软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(7):2003−2017 [doi: 10.13328/j.cnki.jos.005755]

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

同步数据流语言可信编译器 Vélus 与 L2C 的比较*

康跃馨, 甘元科, 王生原

(清华大学 计算机科学与技术系,北京 100084)

通讯作者: 康跃馨, E-mail: kyx16@mails.tsinghua.edu.cn

摘 要: 同步数据流语言(如 Lustre、Signal)在航空、高铁、核电等安全关键领域得到广泛应用.例如,适合这些领域实时控制系统建模和开发的 Scade 工具就是基于一种类 Lustre 语言.这类语言相关开发工具,特别是编译器的安全性问题也自然受到高度关注.近年来,基于形式化验证实现可信编译器构造成为程序设计语言领域的研究焦点之一,也取得了瞩目的成果,如 CompCert 项目实现了产品级的可信 C 编译器.同样,人们也采用这种方法开展了同步数据流语言可信编译器的研发工作.主要关注从事这一工作的两个长线项目,二者均研发面向基于 Lustre 的同步数据流语言编译器,分别以 Vélus 和 L2C 代称.对 Vélus 和 L2C 从多个重要的角度进行较为深入的分析与比较.

关键词: 同步数据流语言;形式化验证的编译器;Lustre 语言

中图法分类号: TP314

中文引用格式: 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. http://www.jos.org.cn/1000-9825/5755.htm

英文引用格式: Kang YX, Gan YK, Wang SY. Comparison of two trustworthy compilers Vélus and L2C for synchronous languages. Ruan Jian Xue Bao/Journal of Software, 2019,30(7):2003−2018 (in Chinese). http://www.jos.org.cn/1000-9825/5755.htm

Comparison of Two Trustworthy Compilers Vélus and L2C for Synchronous Languages

KANG Yue-Xin, GAN Yuan-Ke, WANG Sheng-Yuan

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

Abstract: Synchronous data-flow languages, such as Lustre and Signal, have been widely used in safety-critical industrial areas, such as airplane, high-speed railways, nuclear power plants, and so on, for example, Scade, a tool suitable for modeling and developing a real-time control systems in those areas, is based on a Lustre-like language. Naturally, the safety of development tools, especially compilers, for such languages, has been paid highly attentions. In recent years, the construction of a trustworthy compiler based on formal verification has become one of the research focuses in the field of programming language, and remarkable achievements have also been achieved, for example, a product level trustworthy C compiler has been developed in the CompCert project. Similarly, this method has been used to develop the trustworthy compilers of a synchronous data flow language. Two long term projects for such research work, called Vélus and L2C respectively in this study, are focused here. Either of them has been developing a compiler for a Lustre-like synchronous data-flow language. The analysis and comparison are deeply carried out in terms of various points between Vélus and L2C.

Key words: synchronous data-flow language; formally certified compiler; Lustre language

应用于航空航天、高速铁路、核电能源和医疗卫生等领域的安全攸关系统[1]一旦失效,将给人类生命财产、社会生产、生活环境带来巨大的破坏.如何为安全关键系统构造一个基础的安全软件环境是需要面对的首要问

* 基金项目: 国家科技重大专项(MJ-2015-D-066)
Foundation item: National Science and Technoloy Major Project of China (MJ-2015-D-066)
本文由“软件形式化验证”专题特约编辑贺飞副教授、张立军研究员推荐.
收稿时间: 2018-07-15; 修改时间: 2018-09-28; 采用时间: 2018-12-13; jos 在线出版时间: 2019-03-28
CNKI 网络优先出版: 2019-03-29 09:14:24, http://kns.cnki.net/kcms/detail/11.2560.TP.20190329.0914.007.html

题,尤其是对操作系统、编译器等基础软件而言.

作为用来产生代码的工具,编译器的实现和维护自然经过了“精雕玉琢”,因此编译器的正确性问题容易被人们忽视.但了解实际情况的人都知道,编译器的“误编译”问题是司空见惯、习以为常之事.对于许多领域来说,由于一般不会引发本质的问题,又是小概率事件,人们也往往容易忽视误编译所带来的负面影响.然而,对于安全关键系统而言,则必须考虑因编译器引入的错误,否则,花大力气在源程序级的验证工作可能在目标程序级失效.实际上,如航空领域的RTCA DO-178B/C标准,编译器属于需要鉴定的工具类软件,需要按照机载软件的要求那样对待.

要保障编译器的正确性/可靠性,传统上采用大量的测试以及严格的软件过程管理来实现,这并不能杜绝“误编译”现象.对编译器进行正确性验证,而不仅仅是测试,是解决问题的根本途径.最严格的验证手段莫过于采用形式化方法,近年来,有关编译器形式化验证的研究工作取得了长足的进步,达到了实用化水平,为未来新的工业标准的制定奠定了强有力的基础.CompCert 编译器[2]是经过形式化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(后来也支持 IA、ARM 以及 RISC-V 后端),其编译过程划分为多个阶段,每个阶段的翻译正确性都借助证明辅助工具 Coq[3]进行了证明,且这些证明可由独立的证明检查器检查,这是迄今最强的形式化验证手段,达到了人们所能期望的最高可信程度[4].Yang 等人关于Csmith[5]的研究工作表明:CompCert 在正确性方面的表现明显优于常用的开源或商用 C 编译器.因 CompCert 编译器的杰出成就,其代表性论文[6]的作者 Leroy 获得了 2016 年度的 10 年前最有影响 POPL 论文奖(Most Influential POPL Paper Award).

C 语言一直以来是安全关键领域中使用最为普遍的开发语言,人们下了很大功夫,使基于 C 语言的系统开发更加安全、高效.然而,毕竟 C 语言程序层次较低,不易使人们聚焦于问题本身,开发效率受到很大影响,并且也难于验证,于是基于模型/模型驱动的开发逐步兴起并成为业界主流,由模型自动生成的代码(C 代码为主)占比越来越高.比较著名的建模语言/工具,如 Simulink[7]、Scade[8]等.

有一类建模语言称为同步语言,特别适合于实时控制系统的开发.所有同步语言均遵循同步假设,即每个周期内的计算(从输入值得到输出值)都是瞬间完成的.另外,同步语言的语义被要求具有确定性.同步假设以及确定性可以极大地简化实时系统的验证.Esterel[9]、Argos[10]、Lustre[11,12]和 Signal[13]是几种有代表性的同步语言.其中,Esterel 是命令式语言;Argos 是基于并行和分层自动机的图形化语言;Lustre 和 Signal 是陈述式语言,具有数据流特征,常称为同步数据流语言.Lustre 是函数风格的语言,而 Signal 是关系型的语言.这些同步语言各自的特点有利于进行一些实质性的静态检查甚至于形式化分析和验证,从而有益于产生安全的代码.在基于同步语言的开发工具中,最著名的是 Scade,其代码生成器 KCG 将一种基于 Lustre 的建模语言翻译到 C 语言,KCG 应该是获得民用航空软件生产许可(DO-178B/C)的第一个商用代码生成器.目前,Scade 工具已渗透到我国航空、航天、轨道交通及核电等若干安全关键领域.

与 C 语言编译器的情形类似,人们同样不会止步于像 Scade 这样的安全级代码生成器.基于高级建模语言代码生成器的形式化验证近年来已受到极大关注.本文主要分析与比较两个关于同步数据流语言 Lustre 的可信编译器项目,二者均采用如 CompCert 那样的方法对翻译过程本身进行验证.这两个项目是目前仅有的从事形式化验证的 Lustre 可信编译器研发的长线项目,一个是法国 Pouzet 教授的项目组,另一个是作者所在的 L2C 项目组.前者较早的工作[14]将具有 Lustre 语言关键特征的 Lustre 子集翻译至一种基于对象的中间语言,最近的工作是将这一中间语言翻译至 Clight(见 CompCert 项目),并与 CompCert 衔接后称为 Vélus 编译器[15].L2C 项目致力于基于 Lustre 的语言到 C 语言的可信代码生成器(或称 L2C 编译器)的研发,从一开始便以 Clight 作为目标语言,从而与 CompCert 实现对接.有关这两个可信编译器项目的进一步发展情况会在正文中详细介绍.为方便起见,文中分别以 Vélus 和 L2C 代称这两个项目以及相应的编译器.

