软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007536] [CSTR: 32375.14.jos.007536]

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

形式化验证的类 Scade 时态算子编译*

甘元科, 王生原

(清华大学 计算机科学与技术系, 北京 100084)

通信作者: 甘元科, E-mail: anshuer@foxmail.com; 王生原, E-mail: wwssyy@tsinghua.edu.cn

摘　要: Scade 是一种广泛用于安全攸关嵌入式控制软件开发的著名商业工具, 其建模语言是从同步数据流语言 Lustre 扩展而来的同步语言. 包括 Lustre 在内的同步语言的正确编译近年来备受关注, 并在许多研究中通过形式化验证方法来解决. 对此类语言构建形式化验证的编译器, 实践中常见的做法是先将源程序编译为类 C 程序, 然后使用形式化验证的后端编译器 (如 CompCert 编译器) 将其编译为机器相关的低级代码. 其中, 时态算子的正确编译是至关重要的. 介绍一种形式化验证的类 Scade 时态算子编译并将其应用于形式化验证的编译器项目中, 该项目将 Lustre 扩展的同步语言翻译成 CompCert 编译器的前端中间语言 Clight. 时态算子的编译和形式化验证分为两个核心阶段, 在交互式证明辅助器 Coq 中实现.

关键词: 形式化验证; 编译; 时态算子; 同步数据流语言; Lustre; Scade; CompCert; Coq

中图法分类号: TP314

中文引用格式: 甘元科, 王生原. 形式化验证的类Scade时态算子编译. 软件学报. http://www.jos.org.cn/1000-9825/7536.htm

英文引用格式: Gan YK, Wang SY. Formally Verified Compilation of Scade-like Temporal Operators. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7536.htm

Formally Verified Compilation of Scade-like Temporal Operators

GAN Yuan-Ke, WANG Sheng-Yuan

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

Abstract: Scade is a well-known commercial tool widely used in the development of safety-critical embedded control software, whose modeling language is a synchronous language extended from Lustre, a synchronous data-flow language. Correct compilation of synchronous languages, including Lustre, has attracted much attention in recent years, and has been addressed in many studies through formal verification. To build a formally verified compiler for such a language, it is a common practice to compile the source program into a C-like program first, and then to compile it into low-level machine-dependent code using a formally verified backend compiler such as the CompCert compiler, where the correct compilation of temporal operators is crucial. In this study, the formally verified compilation of Scade-like temporal operators is introduced, which is used in a formally verified compiler projects, where a Lustre-extended synchronous language is translated into the front-end intermediate language Clight in the CompCert compiler. The compilation and formal verification of temporal operators are divided into two key stages, which are implemented in the interactive proof assistant Coq.

Key words: formal verification; compilation; temporal operator; synchronous data-flow language; Lustre; Scade; CompCert; Coq

近年来, 同步语言 (包括 Lustre)[1–4]和其他高级建模语言的可信编译构造广受关注[5], 催生了许多基于形式化验证的相关研究工作[6–12]. 为此类语言构建形式化验证的编译器, 常见的做法是先将源程序编译为类 C 程序, 然后借助于形式化验证的后端编译器 (如 CompCert[13–15]) 将其编译为低级机器相关代码.

商用的工业工具 Scade[16,17]提供了一个航空认证的编译器, 将基于 Lustre 扩展的同步语言翻译至 C 语言, 其

* 基金项目: 国家重点研发计划 (2022YFB3305204)
收稿时间: 2025-04-15; 修改时间: 2025-06-22; 采用时间: 2025-08-26; jos 在线出版时间: 2026-01-14

代码生成器 KCG 通过了大规模的测试, 并经受了严格的质量控制. 然而, KCG 的正确性尚未得到形式化的证明, 因此在可信度方面仍有很大的提升空间.

为构造可信编译器, 一种方法是基于证明辅助器 (如 Coq[18,19]), 首先形式化描述源语言、目标语言、中间语言的语法语义以及翻译算法, 然后机器证明源和目标之间的语义保持关系. CompCert 可信编译器[13–15]是采用该方法最有代表性的工作之一, 它将一个严格定义的 C 语言的大子集 Clight[20]编译至多种目标机器的汇编代码. 基于证明辅助器, 可以对编译过程的正确性证明实行机器检查, 因而可以确信这一方法能够达到我们所希望的最高等级的可信性[21].

本文介绍一个形式化验证的可信编译器项目 (L2C) 中的一个核心环节, 即形式化验证的时态算子编译. 该项目是我们的一个长线项目, 采取类似于 CompCert 项目的技术路线, 编译器的源语言 (本文称之为 Lustre*) 是一种基于 Lustre 扩展的同步语言, 目标语言是 CompCert 中的 Clight.

Lustre*语言是一种同步语言, 它涵盖经典同步数据流语言 Lustre[1,2]以及 Lustre V6[22]的全部核心特性, 另外增加了 Scade[16,17]中某些实用特性. Lustre*支持 Scade 建模语言中的所有时态算子 (非常用算子 times 除外) 和时钟算子, 前者包括 *pre*、-> (arrow)、二元 *fby* 和三元 *fby*, 而后者包括 when 和 merge. 这里, 我们称 when 和 merge 为时钟算子, 它们在某些场合也被称为时态算子.

在我们的可信编译器中, 时钟算子 when 和 merge 在一些早期阶段进行了处理; 时态算子 *pre* 被 binary *fby* 所取代, 因为在工程上传统的初始化约束中, 它只能出现在类似 *a* -> *pre*(*b*) 的表达式 (相当于 *a fby b*), 或者通过其他方式在第 1 个激活周期关联一个初始值, 其行为等同于二元 *fby* 算子. 因此, 在本文大部分内容里, 我们聚焦于讨论时态算子 ->、二元 *fby* 和三元 *fby*.

不同于文献中其他类似的研究工作[6–8,23–29], 为了简化语义定义和证明, 我们对时态算子使用了两阶段的编译和验证. 第 1 阶段将时态算子的语义划分为计算阶段 (computation) 和尾声阶段 (epilogue), 第 2 阶段完成后续翻译和验证. 本文其余部分详细介绍该两阶段方法, 其组织如下.

