软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(5):1233−1246 [doi: 10.13328/j.cnki.jos.005213]

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

可信编译器 L2C 的核心翻译步骤及其设计与实现*

尚 书, 甘元科, 石 刚, 王生原, 董 渊

(清华大学 计算机科学与技术系,北京 100084)

通讯作者: 尚书, E-mail: shangs14@mails.tsinghua.edu.cn

摘 要: 同步数据流语言(如 Lustre)近年来在航空、高铁、核电等安全攸关领域得到广泛应用.这些领域对相关开发工具本身的安全性有着相当高的要求.为尽力解决好“误编译”问题,近期人们借助 reliable-by-construction 辅助定理证明器实现常规命令式语言编译器的构造和验证,取得了很大的成功,如 CompCert C 编译器.L2C 是基于这种方法开发的可信编译器.它以扩展的 Lustre 语言为源语言,以 Clight(CompCert 中的 C 语言子集)为目标语言.L2C 是面向实际工业应用的同步数据流语言编译器.重点介绍 L2C 编译器的核心翻译步骤及其设计与实现过程中考虑的主要问题和相关经验.

关键词: 经过验证的编译器;同步数据流语言;L2C;Coq 证明辅助器;核心翻译步骤

中图法分类号: TP314

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/5213.htm

Key Translations of the Trustworthy Compiler L2C and Its Design and Implementation

SHANG Shu, GAN Yuan-Ke, SHI Gang, WANG Sheng-Yuan, DONG Yuan

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

Abstract: Synchronous data-flow languages, such as Lustre, have been widely used in safety-critical industrial areas, such as airplanes, high-speed railways, and nuclear power plants. The safety of development tools themselves for these types of applications is highly required. In better solving the “miscompilation” problem, very successful progress has been made recently to implement the construction and verification of a conventional imperative language compiler, such as the CompCert C compiler, by using reliable-by-construction proof assistants. L2C is a trustworthy compiler developed based on such an approach, with an extended Lustre language as its source, and Clight, a C subset used in ComperCert, as its target. L2C is an industry-level synchronous data-flow language compiler developed by using the same technique. The paper focuses on the key translations of L2C and the main issues and experience in its design and implementation.

Key words: certified compiler; synchronous data-flow language; L2C; Coq proof assistant; key translation

同步数据流语言,如 Lustre[1,2]和 Signal[3],近年来在实时嵌入式安全攸关系统开发中得到了广泛应用,此类语言相关开发工具本身的安全性在这些领域中的准入标准也越来越严格.例如,以 Lustre 为基础定义建模语言

* 基金项目: 国家自然科学基金(90818019, 61462086); 国家科技重大专项(MJ-2015-D-066); Sino-European Laboratory of Informatics, Automation and Applied Mathematics 资助项目

Foundation item: National Natural Science Foundation of China (90818019, 61462086); National Science and Technology Major Project (MJ-2015-D-066); Sino-European Laboratory of Informatics, Automation and Applied Mathematics Grants

收稿时间: 2016-07-15; 修改时间: 2016-09-25; 采用时间: 2016-12-07; jos 在线出版时间: 2017-01-20

CNKI 网络优先出版: 2017-01-20 16:06:41, http://www.cnki.net/kcms/detail/11.2560.TP.20170120.1606.018.html

的 Scade[4]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 1 个商用编译器(在我国航空、高铁及核电等领域也有广泛使用),其设计开发过程(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".虽然 Scade 的 KCG 开发经受了很严格的测试和过程管理,然而,和其他软件一样,通过测试只能发现错误,严格的过程管理会改进产品质量,这些努力并不足以保证编译器是正确的.为了进一步提高编译器的安全和可信程度,仅通过传统的方法显然是不够的,人们很自然地会想到形式化验证的途径.

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[5],随后,Milner 等人在 1972 年给出了相应的机器证明[6].Dave 于 2003 年的综述列举了 1967 年~2003 年的大部分相关工作[7],包含从简单语言的单遍编译器到较成熟的代码优化遍历等形形色色的工作.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.CompCert 编译器[8]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 IA32 和 ARM 后端),使其可直接应用于范围广泛的嵌入式应用开发.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)都借助证明辅助工具 Coq[9,10]进行了证明,且这些证明可由独立的证明检查器检查,达到了人们所能期望的最高可信程度[11].Yang 等人关于 Csmith 的研究工作[12]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.因为 CompCert 编译器的杰出成就,其代表性论文[13]的作者 Leroy 获得了 2016 年度的 10 年内最有影响 POPL 论文奖(Most Influential POPL Paper Award).

CompCert 编译器实现了对翻译过程本身的验证.编译器验证的另外一种可选方案是翻译确认(translation validation),由 Pnueli 等人首先提出[14,15],所给出的示范例子的源语言是同步数据流语言 Signal[3],目标语言是 C.翻译确认的方法只对编译器的输入和输出做检查,而不关心编译器的具体实现,因而具有较好的可重用性.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,设计一种可证明二者等价性的自动确认程序.确认程序本身是编译器的一部分,因此必须是一个自动化的过程,从理论上决定了它往往只能关注部分或抽象性质的保持性,这是翻译确认方法的不足之处.相对来说,对编译器的翻译过程本身进行形式化验证是一种比较完美的解决方案,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,但这一方法的缺点是扩展性较差.实践中,较好的选择是针对较为稳定的主体翻译过程本身进行形式化验证,而对于容易变化的步骤(比如编译优化)进行翻译确认.比如,一些针对优化算法的翻译确认工作[16,17]可以很好地融合到 CompCert 编译器中.

L2C 是采用类似于 CompCert 编译器的方法开发的可信编译器.它以扩展的 Lustre 语言作为源语言,以 CompCert 的 Clight 作为目标语言,并且从验证方面也与 CompCert 完全对接.L2C 是采用对翻译过程本身进行验证的可信编译器中,面向实际工业应用的同步数据流语言编译器.

本文第 1 节概述 L2C 编译器的基本信息.第 2 节简介源语言的特性.第 3 节叙述 L2C 编译器的主体翻译框架.第 4 节结合具体例子介绍 L2C 编译器的几个有特色的核心翻译步骤.第 5 节以核心翻译步骤为重点,介绍翻译过程的验证中所考虑的主要问题以及相关的经验.第 6 节是相关工作的分析.第 7 节给出本文工作总结以及对未来工作的展望.

1 L2C 编译器简介

为了满足国内某安全攸关领域的需求,L2C 编译器的开发始于 2010 年 9 月,其目标是设计实现一个经过形式化验证的可信编译器,其源语言是面向领域的同步数据流语言 Lustre*(Lustre 语言的一个变种,参考下一节),目标语言是 C,最终可用作相关领域数字化仪控系统的安全级代码生成器.