编译器形式化验证的另外一种重要方案是翻译确认(translation validation),由 Pnueli 等人于 1998 年首先提出[16].翻译确认不是直接验证翻译程序,而是对翻译前后代码前的语义等价关系进行确认.比起直接对翻译过程进行验证,这种方法具有较好的可重用性.然而,它必须是一个自动化的过程,因而从理论上决定了它可能出现

误报(false alarm).值得指出的是,Pnueli 等人首次提出翻译确认方法时,是以 Signal 作为示范例子,即编译器源语言是同步数据流语言 Signal,目标语言是 C.翻译确认的话题与本文主题关系不大,因此不再进一步展开讨论.

本文第 1 节首先介绍 Vélus 和 L2C 的基本情况.第 2 节~第 6 节分别就源语言特性、编译器结构、核心翻译步骤、同步数据流语义与翻译正确性验证、实现与性能等多个角度对 Vélus 和 L2C 进行较为深入的分析与比较.最后,第 7 节是全文总结.

1 基本情况

同步的概念[17]早在 20 世纪 80 年代初就开始被关注.到 20 世纪 80 年代后期出现了包括 Lustre[12]的几个最著名的同步语言,其他,如 Esterel[9]和 Signal[13]等.Lustre 语言最初是由 Caspi 和 Halbwachs 等人提出来的[18],然后在工业领域需求的基础上发展起来[19].先是在 Merlin-Gerin 公司开发了基于 Lustre 的 Saga 工具,用于核电保护系统开发.接着,Verilog 公司接手了 Saga 商品化的工作.同时期,Aerospatiale 公司(现隶属于空客)为空客 320 机载软件开发设计了 Sao 工具,其思想也类似于 Saga.后来,在 Saga 和 Sao 的基础上,Aerospatiale、Merlin-Gerin 以及 Verilog 联手启动了新工具(Scade)的开发计划.当时,Verimag 实验室建立了 Verilog 公司和 Lustre 项目组合作的一个共同实验室,专攻 Scade 的设计,并由 Lustre 项目组的 Pilaud 领导 Scade 团队.再后来,Scade 又吸收了其他公司的一些技术,并经多次转卖,目前全球闻名.

Lustre 语言被应用于商用工具后得到更多的重视,Verimag 的 Lustre 团队也一直在维护和更新版本,目前的最新版本是 Lustre V6[20].

Pouzet 教授课题组很早开始从事同步语言相关的研究工作,曾与 Caspi 等人合作在数据流语言 Lucid[21]基础上加入同步特征,设计实现了同步数据流语言 Lucid Synchrone[22],实际上也可以看作是 Lustre 的函数式语言(ML)的扩展[23].

大致在 2006 年前后,Pouzet 与 Paulin 共同启动了形式化验证 Lustre 语言编译器的一个长线项目[24].最早的参与者有 Bertails 和 Biernacki.在 TYPES 2007 会议上,Bertails[25]报告了该项目的目标以及最初始的一些工作(类型检查、时钟演算以及因果性分析等问题).Biernacki 等人在 2008 年发表的论文[14]中刻画了一个更加清晰的蓝图并给出一些实质性进展,为该项目奠定了基调.后来,Auger 完成了其博士论文工作[26,27],该项目取得进一步的成果.从这些工作可知,该课题组这一时期实现的形式化验证编译器,源语言为包含 Lustre 语言关键特征的小型同步数据流语言,目标语言是一种基于对象的中间语言.最近几年,Bourke 等人基于 Biernacki 和 Auger 等人的工作,实现了从上述中间语言到 CompCert 的前端中间语言 Clight 的翻译与验证,从而得到一个核心 Lustre 子集的可信编译器 Vélus[15].

本文作者所在课题组的 L2C 项目,源于国内核电领域的实际需求,于 2010 年 9 月正式启动.L2C 可信编译器的预研工作于 2013 年 6 月完成验收[28],源语言为一个单时钟的核心 Lustre 子集.L2C 可信编译器的一个完整企业版于 2015 年 4 月完成验收[29],源语言为一个面向领域的类 Lustre 同步数据流语言.L2C 项目开展后,合作企业的建模与代码生成工具从原 Scade 的用户走向自主研发之路,目前已被实际应用.最近几年,L2C 项目组主要在做开源 L2C 可信编译器的工作[30−32].

后续各节中,会涉及到有关 Vélus 编译器和 L2C 编译器的更多细节.

2 源语言特性

本节在讨论 Vélus 和 L2C 的源语言特性时,会涉及到原始论文中的 Lustre 语言、Lustre V6 以及 Scade 源语言的特性.

Caspi 和 Halbwachs 等人在其早期论文[11,12]中给出了 Lustre 语言最基本的特性.Lustre 程序由若干节点(node)组成,执行时会明确一个主节点.每个节点相当于普通语言中的子程序.不同于普通语言,Lustre 语言的所有数据对象(变量、常量以及表达式)的取值都是流(stream).每个流都有各自的时钟(clock);每个节点有一个基本时钟,是该节点中所有数据对象的时钟里面最快的一个.程序的执行是无限循环的过程,每次循环均在一个时钟

周期(cycle)内完成,每次执行都针对不同的输入和输出.对于每个时钟周期,一个流或者有值,或者没有值,前者是当前时钟周期包含在这个流的时钟的激活(activation)周期,而后者是当前时钟周期不属于激活周期的情形.时钟相同的流之间可以进行运算,相当于这些流在该时钟的每个激活周期里的取值都进行相应的运算.节点可以被调用,可以看作特殊的运算.节点调用相比传统语言(如 C 语言)的函数调用,允许有多个返回值.每个节点中包含一组等式,每个等式的执行效果类似于赋值.

Lustre 语言中有几个与时钟和流相关的特殊运算(或算子).when 和 current 均为一元运算,它们产生新的流,其时钟会不同于原来的流;when 会使时钟变慢,而 current 会使时钟变快,二者为互补的运算.pre 为一元运算,所产生的流会比原来的流延迟一个激活时钟周期.二元运算 arrow(→),所产生的流第 1 个激活周期的值取自第 1 个流,其余激活周期的值取自第 2 个流;特别是,通常会与 pre 运算配合使用,为新流置第 1 个激活周期的值.fby 为二元运算,所产生的流第 1 个激活周期的值取自第 1 个流,其余激活周期的值取自第 2 个流进行 pre 运算之后的流,其效果相当于前面提到的 arrow(→)与 pre 运算的配合使用.为叙述方便,下面将 pre、arrow 以及 fby 之类的算子称为时态(temporal)算子,而将 when 和 current 之类的算子称为时钟(clock)算子.

为方便起见,我们称上述原始论文中的 Lustre 语言为经典 Lustre 语言.为便于实际应用,Lustre V6 以及 Scade 在经典 Lustre 语言核心特性的基础上增加了更为丰富的语言特性,在下面讨论 Vélus 和 L2C 时会涉及到其中一些.

Vélus 编译器目前未开源,其源语言的信息主要依据其里程碑论文[14,15,27]以及该项目最新论文[15]的作者 Bourke 提供给我们的 Vélus 编译器部分源程序测例.

从实现角度看,无论是 Lustre V6 和 Scade,还是本文讨论的 Vélus 和 L2C,对于经典 Lustre 语言核心特性的实现基本上是遵循原始论文中所定义的语义.有一个值得注意的例外,即关于节点的调用.在 Caspi 和 Halbwachs 等人的论文中以及之后相当一段时间内的学术探讨中,Lustre 语言编译器在翻译时都是将节点调用进行宏展开(有必要时会进行一系列换名),形成实际上只含一个节点的中间语言.这在实际应用中显然是不合适的,因此在 Lustre V6、Scade、Vélus 以及 L2C 中,节点调用的实现均与传统语言类似.对此,在 Vélus 中,将其称为模块化(modular)实现[14,15,27].