第 1 节简要介绍 Lustre*语言中所包含的类 Scade 时态算子, 以及一些相关的同步数据流特性. 第 2 节介绍时态算子的两阶段编译. 第 3 节聚焦于讨论两阶段编译的正确性验证. 第 4 节是相关工作介绍. 第 5 节总结全文.

1 Scade 时态算子

首先对 Lustre*语言所包含的 Scade 时态算子以及与本文相关的基本同步数据流特性进行简要介绍.

图 1 给出一个 Lustre*示例程序, 当删除前 10 行时, 可以得到与文献 [6] 中基本相同的 Lustre 程序, 除第 38 行外 (文献 [6] 中的可信编译器不支持时态算子 *pre*), 其余部分均相同.

如同其他类 Lustre 同步语言, 一个 Lustre*程序由节点 (node) 以及其他 (类型和常量) 声明的列表组成. 如同 Lustre V6 和 Scade, Lustre*通过关键字 function 和 node 区分两类节点. 不同于 node, function 不包含时态算子, 其动态语义也不需要记忆过往的历史值. 为简化讨论, 本文中, 节点和函数都被假设为“节点”. 节点是可重用的用户定义算子, 类似于常规语言中的函数, 但其输入数据为多个流 (stream), 也可能有多个流作为返回值. 一个节点可以由其他节点调用, 但不允许递归调用. 每次执行程序时都必须指定一个主节点. 例如, 我们假设 tracker_t_sum 是图 1 示例程序中的主节点.

在 Lustre*程序中, 每个节点都由一组等式 (equation) 组成. 每个与输入不对应的变量都应该在某个等式中给出一个且只有一个定义. 如图 1, 节点 tracker_cdiff 和 tracker 分别拥有 4 和 5 个等式. 一种基本类型的等式形如 $x = e$, 它将变量 x 定义为与表达式 e 相同. 第 2 种基本类型的等式形如 $x_1,\ldots, x_m = op(e_1,\ldots, e_m)$, 它描述了一个节点的调用. 例如, 图 1 中节点 tracker 的等式 $(s, p) = d\_integrator(acc)$. 等式的另一个基本类型涉及高阶迭代器, 不在本文讨论范围, 这里忽略.

时钟是类 Lustre 同步语言中的一个重要特性. 一个 Lustre*程序每个时钟周期执行一次, 以控制安全关键设备. 在第 1 个周期中, Lustre*程序被初始化并从主节点启动执行, 使用一个从外部环境赋予的时钟, 后者称为基

准时钟 (base clock), 每个周期的时间间隔基于该基准时钟. 之后, 主节点在每个时钟周期被重复调用. 其余各节点都由主节点直接或间接调用, 这些被调用节点的局部基准时钟 (local base clock) 源于当前调用方节点实例的环境, 可以在编译时间获得. 这样一直执行, 直到接收到重置 (reset 或 restart) 信号, 此时程序将返回新的初始周期重新执行.

```
(0)   const k:int = 3;
(1)
(2)   node tracker_cdiff (acc, limit: int) returns (td: int)
(3)       var p, t : int; c : bool; pt : int when c;
(4)   let
(5)       td = merge c (t when c - pt) (0 when not c);
(6)       pt = fby(t when c; k; 0);
(7)       c = limit > 0;
(8)       (p, t) = tracker(acc, limit);
(9)   tel
(10)
(11)  node counter(ini, inc: int; restart: bool) returns (n: int)
(12)      var c : int; f : bool;
(13)  let
(14)      n = if f or restart then ini else c + inc;
(15)      f = true fby false;
(16)      c = 0 fby n;
(17)  tel
(18)
(19)  node d_integrator(gamma: int) returns (speed, position: int)
(20)  let
(21)      speed = counter(0, gamma, false);
(22)      position = counter(0, speed, false);
(23)  tel
(24)
(25)  node rising(s: bool) returns (edge: bool)
(26)      var ps : bool;
(27)  let
(28)      edge = not ps and s;
(29)      ps = true fby s;
(30)  tel
(31)
(32)  node tracker(acc, limit: int) returns (p, t: int)
(33)      var s, pt : int; x : bool; c : int when x;
(34)  let
(35)      c = counter(1 when x, 1 when x, false when x);
(36)      x = rising(s > limit);
(37)      (s, p) = d_integrator(acc);
(38)      pt = 0 fby t;
(39)      t = merge x c (pt when not x);
(40)  tel
```

图 1　Lustre*程序示例

当程序执行时, 只要满足程序依赖关系, 节点实例中的所有等式都会在其被活动的周期中并发执行. 在 Lustre* 中, 一个变量在不同的时钟周期可能具有不同的值. 因此, 变量的值形成了一个数据的流 (stream). 一个普通的 Lustre*表达式, 一般只能直接访问变量在当前周期的值, 先前时钟周期的任何值只能使用时态算子 (temporal operator) 进行读取, 但后者也无法修改前面周期的值.

Lustre*提供 4 个时态算子: *pre*、-> (arrow)、二元 *fby* 和三元 *fby*. 前 3 个与经典 Lustre 中的时态算子相同, 它们同样也是 Lustre V6 和 Scade 中支持的算子. 二元表达式 *a* -> *b* 的值 (流), 在第 1 个激活周期 (时钟同 *a* 或 *b*) 的值取 *a* 在该周期的值, 其他激活周期中的值取 *b* 在相应周期的值. 一元表达式 *pre*(*a*) 的值 (流), 当前激活周期取 *a* 在上一个激活周期的值, 通常使用 *nil* 或“_”表示一个未定义 (undefined) 的值. 二元 *fby* 算子可表示为->和 *pre* 的组合, 表达式 *a fby b* 等价于 *a* -> *pre*(*b*). 三元 *fby* 算子在 Scade 中支持, 而在 Lustre V6 中不支持. 三元 *fby* 表达式 *fby*(*a*, *n*, *b*), 当 *n* > 1 时等价于 *b* -> *pre*(*fby*(*a*, *n*–1, *b*)), 以及 (作为归纳基础) 当 *n* = 1 时, *fby*(*a*, 1, *b*) 等价于二元 *fby* 表达式 *b fby a*.