L2C 编译器的发展进程可归为 3 个里程碑.一个是面向 Lustre*的一个核心子集,设计实现了 L2C 编译器的一个原型系统[18],于 2013 年 6 月完成验收.另一个里程碑是已实现除嵌套时钟外 Lustre*全部特性的一个单时钟 L2C 编译器版本,完全能够满足国内该安全攸关领域目前的实际应用需求,并于 2015 年 4 月完成严格的企业级

验收,这些工作的相关技术和代码已在实际应用中发挥作用.

在上述第 2 个里程碑之后,项目组对 L2C 编译器的设计框架进行了较大程度的优化调整,目标是拓展应用领域以及开源系统的建设.目前,L2C 编译器进入了第 3 个里程碑的发展阶段,其目标是在目前面向企业的版本(不开源)基础上裁减并适当改造,形成了覆盖 Lustre V6[19]全部特性的可开源版本.目前,这一 L2C 编译器的单时钟版本(L2C-MC)已经开放源码(https://github.com/l2ctsinghua/l2c/releases/tag/version-0.8),支持嵌套时钟的版本处于测试与完善的周期,其源码不久也将开放.

2 源语言的特性

L2C 编译器的不同版本,其源语言(Lustre*)可能有所不同,本文以目前支持的最多特性为准.Lustre*覆盖了 Lustre V6[19]的全部特性,并且根据实际应用的需求在此基础上有许多扩展,特别是在高阶运算方面比 Lustre V6 更加丰富.

图 1 展示一个简单的 Lustre*程序,可见,一个 Lustre*程序(program)由多个节点(node)组成,节点中包含输入参数(parameters)、输出参数(returns)和函数体(body),其中函数体又由局部变量(local variables)和等式(或语句)序列组成,结构清晰.

```
1:  node Main(x1:int^2;x2:int^2;x3:int;b:bool)
2:  returns (p:int^5;m:int;n:int;k:int;s:int)
3:  var
4:     y1:int;
5:     y2:int;
6:  let
7:     k=y1;
8:     y1=x3 when b;
9:     y2=x3 when not b;
10:    p=map<<Max;2>>(x1,x2);
11:    m=Stay(y1);
12:    n=Stay(y2);
13:    s=fby(y1;1;5);
14: tel

15: node Max(a:int;b:int)
16: returns (c:int)
17: let
18:    c=if a>b then a else b;
19: tel

20:  node Stay(a:int)
21:  returns(b:int)
22:  var
23:     m:int;
24:     n:int;
25:     k:int;
26:  let
27:     m=fby(a;1;5);
28:     n=fby(a;2;8);
29:     k=Max(m,n);
30:     b=Max(k,a);
31:  tel
```

Fig.1 A simple Lustre* program

图 1 一个简单的 Lustre*程序

下面以图 1 所示的 Lustre*程序为例来阐述 Lustre*的某些重要语言特性,并且在图 2 中给出图 1 中主节点 Main 的一组合理输入与输出来直观展示 Lustre*语言的这些特性.

(1) 流数据对象.图 2 展示的输入输出很直观地体现了 Lustre*程序区别于 C 语言程序的一个很重要的特性,即在 Lustre*中每一个变量都是一个无穷长的流(stream)数据对象,而不仅仅是一个单个的值.每一个周期(cycle),变量都可能会有不同的值或者没有值,其中图 2 截取了输入输出前 10 个周期的值,后面还有无穷个周期.虽然每个周期的值可能会变化,但处理逻辑每个周期都一样,如图 1 所示的主节点 Main 的代码逻辑,每个周期都一样.Lustre*是同步语言的一种,所有同步语言均满足同步假设:当前周期的输入时间出现时,系统能够在下个周期的输入时间出现之间计算出当前周期的输出.这一重要的同步假设使得同步语言在嵌入式实时控制系统中得到大规模应用.

(2) 数据流并发性.不同于 C 程序,Lustre*程序具有数据流并发性,Lustre*节点中的等式(相当于语句)虽然在书写时有先后顺序,但都是并发执行的.更复杂的情况是,Lustre*程序中的等式(或语句)之间存在因果关系,所以在并发执行时还需要考虑等式(或语句)间的拓扑关系.如图 1 中第 7 行~第 8 行所示,这两行就存在因果关系.第 7 行 k 的赋值依赖于第 8 行执行的结果,所以第 7 行应该在第 8 行之后执行.这里拓扑关系的具体含义可参见

第 4 节.

(3) 高阶算子.Lustre*支持如 map,re 等 10 多个高阶算子,这些高阶算子可以很方便地操作数组对象,在编写程序时会更加便利.如图 1 中第 10 行所示的 map 算子,以节点(或称函数)Max 以及两个数组 $x1$ 和 $x2$ 作为参数,返回另一个数组 p.语义上,相当于并行执行"p[i]=Max(x1[i],x2[i])"(i 从 0 变到 1).

(4) 时态与时钟算子.由于 Lustre*的流数据对象特性,语言提供了时态算子,比如 pre,fby,arrow(→)等,可以对流数据进行操作.另外,Lustre*还支持嵌套的时钟,提供了可改变时钟快慢的算子,如 when 使时钟变慢形成下一层嵌套的时钟,而 current 与 merge 使时钟变快,回到上一层嵌套时钟.如图 2 所示,使用 when 算子后,主节点中 $y1$ 的时钟相对于 $x3$ 的时钟变慢了(根据布尔量 b 取值为 true 时进行采样);使用了 fby 算子之后,主节点中 s 相对于 $y1$ 时钟周期不变,但每个周期的值向后 shift 了 1 个周期(注意,这里指相对于 $y1$ 时钟的 shift,即对应于 b 取值为 true 时的时钟周期).

Cycle	1	2	3	4	5	6	7	8	9	10
*x*1	(1,2)	(4,3)	(2,6)	(5,4)	(6,9)	(0,10)	(1,5)	(9,2)	(7,3)	(10,1)
*x*2	(1,1)	(7,2)	(3,3)	(4,5)	(10,4)	(8,4)	(9,1)	(2,5)	(4,7)	(3,2)
*x*3	2	4	6	8	10	12	14	16	18	20
b	T	F	F	F	T	T	F	T	F	F
*y*1	2				10	12		16		
*y*2		4	6	8			14		18	20
p	(1,2)	(7,3)	(3,6)	(5,5)	(10,9)	(8,10)	(9,5)	(9,5)	(7,7)	(10,2)
m	8				10	12		16		
n		8	8	8			14		18	20
k	2				10	12		16		
s	5				2	10		12		

Fig.2 A rational input and output of the Lustre* program in Fig.1

图 2 图 1 中 Lustre*程序的一组合理输入与输出

由于源语言 Lustre*这些 Clight 语言中没有的特性,导致了设计翻译算法的难度较大,正确性证明工作也显得相当复杂.正是这些原因,我们对整个可信翻译任务进行了合理的拆分,形成主体翻译框架,参见下一节.