从传统语言特性方面看,Vélus 与 Lustre V6 基本类似,而 L2C 和 Scade 除了这些特性外,还增加了一些面向实际应用的特性,主要包括 case 表达式以及更多的数组和结构体运算.

在时态算子方面,Vélus 目前仅支持经典 Lustre 语言中的 fby 算子,而 L2C、Lustre V6 以及 Scade 均支持经典 Lustre 语言中的全部时态算子(pre、arrow 和 fby).在 Bourke 提供的 Vélus 测例程序中,对原始例子中出现 pre 和 arrow 之处均用 fby 及条件(if)表达式进行了等价替换,参见 PLDI 2017 论文[15].另外,L2C 与 Scade 还增加了一个三元 fby 算子.二元 fby 算子可通俗理解为:结果流是将原来的流(第 2 个参数)延后一个激活周期,且第 1 个激活周期取第 1 个参数的流在该周期的值.三元 fby 算子是一种扩展:结果流是将原来的流(第 3 个参数)延后 n 个激活周期,且前 n 个激活周期均取第 1 个参数的流在第 1 个激活周期的值,这里,n 为第 2 个参数.

在时钟算子方面,Lustre V6 是最全面的,包含 when、current 和 merge 算子.merge 算子是作用于互补的慢速流,将它们归并后得到一个快速的流.Vélus 与 Scade 均不支持 current 算子,认为 merge 完全可以替代 current 的作用.虽然 Vélus 与 Scade 同样都是支持 when 和 merge 算子,但二者有一个本质的差异:Scade 既有布尔型的时钟,又有枚举型的时钟.布尔型时钟是通过布尔量的 true 或者 false 进行二选一取样(或确定是否激活周期),而枚举型时钟则可以通过枚举量进行多选一取样.Scade 和 Lustre V6 都同时支持布尔型时钟和枚举型时钟,Vélus 与 L2C 目前都只支持布尔型时钟.与 Vélus 不同的是,L2C 除支持 when 和 merge 算子外,也支持 current 算子.L2C 规划中拟支持枚举型时钟,但目前尚未实现.

对于一般运算,各操作数的时钟要求是相同的.但对于节点调用的运算,在一些语言中各个参数的时钟可以是不同的,L2C、Lustre V6 以及 Scade 都是这样,但 Vélus 目前要求所有参数的时钟是相同的.

在高阶迭代算子方面,L2C 的两个主要版本有不同的支持.对于面向核电领域的企业版 L2C,支持 Scade 的 9 个高阶迭代算子[29];对于开源版的 L2C,则支持 Lustre V6 的全部 5 个高阶算子[31].这二者之间,有 3 个算子的含义

是等价的.目前,Vélus 未支持高阶迭代算子.

表 1 是有关 Vélus 和 L2C 的源语言特性对照的一个简单小结.

Table 1 Source language features (L2C vs. Vélus)

表 1 源语言特性(L2C vs. Vélus)

特性	L2C	Vélus
时钟算子	when,current,布尔型 merge,枚举型 merge	when,布尔型 merge
时态算子	pre,arrow,二元 fby,三元 fby	二元 fby
高阶迭代算子	开源版:map,fill,red,fillred,boolred 开源版:共 9 个,其中有 3 个等价于 map,red,fillred	不支持
常规语言特性	比 Lustre V6 多了一些面向实际应用的特性,比如 case 表达式以及更多的数组和结构体运算	与 Lustre V6 基本类似

3 编译器结构

Vélus 编译器的架构如图 1 所示,该图与 Bourke 等人在 PLDI 2017 论文[15]中的编译架构图(该文的图 1)相对应,包括所有中间语言和翻译过程所用的名称.Vélus 编译器继承了 Pouzet 课题组的前期工作[14,27],下面的介绍并未刻意区分这些集成进来的部分,欲了解详情可参考文献[15].

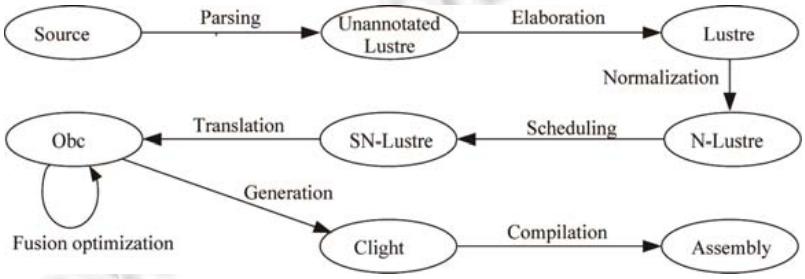

Fig.1 Compiler architecture of Vélus

图 1 Vélus 编译器架构

图 2 刻画了最新的 L2C 编译器架构,是目前的开源版 L2C[31]所采用的架构.以前的 L2C 版本(主要有原型版与企业版)的架构与此有所不同,不在本文讨论范围之内,这里不再赘述.

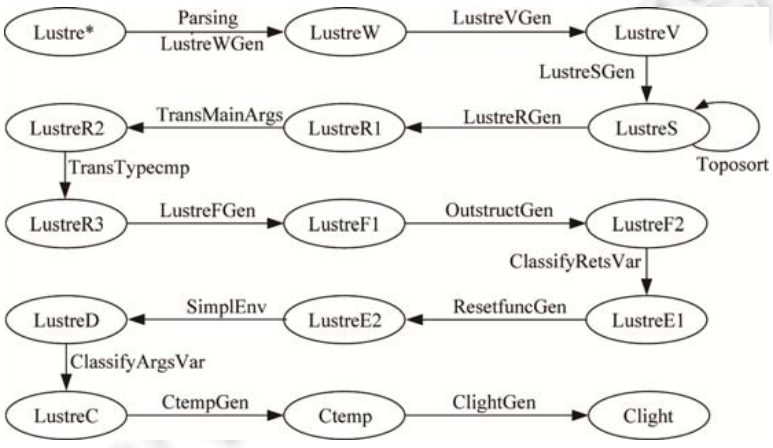

Fig.2 Compiler architecture of Open L2C

图 2 Open L2C 编译器架构

两个编译器的前期处理工作很相似.图 1 中 Vélus 将源程序(Source)翻译至中间表示 SN-Lustre 的过程,对应

于 L2C 将源程序(Lustre*)翻译至中间表示 LustrS 的过程.

二者的 Parsing 过程如同常规的编译器,实现词法和语法分析,并生成抽象语法树(AST).经过 Parsing 过程,Vélus 将源程序变换至 AST,图 1 中标为 Unannotated Lustre.同样,L2C 经过 Parsing 过程也将源程序变换至 AST,之后附加了一个 LustreWGen 过程,最后生成 LustreW 中间表示.LustreWGen 最核心的任务是完成全局和部分局部常量表达式求值(含常量定义的依赖性检查和排序),以方便后续技术环节高效地处理需要单个常量的情形,如三元 fby 运算中的延后激活周期数(正整数常量值)、高阶迭代运算的迭代次数常量、数组定义或访问区间有关的常量,等等.Vélus 目前不支持这些特性,故此类需求不足.LustreWGen 还有一些个别很细节的任务,比如根据调用关系自动设定主节点.主节点不一定非要自动设定,也可以在执行时手动设定,对于此类细节,在 Vélus 的相关论文中并未提及,因此在本文后续内容中,对于此类细节也并未作为关注重点.

随后,是类型和时钟检查及其标注过程,Vélus 和 L2C 所完成的工作是一致的:对程序(AST)进行常规类型检查以及时钟检查.若检查不通过,则报错;检查通过后,进行类型和时钟标注,类型检查与传统语言相同,时钟检查二者遵循相同的时钟演算规则.如图 1 所示,Vélus 称这一过程为 Elaboration,标注后的中间语言称为 Lustre.同样的工作在 L2C 中是由 LustreVGen 过程完成的.除了类型和时钟检查及其标注过程之外,LustreVGen 还包括一些后续阶段所需要的准备工作,参见随后的介绍.