Lustre*表达式中任何运算 (merge 除外) 的操作数都必须具有相同的时钟. 具有不同时钟的流不能相互操作, 除非操作数是任何时候均为恒定常量的流. 流可以使用时钟算子进行同步, 算子 when 可以产生较慢的时钟, 而算子 merge 则可以使时钟更快.

可以从表 1 中初步了解时态算子和时钟算子的含义, 其中, 0–7 为时钟周期 (cycles), T 表示 true, F 表示 false. 两个表达式 *x* when *ck* 和 *x* when not *ck* 有不同但互补的时钟. 表达式 *fby*(*x* when not *ck*, 2, 5) 与 *x* when not *ck* 具有相同的时钟. 一个表达式在不被激活的周期是没有值的, 这在表 1 中表示为“—”. 算子 merge 可作用于时钟互补的两个表达式, 参见表 1 的最后 3 行.

表 1 包含时态和时钟算子的表达式执行示例

变量或表达式	0	1	2	3	4	5	6	7	…
ck	F	T	F	T	F	F	T	T	…
x	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	…
5 -> *x*	5	v_2	v_3	v_4	v_5	v_6	v_7	v_8	…
pre(*x*)	—	v_1	v_2	v_3	v_4	v_5	v_6	v_7	…
5 -> *pre*(*x*)	5	v_1	v_2	v_3	v_4	v_5	v_6	v_7	…
5 *fby x*	5	v_1	v_2	v_3	v_4	v_5	v_6	v_7	…
fby(*x*, 3, 5)	5	5	5	v_1	v_2	v_3	v_4	v_5	…
x when *ck*	—	v_2	—	v_4	—	—	v_7	v_8	…
x when not *ck*	v_1	—	v_3	—	v_5	v_6	—	—	…
merge *ck* (*x* when *ck*) (*x* when not *ck*)	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	…
5 *fby* (*x* when *ck*)	—	5	—	v_2	—	—	v_4	v_7	…
fby(*x* when not *ck*, 3, 5)	5	—	5	—	5	v_1	—	—	…

默认读者可以理解图 1 第 11–40 行代码, 不妨可参考 Bourke 等人[6]的工作. 若不熟悉 Lustre 语言的基本特性, 表 2 下半部分刻画了图 1 中 tracker 节点实例的执行. 该 tracker 实例的执行返回流 *p* 和 *t*, 其实际含义分别可以解释为跟踪设备/车辆从其在初始周期的起点到当前周期为止的位移量, 以及在超过速度限制条件时触发上升沿的次数. 表 2 上半部分刻画了图 1 中程序的执行, 是以 tracker_cdiff 作为主节点, 以及输入流读取自 *acc* 和 *limit* 对应的源 (设备), 其输出流的含义不难通过图 1 中第 2–9 行代码及表 2 中流变量 *td* 来理解.

表 2 中列举了主节点 tracker_cdiff 以及节点 tracker 实例中所有的流变量, 忽略了其他节点实例的变量. 此外, 为便于更好地理解 tracker_cdiff 中三元 *fby* 表达式 *fby*(*t* when *c*, *k*, 0), 表中也同时给出了流表达式 *t* when *c* 所对应的行. 其中, 0–12 为时钟周期.

表 2 图 1 中 Lustre*程序 (以 tracker_cdiff 为主节点) 的执行

变量或表达式		0	1	2	3	4	5	6	7	8	9	10	11	12	…
tracker_cdiff (main node)	*acc*	0	2	4	−2	0	3	−3	2	0	0	−1	3	2	…
	limit	0	5	0	5	0	0	5	0	5	5	5	5	5	…
	c	F	T	F	T	F	F	T	F	T	T	T	T	T	…
	p	0	2	8	12	16	23	27	33	39	45	50	58	68	…
	t	0	0	1	1	2	3	3	4	5	6	6	7	8	…
	t when *c*	—	0	—	1	—	—	3	—	5	6	6	7	8	…
	pt	—	0	—	0	—	—	0	—	0	1	3	5	6	…
	td	0	0	0	1	0	0	3	0	5	5	3	2	2	…
tracker	*acc*	0	2	4	−2	0	3	−3	2	0	0	−1	3	2	…
	limit	0	5	0	5	0	0	5	0	5	5	5	5	5	…
	s	0	2	6	4	4	7	4	6	6	6	5	8	10	…
	p	0	2	8	12	16	23	27	33	39	45	50	58	68	…
	pt	0	0	0	1	1	2	3	3	4	5	6	6	7	…
	x	F	F	T	F	T	T	F	T	T	T	F	T	T	…
	c	—	—	1	—	2	3	—	4	5	6	—	7	8	…
	t	0	0	1	1	2	3	3	4	5	6	6	7	8	…

2 时态算子的编译

2.1 时态算子编译概述

本文以形式化验证的可信编译器 L2C 的一个开源版本 (Open-L2C[30]) 为例, 图 2 刻画了 Open-L2C 可信编译

器架构中的一个侧面, 聚焦于与时态算子编译密切关联的一些核心翻译步骤, 分为前期处理、第 1 阶段 (包含于图 2 的 LustreRGen 步骤)、第 2 阶段 (包含于图 2 的 LustreFGen 步骤) 以及后续步骤. 对此, 将在第 2.2–2.5 节进一步介绍.

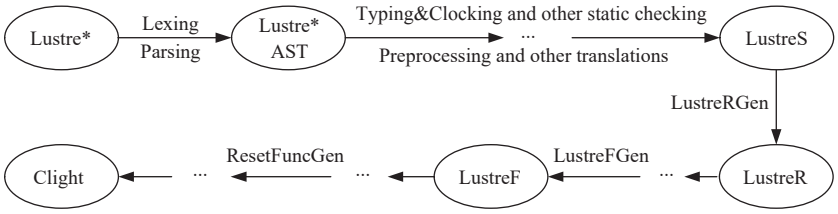

图 2 可信编译器架构中时态算子编译的核心翻译步骤

Open-L2C 可信编译器的后端起点为中间表示 LustreS. Open-L2C 之后, 其他版本 L2C 可信编译器的后端起点有所不同, 但本文介绍的时态算子编译和形式化验证方法并未发生改变. 附录 A 给出当前增加类 Scade 状态机与条件块等特性的 L2C 可信编译器架构, 其中 LustreS 不再是后端起点, 时态算子编译和形式化验证的分阶段处理位置会有所调整.