3 L2C 编译器的主体翻译框架

随着源语言(Lustre*)特性的增加以及不同开发阶段的不同理念,L2C 可信编译器的翻译框架也在不断变化,先后出现过接近 10 个稳定的框架图,图 3 是其中最近的一个.

如图 3 所示,Lustre*源程序经过词法、语法以及静态语义分析(类型检查),翻译到类型良好的抽象语法树(abstract syntax tree,简称 AST)形式的高级中间语言 Well-typed Lustre* AST 代码. Well-typed Lustre* AST 代码经过一些简单的预处理变换(LustreSGen)生成一种具有规范形式的中间语言 LustreS 代码.预处理变换主要包括全局常量和类型的合并、拆分表达式列表、通过引入新变量提升某些特殊表达式(如 call 和 fby 表达式)至等式(或语句)级别等.

对于这部分的翻译过程,目前我们没有去进行形式化验证,故在图 3 中用虚线箭头来表示.词法和语法分析算法或者相关的工具是比较成熟的,比较可信,但若要验证它们的正确性却是很困难的,因此目前的可信编译器(包括 CompCert)基本上不验证这一部分工作.此外,其余的翻译工作(包括类型检查)相对比较简单,也比较容易验证,同时也因为它们是受 Lustre*变化影响较大的部分,所以我们在整个 L2C 可信编译器的实现中将这部分验证工作放在了最低的优先级.另外,这些翻译过程的翻译确认程序是比较容易实现的,这也是我们不急于完成这部分验证工作的重要原因之一.

图 3 中后续的翻译过程均表示为实线,其中所有的变换均经过了形式化验证(最后一步从 Clight 到汇编的变换对应于 CompCert 编译器的一系列翻译过程).这些翻译过程(CompCert 编译器除外)的主要工作分别是:

(1) Toposort:对 LustreS 代码的拓扑排序;

(2) LustreRGen:将高阶运算翻译为 for 循环,同时完成嵌套时钟的消去;

(3) TransMainArgs:将主节点的输入参数翻译为结构体;

(4) TransTypecmp:将复杂类型比较运算翻译为比较函数;

(5) LustreFGen:消去所有时态算子;

(6) OutstructGen:翻译生成输出结构体;

(7) ClassifyRetsVar:将输出变量从普通变量中分离出来;

(8) ResetfuncGen:生成 reset 函数;

(9) SimplEnv:将节点输入参数翻译为结构体以简化存储模型;

(10) ClassifyArgsVar:将输入变量从普通变量中分离出来;

(11) CtempGen:生成 Clight 代码的第 1 步,主要是完成到 C 的语法翻译;

(12) ClightGen:生成 Clight 代码的第 2 步,主要是将 Ctemp 中的 memcpy 语义翻译为内存拷贝函数.

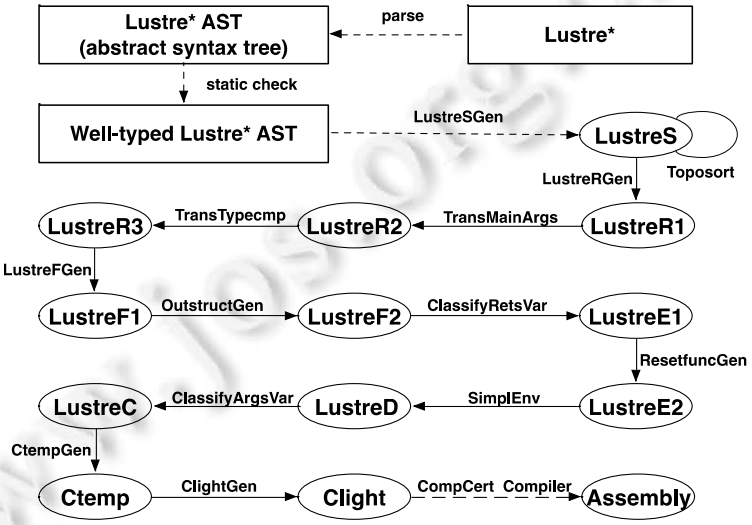


Fig.3 Translation framework of L2C

图 3 L2C 翻译过程框架

对完整翻译过程进一步的细化以及相关的对各翻译过程本身的正确性验证超出了本文的范围.下面我们分两节分别讨论核心的翻译步骤,以及有关翻译正确性验证的重点疑难问题及其设计实现方案.

4 核心翻译步骤示例

本节我们以第 2 节提到的 Lustre*语言的主要特性为线索来解释 L2C 在翻译过程中的关键节点是如何处理的,并以图 1 的实例来解释 Lustre*程序是如何被一步步地翻译到 Clight 语言的.

4.1 数据流并发性

Lustre*程序具有数据流并发性,而 Clight 程序却是串行执行的.因此,翻译过程中的一大难题就是要将 Lustre*语句串行化.现在一般采用的因果分析和排序大多采用测试或翻译确认的方法,未进行形式化验证.对于 L2C 构建经过形式化证明的可信编译器的目标,对排序做严格证明是必要的.我们首先定义 Lustre*中的因果关系.

(1) 如果一个等式 A 的左值出现在等式 B 的右值中,则说等式 B 依赖于等式 A.

(2) 如果一个节点 A 的左值出现在等式 B 的时钟中,则说等式 B 依赖于等式 B.

然后,我们定义拓扑排序的性质,并且定义拓扑排序等价性定理,即任意两个满足拓扑排序性质的程序在串

行执行语义中是等价的,以此来保证串行化方案的正确性.最后利用 Coq 实现拓扑排序算法,将并行的 LustreS 串行化,并以此算法完成之前定义的拓扑排序性质和语义等价性证明.

实际翻译效果以图 1 中的第 7 行~第 8 行为例,按照第 1 条因果关系的定义,图 1 中第 8 行的左值出现在了第 7 行的右值中,所以第 7 行语句依赖于第 8 行.图 4 显示了拓扑排序后 LustreS 程序中与图 1 第 7 行~第 8 行对应的代码片段,在排序后,图 1 的第 7 行被翻译为图 4 中的第 4 行,图 1 中的第 8 行被对应到图 4 中的第 2 行,可见排序后的语句满足因果关系的定义.

```
1:  …
2:  y1=x3;(b)
3:  …
4:  k=y1;()
5:  …
```

Fig.4 Topsorted LustreS intermediate representation of the statement of line 7~8 in Fig.1

图 4 图 1 中第 7 行~第 8 行翻译到排序后的 LustreS 的中间表示

4.2 高阶算子

Lustre*提供了 10 多个高阶算子(涵盖了 LustreV6 支持的所有高阶算子),包括 map,red 等.图 1 第 10 行使用了 map 算子,本节以 map 为例来介绍高阶算子消去.Lustre*中 map 算子的形式为 map⟨⟨op;size⟩⟩($a1$,$a2$,…,an),其中,$a1$~an 是 n 个输入数组,size 为数组的大小;而 op 为一个操作,可以是运算符或一个节点,拥有 n 个输入参数.map 算子运算的结果为一个数组,记为 res,其中,res[i]=op($a1$[i],$a2$[i],…,an[i]).图 1 中的 p 值展示了图 1 第 10 行 map 算子运算的结果,可见高阶算子简化了对数组的循环操作.