完成类型和时钟检查及其标注之后,Vélus 进行 Normalization 变换,其最核心的工作是使每个 fby 表达式和节点调用实例从上层表达式分解出来,确保它们只出现在专用等式的右端,而不是嵌在其他表达式中.这与 Lustre 官方编译器以及 Scade 工具的做法是一致的,主要是进行一些使代码规范化以及新生成某些结构信息,为后续阶段做好铺垫:对于 fby 等时态算子,要准备好如何对 fby 等时态算子在有关历史数据的访问方面得到妥当处理;以及针对节点的每一个调用实例要如何做好输入流和输出流相关的处理.首先是不同输入流和输出流的时钟有可能不同(Vélus 目前不支持这种情形),其次是输出流可能有多个(比较,C 函数仅有一个返回参数).

在 L2C 框架中,与 Vélus 的 Normalization 相对应的工作体现在 LustreVGen 和 LustreSGen 过程中,如图 2 所示.LustreVGen 首先进行如前所述的工作(类型和时钟检查及其标注),之后是生成 fby 和 pre 等时态运算所需要的特殊标识符和类型;以及生成类型比较函数的函数名;结果得到 LustreV 中间表示.使每个 fby 表达式和节点调用实例从上层表达式分解出来的工作在 LustreSGen 过程完成.除此之外,LustreSGen 还实现以下功能:将含有表达式列表的平行等式分解为多个不含表达式列表的等式集合,将 arrow(→)和 pre 算子的组合转换成二元 fby 算子,以及将高阶迭代算子处理成类似 for 循环的结构(为后续阶段的高阶算子消去,即生成 for 循环作准备)等.

在 LustreVGen 和 LustreSGen 过程中有许多方面对于当前的 Vélus 来说是不需要的.另外,在对于包含条件的表达式处理上,Vélus 和 L2C 略有不同.经 Normalization 变换后,Vélus 将所有 if 和 merge 表达式置为顶层,而 L2C 是在排序(见后文)之后才处理 if、case 和 merge 等算子.

接下来,Vélus 和 L2C 均对上述变换之后的中间代码以等式为粒度进行排序.这些等式之间具有数据流并发关系,排序之后等式之间具有先后顺序关系,等式的语义与命令式语言的赋值语句相类似.排序的结果应不违背数据依赖关系,即满足先写(或先有值)后读的原则.Vélus 和 L2C 的排序过程分别称为 Scheduling 和 Toposort,如图 1 和图 2 所示.

后续翻译过程中,Vélus 从 SN-Lustre 到 Obc 的过程,以及 L2C 从 LustreS 到 LustreC 的过程,集中了多数核心同步数据流语言特征的翻译.同样,Vélus 从 Obc 到 Clight 以及 L2C 从 LustreC 到 Clight 的变换将同步数据流语言的可信编译与可信编译器 CompCert 衔接起来,对二者来说都是整个编译器最后的重要环节.对于这些核心翻译步骤,我们将在下一节加以专门讨论.

在当前的开源版 L2C 中,已发布了从 LustreS 到 LustreC 全部翻译过程(包含了所有的核心翻译阶段)的正确性验证代码;LustreWGen、LustreVGen 和 LustreSGen 等非核心过程的验证工作多数也已完成,将会在后续版本中陆续发布(试对比,CompCert 的前端一些非核心翻译过程如语法分析、类型检查等的验证工作也是在新近的版本中才陆续发布).因 Vélus 至今尚未开源,其核心翻译步骤的验证工作可以从相关论文中有所了解,但非核心翻译过程验证的工作在论文中很少提到具体的实现情况.有关二者核心翻译阶段的验证工作,在第 5 节有进

一步的讨论.

相比 Vélus,L2C 之所以有更多的翻译阶段,主要有如下原因:(1) L2C 立项时是面向实际的领域需求,所以支持较多和较为复杂的语言特性,而 Vélus 相关的工作主要面向学术研究,二者在源语言特性方面的对照参见表 1;(2) 实用语言的编译器开发需要不断维护,多阶段的翻译有利于可重用性及可扩展性,如 CompCert 面向可实用的 C 语言,其编译器结构中包含了许多翻译阶段;(3) 安全攸关领域一般都有可追溯的要求,多阶段的翻译有利于提高可追溯性.

4 核心翻译步骤

从 Lustre 语言翻译到 C,由同步数据流范型变换为串行命令式范型,必须实现几个核心同步数据流语言特征的翻译,Vélus 和 L2C 的翻译阶段均包含了对这些核心特征的处理,主要包括:(1) 通过以等式为粒度的排序或调度,完成数据流并发特征的消去,实现等式集合的串行化;(2) 实现同步数据流语言时钟算子特征的消去;(3) 实现同步数据流语言时态算子特征的消去;(4) 结合前述这些特征的处理,实现同步数据流范型向串行命令式范型的转换.

在上述几个核心同步数据流语言特征的翻译步骤中,以等式为粒度的排序或调度容易解释,在前面一节已经介绍过(Vélus 中对应 Scheduling,L2C 中对应 Toposort),剩余的 3 个方面在本节进行讨论.对于经历这几个消去同步数据流特征的翻译步骤之后所得到的中间代码,Vélus 和 L2C 最终将其翻译至 Clight 代码,从而实现与 CompCert 的对接.翻译至 Clight 代码的步骤也是本节要讨论的重点内容之一.

L2C 支持高阶迭代算子,高阶算子的消去也是 L2C 的核心翻译步骤之一.这项工作主要是在 LustreRGen 过程(如图 2 所示)中实现的,它将高阶运算翻译为等价的 for 循环语句.由于 Vélus 未支持高阶迭代算子,所以我们不对这一核心翻译步骤作进一步讨论.顺便,对于其他 L2C 支持而 Vélus 不支持的非核心语言特征,其处理过程也同样不作讨论,比如,将复杂类型比较运算翻译为比较函数的过程(图 2 中的 TransTypecmp).

值得注意的是,在上述核心翻译步骤中,除了排序(调度)和生成到 Clight,对于其余几个方面,Vélus 均集中在从 SN-Lustre 到 Obc 两个层次的变换,而 L2C 则是分散在 LustreS 到 LustreC 的多个层次的变换中.

```
node counter(ini, inc: int; restart: bool)
returns (n: int)
   var c: int; f: bool;
let
   n=if f or restart then ini else c+inc;
   f=true fby false;
   c=1 fby n;
tel

node d_integrator(gamma: int)
returns (speed, position: int)
let
   speed=counter(0, gamma, false);
   position=counter(0, speed, false);
tel
```

```
node rising(s: bool) returns (edge: bool)
   var ps : bool;
let
    edge=not ps and s;
       ps=true fby s;
tel

node tracker(acc, limit: int) returns (p, t: int)
var s, pt : int; x : bool; c : int when x;
let
    c=counter(1 when x, 1 when x, false when x);
    x=rising(s>limit);
      (s, p)=d_integrator(acc);
       pt=0 fby t;
    t=merge x c (pt when not x);
tel
```

Fig.3 An example program for both Vélus and L2C

图 3 Vélus 和 L2C 共享的一个例子程序

(a) 时钟算子的消去

我们仅讨论 Vélus 和 L2C 均支持的时钟算子 when 和 merge(此外,L2C 还支持 current 算子).

Vélus 和 L2C 遵循相同的时钟演算规则,在仅限于布尔型时钟的情况下,嵌套时钟可由时钟变量(或其否定)的序列来表示(可参见 Biernacki 等人在 2008 年发表的论文[14]):*base* on b_1 on b_2 … on b_n.其中,*base* 为当前节点的基准时钟,每个 b_i 或者是某个时钟变量(布尔型)*x* 或者是其否定 not *x*.

图 3 给出 Vélus 和 L2C 均可以接受的一个源程序例子.我们来观察节点 tracker 的代码.假设该节点的基本

时钟为 *base*.根据时钟演算规则,变量 *c* 的时钟为 *base* on *x*(与 *counter* 调用中的两个参数的时钟相同),而表达式(*pt* when not *x*)的时钟为 *base* on not *x*;*x*,*t*,*pt* 等变量的时钟均为 *base*.表达式 merge *x c* (*pt* when not *x*)的作用是将时钟分别为 *base* on *x* 和 *base* on not *x* 的两个表达式 *c* 和(*pt* when not *x*)归并为一个时钟为 *base* 的表达式.这里,需要注意的是,when 算子会使时钟变慢,而 merge 算子会使时钟变快.