图 3 通过一个特例刻画了可信编译器中三元 *fby* 算子翻译的核心技术路线. 图 3 的左边给出了等式 $x = fby(e_1, n, e_2)$ 的编译结果概要, 这里假设 $n = 3$ 以及当前时钟为基准时钟. x_s 是用来模拟该时态算子输入流的状态变量 (state variable), *init* 是标识这一 x_s 流首个激活周期的标志变量 (flag variable), 而 x_h 是 x_s 流的类型 (结构体类型). 中间 (computation stage) 的代码行负责 x_s 流的初始化, 以及读取 x_s 流上一个激活周期的值. 尾部 (epilogue stage) 的 3 行代码用来为下一激活周期准备数据, 并将标志变量复位. 后续会进一步介绍相关细节.

$x = fby(e_1, n, e_2)$ trans

x_h: struct
{*idx*: int;
items: array[3] of τ;
};
(τ is the type of e_1 or e_2)

```
if (init) {
    for (i = 0; i < 3; i++) {
        x_h.items[i] = e_2;
    }
    x_h.idx = 0;
}
... // statements in the same node
x = x_h.items[x_h.idx];
x_h.items[x_h.idx] = e_2;
x_h.idx = (x_h.idx + 1) % 3;
```

Cycle	0	1	2	3	4	5	6	7	8
e_1	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
e_2	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
x	b_0	b_0	b_0	a_0	a_1	a_2	a_3	a_4	a_5

init	T	F	F	F	F	F	F	F	F
$x_h.idx$	0	1	2	0	1	2	0	1	2
$x_h.items[0]$	a_0	a_0	a_0	a_3	a_3	a_3	a_6	a_6	a_6
$x_h.items[1]$	b_0	a_1	a_1	a_1	a_4	a_4	a_4	a_7	a_7
$x_h.items[2]$	b_0	b_0	a_2	a_2	a_2	a_5	a_5	a_5	a_8
x	b_0	b_0	b_0	a_0	a_1	a_2	a_3	a_4	a_5

图 3 等式 $x = fby(e_1, n, e_2)$ 的翻译 (取 $n = 3$, 时钟为基准时钟)

如图 3 右边的 x 流所示, $x = fby(e_1, 3, e_2)$ 将输入流 e_1 延迟 3 个周期 (假定当前时钟为基准时钟), 且在头 3 个激活周期被初始化为输入流 e_2 在首个激活周期的值. 图 3 右边的虚线对应时态算子翻译前后的分界线.

时态算子->和二元 *fby* 翻译的核心思想与三元 *fby* 翻译相类似.

2.2 时态算子编译的前期处理

如图 2 所示, Lustre*源程序在词法和语法解析之后, 伴随着生成抽象语法树 (Lustre* AST). 图 4 给出一个简化版的 Lustre*抽象语法, 省略了源语言的许多其他特性, 后者多数与本文的关联性不直接, 包括表达式列表、调用子表达式、高阶迭代算子以及数组和结构体相关的算子等.

时态算子编译的前期处理工作集中于 LustreS 中间表示生成之前. 如图 2 所示, 从 Lustre* AST 到 LustreS 经

历了静态语义检查、预处理以及其他翻译过程. 其他翻译过程如状态机和条件块的翻译、拓扑排序等, 与时态算子的编译不很密切, 虽然时态算子对这些翻译过程有一定影响 (如拓扑排序过程与程序依赖关系密切相关, 而时态算子对于后者有特殊的影响). 预处理工作中, 有一些环节对于时态算子编译来说是至关重要的 (前期处理工作).

prog	::=	$(\overrightarrow{td}, \overrightarrow{cd}, \overrightarrow{node}, main\_id)$	e	::=	x
td		(type decl)		|	c (constant)
cd		(constant decl)		|	$uop\ e$ (unary operation)
node	::=	$op\ (args)$ **returns** $(rets)\ body$ (node decl)		|	$e\ bop\ e$ (binary operation)
body	::=	**var** *vars* **let** $\overrightarrow{eq}$ **tel**		|	**if** e **then** e **else** e
args, vars, rets	::=	$\overrightarrow{vd}$		|	**pre** e
vd	::=	(ID, τ) (variable decl)		|	$e \rightarrow e$
	|	(ID, τ, ck) (variable decl with sampling)		|	e **fby** e
τ		(data type)		|	**fby** (e, n, e) (n: positive integer)
ck	::=	x | **not** x		|	e **when** *ck*
eq	::=	$x = e$ (common equation)		|	**merge** (x, e, e)
	|	$\overrightarrow{x} = op\ (\overrightarrow{e})$ (call equation)		...	(other expressions)
main_id, op	::=	*ID*	x		(variables or things with L-value)

图 4 Lustre* 抽象语法 (节选)

图 5 所示是节选的 LustreS 中间语言的抽象语法. 就时态算子而言, 如前述原因 (参见引言和第 1 节), 本文聚焦于讨论算子-> (arrow)、二元 *fby* 和三元 *fby*, 因而省略了算子 *pre*.

node	::=	(*fd, svars*)	e	::=	*se*
fd	::=	$op\ (args)$ **returns** $(rets)\ body$		|	**if** (se, se, se)
body	::=	**var** *vars* **let** $\overrightarrow{eq}$ **tel**		|	**merge** (x, se, se)
args, vars,	*rets,*	*svars*		...	(other expressions)
	::=	$\overrightarrow{vd}$	*se*	::=	x
vd	::=	(ID, τ)		|	c (constant)
eq	::=	$(s, clock)$		|	$uop\ se$
s	::=	$x = e$		|	$se\ bop\ se$
	|	$\overrightarrow{x} = op\ (\overrightarrow{se})$ **inst** *inst_id*		...	(other simple expressions)
	|	$x =$ **fby** (se, se) **reg** x_s	*clock*	::=	$\overrightarrow{ck}$
	|	$x =$ **fbyn** (se, n, se) **reg** (x_s, x_h)	*ck*	::=	x | **not** x
	|	$x =$ **arrow** (se, se) $(se \rightarrow se)$	*op, inst_id*	::=	*ID*
	|	**skip**	x, x_s, x_f		(variables or things with L-value)
	...	(other statements)	x_h		(struct types for fbyn)

图 5 LustreS 抽象语法 (节选)

从 Lustre* AST 到 LustreS 的预处理工作有很多, 与时态算子编译直接相关的工作包括如下.