由于目标语言 Clight 不支持高阶运算,所以在翻译过程中,需要消去 Lustre*中的高阶算子.基于高阶算子对数组处理的特点,我们最终在 LustreS 到 LustreR1 阶段将所支持的各个高阶算子按其语义特性展开为 Clight 支持的语法结构,主体为 for 循环结构.

图 5 展示了图 1 中第 10 行的 map 运算在 LustreS 这一层的中间表示,可见,在 LustreS 这一层 L2C 不会对 map 高阶算子做翻译.

```
1:  for 2 {
2:      mapcall: int p[2]=Max.acg_context3(x1,x2);
3:  }
```

Fig.5 LustreS intermediate representation of the statement of line 10 in Fig.1

图 5 图 1 中第 10 行翻译到 LustreS 的中间表示

经过 LustreRGen 过程,L2C 会将 map 高阶表达式转换成 for 循环,如图 6 所示.

```
1:  for (acg_i=0;acg_i<2;acg_i=acg_i+1){
2:      p[acg_i]=Max.acg_context3[acg_i](x1[acg_i],x2[acg_i]);
3:  }
```

Fig.6 LustreR1 intermediate representation of the statement of line 10 in Fig.1

图 6 图 1 中第 10 行翻译到 LustreR1 的中间表示

通过设计翻译算法将 LustreS 中的高阶复杂运算分解成 LustreR1 中多个基础运算的循环,我们消除了 Lustre*程序中的所有高阶算子.

4.3 时态和时钟算子

Lustre*支持如 LustreV6 的一些可以操作流数据的时态算子,如 fby,以及操作变量时钟的时钟算子,如 when 算子.在 Lustre*程序中,每个变量均有自己的时钟,默认情况下为一个全局基本时钟,该时钟在每个时钟周期都

为 True.如图 2 中拥有全局基本时钟的 *x*3,在每个时钟周期都有值.而 Lustre*提供的 when 算子则可以用来改变某个变量的时钟,如图 2 中的 *y*1,*b* 为 True 的周期有值且与 *x*3 当前周期值相同,但 *b* 为 False 的周期的值则是未定义.再如 Lustre*提供的 fby 算子,可以用来访问流数据的历史值,它不改变时钟,返回值相当于向右 shift 流数据的值,如图 2 中的 *s*,它的值在 *y*1 上整体向右 shift 了一位,并且在 shift 产生的空缺值中补上了 fby 函数指定的默认值 5.

L2C 在 LustreS 到 LustreR1 阶段会处理所有时钟算子,在 LustreR3 到 LustreF1 阶段会处理所有的时态算子.以 fby 算子为例,在 LustreF1 中,图 1 中 Stay 节点内的第 27 行和第 28 行,fby 算子会被最终翻译成如图 7 所示的 LustreF1 代码,其中 *a* 为 Stay 节点的输入参数.

```
1:  int acg_j;
2:  if (acg_init){
3:      m=5;
4:  } else {
5:      m=acg_fby1;
6:  }
7:  if (acg_init){
8:      for (acg_j=0;acg_j<2;acg_j=acg_j+1){
9:          acg_fby2.items[acg_j]=8;
10:     }
11:     acg_fby2.idx=0;
12: }
13: n=acg_fby2.items[acg_fby2.idx];
14: acg_fby1=a;
15: acg_fby2.items[acg_fby2.idx]=a;
16: acg_fby2.idx=(acg_fby2.idx+1)%2;
17: acg_init=false;
```

Fig.7 LustreF1 intermediate representation of the statement of line 27~28 in Fig.1

图 7 图 1 中第 27 行~第 28 行翻译到 LustreF1 的中间表示

L2C 引入 acg_init 变量来标识当前周期是否为第 1 个周期,如果是第 1 个周期,那么将对变量 *m* 和 *n* 赋初值,赋初值时会根据 fby 指定的周期数来初始化存储 *m* 和 *n* 的数组大小,并在数组中对每个值赋予 fby 指定的默认值,然后在之后的周期依次循环遍历数组的值,并在每次读取值之后在数组当前位置记录变量在 fby 之前当前周期的值.以图 2 的输入值为例,我们将图 7 中 *n* 对应的 fby 相关参数 acg_fby2 前 4 个周期的变化过程展示出来,如图 8 所示,便于更好地理解图 7 的翻译结果.

Cycle	1	2	3	4
Main:y2	–	4	6	8
Stay(y2):n	8	8	–	4
acg_init	1	0	0	0
acg_fby2.idx	0	1	0	1
acg_fby2.items	(8,8)→(,8)	(,8)→(,4)	(,4)→(6,4)	(6,4)→(6,8)

Fig.8 Change process of acg_fby2 in Fig.7

图 8 图 7 中 acg_fby2 的变化过程

4.4 翻译至Clight

经过前面几个核心步骤后,已经消除了 Lustre*中最显著的同步数据流特征,程序已经十分接近常规的串行命令式语言.又经过后续若干个关键步骤(包含初始化函数生成 ResetfunGen)到 LustreC,已经十分接近 Clight,经由最后两个步骤实现与 CompCert 完全对接(如图 3 所示).从 LustreC 到 Clight 的翻译过程中,在语法上几乎没有太大的变化,但在语义环境上差异较大,证明工作繁重.图 9 展示了 L2C 翻译图 1 所示 Lustre*程序中 Main 节点的最终结果,具体翻译验证的难点我们将在下一节详细描述.

```
typedef struct {
  int idx;
  int items[2];
} acg_s2;

typedef struct {
  int x1[2],x2[2],x3;
  _Bool b;
} inC_Main;

typedef struct {int c;} outC_Max;

typedef struct {
  _Bool acg_init;
  int b,acg_fby1;
  acg_s2 acg_fby2;
  outC_Max acg_context1,acg_context2;
} outC_Stay;

typedef struct {
  int p[2];
  int m,n,k,s,acg_fby3;
  _Bool acg_init;
  outC_Max acg_context3[5];
  outC_Stay acg_context4,acg_context5;
} outC_Main;

… // reset functions
… // composite types compare functions
… // Max & Stay node

void Main(inC_Main *inC,outC_Main *outC) {
  int y1,y2,acg_i=0;
  if ((*inC).b){y1=(*inC).x3;}
  if (!(*inC).b){y2=(*inC).x3;}
  for (;1;acg_i=acg_i+1){
    if (! (acg_i<2)){break;}
    Max((*inC).x1[acg_i],(*inC).x2[acg_i],
      &(*outC).acg_context3[acg_i]);
    (*outC).p[acg_i]=((*outC).acg_context3[acg_i]).c;
  }
  if ((*outC).acg_init){(*outC).s=5;}
  else {(*outC).s=(*outC).acg_fby3;}
  if ((*inC).b){(*outC).k=y1;}
  if ((*inC).b){
    Stay(y1,&(*outC).acg_context4);
    (*outC).m=(*outC).acg_context4.b;
    (*outC).acg_fby3=y1;
  }
  if (!(*inC).b){
    Stay(y2,&(*outC).acg_context5);
    (*outC).n=(*outC).acg_context5.b;
  }
  (*outC).acg_init=0;
}
```