在得到各变量以及表达式的时钟后,when 算子实际上就可以消去了.翻译的结果就是将等式(左边的变量和右边的表达式具有相同的时钟)的执行加上时钟为真的条件,生成 if 条件语句.这一条件的构成是将时钟表达式中所有的时钟变量(或者其否定)进行合取,而基本时钟 *base* 可认为是 true.例如,若时钟为 *base* on not *x*,则条件即为 not *x*.

对于 merge 算子,则会翻译为 if-else 条件语句或 case 语句.例如,图 3 中的 merge *x c* (*pt* when not *x*),Vélus 会翻译为类似于 if (*x*) {*c*=…} else {*pt*=…}之类的语句.在 L2C 中也类似,不同的是,L2C 的设计要支持枚举型时钟,所以目前选择了一种更为一般的情况,会将 merge 表达式翻译至 case 语句.具体细节可分别对应于附录中图 4 所示右半边的 if 语句和图 5 所示右半边的 case 语句.

顺便,Vélus 在生成 Obc 表示后会针对这些 if 语句进行 Fusion Optimization 的优化处理(如图 1 所示),使条件相容(多为时钟相同)的分支语句序列进行合并,这样仅需一次条件判断即可.目前,L2C 尚未开展此类优化工作.这个问题,后面还会提到.

(b) 时态算子的消去

Vélus 经过 Normalization 变换,或者 L2C 经过 LustreVGen 和 LustreSGen 过程,已确保了每个 fby 表达式只出现在专用等式的右端,而没有嵌在其他表达式中.图 3 中的所有 fby 表达式均已满足这一要求,对应的等式为:节点 *counter* 中的 *f*=true fby false 和 *c*=1 fby *n*,节点 *rising* 中的 *ps*=true fby *s*,以及节点 *tracker* 中的 *pt*=0 fby *t*.我们以 *tracker* 中的 *pt* = 0 fby *t* 为例解释 fby 算子的消去.

每个 fby 对应一个时态变量,用于记录历史状态.对于图 3 中节点 *tracker* 的含 fby 等式 *pt*=0 fby *t*,这一变量为 *pt*.无论是 Vélus 还是 L2C,都会将 *pt* 包装为节点 *tracker* 内可访问的特殊变量.下一小节会看到,Vélus 会将 *pt* 看作节点 *tracker* 所对应的类的属性变量,每个 *tracker* 实例都会维护一份 *pt* 的存储,可以保存以前周期的信息,而其他普通变量(比如 *t*)只能看作节点 *tracker* 所对应的类的方法的内部变量.类似地,L2C 会将 *pt* 看作与每个 *tracker* 的实例相关联的某个全局结构体的域变量,而其他普通变量(比如 *t*)不在这个全局结构体的域中.

若从对应到 Vélus 翻译后的 Clight 代码角度来看,图 4 所示的结构体类型 *tracker* 包含了域变量 *pt*,对应于上述节点 *tracker* 所对应的类的属性变量 *pt*.若从对应到 L2C 翻译后的 Clight 代码角度来看,图 5 所示的结构体类型 *outC_tracker* 包含的域变量 *acg_fby*4,对应于上述节点 *tracker* 所对应的类的属性变量 *pt*.这种时态变量在 L2C 中是自动产生的新变量,比如对于图 3 所示的程序,将会产生 *acg_fby*1、*acg_fby*2、*acg_fby*3 和 *acg_fby*4,分别对应于该程序中的 4 个 fby 的时态变量 *f*、*c*、*ps* 和 *pt*.

对于每个 *tracker* 实例中 *pt*=0 fby *t* 的执行过程,Vélus 和 L2C 的处理是类似的.在该实例执行的第 1 个激活周期开始时,会将相应于这一实例的时态变量 *pt* 的值置为 0,而在之后的每个激活周期开始时会将 *pt* 的值置为前一个激活周期结束时 *pt* 的值.在每个激活周期的结束阶段,会将在该周期中已经计算好的 *t* 的值存入 *pt* 的存储空间,存入之后在该周期不会再有对 *pt* 的读取.

若从对应到翻译后的 Clight 代码角度来看,图 4 中省略了 reset 相关的部分,而从图 5 右半边上部的 if 语句来看,我们会发现:第 1 个激活周期(*acg_init* 为 1 时)开始时,*pt* 会被置为 0,而在之后的每个激活周期开始时会将 *pt* 的值置为相应时态变量 *acg_fby*4 在前一个激活周期结束时的值.关于每个激活周期结束时对时态变量的修改,图 4 中是由最后一句(* self).*pt*=(* out).*t* 来完成的,而在图 5 中是由倒数第 2 句(*outC).*acg_fby*4=(*outC).*t* 来完成的.

另外,L2C 还支持三元的 fby 算子,其实现更加复杂一些.L2C 的实现方案是将自动产生的新增时态变量(类似上述的 *acg_fby*4)扩展为数组,其大小即为三元 fby 比起前面讨论的二元 fby 多出来的常量参数值.因 Vélus 目前不支持三元 fby 算子,所以我们不在这里讨论相应的翻译细节.

除了二元和三元 fby 算子,L2C 还支持传统 Lustre 的时态算子 pre.虽然 Vélus 目前未涵盖 pre 算子,但其实现与 fby 雷同.

(c) 同步数据流范型转换为串行命令式范型

在处理好前述几个核心同步数据流语言特征翻译的基础上,再解决好流数据对象的表示及其周期性处理的环节,基本上就实现了同步数据流范型向串行命令式范型的转换.在 Vélus 中,这种范型转换体现在生成 Obc 中间表示.相应地,在 L2C 中,这种转换导致了 LustreC 中间表示的产生.

Vélus 的 Obc 是一种基于对象的中间表示,用于封装从同步数据流代码到命令式代码的翻译结果.在翻译到 Obc 时,每一个节点会对应到一个类.这个类中包含的属性有两类:一类是该节点中的所有时态变量(每个 fby 对应一个这样的变量);另一类是该节点中调用其他节点的实例(注:不允许调用到本节点的实例).这两类属性都是有状态的,即要维护其以前周期的历史信息(注:节点实例内部也可能有自己的时态变量).该类中包含的方法有两个:一个是 reset,用于初始化时态变量属性以及调用实例,仅在第 1 个激活周期执行;另一个是 step,用来执行该节点内部的所有等式,每个激活周期都会被重复调用.每个节点的输入输出参数以及局部变量均在 step 方法中得以体现.

与 Vélus 不同,L2C 从同步数据流代码到命令式代码的翻译结果(LustreC 中间表示),并未将每个节点对应的方法及 reset 方法与它们要处理的状态数据(时态变量和调用实例)封装在一起,而是采用了类似于 C 程序的组织方式.每个节点对应 C 的一个函数,它的每个实例(对应于程序中实际调用该节点的位置)对应于一个结构体,其中封装了对应的时态变量和要调用的其他节点实例.为方便每个周期的循环调用,每个节点的输入输出参数变量也会被封装在一个该节点专用的结构体中,对于主节点要封装所有输入和输出变量,而对于其他节点仅封装输出.因为有可能有多个输出,所以对应到 C 时用一个结构体类型的变量存放多个返回值,所以,所有节点的输出变量均被分装在结构体中.主节点的输入变量要作为外部输入,为方便起见,也被封装在结构体中.同时,每个节点均有一个对应的 reset 函数.由此可见,虽然没有对每个节点对应的函数和 reset 函数以及它们要处理的数据对象进行封装,但这些内容均包含在生成的 LustreC 代码中,如同 C 语言的代码结构.

从翻译后的 Clight 代码可以看到一些具体的细节,如图 4 和图 5 所示.

(d) 翻译至 Clight

综合前面几个核心同步数据流语言特征的翻译步骤,应该能够看明白从图 3 所示的 Lustre 源程序分别经 Vélus 和 L2C 翻译至图 4 和图 5 所示的 Clight 代码(二者虽然形式上不同,但体现的行为是相通的),从而也可以了解最后阶段的这一核心翻译步骤所起到的作用.

