5. [doi: 10.1145/966221.966235]

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

[23] Chlipala A. A certified type-preserving compiler from lambda calculus to assembly language. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2007). New York: ACM, 2007. 54−65. [doi: 10.1145/1273442.1250742]

[24] Klein G, Nipkow T. A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. on Programming Languages and Systems, 2006,28(4):619−695. [doi: 10.1145/1146811]

[25] The Coq Development Team. The Coq proof assistant reference manual. Version V8.3, 2010. http://coq.inria.fr/

[26] Bertot Y, Castéran P. Interactive theorem proving and program development—Coq'art: The calculus of inductive constructions. In: Proc. of the Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[27] Morrisett G, Technical perspective: A compiler's story. Communications of the ACM, 2009,52(7):106−106. [doi: 10.1145/1538788.1538813]

[28] Pnueli A, Siegel M Singerman E. Translation validation. In: Proc. of the TACAS'98. LNCS 1384, Springer-Verlag, 1998. 151−166.

[29] Pnueli A, Shtrichman O, Siegel M. Translation validation for synchronous languages. In: Proc. of the ICALP'98. LNCS 1443, Springer-Verlag, 1998. 235−246. [doi: 10.1007/BFb0055057]

[30] Ngo VC, Talpin JP, Gautier T, Le Guernic P, Besnard L. Formal verification of synchronous data-flow compilers. Project-Team ESPRESSO Research Report, No.7921, 2012.

[31] Ngo VC, Talpin JP, Gautier T, Le Guernic P. Formal verification of transformations on abstract clocks in synchronous compilers. Project-Team ESPRESSO Research Report, No.8064, 2012.

[32] Ngo VC, Talpin JP, Gautier T, Le Guernic P, Besnard L. Formal verification of compiler transformations on polychronous equations. In: Latella D, Treharne H, eds. Proc. of the IFM 2012. LNCS 7321, Springer-Verlag, 2012. 113−127. [doi: 10.1007/978-3-642-30729-4_9]

[33] Dutertre B, de Moura L. The YICES SMT solver. 2009. http://yices.csl.ri.com

[34] Tristan JB, Govereau P, Morrisett G. Evaluating value-graph translation validation for LLVM. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming and Language Design Implementation (PLDI 2011). New York: ACM, 2011. 295−305. [doi: 10.1145/1993316.1993533]

[35] Paulin C, Pouzet M. Certified compilation of Scade/lustre. 2006. https://www.lri.fr/~paulin/lustreincoq.pdf

[36] LUCID SYNCHRONE home. http://www.di.ens.fr/~pouzet/lucid-synchrone/

[37] Bertails A, Biernacki D, Paulin C, Pouzet M. A certified compiler for the synchronous language Lustre. In: Proc. of the TYPES 2007. 2007. http://users.dimi.uniud.it/types07/slides/Bertails.pdf

[38] Biernacki D, Colaco JL, Hamon G, Pouzet M. Clock-Directed modular code generation of synchronous data-flow languages. In: Proc. of the ACM Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES). New York: ACM, 2008. [doi: 10.1145/1375657.1375674]

[39] Biernacki D, Colaco JL, Pouzet M. Clock-Directed modular code generation from synchronous block diagrams. In: Proc. of the Workshop on Automatic Program Generation for Embedded Systems (APGES 2007). New York: ACM, 2007. 78-89. [doi: 10.1145/1480881.1480893]

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

[41] http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf

附录 A. 当前源语言原型 Lustre*的语法定义(EBNF)

⟨program⟩::={⟨decls⟩}

⟨decls⟩::=⟨type_block⟩|⟨const_block⟩|⟨user_op_decl⟩

⟨type_block⟩::="**type**"{⟨type_decl⟩";"}

⟨type_decl⟩::=ID"="⟨type_expr⟩

⟨type_expr⟩::="**bool**"|"**int**"|"**real**"|"**char**"|ID

|"{"⟨field decl⟩{","⟨field_decl⟩}"}"|⟨type_expr⟩^⟨expr⟩

⟨field_decl⟩::=ID":"⟨type_expr⟩

⟨const_block⟩::="**const**"{⟨const_decl⟩";"}$^{+}$

⟨const_decl⟩::=ID":"⟨type_expr⟩"="⟨expr⟩

⟨user_op_decl⟩::=⟨op_kind⟩ID⟨params⟩"**returns**"⟨params⟩⟨opt_body⟩

⟨op_kind⟩::="**node**"|"**function**"

⟨params⟩::="("[⟨var_decls⟩{";"⟨var_decls⟩}]")"

⟨var_decls⟩::=⟨var_id⟩{","⟨var_id⟩}":"⟨type_expr⟩[⟨when_decl⟩]

⟨var_id⟩::=["**clock**"]ID

⟨when_decl⟩::="**when**"⟨clock_expr⟩

⟨opt_body⟩::=";"|⟨equation⟩";"|[⟨local_block⟩]"**let**"{⟨equation⟩";"}"**tel**"[";"]