Fig.9 Clight program translated from the Main node in the Lustre* program in Fig.1

图 9 由图 1 所示 Lustre*程序的 Main 节点翻译而来的 Clight 程序

4.5 流数据对象

如图 1 所示,主节点 Main 的输入和输出都是无穷长的流数据,简单来说,每个时钟周期,Lustre*程序以输入流数据当前时钟周期的值传入 Main 节点作为输入,执行得到输出流中当前时钟周期的值.

由于程序获取输入和处理输出的方式不尽相同,所以 L2C 只会翻译 Lustre*程序中的节点,并不会给最终的 Clight 程序中加入 C 语言的入口函数 Main 函数.在实际使用中,如图 10 所示,我们会编写 C 语言的 main 函数循环调用调用生成 Main 函数,以此来处理流数据.

```
1:  int main() {
2:      inC_Main* in=malloc(sizeof(inC_Main));
3:      outC_Main *out=malloc(sizeof(outC_Main));
4:      init(in);
5:      while(true) {
6:          getNextInput(in);
7:          Main(in,out);
8:          printOutput(out);
9:      }
10:     return 0;
11: }
```

Fig.10 A solution for Infinite input and output stream

图 10 一种针对 Lustre*无限长输入输出流的解决方案

5 有关翻译正确性验证的重点疑难问题及其设计实现方案

在 L2C 可信编译器的设计与实现中,对于图 3 中实线所对应的翻译过程(CompCert 编译器除外)均借助于

Coq 证明了正确性(语义保持性),然后得出 LustreSGen 所产生的 LustreS 代码到 Clight 代码整个翻译过程的正确性.从 LustreS 到 Clight 的任意两个中间语言 S 和 T(设 S 在前)之间的语义保持性可描述为

$$\forall P.sound(P)\Rightarrow sound(\tau(P))\wedge S_S(P)\approx S_T(\tau(P)),$$

其中,τ是翻译函数,可将 S 中间语言的程序 P 翻译至 T 中间语言的程序 $\tau(P)$;S_S 是 S 中间语言的语义函数,S_T 是 T 中间语言的语义函数;$sound(P)$和 $sound(\tau(P))$分别刻画翻译前后的程序应满足的一些 Well-formedness 性质(比如第 5.4 节所述的左右值不相交的性质.另外,各层中间表示的性质会有所不同),用以将各阶段翻译过程的证明串连起来得到整个翻译过程的语义保持性,同时保证在每一阶段可以正常得到 $S_S(P)$和 $S_T(\tau(P))$;$\approx$是单向模拟等价关系:$S_S(P)\approx S_T(\tau(P))$意味着:$P$ 的所有环境变量在 $\tau(P)$都有匹配对象,且 $S_S(P)$对环境的改变可以由 $S_T(\tau(P))$对相匹配环境的改变进行模拟.对于合法的 LustreS 程序,求值结果总是确定的,这意味着语义计算的确定性(在非串行语义的情形,不同求值过程的结果也是确定的).因此,证明上述单向模拟关系就足够了.

这一验证目标看似比较明确,然而在具体设计和实现时,LustreS 到 Clight 的验证过程很复杂,会遇到许许多多的疑难问题.一方面是因为这两种语言差异非常大,LustreS 自身有很多独特的特点;另一方面是因为形式化验证的特点决定证明的过程可能会经过多次反复.限于篇幅,本文有选择地与读者分享在实际的设计和实现过程中遇到的一些重点疑难问题以及采取的具体方案,首先讨论若干共性的问题,然后再针对前一节提到过的核心翻译步骤详细加以说明.

5.1 整体证明框架的定义

形式化验证的特点决定了工作最难点在于整体证明框架的定义.形式化证明虽然能够为验证的正确性提供可靠保证,但验证过程的可复用性、可移植性差.如果在整体框架的定义上出现问题,即使是微小的问题,也可能造成整个证明过程需要推倒重来.所以,如何定义证明框架是能否完成证明的基础,这直接决定了整体证明的工作量.

为了解决证明框架的定义问题,我们在总结经验后采取了逆序分层验证的方式.一是由正常的 LustreS 到 Clight 的顺序验证,改为由 Clight 到 LustreS 的逆序验证过程.二是尽量将翻译过程拆分为小的步骤,每步只完成 1 个功能,使每一步的的验证尽量简单,以减少出错的可能性.

逆序验证能够保证每一步验证完成后,都能形成一个 Lustre*子集到 Clight 的已验证翻译过程.而这部分已验证的工作不会因为前端的翻译和证明的修改而变化.如果按顺序方式来验证,即使前面的证明已完成,也可能由于考虑不周而无法完成后续到 Clight 的证明,这样,前面已经完成的证明还是不得不反复修改.而小步分层的验证过程能够使每一层的证明过程简单化,而越简单的验证越不容易出现错误.而且由于是逆序验证,已完成的验证和前端待验证的层是独立的,以减少证明工作的耦合性,从而减少验证工作移植性差、可复用性差带来的工作量.

5.2 语义值和语义环境的定义

Lustre*的操作语义定义是实现 L2C 编译器中主体翻译阶段正确性验证的基础,而语义值和语义环境又是操作语义定义的基础.

在实现 L2C 编译器的证明过程中语义值的定义反复经过多次改动.每次改动都给证明过程带来很大的影响.最初 Lustre*语义值的定义与 Clight 相差非常大,这一方面导致 Lustre*在自身的简化证明过程中难度很大,另一方面又导致很难和 Clight 进行对接.最开始的定义尤其在 Lustre*语义值的写入和读取时,不支持读写范围的限制,这导致最后和Clight对接时,无法提供相应的条件.经过多次证明的反复修改,最终决定让Lustre*语义值的定义尽量接近 Clight,这样一方面可以吸收 Compcert 的成功经验,另一方面减少语义值的差异能够简化证明的过程.