(1) 时态子表达式抽取. 变换后, 任何时态表达式不可能是其他表达式的子表达式. 换言之, 任何时态运算只能是独自出现在某个语句 (等式) 的右端, 如图 5 中的 arrow (->), *fby* (二元 *fby*) 和 *fbyn* (三元 *fby*).

(2) 为时态运算生成特殊标识符. 由于需要获取历史值, 需要为时态算子 (arrow 除外) 注册 (自动产生) 一个状态变量. 对于三元 *fby*, 需要缓存多个历史值, 因而状态变量被声明为特定的结构体类型, 参见图 3 中的示例. 该结构体类型的名称也需要注册 (自动产生). 如图 5, 为每个二元 *fby* 运算注册一个状态标识符 x_s, 为每个三元 *fby* 运算注册状态标识符 x_s 及其类型标识符 x_h. 因为这些状态变量需要静态存储分配, 故需要为每个节点调用注册一个实例标识符. 在图 5 的 LustreS 抽象语法中, 所注册的变量或标识符对应有保留字 reg 和 inst. 我们为每个节点绑定一个 *svars*, 用来收集该节点中的所有 x_s. 此外, 所注册的结构体类型标识符 x_h 属于全局作用域.

(3) 将->和 *pre* 两个算子的组合转换为二元 *fby* 算子. 如前所述, 遇到 *a* -> *pre*(*b*) 时, 将其变换为 *a fby b*.

相比图 4, 图 5 中的每个 LustreS 语句 (等式) 均对应一个时钟 (我们称之为等式时钟). 一个语句 (等式), 如 $x = e$, x 和 e 应该拥有相同的时钟, 即为该语句 (等式) 的等式时钟. 每个表达式的时钟, 可以通过时钟演算规则获得.

图 6 中节选了部分时钟演算规则. 这里, $\mathcal{H}$ 为对应于节点的局部时钟环境. 设节点的局部基本时钟为 *base*, 则时钟表达式可归纳定义为:

ck ::= *base* | *ck* on *ck_id*

ck_id ::= *id* | not *id*

$$\frac{\mathcal{H}\vdash e_1 : ck \qquad \mathcal{H}\vdash e_2 : ck}{\mathcal{H}\vdash e_1 \ \mathbf{fby}\ e_2 : ck} \qquad \frac{\mathcal{H}\vdash e_1 : ck \qquad \mathcal{H}\vdash e_2 : ck}{\mathcal{H}\vdash \mathbf{fby}\ (e_1, n, e_2) : ck}$$

$$\frac{\mathcal{H}\vdash e : ck \qquad \mathcal{H}\vdash x : ck}{\mathcal{H}\vdash e\ \mathbf{when}\ x : ck\ on\ x} \qquad \frac{\mathcal{H}\vdash e : ck \qquad \mathcal{H}\vdash x : ck}{\mathcal{H}\vdash e\ \mathbf{when}\ \mathbf{not}\ x : ck\ on\ not\ x}$$

$$\frac{\mathcal{H}\vdash x : ck \qquad \mathcal{H}\vdash e_1 : ck\ on\ x \qquad \mathcal{H}\vdash e_2 : ck\ on\ not\ x}{\mathcal{H}\vdash \mathbf{merge}\ (x, e_1, e_2) : ck} \qquad \frac{\mathcal{H}\vdash e_1 : ck \qquad \mathcal{H}\vdash e_2 : ck}{\mathcal{H}\vdash e_1\ \ bop\ e_2 : ck}$$

图 6　时钟演算规则 (节选)

从变量声明附带的取样信息 (以保留字 when 开头) 可以初始化时钟环境 $\mathcal{H}$, 例如, 图 1 中节点 tracker_cdiff 中的变量 *td*、*c*、*t* 声明时并未附带取样信息, 因而均拥有局部基本时钟为 *base*, 即有 $\mathcal{H}$ *td*: *base*, $\mathcal{H}\vdash$ *c*: *base*, 以及 $\mathcal{H}\vdash$ *t*: *base*; 而 *pt* 声明时附带取样信息 (布尔量 *c*), 因而有 $\mathcal{H}\vdash$ *pt*: *base* on *c*. 进一步, 基于时钟演算规则可以推断出: $\mathcal{H}\vdash$ merge *c* (*t* when *c* – *pt*) (0 when not *c*): *base*, 以及 $\mathcal{H}\vdash$ *fby*(*t* when *c*, *k*, 0): *base* on *c*.

不难推断, 图 1 中节点 tracker_cdiff 中第 5 和 6 行等式两端的变量或表达式均拥有相同的时钟. 因此, 第 5 行等式的等式时钟为 *base*, 而第 6 行等式的等式时钟为 *base* on *c*.

因为每个 LustreS 语句 (等式) 均对应一个等式时钟, 实际上就相当于消去了 when 算子. 例如, 图 1 中第 5 行的等式 *td* = merge *c* (*t* when *c* – *pt*) (0 when not *c*), 对应于 LustreS 中的等式 (*td* = merge (*c*, *t*–*pt*, 0), *base*).

对于图 1 中第 6 行的等式 *pt* = *fby*(*t* when *c*, *k*, 0), 由于等式右端是三元 *fby* 运算, 因而为之注册了状态变量标识符及其结构体类型标识符, 分别为 *acg_fby*1 和 *acg_s*1. 这样, 该等式 *pt* = *fby*(*t* when *c*, *k*, 0) 对应的 LustreS 等式为 (*pt* = *fbyn*(*t*, 3, 0) reg (*acg_fby*1, *acg_s*1), *base* on *c*).

考虑到基于程序依赖关系进行的拓扑排序 (本文不展开讨论), 图 1 中第 5–8 行的等式对应到 LustreS, 可以是图 7 中的顺序 LustreS 语句 (等式) 序列.