在这个阶段,Vélus 就是将 Obc 中间代码的语法设法与翻译后的 Clight 代码的语法对应起来.L2C 也类似,通过 CtempGen 完成从 LustreC 到 Ctemp 的翻译,又通过 ClightGen 将 Ctemp 中的 memcpy 语义翻译为内存拷贝函数,最终翻译至 Clight 代码.L2C 后面这一步骤(从 Ctemp 到 Clight)在 Vélus 目前的版本中没有对应的需求.这些翻译步骤的图示可分别如图 1 和图 2 所示.

5 同步数据流语义与翻译正确性验证

这部分内容的准确论述需采用形式化或半形式化方式,但受限于篇幅,本文仅就其要点进行非形式描述.

(a) 同步数据流语义

在从 Lustre 到 C 的转换过程中,经历了同步数据流语义逐步向串行命令式语义的过渡.在现阶段,Vélus 的动态语义的形式化定义是从 SN-Lustre 中间表示开始的,在 Obc,中间表示被变换至一种基于对象的串行命令式语义;相应地,L2C 的动态语义形式化定义是从 LustreS 中间表示开始的,在 LustreC,中间表示彻底变换为一种类 C 的串行命令式语义.这些中间层语言所处位置,如图 1 和图 2 所示.

可以看出,Vélus 的同步数据流语义消去仅在一个阶段完成,而 L2C 则经历了多个阶段逐步消去.由于 L2C 的源语言特征相比 Vélus 要复杂很多,实践过程决定了跨多层的语义过渡是必然的选择.

Vélus 在定义 SN-Lustre 语义时已完成排序(scheduling),因此无需考虑同步数据流语言的数据流并发特

性.L2C 是在 LustreS 层进行排序(toposort),在 LustreS 层有串行语义定义(节点内部的所有等式相互之间存在全序关系),同时也定义有超粗粒度的并发语义(以等式为粒度,等式操作是不可分割的原子操作,等式之间存在由数据依赖所确定的偏序关系).

Vélus 和 L2C 的同步数据流语义都选择了基于操作语义来定义(虽然也存在看似很不错的基于指称语义的基础工作[33]).在 Lustre 语言的原始论文[11]中,作者给出了 Lustre 语言的基本操作语义规则.Vélus 和 L2C 的操作语义定义首先要与学界公认的这些工作相符,同时也应考虑实现的因素(交互式定理证明的方便性).比如,采用 Coinductive 来定义流看似是非常自然的(因为要处理的是 infinite 对象),然而无论是 Vélus 还是 L2C 均放弃选择这种方法,L2C 团队当初遇到的问题主要是因为采用 Coq 实现整体任务需求的技术难度无法预测.

在 Vélus 和 L2C 的同步数据流语义中,对于流数据对象的基本理解是一致的,均将其看作是从自然数到取值论域的函数.比如,可以将一个流 s 在第 n 个周期的值记为 $s(n)$.然而,在具体实现上,L2C 采取了一些不同的措施.特别是针对同步数据流语义中最具代表性的 fby 算子.

在定义 fby 算子的语义时,Vélus 使用了一个辅助算子 hold,会将所处理的流数据对象在上一个激活周期的取值保持到其后续的无效周期,这样,若当前周期是激活周期,则 fby 运算的结果就相当于当前周期 hold 运算的结果.这一思想在 L2C 项目开展的初期用来实现一种时钟归一化(clock unifying)算法[34],其中定义了类似于 hold 的算子,并用来刻画 fby 算子的语义.然而,在后来的工作中,L2C 摈弃了这种做法.主要原因是,这种方法仅适合于定义二元 fby 算子,而难以推广到实际应用中很有用的三元 fby 算子.L2C 对二元和三元 fby 算子的语义定义采用了一致的框架,定义一个大小为 n 的循环缓冲区,并设法限定仅在激活周期可访问缓冲区,这里,n 即为三元 fby 算子的第 3 个参数(对于二元 fby 算子,n=1).

定义同步数据流语义的另一个重要方面,就是语义环境中要保留子环境的状态,这些子环境对应于当前环境下调用其他节点而形成的实例环境.保存这些子环境的原因是因为存在时态变量(见第 4.2 节),需要保存历史状态,这就导致了在调用的节点实例返回时不能像传统的函数调用那样释放掉变量的存储空间.因而,我们必须要定义并维护一个树状的语义环境,这对语义定义以及相应的证明来说,难度增加了许多,但这是我们必须要应对的.L2C 在定义语义时对是否是时态变量进行了区分,以方便对应到 Clight 的语义环境中.Vélus 在文献[15]中对这种树状语义环境也有许多描述,但仅从论文无法了解其进一步的实现详情.

另外,L2C 因支持高阶迭代算子以及更多的数组和结构体算子,使语义定义的难度增大许多,但相关细节不属本文所要讨论的范围.L2C 的多阶段翻译架构(如图 2 所示)正是为了适应各种各样的语义复杂度以及与 Clight 语义之间的巨大差异,同样,图 2 所示的许多翻译步骤也不在本文所讨论的议题之内.

(b) 翻译正确性验证

在形式化定义动态语义阶段,Vélus 和 L2C 的各阶段翻译正确性的验证目标与 CompCert 所建议的阶段翻译正确性目标从形式上是一致的,都是要证明一种从高层到低层单向的语义模拟等价关系.这种模拟等价关系之所以可以是单向的,是由于每一层语言均具有确定的语义(deterministic semantics).

对于同步数据流语义来说,这种单向的语义模拟等价关系可大致描述为:任意数据流程序 G 的每个节点 f,若可以将输入流 xs 映射到输出流 ys,则对于翻译后得到的程序 G'中与 f 对应的节点或函数 f'以及这一层次中相应于 G'的初始化函数 reset,如果 G'在其第 1 个激活周期先执行 reset 后执行 f',而在其他后续激活周期中都重复执行 f',那么同样可以将 xs 映射到输出流 ys.注意:这一命题中的 G、f、xs 和 ys 都是任意的.

虽然各阶段要证明的单向语义模拟等价关系形式上看都是相似的,但具体到不同阶段,其含义却都有所不同.首先,各个阶段的语法和语义互不相同;其次,每个阶段可能有或没有语法层面的 reset 函数,如果有的话,不同阶段的 reset 函数也可能有所不同;以及还有其他各种不同,这里不再赘述.

一般来说,从 Lustre 源程序到 Clight 程序的各阶段翻译过程中,同步数据流语言的语法及语义特征逐渐被消去,C 语言语法及语义特征逐渐增加.在定义上述单向语义模拟等价关系时,所消失的语义特征描述通常会由新增加的语法特征来弥补.比如前面提到的 reset 特征,在较高的语言层次,语法层面并没有对应的函数,所以是在语义定义中体现其行为,而在靠后的语言层次(比如图 2 所示的 LustreE2 层)已有语法层面的 reset 函数,就不

会也不可能通过语义定义来体现 reset 行为.

因 Vélus 和 L2C 的编译框架之间差异较大(如图 1 和图 2 所示),可以想象到二者在翻译正确性验证方面存在较大的差异.这种差异体现在各个方面.这里仅列举一例.如果考虑第 4.1 节~第 4.3 节提到的核心翻译步骤,在 Vélus 中仅由 Translation 一个阶段完成(如图 1 所示),而在 L2C 中则是分散到从 LustreR1 到 LustreD 的许多阶段实现的(如图 2 所示).一般来说,一个翻译阶段所负责的工作越多,其正确性证明的难度也越大,重用性也越差.然而,另一方面,翻译阶段也不能太多,否则会导致语言层次增加太多,反而加重维护的负担.因此,应根据源语言的规模合理设计翻译阶段的数量.

Vélus 中 SN-Lustre 之前的阶段未定义动态语义,这些阶段所保持的特性主要是静态方面的.多数这些阶段,甚至从 SN-Lustre 到 Obc 的过程(Translation),基本上是继承前人(Bertails、Biernacki 和 Auger 等人)的工作.Bourke 等人的论文[15]对这些静态语义相关阶段的介绍很少,同时,Vélus 未开源,所以具体继承情况的详情未能很好地得到了解,从 Biernacki 和 Auger 等人的论文[14,27]仅可了解到其中的一些.