Lustre*环境的定义是整个工作的难点之一.Lustre*程序的特点决定了 Lustre*不能简单使用 Clight 类似的环境定义.Lustre*的时态运算需要保留历史环境;Lustre*拓扑排序的证明需要环境在等式交换顺序时便于证明;Lustre*翻译过程中会对多种变量进行分类,要求不同类型变量的环境要有一定的隔离性;call 过程的证明是

每层证明的重点,要求环境在一定程度上反映 call 的调用关系;Lustre*程序执行过程中对类型和时钟的限制很严格,这要求环境在存储值的同时也要保存变量类型和时钟.环境定义的难点就在于在工作的初期很难预见到Lustre*对环境有如此多的要求,所以初期所定义的环境自然是很难满足证明需要的,在证明过程中不断地被修改,导致证明的过程也反复地被修改.比如,在第 5 版也是最终版的环境定义中,我们修改了第 4 版将普通变量和需要存储到输出结构体的特殊变量混合存放的做法(会使证明的难度增加),分开存放为普通变量和时态运算自定义变量等特殊变量提供了很好的隔离性,既简化了各层的证明,又解决了时态运算证明的问题.

5.3 ID管理

翻译过程中可能产生大量的中间变量,而且翻译的层数比较多.如何定义程序中已有的 id 集合不重复,如何保证翻译新生成的 id 集合不重复,以及如何保证新生成的 id 集合与已有 id 集合不相交,是贯穿整个验证过程的问题.为了解决这个问题,我们对整个翻译过程中的 id 进行了分类:程序自带的 id、翻译新增中间变量的 id、预定义 id 以及特殊预定义的 id.

5.4 左右值不相交的保持性证明

这一难点是在 CtempGen 的证明中处理结构体和数组类型的赋值运算时发现的.结构体和数组类型赋值运算要翻译为 Ctemp 中 memcpy 运算,这需要有一个对地址范围的限制:要么源地址和目的地址的指针不同,要么源地址和目的地址的偏移量不同,要么源地址范围和目的地址范围不相交.而 Lustre*中不存在指针,在这一步的证明之前也没有对 Lustre*赋值运算左右值的地址进行限制,所以需要定义 Lustre*中赋值的左右值地址不相交.后来发现在 call 运算中也存在类似的问题.由于地址不相交性质贯穿 Lustre*的所有层,还涉及 call 运算,所以带来的工作量非常大.尤其是在部分类型节点的输入和输出参数需要被翻译成结构体的情况下,这时结构体变量的地址相同,要证明其偏移范围不相交,证明过程很复杂.又因为这个定义涉及对表达式的地址求值,这也增加了证明的工作量.

5.5 程序初始化及reset函数的证明

Lustre*程序的主节点和节点列表都有对应的 reset 函数,主要用于初始化时态运算中所需的第 1 周期标识变量 acg_init.Reset 函数的生成和证明本身难度不大,难点在于在哪些层生成 reset 函数.Reset 函数的生成位置选择不对,会使证明难度加大,还可能因为提供的条件不够造成无法证明.经过多次试证,才发现节点类 reset 函数的生成 ResetfuncGen 放在返回值分类 ClassifyRetsVar 之后最合适.因为返回值分类后,将需要翻译为输出结构体的变量都归为一类,这为 reset 函数的翻译证明提供了合适的条件.而且 reset 函数翻译后,和普通节点类型统一起来,后续不必单独证明 reset 函数.reset 函数和程序初始化紧密相连,都是贯穿整个证明的过程,整体证明的工作量也较大.

5.6 拓扑排序Toposort的证明

经过预处理的 LustreS 程序是未经排序的.需要对节点列表和每个节点内的等式列表进行排序.使排序后的程序的执行按顺序语义执行,为后续证明提供条件.其中,节点的排序能够为后续只翻译单个节点时的证明提供条件.这一部分的证明分 3 个方面:(1) 排序后的程序和排序前的程序是同一个程序的排列;(2) 排序后的程序满足拓扑排序的性质;(3) 对同一程序经排序后,任意两个满足拓扑排序性质的程序按照 LustreS 顺序语义执行的结果是等价的.当前版本中定义了 LustreS 的并发语义,从并发语义到顺序语义的语义保持性证明本质上与这 3 个方面的证明是等效的.

5.7 高阶算子与嵌套时钟消去过程LustreRGen的证明

为了描述 Lustre*复杂的高阶运算,以及 mix 等各种特殊运算,LustreS 的语法定义比较复杂.LustreS 中的高阶运算以及各种特殊运算实际上是由多种简单运算组合而成.比如,普通的高阶运算可以化简为 for 循环运算;mapw 等特殊高阶运算可以化简为 for 循环和 if 运算的组合;flatten 运算可以拆分为多个赋值运算的序列;aryprj 运算可以拆分为 if 语句和赋值运算的组合.高阶算子消去的作用就是将 LustreS 中各种复杂运算拆分

为多步简单运算.另外,嵌套时钟的消除仅需要在等式/语句之前添加相应的条件语句.LustreRGen 翻译的过程不产生中间变量,其证明不涉及环境匹配的问题,因为翻译前后执行环境都完全相同.但因为需要拆分的复杂运算比较多,部分复杂运算的拆分过程比较复杂,使得证明的过程虽然不是很难,但证明的量比较大.

5.8 时态算子消去过程LustreFGen的证明

这一层的翻译主要是消去节点列表中的时态运算,这也是整体翻译和证明的难点.但通过前面一系列的分类和简化,使得这一部分的证明得以完成,需要处理 3 个时态算子 fby,fbyn 和 arrow,而且以语句的形式独立呈现.

Fby 语句的来源分为两部分,一是由 arrow(→)和 pre 算子转换过来的,二是 fbyn 中 n 为 1 的语句.Fby 和 fbyn 的翻译需要生成后置赋值等式(参见第 5.7 节的示例),负责延续历史值的生成.Fby 的翻译比较简单,通过第 1 周期标志变量 FLAGID,生成主体的 if 分类赋值语句,再生成预定义的中间变量和后置赋值等式.Fbyn 语句的翻译较为复杂,因为其历史值是存放在对应的数组当中的.在第 1 周期是需要利用 for 循环对其进行初始化;在后续周期循环取数组对应位置的值;后置等式要对 fbyn 下标进行模加运算,还需循环更新对应下标的值.Arrow 的翻译比较简单,通过第 1 周期标志变量 FLAGID,生成 if 分类赋值语句即可.

Fbyn 的翻译还会生成循环变量 *acg_j*,这需要根据是否有 fbyn 运算动态生成.后置等式的生成也是一样,需要根据是否有时态运算动态生成.

LustreFGen 的证明比较难,尤其是 fbyn 运算的证明.这种复杂的运算证明,必须在定义语义时与翻译过程相匹配,从而将复杂的运算拆分成小步的证明.而且之前的各层已经对 Lustre*语法和语义进行了大幅度简化,这使得本层的证明也得到了简化.