(1) (*c* = *limit* > 0, *base*)
(2) (*pt* = *fbyn*(*t*, 3, 0) reg (*acg_fby*1, *acg_s*1), *base* on *c*)
(3) ((*p*, *t*) = *tracker*(*acc*, *limit*), *base*)
(4) (*td* = merge (*c*, *t*–*pt*, 0), *base*)

图 7　图 1 第 5–8 行的等式翻译至 LustreS 所对应的语句 (等式) 序列

2.3　第 1 阶段编译的核心工作

参见图 2, 从 LustreS 经过翻译过程 LustreRGen, 得到另一个关键的中间表示 LustreR. 这一过程有不少核心变换, 其中包括: 时钟消去、 merge 和高阶算子的翻译, 以及时态算子编译的一个重要步骤 (第 1 阶段编译).

图 8 所示是节选的 LustreR 中间语言的抽象语法.

node	::=	(*fd*, *flags*, *svars*, $\overrightarrow{eqt}$)
fd	::=	*op* (*args*) **returns** (*rets*) *body*
body	::=	**var** *vars* **let** *s* **tel**
args, *vars*,		*rets*, *flags*, *svars*
	::=	$\overrightarrow{vd}$
vd	::=	(*ID*, τ)
se	::=	*x*
	|	*c* (constant)
	|	*uop se*
	|	*se bop se*
	...	(other simple expressions)
eqt	::=	(*sa* | *si*, $\overrightarrow{(x, bool)}$)

sa, *si*	::=	$x = se$
s	::=	$x = se$
	|	$\overrightarrow{x} = op\ (\overrightarrow{se})$ **inst** *inst_id*
	|	**ifs** (*se*, *s*, *s*)
	|	$x =$ **fby** (*se*, *se*) **reg** (x_s, x_f)
	|	$x =$ **fbyn** (*se*, *n*, *se*) **reg** (x_s, x_h, x_f)
	|	$x =$ **arrow** (*se*, *se*) **reg** x_f
	|	**seq** (*s*, *s*)
	|	**skip**
	...	(other statements)
op, *inst_id*	::=	*ID*
x, x_s, x_f		(variables or things with L-value)
x_h		(struct types for fbyn)

图 8　LustreR 抽象语法 (节选)

对于时态算子的编译, 第 1 阶段的核心工作是将时态算子的语义划分为计算阶段和尾声阶段, 具体包括如下.

(1) 为每个时态算子注册 (自动产生) 一个标志 (flag) 变量, 负责告知当前周期是否是所对应状态变量的第 1 个激活周期. 以标志变量为基础, 可进行计算阶段代码的设计.

(2) 生成特别的时钟卫士 (clock-guarded) 语句集合, 为进一步产生尾声阶段作好准备.

每个 LustreS 语句 (等式) 所附带的等式时钟将被消除, 代之以该时钟作为对应条件的 LustreS 语句 ifs. 例如, 图 7 中的第 2 个语句 (等式) 所附带的等式时钟为 *base* on *c*, 因此将其变换为 LustreR 中的语句 ifs (*c*, *pt* = *fbyn*(*t*, 3, 0) reg (*acg_fby*1, *acg_s*1), skip). 对于图 7 中另外 3 个语句 (等式), 由于所附带的等式时钟为 *base*, 因而不需要生成以附带时钟作为对应条件的 ifs 语句.

类似地, LustreS 中的 merge 表达式将被转换为 LustreR 中的 ifs 语句, 这里不赘述.

由于每个含时态表达式的 LustreS 语句 (等式) 可能有各自不同的附带等式时钟, 因而状态变量的第 1 个激活周期不尽相同, 需要拥有各自的标志 (flag) 变量, 参见图 6 中的 x_f. 这样, 图 7 中的第 2 个语句 (等式) 将最终转换为 LustreR 中的语句 ifs (*c*, *pt* = *fbyn*(*t*, 3, 0) reg (*acg_fby*1, *acg_s*1, *acg_cinit*), skip). 这里, *acg_cinit* 是自动产生的标志 (flag) 变量名称, 其中的第 2 个 *c* 直接拷贝自时钟 *base* on *c* 的名字串"*c*". 相比较, 如果时钟是 *base* on not *c*, 则自动产生的标志变量名称为 *acg_cinit*1; 若时钟是 *base*, 则标志变量名称使用 *acg_init* 即可. 不失一般性, 图 3 中使用了 *init* 来替代 *acg_init*. 总之, 我们采用了一种特殊的标志变量名称自动产生算法, 可以确保不同时钟所对应的标志变量名称互不相同.

图 8 中, 各个节点中所有 x_f 的集合用 *flags* 表示.

进一步, 为简化尾声阶段的生成, 每一节点均维护一个时钟卫士语句集合. 参见图 8 中的 *eqt*, 时钟卫士语句分两大类: 第 1 类 (*sa* 语句) 对所有状态变量 x_s 和标志变量 x_f 进行直接更新. 前者为下一激活周期准备数据, 后者是在对应的第 1 个激活周期复位标志变量 x_f (即置为 false). 第 2 类 (*si* 语句) 用于更新状态变量 x_s 的下标域. 时钟卫士是 (*id*, true) 或者 (*id*, false) 的序列, 对应于某个等式时钟. 参见图 3 例子中最末 3 行的语句, 其中前两行语句用来更新 x_s, 一条是 *sa* 语句 $x_s.items[x_s.idx] = e_1$, 另一条是 *si* 语句 $x_s.idx = (x_s.idx + 1)\ \%\ 3$; 最后一行语句用于复位标志变量 *init*, 对应一条 *sa* 语句 *init* = false. 对于二元 *fby* 算子, 仅需要一条 *sa* 语句来更新 x_s. 如果时钟是 *base*, 则不需要生成时钟卫士. 例如, 图 3 中最末 3 行语句不含时钟卫士条件. 然而, 如果是图 1 第 6 行的三元 *fby* 运算, 则需要生成时钟卫士 (参见第 2.4 节).

对于 arrow, 由于仅注册有标志变量 x_f, 没有状态变量 x_s, 因而仅对应有 *sa* 类别的时钟卫士语句.

2.4 第 2 阶段编译的核心工作

时态算子编译的第 2 阶段主要完成时态算子的最终消去. 核心工作主要在翻译过程 LustreFGen 完成. 参见图 2, LustreFGen 之前还有一些其他较为简单的翻译工作, 与本文关系不太密切, 这里忽略.