Vélus 的语法分析器是 Bourke 等人重新写的,它基于 Jourdan 等人提出的对 LR(1)自动机进行确认并对确认程序进行验证的方法[35],因此可以认为是一个经过形式化验证的语法分析器.L2C 中,最近刚完成与此平行的工作[32],并以编译选项的方式集成到了开源 L2C 编译器[31]中.

对于 Vélus 中的类型、时钟检查(elaboration)以及接下来的规范化过程(normalization),仅了解 Bertails、Biernacki 和 Auger 等人在 Coq 中实现了这些过程以及进行了验证,但他们的论文中未提到是如何进行验证的.L2C 中与此对应的过程(LustreWGen、LustreVGen 和 LustreSGen)的验证在目前的开源版本中尚未发布,相关工作仍在完善和整理之中.这些翻译过程在 Vélus 和 L2C 编译框架中所处位置,如图 1 和图 2 所示,下同.

Vélus 对于排序过程(Scheduling)的验证是基于翻译确认的方法[16],具体实现在相关论文[14,15,27]中未加介绍.L2C 中的排序过程(Toposort)是经过严格形式化验证的,证明了经过排序后的 LustreS 程序的串行语义行为,与排序前 LustreS 程序的等式粒度并发语义所刻画的行为是兼容的.从实现角度来看,对排序过程的形式化验证要比翻译确认的工作量大很多,所以从通常的观点来看,是否有必要这样做是值得商榷的.但 L2C 的这项工作,对于在项目开展初期积累团队成员在使用 Coq 进行编程和验证的经验以及检查 LustreS 操作语义定义的合理性等方面起着重要的作用.

6 实现与性能

Vélus 和 L2C 的开发环境与 CompCert[2,36]一致,有验证需求的部分均在 Coq[3,37]工具之中实现,并自动抽取得到相关的 Ocaml 代码.词法分析器的 Ocaml 代码由 Ocamllex 自动产生.语法分析器使用了 Ocamlyacc,或者 Menhir[38],前者针对未经验证的语法分析器选项,后者针对经过形式化验证的语法分析器选项.驱动程序以及其他辅助工具(如各层的语法树或代码的格式输出程序)通过手工编写 Ocaml 代码实现.

由于 Lustre 没有标准测试集,在 Bourke 等人的论文[15]中从前人相关工作中选取了 14 个有代表性的程序进行性能测试,在开源 OTAWA 框架[39]下,对 Vélus 以及其他一些 Lustre 编译器(Lustre V6 和 Heptagon 与 CompCert 或者 GCC 的组合)的最坏执行时间(worst-case execution time,简称 WCET)指标进行了测试评估.CompCert 的版本是 2.6,GCC 的版本是 4.8.4,目标体系结构是 armv7-a(含硬件浮点单元 vfpv3-d16).

我们从 Bourke 那里获得了文献[15]中使用的全部 14 个测例程序源码,并在与上述测试环境基本相同的配置下(配置选项为-march=armv7-a-mfloat-abi=softfp-mfpu=vfpv3-d16),针对 L2C 完成了测例程序的评估测试.我们将得到的测试结果汇总于表 2.

表 2 中的第 1 列是 14 个测例程序的名称,第 2 列是 L2C 和 CompCert 2.6 组合编译器的测试结果(WCET 值),第 3 列是从文献[15]中摘录的 Vélus 测试结果,第 4 列是 L2C 和 GCC4.8.4(加-O1 优化选项)组合编译器的测试结果,最后一列是 Lustre V6 和 GCC4.8.4(加-O1 优化选项)组合编译器的测试结果.

Table 2 WCET estimates in cycles for an armv7-a/vfpv3-d16 target

表 2 周期为单位的 WCET 评估值(目标处理器:armv7-a/vfpv3-d16)

测例程序名	L2C+CompCert	Vélus	L2C+gcc-O1
avgvelocity	515	315	275
count	90	55	55
tracker	1 035	680	580
pip_ex	6 620	4 415	2 745
mp_longitudinal	7 415	5 525	3 140
cruise	3 200	1 760	1 645
risingedgeretrigger	640	285	265
chrono	250	410	80
watchdog3	1 080	610	435
functionalchain	17 565	11 550	7 795
landing_gear	14 325	9 660	5 955
minus	1 555	890	560
prodcell	3 315	1 020	1 190
ums_verif	4 210	2 590	855

从表 2 的结果粗略来看,L2C+CompCert 的性能比 Vélus 低 50%左右.L2C 性能不如 Vélus 的原因,目前能想到的主要是因为 Vélus 在 Obc 中间表示进行了 if 语句的 Fusion Optimization 优化(如图 1 所示),而目前 L2C 未作任何优化,CompCert 目前也没有支持有关的优化.

附件 A.1 和 A.2 分别给出了测例程序 tracker(源码如图 3 所示)经 Vélus 和 L2C 编译生成的 Clight 代码,如图 4 和图 5 所示.从图 5 可以看出,L2C 生成的代码中包含了大量可融合的 if 语句.从图 4 可以看出,Vélus 生成的代码中时钟相同的操作已被合并到相同的条件分支下.

表 2 的第 4 列给出了 L2C 和 GCC4.8.4(加-O1 优化选项)组合编译器的测试结果.从中可以看出,测例程序 tracker 的 WCET 值比起第 2 列有显著降低(由 1 035 降至 580),其他测例程序的情况也是如此,性能也优于 Vélus 的测试结果(见第 3 列).在 GCC 的“-O1”优化选项中,包含了针对 if 语句的“-fif-conversion”优化,可以缩短 if-then 语句所花费的时间.

Lustre V6 编译器设计时也未过多考虑相关优化,主要是寄望于 C 编译器会负责这些优化工作(L2C 也曾有类似的考虑,但更主要的是在开发 L2C 企业版时,用户方出于可追溯性的硬性要求,原则上不允许对代码进行编译优化,这种理念一直延续至今,但不一定适合各种场合).从文献[15]的测试结果(如该文中图 12 第 6 列所示)看,Lustre V6 和 CompCert 组合的编译器对于各测例程序的 WCET 性能指标均不如 L2C 和 CompCert 组合的编译器.目前,我们对 Lustre V6 和 CompCert 组合编译器的性能测试工作也在进行之中.

7 总 结

Vélus 和 L2C 都是基于辅助定理证明器(Coq)构造 Lustre 可信编译器的长线项目,它们都是先将源语言翻译至 Clight,并与 CompCert 编译器实现衔接.Vélus 和 L2C 课题组在立项目标、发展过程和方向、源语言需求以及工作基础等方面都不同,因而导致 Vélus 和 L2C 编译器在许多方面有明显的差异,形成了各自的特色.

本文从基本情况、源语言特性、编译器结构、核心翻译步骤、同步数据流语义与翻译正确性验证以及实现与性能等多个角度对 Vélus 和 L2C 进行了较为深入的分析与比较.当然,也不排除漏掉一些比较重要的方面,今后会通过其他机会加以补充.

据 Bourke 介绍,Vélus 项目组近期在开展将有限自动机融入 Vélus 的研究工作.Scade[8]工具中包含对有限自动机的支持,其目的是增加控制流特性.L2C 项目组目前尚无计划开展类似的工作.

References:

[1] Knight, JC. Safety critical systems: Challenges and directions. In: Proc. of the 24th Int'l Conf. on Software Engineering, ICSE 2002. 2002. 547–550.

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

[3] Bertot Y, Castéran P. Interactive theorem proving and program development—Coq'Art: The calculus of inductive constructions. In:

Proc. of the EATCS Series on Theoretical Computer Science. Springer-Verlag, 2004.

[4] Morrisett G. Technical perspective: A compiler's story. Communications of the ACM, 2009,52(7):106.

[5] 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). 2011. 283–294.

[6] Leroy X. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In: Proc. of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2006,41(1):42–54.