⟨equation⟩::=⟨lhs⟩“=”⟨expr⟩
⟨lhs⟩::=⟨lhs_id⟩{“,”⟨lhs_id⟩}
⟨lhs_id⟩::=ID|“_”
⟨local_block⟩::=“**var**”{⟨var_decls⟩“;”}
⟨expr⟩::=ID|⟨atom⟩|⟨list_expr⟩|⟨tempo_expr⟩|⟨arith_expr⟩|⟨relation_expr⟩
|⟨bool_expr⟩|⟨array_expr⟩|⟨struct_expr⟩|⟨switch_expr⟩|⟨apply_expr⟩
⟨atom⟩::=⟨bool_atom⟩|CHAR|INTEGER|FLOAT
⟨bool_atom⟩::=“**true**”|“**false**”
⟨list_expr⟩::=“(”⟨list⟩“)”
⟨list⟩::=[⟨expr⟩{“,”⟨expr⟩}]
⟨tempo_expr⟩::=“**pre**”⟨expr⟩|⟨expr⟩“→”⟨expr⟩
|“**FBY**”“(”⟨list⟩“;”⟨expr⟩“;”⟨list⟩“)”|⟨expr⟩“**when**”⟨clock_expr⟩
⟨clock_expr⟩::=“(”ID“)”
⟨arith_expr⟩::=⟨unary_arith_op⟩⟨expr⟩|⟨expr⟩⟨bin_arith_op⟩⟨expr⟩
⟨unary_arith_op⟩::=“−”|“+”|“**int**”|“**real**”
⟨bin_arith_op⟩::=“+”|“−”|“*”|“/”|“**mod**”|“**div**”
⟨relation_expr⟩::=⟨expr⟩⟨bin_relation_op⟩⟨expr⟩
⟨bin_relation_op⟩::=“=”|“<>”|“<”|“>”|“⩽”|“⩾”
⟨bool_expr⟩::=“**not**”⟨expr⟩|⟨expr⟩⟨bin_bool_op⟩⟨expr⟩
⟨bin_bool_op⟩::=“**and**”|“**or**”|“**xor**”
⟨switch_expr⟩::=“**if**”⟨expr⟩“**then**”⟨expr⟩“**else**” ⟨expr⟩
|“(”“**case**”⟨expr⟩“**of**”{|⟨case_expr⟩}$^{+}$“)”
⟨case_expr⟩::=⟨pattern⟩“:”⟨expr⟩
⟨pattern⟩::=ID|CHAR|[“−”]INTEGER|⟨bool_atom⟩|“_”
⟨struct_expr⟩::=⟨expr⟩“.”ID|“{”⟨label_expr⟩{“,”⟨label_expr⟩}“}”
⟨label_expr⟩::=ID“:”⟨expr⟩
⟨array_expr⟩::=⟨expr⟩⟨index⟩|⟨expr⟩^⟨expr⟩
⟨index⟩::=“[”⟨expr⟩“]”
⟨apply_expr⟩::=ID“(”⟨list⟩“)”

附录 B. Lustre*程序的一个样例

Lustre*程序:

```
node counter (ck: bool) returns (x: int);
var y: int;
let
    y=0→x;
    x=0→if ck
        then pre(y)+1
        else pre(y);
tel
node sample (c1:bool;clock c2:bool) returns (x: int when c2)
let
```

x=counter(c_1) when c_2;

tel

执行效果:

Variables	Cycles:	0	1	2	3	4	5	6	7	...
c_1, c_2, ck		F	T	F	T	F	F	T	T	...
x *in* node counter		0	1	1	2	2	2	3	4	...
y *in* node counter		0	1	1	2	2	2	3	4	...
pre(y) *in* node counter		nil	0	1	1	2	2	2	3	...
x *in* node sample			1		2			3	4	...

注:Lustre*语言中,所有的数据都是流(stream),对应于无穷多个逻辑时钟周期(cycle),每个周期有一个取值.所有的运算都作用于流.每个逻辑时钟周期都执行同样的计算(即相同的代码),并假设在逻辑时钟周期内计算一定可以完成,即满足所谓的同步假设(synchronization hypothesis).各个周期内的计算由多个计算节点(node)组成,每个计算节点内部是一个等式的集合,每个等式都定义流上的运算,等式之间是并发执行,是数据驱动的计算过程.时态(temporal)运算是 Lustre 语言中具有特色的流运算,基本的时态运算包括 *pre*,*FBY*,*when*,*arrow*(→)等算子.从上述代码及其执行效果,不难理解部分这些算子的含义.

石刚(1972－),男,重庆人,博士生,讲师,CCF 学生会员,主要研究领域系统软件的形式化验证,嵌入式软件环境.
E-mail: shigang1022@163.com

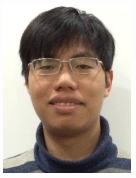
王生原(1964－),男,博士,副教授,CCF 高级会员,主要研究领域为程序设计语言与系统,并发/分布对象计算,Petri 网应用.
E-mail: wwssyy@tsinghua.edu.cn

董渊(1973－),男,博士,副研究员,CCF 会员,主要研究领域为操作系统,编译系统,基于语言的可信软件.
E-mail: dongyuan@tsinghua.edu.cn

嵇智源(1956－),男,高级工程师,主要研究领域为计算机专业及嵌入式软件,国家科技计划信息技术领域技术管理.
E-mail: jzy@htrdc.com

甘元科(1983－),男,硕士,主要研究领域为形式化验证.
E-mail: laogan@gmail.com

张玲波(1987－),男,硕士,助理研究员,主要研究领域为操作系统与形式化验证.
E-mail: zhanglingb@gmail.com



张煜承(1989－),男,主要研究领域为编程语言.
E-mail: yczhang89@gmail.com

王蕾(1983－),男,硕士,工程师,主要研究领域为编译原理,操作系统,计算机网络.
E-mail: Leopardguo25@gmail.com

杨斐(1984－),男,助理工程师,主要研究领域为可信编译器形式化验证.
E-mail: Guo_guo_guo@yeah.net