图 9 所示是节选的 LustreF 中间语言的抽象语法. 可见, LustreF 语法中已不含显式的时态算子.

node	::=	(*fd*, *flags*, *svars*)	*s*	::=	$x = se$
fd	::=	*op* (*args*) **returns** (*rets*) *body*		|	$\overrightarrow{x} = op\ (\overrightarrow{se})$ **inst** *inst_id*
body	::=	**var** *vars* **let** *s* **tel**		|	**ifs** (*se*, *s*, *s*)
args, *vars*,	*rets*,	*flags*, *svars*		|	**for** (*init*, *se*, *eqf*, *s*)
	::=	$\overrightarrow{vd}$		|	**seq** (*s*, *s*)
vd	::=	(*ID*, τ)		|	**skip**
τ		(data type)		...	(other statements)
op, *inst_id*	::=	*ID*	*init*	::=	*eqf* | ε
se		(simple expressions as before)	*eqf*	::=	$x = se$
			x		(variables or things with L-value)

图 9 LustreF 抽象语法 (节选)

翻译过程 LustreFGen 中的一些核心翻译函数如图 10 所示. 多数函数的含义比较直接, 可特别关注时态算子编译的计算阶段和尾声阶段. 计算阶段的 LustreF 代码可由翻译函数 *trans_stmt* 应用于 3 个 LustreR 时态运算语句 (分别针对 *fby*、*fbyn* 和 arrow 算子) 来生成, 而尾声阶段的 LustreF 代码可由函数 *trans_node* 内部调用函数 *trans_pegs* 时返回并置于本节点代码的尾部.

$$
\begin{array}{rcl}
trans\_expr\ (c) & \equiv \triangleright & c \quad \text{(constant)} \\
trans\_expr\ (x) & \equiv \triangleright & x \quad \text{(variable)} \\
trans\_expr\ (se_1\ bop\ se_2) & \equiv \triangleright & trans\_expr\ (se_1)\ bop\ trans\_expr\ (se_2) \\
trans\_stmt\ (\overrightarrow{x} = op\ (se_1, \cdots, se_m)\ \mathbf{inst}\ inst\_id) & \equiv \triangleright & \overrightarrow{x} = op\ (trans\_expr(se_1), \cdots, trans\_expr(se_m))\ \mathbf{inst}\ inst\_id \\
trans\_stmt\ (\mathbf{ifs}\ (se, s_1, s_2)) & \equiv \triangleright & \mathbf{ifs}\ (trans\_expr(se), trans\_stmt(s_1), trans\_stmt(s_2)) \\
trans\_stmt\ (x = \mathbf{fby}\ (se_1, se_2)\ \mathbf{reg}\ (x_s, x_f)) & \equiv \triangleright & \mathbf{ifs}\ (x_f, x = trans\_expr(se_2), x = x_s) \\
trans\_stmt\ (x = \mathbf{fbyn}\ (se_1, n, se_2)\ \mathbf{reg}\ (x_s, x_h, x_f)) & \equiv \triangleright & \mathbf{seq}\ (\mathbf{ifs}\ (x_f, \mathbf{seq}\ (\ \mathbf{for}\ (\ acg\_j = 0, acg\_j \lrcorner n, acg\_j = acg\_j + 1, \\
 & & \qquad x_s.items[acg\_j] = trans\_expr\ (se_2)), \\
 & & \qquad x_s.\text{idx} = 0), \mathbf{skip}), \\
 & & \quad x = x_s.\text{items}[x_s.\text{idx}]\ ) \\
trans\_stmt\ (x = \mathbf{arrow}\ (se_1, se_2)\ \mathbf{reg}\ x_f\ ) & \equiv \triangleright & \mathbf{ifs}\ (x_f, x = trans\_expr(se_1), x = trans\_expr(se_2)) \\
trans\_stmt\ (\mathbf{seq}\ (s_1, s_2)) & \equiv \triangleright & \mathbf{seq}\ (trans\_stmt(s_1), trans\_stmt(s_2)) \\
trans\_stmt\ (\mathbf{skip}) & \equiv \triangleright & \mathbf{skip} \\
trans\_node\ (\ (op\ (args)\ \mathbf{returns}\ (rets) & \equiv \triangleright & (op\ (args)\ \mathbf{returns}\ (rets)\ \mathbf{var}\ \ vars + + mkloopid\ (\overrightarrow{eqt}) \\
\quad \mathbf{var}\ \ vars\ \ \mathbf{let}\ \ s\ \ \mathbf{tel}, & & \qquad \mathbf{let}\ \ \mathbf{seq}\ (\ trans\_stmt(s), trans\_peqs\ (\overrightarrow{eqt})\ )\ \ \mathbf{tel}, \\
\quad flags, svars, \overrightarrow{eqt}\ )\ ) & & \quad flags, svars\ )\ )
\end{array}
$$

图 10　LustreFGen 的核心翻译函数 (节选)

函数 *trans_pegs* 可定义为如下:

$$
\begin{array}{l}
trans\_peqs\left(\overrightarrow{eqt}\right) : stmt := \\
\quad \text{match } \overrightarrow{eqt} \text{ with} \\
\quad\quad \mid nil \Rightarrow \text{return skip} \\
\quad\quad \mid (sa, conds)\ \overrightarrow{eqt'} \Rightarrow \text{return seq } \left(\text{if } (conds)\ \{sa\},\ trans\_peqs\left(\overrightarrow{eqt'}\right)\right) \\
\quad\quad \mid (si, conds)\ \overrightarrow{eqt'} \Rightarrow \text{return seq } \left(\text{if } (conds)\ \{si\},\ trans\_peqs\left(\overrightarrow{eqt'}\right)\right)
\end{array}
$$

这里, 为简洁明了, 在 *trans_pegs* 的翻译结果中, 我们没有使用 LustreF 语法中的 ifs 语句来替换 if.

另外, 函数 *trans_node* 中还有一个简单的函数 *mkloopid* 返回一个共享的局部循环变量, 用来生成针对三元 *fby* (*fbyn*) 状态变量 (x_s) 进行初始化的代码.