Fbyn 运算的复杂性使得证明语义保持性时环境的匹配过程相当复杂,它需要利用 for 循环对数组进行初始化操作,这样翻译后的程序环境中就可能增加循环变量 LOOPJ,而且 LOOPJ 变量需要根据是否有 fbyn 运算动态生成,为了适应这种情况的环境匹配,需要定义以下 5 条性质:(1) LOOPJ 不在全局常量中;(2) 翻译前后局部环境匹配;(3) 翻译后的环境已动态分配 LOOPJ 变量;(4) 全局常量中的值不在翻译前后的环境当中;(5) LOOPJ 不在翻译前的局部环境当中.

进一步地,还需要处理好 Fbyn 循环初始化执行的等价证明,以及后置等式列表的执行等价证明.

5.9 LustreC到Ctemp的证明

LustreC 到 Ctemp 翻译过程 CtempGen 的证明是整个证明过程中工作量最大的,也是最难的证明.由于证明的量过大,不得不分成两部分,第 1 部分主要是完成一些基本定义和底层证明,第 2 部分完成主体的证明.这里我们抛开翻译细节,主要分析一下证明量大的原因并描述解决方案.

首先是语义环境的差异较大.LsemC 的语义环境主要包括全局常量环境 gc、存储节点本地变量的环境 te、存储节点输入参数变量的 ta 和存储输出参数的 se.Ctemp 的环境主要包括全局环境 ge、存储节点本地变量地址的环境 eC、存储输入参数的环境 leC、存储输出参数的环境 teC 和存储数据的内存 mem.一是两者的环境的种类多,二是环境之间差异大.种类多造成工作量大,执行每步语义都需要证明所有环境都匹配.环境差异大造成环境匹配的定义较为复杂,从而证明的难度增加.语义环境中变量种类多所造成的证明工作量的问题是小问题,通过这种分类和环境隔离的方式,可以起到通过增加工作量来降低证明难度的目的.这正是前面各层拆分化简想要达到的目的.如果之前不对不同种类变量进行分类隔离,证明的难度将会更高.通过变量的分类隔离相对缩小了两者环境的差异.

其次,输出 C 代码的规范要求造成证明分支多.C 代码的规范化是企业版 L2C 编译器用户方的需求.仅举一个例子,比如主节点的输入输出结构体的生成.当输入参数为空时,不生成输入结构体;当输入参数不为空时,生成输入结构体.当输出参数为空时,不生成输出结构体;当输出参数不为空时,生成输入参数结构体.输入参数的两种情况和输出参数两种情况组合起来就是 4 种情况.这意味着 Ctemp 程序初始化的语义定义要分 4 种情况,Ctemp 程序的执行过程也要分 4 种情况.程序初始化和执行过程的证明就要分 4 种情况.诸如此类的情况是证明过程中不得不面对的问题,只能尽量将各种性质和语义定义得更加抽象,以适应更普遍的情况.在不考虑 C 代码

输出时(如只关心通过 CompCert 将 Clight 翻译为会编码),情况会简化一些.

第三,Ctemp 语义环境内的隔离性定义.虽然 Ctemp 语义环境的几种变量都是隔离的,但在 Ctemp 中的值主要还是存储在 mem 中.Ctemp 输入参数环境只存储普通值,结构体和数组类型的值只存储其指针,而指针对应的存储数据还是在 mem 中.Ctemp 中的输出参数也是一样.还有,Ctemp 的全局常量也是存储在 mem 中.这就使得在 Ctemp 的 mem 中写入任意一类变量,都要证明其不影响其他类型变量的值.即将 mem 中的多种变量隔离起来.为了解决这个问题,我们将 mem 划分几个范围,并通过多个性质的约束来实现不同类型变量地址范围的隔离性,包括:(1) 全局常量的存储范围;(2) 输入参数的地址范围;(3) 输出参数的地址范围不相交的性质;(4) 子函数的输出参数和父函数的输出参数的地址范围的关系定义;(5) 函数内局部变量的地址范围.

最后,函数和语句的互归纳证明.所有各层的节点和语句的证明都要通过互归纳的方式来实现.但由于 LustreC 到 Ctemp 的证明比较复杂,其函数和语句的互归纳的证明也就成为难点.最难之处在于没法一次完成该部分证明目标的定义和互归纳证明分支的定义.因为只有完成该部分的整体证明才能最终说明定义的目标是可证的.事实上,我们在实现这部分证明时,证明目标的定义和互归纳证明分支定义通过数十遍的反复证明才最终确定可证.这也是形式化验证证明复杂问题的一个困境,不到最后完成证明,很难确定之前的证明是否有用,甚至可能需要推倒重来.解决的方法只能通过不断拆分化简,将复杂的问题尽量分解为简单问题,以避免直接证明复杂的问题.好在之前已经经过了十几层的化简,否则很难想象证明的过程会有多复杂.

6 相关工作

学术界在关于编译器验证的工作中,面向 C、Java 之类的常规语言较多[8,20,21],而高级建模语言则并不多见.对于同步数据流语言,目前尚未见到有开源的或者在实际应用中发挥作用的经过形式化(机器)验证的编译器.

与本文工作密切相关的是以 Lustre 和 Signal 为代表的同步数据流语言的可信编译器研究.根据安全攸关领域的实际应用需求,目前这些编译器的目标语言均为 C 语言.就已知的项目情况而言,Lustre 到 C 可信编译器的代表性研究主要集中于对翻译过程本身进行验证;而 Signal 到 C 可信编译器的代表性研究则是以翻译确认的方法为主.

除了 Pnueli 等人首次提出翻译确认方法时的示范例子以外[14,15],近年来,Van Ngo 等人基于翻译确认的思想进一步开展了 Signal 到 C 的编译器验证工作[22,23],其主要工作是对源代码和目标代码使用统一的语义框架建模,给出源和目标之间的各种抽象等价关系,通过对等价关系进行验证来保证某些语义特征的一致性(比如时钟一致性),采用求解器自动验证等价关系的成立.

关于 Lustre 到 C 可信编译器,据我们所知,具有一定规模的研究小组,除作者课题组与国内面向安全关键领域的某技术企业建立的 L2C 项目组之外,国外主要是法国 INRIA 的 M.Pouzet 项目组.Pouzet 等人于 2006 年启动了一个有关同步数据流语言 Lustre 编译器验证的长线项目.该项目的工作[24−26]是将一种具有 Lustre 语言关键特性的小语言 MiniLS 翻译至一种基于对象的中间语言(这一中间语言容易翻译至 Java 和 C),采用 Coq 构造其核心翻译步骤并进行证明性证明.