[7] Simulink home. http://www.mathworks.com/products/simulink/

[8] Scade home. http://www.ansys.com/products/embedded-software/ansys-scade-suite

[9] Berry G, Gonthier G. The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 1992,19(2):87–152.

[10] Maraninchi F. The ARGOS language: Graphical representation of automata and description of reactive systems. In: Proc. of the IEEE Workshop on Visual Languages. Kobe, 1991.

[11] 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. Munchen, 1987.

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

[13] 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.

[14] Biernacki D, Colaco J, 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). Tucson, 2008.

[15] Bourke T, Brun L, Dagand PÉ, Leroy X, Pouzet M, Rieg L. A formally verified compiler for Lustre. In: Proc. of the Programming Language Design and Implementation. 2017. 586–601.

[16] Pnueli A, Siegel M, Singerman E. Translation validation. In: Proc. of the TACAS'98. Lecture Notes in Computer Science 1384, 1998. 151–166.

[17] Milner R. Calculi for synchrony and asynchrony. TCS, 1983,25(3).

[18] Caspi P, Halbwachs N. A functional model for describing and reasoning about time behaviour of computing systems. Acta Informatica, 1986,22:595–697.

[19] 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.

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

[21] Wadge WW, Ashcroft EA. Lucid, the Dataflow Programming Language. San Diego: Academic Press Professional Inc., 1985.

[22] Lucid Synchrone home. http://www.di.ens.fr/~pouzet/lucid-synchrone/

[23] Caspi P, Pouzet M. A functional extension to LUSTRE. In: Proc. of the 8th Int'l Symp. on Languages for Intensional Programming, ISLIP'95. Sydney, 1995.

[24] Paulin C, Pouzet M. Certified compilation of scade/Lustre. 2006. http://www.lri.fr/paulin/lustreincoq.pdf

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

[26] Auger C. Compilation Certifiée de SCADE/LUSTRE [Ph.D. Thesis]. Université Paris Sud, 2013.

[27] Auger C, Colaçob J-L, Hamon G, Pouzet M. A formalization and proof of a modular Lustre compiler. 2012. http://www.di.ens.fr/~pouzet/cours/mpri/cours4/scp12.pdf

[28] 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 with English abstract). http://www.jos.org.cn/1000-9825/4542.htm [doi: 10.13328/j.cnki.jos.004542]

[29] Liu Y, Gan YK, Wang SY, Dong Y, Yang F, Shi G, Yan X. Trustworthy translation for eliminating high-order operation of a synchronous dataflow language. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):332–347 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4785.htm [doi: 10.13328/j.cnki.jos.004785]

[30] 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]

[31] Open L2C home. http://soft.cs.tsinghua.edu.cn:8000/

[32] Yang P, Wang SY. Survey on trustworthy compilers for synchronous data-flow languages. Computer Science, 2019,46(5):21–28 (in Chinese with English abstract).

[33] Paulin-Mohring C. A constructive denotational semantics for Kahn networks in Coq. In: From Semantics to Computer Science. Essays in Honour of Gilles Kahn, 2009. 383–414.

[34] Shi G, Zhang YC, Shang S, *et al*. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. Science China-Information Sciences, 2019,62(1). http://scis.scichina.com/en/2019/012801.pdf

[35] Jourdan JH, Pottier F, Leroy X. Validating LR(1) parsers. In: Proc. of the Programming Languages and Systems—the 21st European Symp. on Programming, ESOP 2012. Lecture Notes in Computer Science 7211, Springer-Verlag, 2012. 397–416.

[36] CompCert home. http://compcert.inria.fr/

[37] Coq home. https://coq.inria.fr/

[38] Pottier F, Régis-Gianas Y. Menhir Reference Manual. 2018. http://gallium.inria.fr/~fpottier/menhir/manual.pdf

[39] Ballabriga C, Cassé H, Rochange C, Sainrat P. OTAWA: An open toolbox for adaptive WCET analysis. In: Proc. of the 8th IFIP WG 10.2 Int'l Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010). Lecture Notes in Computer Science 6399, Springer-Verlag, 2010. 35–46.

附中文参考文献:

[28] 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 [doi: 10.13328/j.cnki.jos.004542]

[29] 刘洋,甘元科,王生原,董渊,杨斐,石刚,闫鑫.同步数据流语言高阶运算消去的可信翻译.软件学报,2015,26(2):332–347. http://www.jos.org.cn/1000-9825/4785.htm [doi: 10.13328/j.cnki.jos.004785]

[30] 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. http://www.jos.org.cn/1000-9825/5213.htm [doi: 10.13328/j.cnki.jos.005213]

[32] 杨萍,王生原.同步数据流语言可信编译器的研究进展.计算机科学,2019,46(5):21–28.

附录

A.1

例:Vélus 生成的 Clight 代码.如图 4 所示.

```
struct tracker {
  int pt;
  struct d_integrator s;
  struct rising x;
  struct counter c;
  f=true fby false;
  c=1 fby n;
};

struct tracker$step {
  int p;
  int t;
};
```

```
void tracker$step(struct tracker *self,
    struct tracker$step *out, int acc, int limit)
{
    struct d_integrator$step out$s$step;
    register int step$n, s, c;
    register bool step$edge, x;
    d_integrator$step(&(*self).s, &out$s$step, acc);
    s=out$s$step.speed;
    (*out).p=out$s$step.position;
    step$edge = rising$step(&(*self).x, s>limit);
    x=step$edge;
    if (x) { step$n=counter$step(&(*self).c, 1, 1, 0);
             c=step$n; (*out).t=c;}
    else { (*out).t=(*self).pt;}
    (*self).pt=(*out).t;
}
```

Fig.4 Clight code procuded from Fig.3's *tracker* node by Vélus (the reset function being left out)

图 4 用 Vélus 编译图 3 中 *tracker* 节点生成的 Clight 代码(略去了 reset 函数)

A.2

例:L2C 生成的 Clight 代码.如图 5 所示.

```
typedef struct {
   int acc;
   int limit;
}inC_tracker;

typedef struct {
   int p;
   int t;
   bool acg_init;
   int acg_fby4;
   outC_counter acg_context1;
   outC_d_integrator acg_context3;
   outC_rising acg_context2;
}outC_tracker;

void tracker_reset(outC_tracker *outC)
{
   (*outC).acg_init=1;
   rising_reset(&(*outC).acg_context2);
   d_integrator_reset(&(*outC).acg_context3);
   counter_reset(&(*outC).acg_context1);

}

void tracker(inC_tracker *inC, outC_tracker *outC)
{
   int s;
   int pt;
   bool x;
   int c;
   int acg_L1;
   int acg_L2;
   bool acg_L3;
   int acg_L4;
   d_integrator((*inC).acc, &(*outC).acg_context3);
   s=(*outC).acg_context3.speed;
   (*outC).p=(*outC).acg_context3.position;
   if ((*outC).acg_init) {
      pt=0;
    } else {
      pt=(*outC).acg_fby4;
    }
   rising(s>(*inC).limit, &(*outC).acg_context2);
    x=(*outC).acg_context2.edge;
    if (x) {
       acg_L1=1;
    }
    if (x) {
      acg_L2=1;
     }
     if (x) {
       acg_L3=0;
     }
     if (!x) {
       acg_L4=pt;
   }
   if (x) {
      counter(acg_L1, acg_L2, acg_L3,
         &(*outC).acg_context1);
      c=(*outC).acg_context1.n;
    }
              switch (x) {
                 case 1:
        (*outC).t=c;
        break;
      case 0:
        (*outC).t=acg_L4;
        break;
     default:
         /*skip*/;
    }
    (*outC).acg_fby4=(*outC).t;
    (*outC).acg_init=0;
}
```

Fig.5 Clight code procuded from Fig.3's *tracker* node by L2C (with the reset function)

图 5 用 L2C 编译图 3 中 *tracker* 节点生成的 Clight 代码(含 reset 函数)

康跃馨(1993－),男,河南安阳人,硕士,主要研究领域为形式化验证.

甘元科(1983－),男,硕士,主要研究领域为形式化验证.

王生原(1964－),男,博士,副教授,CCF 高级会员,主要研究领域为程序语言与系统,程序验证,Petri 网应用.