CompCert 编译器已被学术界广泛用于构建可信软件的基础平台,如翻译确认程序的验证[16,17]、OS 内核的验证[27]、静态分析程序的验证[28],等等.以 CompCert 编译器为基础构建同步数据流语言可信编译器已成为国际上相关研究团队的兴趣点,包括 M.Pouzet 项目组[25].这样做,其直接优势是可以重用已被广泛认可的 CompCert C 形式规范(如语法、语义、存储模型以及相关性质[29,30]),并继承其从 C 到汇编的可信翻译过程.L2C 项目组从启动伊始就规划了以 Clight 为目标语言,因而在复用 CompCert 编译器方面占有一定的先机.

在国内,有一些关于同步语言建模、调度以及代码生成等方面的工作[31,32],但与编译器的验证关系不大.也有研究人员开展了关于编译器的验证、验证式编译器等相关领域的工作,另有一些研究组借助编译器进行程序的静态分析与验证,但均未涉及到同步语言的编译问题.

7 结论与展望

L2C 是从同步数据流语言 Lustre*(扩展的 Lustre 语言)到 Clight(CompCert 编译器中经过形式化验证部分的源语言)的可信编译器,其主体翻译过程本身经过了严格的形式化验证(借助证明辅助器 Coq),是同类工作中首个面向实际工业应用的同步数据流语言编译器.本文以 L2C 编译器的核心翻译步骤为出发点,结合实际开发经验,列举了 L2C 编译器的设计与实现过程中的若干重要或难点问题以及所采取的解决方案.本文不是对 L2C 可信编译器完整和形式的论述,有许多重要方面未能涉及.同时,它也不是一份技术文档,不可能对所讨论到的问题进行详尽描述.本文旨在实践经验的分享,有利同行读者之间相互学习交流.有意进一步了解 L2C 编译器细节的读者,可关注 L2C 主页(http://soft.cs.tsinghua.edu.cn:8000/).目前可下载单时钟 L2C 编译器的开源版本 L2C-MC,很快也会有支持嵌套时钟的开源版本上线.

References:

[1] 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. 178−188.

[2] Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous dataflow programming language LUSTRE. Proc. of the IEEE, 1991,79(9):1305−1320. [doi: 10.1109/5.97300]

[3] Guernic M, Gautier T, Maire C. Programming real time applications with signal. Proc. of the IEEE, 1991,79(9):1321−1336. [doi: 10.1109/5.97301]

[4] http://www.esterel-technologies.com/products/scade-suite/

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

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

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

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

[9] Coq Development Team. The Coq Reference Manual. 2016. http://coq.inria.fr/

[10] Bertot Y, Castéran P. Interactive theorem proving and program development—Coq'Art: The calculus of inductive constructions. In: Proc. of the EATCS on Texts in Theoretical Computer Science. Springer-Verlag, 2004. [doi: 10.1007/978-3-662-07964-5]

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

[12] 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. [doi: 10.1145/1993498.1993532]

[13] Leroy X. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. ACM SIGPLAN Notices, 2006,41(1):42−54. [doi: 10.1145/1111320.1111042]

[14] Pnueli A, Siegel M, Singerman E. Translation validation. In: Proc. of the TACAS'98. LNCS 1384, 1998. 151−166. [doi: 10.1007/BFb0054170]

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

[16] Tristan J-B, Leroy X. Verified validation of lazy code motion. In: Proc. of the PLDI. 2009. 316−326. [doi: 10.1145/1542476.1542512]

[17] Tristan J-B, Leroy X. A simple, verified validator for software pipelining. In: Proc. of the POPL. 2010. 83−92. [doi: 10.1145/1706299.1706311]

[18] 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]

[19] http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6

[20] 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. 2007. 54−65. [doi: 10.1145/1273442.1250742]

[21] 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: 10.1145/1146809.1146811]

[22] Van Ngo C, 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, 2012. 113−127. [doi: 10.1007/978-3-642-30729-4_9]

[23] 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 (FASE-ETAPS 2015). LNCS 9039, 2015. 171−185. [doi: 10.1007/978-3-662-46675-9_12]

[24] 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). Tucson, 2008. [doi: 10.1145/1375657.1375674]

[25] Auger C, Pouzet M. A formalization and proof of a modular lustre compiler. Technical Report, LRI, 2012. http://www.di.ens.fr/~pouzet/cours/mpri/cours4/scp12

[26] Auger C. Compilation certifiee de SCADE/LUSTRE [Ph.D. Thesis]. Universite de ParisSud, 2013.

[27] Gu RH, Koenig J, Ramanamandro T, Shao Z, Wu XN, Weng SC, Zhang HZ, Guo Y. Deep specifications and certified abstraction layers. In: Proc. of the POPL 2015. Mumbai: ACM Press, 2015. 593−608. [doi: 10.1145/2676726.2676975]

[28] Jourdan J-H, Laporte V, Blazy S, Leroy X, Pichardie D. A formally-verified c static analyzer. In: Proc. of the POPL2015. Mumbai: ACM Press, 2015. 247−259. [doi: 10.1145/2676726.2676966]

[29] 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]

[30] Leroy X, Blazy S. Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of. Automated Reasoning, 2008,41(1):1−31. [doi: 10.1007/s10817-008-9099-0]

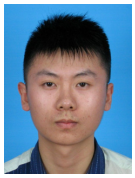
[31] Zhu XY, Yan RJ, Gu YL, Zhang J, Zhang WH, Zhang GQ. Static optimal scheduling for synchronous data flow graphs with model checking. In: Proc. of the 20th Int'l Symp. on Formal Methods (FM 2015). LNCS 9109, At Oslo, 2015. 551−569. [doi: 10.1007/978-3-319-19249-9_34]

[32] Yang ZB, Zhao YW, Huang ZQ, Hu K, Ma DF, Bodeveix JP, Filali M. Time-Predicatable multi-threaded code generation with synchronous languages. Ruan Jian Xue Bao/Journal of Software, 2016,27(3):611−632 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4984.htm [doi: 10.13328/j.cnki.jos.004984]

附中文参考文献:

[18] 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]

[32] 杨志斌,赵永望,黄志球,胡凯,马殿富,Bodeveix JP,Filali M.基于同步语言的时间可预测多线程代码生成方法,软件学报,2016,27(3):611−632. http://www.jos.org.cn/1000-9825/4984.htm [doi: 10.13328/j.cnki.jos.004984]



尚书(1992－),男,湖北枝江人,硕士生,CCF 学生会员,主要研究领域为形式化验证,可信编译.

甘元科(1983－),男,工程师,主要研究领域为形式化验证.

石刚(1972－),男,博士生,讲师,CCF 学生会员,主要研究领域为验证编译器,形式化方法.

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.

董渊(1973－),男,博士,副研究员,CCF 专业会员,主要研究领域为操作系统,编译系统,基于语言的可信软件.