考虑图 7 中的第 2 条 LustreS 语句 ($pt = fbyn(t, 3, 0)$ reg (acg_fby1, acg_s1), *base* on *c*), 经由翻译过程 LustreRGen 的第 1 阶段时态算子编译, 生成 LustreR 语句 ifs (*c*, $pt = fbyn(t, 3, 0)$ reg (acg_fby1, acg_s1, acg_cinit), skip) 作为 LustreR 层面的计算阶段代码, 以及依次产生时钟卫士语句集合: *sa* 语句 $acg\_fby1.items[acg\_fby1.idx] = t$, *si* 语句 $acg\_fby1.idx = (acg\_fby1.idx + 1)\ \%\ 3$, 以及 *sa* 语句 acg_cinit = false, 且这 3 条语句的时钟卫士均为 (*c*, true)::*nil*.

这一 LustreR 语句经函数 *trans_stmt* 翻译生成 LustreF 层面的计算阶段代码:

seq (ifs (acg_cinit, seq (for ($acg\_j = 0$, $acg\_j < 3$, $acg\_j = acg\_j + 1$, $acg\_fby1.items[acg\_j] = 0$),
$acg\_fby1.idx = 0$), skip), $pt = acg\_fby1.items[acg\_fby1.idx]$)

同时, 经函数 *trans_pegs* 翻译生成 LustreF 层面的尾声阶段代码为:

seq (if (*c*) {$acg\_fby1.items[acg\_fby1.idx] = t$},
seq (if (*c*) {$acg\_fby1.idx = (acg\_fby1.idx + 1)\ \%\ 3$},
seq (if (*c*) {acg_cinit = false}, skip)))

2.5　时态算子编译后续的翻译步骤

LustreF 层的代码已经很接近于 C 代码了, 后续的翻译过程逐步向 Clight 靠近, 但仍有与时态算子编译关联较紧密的步骤, 如输出参数结构体的生成, 构造 reset 函数等. 这些步骤中用到针对时态算子注册的一些特殊变量/标识符, 如之前讨论的状态变量 x_s, 结构体类型标识符 x_h 和标志变量 x_f 等.

比如, 所构造的 reset 函数, 需要将与当前节点关联的全部标志变量 x_f 复位, 即初始化为 true, 同时需要调用节点内所有其他节点调用实例的 reset 函数.

3　时态算子编译的正确性

如第 2.1 节所述, 在我们的可信编译器设计中, 时态算子的编译包含两个核心阶段. 参见图 2, 第 1 个核心阶段隶属于从 LustreS 到 LustreR 的翻译过程 LustreRGen 中, 而第 2 个核心阶段隶属于从 LustreR 到 LustreF 的翻译过程 LustreFGen 中. 因此, 对翻译过程 LustreRGen 和 LustreFGen 进行验证, 蕴含了对时态算子编译的两个核心阶段

进行翻译正确性验证.

3.1 翻译过程 LustreRGen (含时态算子第 1 阶段编译) 正确性验证

3.1.1 中间语言 LustreS 语义

如下给出定义 LustreS 操作语义的部分语义规则 (覆盖 3 个主要的时态算子), 可用于 LustreS 节点实例于单个时钟周期的语义计算.

$$\frac{}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},nil>\mapsto<\mathcal{T},\mathcal{L}>} \quad \text{(规则 S1)}$$

$$\frac{\mathcal{G},\mathcal{L}\vdash clock\Downarrow \text{true}\quad \mathcal{G}\vdash<\mathcal{T},\mathcal{L},(s,clock)>\mapsto<\mathcal{T}',\mathcal{L}'>\quad \mathcal{G}\vdash<\mathcal{T}',\mathcal{L}',ss>\mapsto<\mathcal{T}'',\mathcal{L}''>}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(s,clock)::ss>\mapsto<\mathcal{T}'',\mathcal{L}''>} \quad \text{(规则 S2)}$$

$$\frac{\mathcal{G},\mathcal{L}\vdash clock\Downarrow \text{false}\quad \mathcal{G}\vdash<\mathcal{T},\mathcal{L},ss>\mapsto<\mathcal{T}',\mathcal{L}'>}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(s,clock)::ss>\mapsto<\mathcal{T}',\mathcal{L}'>} \quad \text{(规则 S3)}$$

$$\frac{\mathcal{G},\mathcal{L}\vdash se\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},x=se>\mapsto<\mathcal{T},\mathcal{L}'>} \quad \text{(规则 S4)}$$

$$\frac{\mathcal{G},\mathcal{T}\vdash flagid(clock)\Downarrow \text{true}\quad \mathcal{G},\mathcal{L}\vdash se_2\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}\quad \mathcal{T}'=\mathcal{T}\|_{x_s\leftarrow v,\overrightarrow{eqt}\leftarrow\overrightarrow{eqt}::(x_s=se_1,clock)::(flagid(clock)=\text{false},clock)}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x=fby(se_1,se_2))\ \text{reg}\ x_s,clock)>\mapsto<\mathcal{T}',\mathcal{L}'>} \quad \text{(规则 S5)}$$

$$\frac{\mathcal{G},\mathcal{T}\vdash flagid(clock)\Downarrow \text{false}\quad \mathcal{G},\mathcal{T}\vdash x_s\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x=fby(se_1,se_2))\ \text{reg}\ x_s,clock)>\mapsto<\mathcal{T},\mathcal{L}'>} \quad \text{(规则 S6)}$$

$$\frac{\begin{array}{c}\tau=type(se_1)\quad type(x_s)=x_h\quad \text{typedef struct}\{idx:\text{int};\ items:\text{array}[n]\ of\ \tau;\}\ x_h\\ \mathcal{G},\mathcal{T}\vdash flagid(clock)\Downarrow \text{true}\quad \mathcal{G},\mathcal{L}\vdash se_2\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}\quad \mathcal{T}'=\mathcal{T}\|_{x_s.items[i]\leftarrow v\ (i=0,1,\ldots,n-1),\ x_s.idx\Leftarrow 0}\\ \mathcal{T}''=\mathcal{T}'\|_{\overrightarrow{eqt}\leftarrow\overrightarrow{eqt}::(x_s.items[x_s.idx]=se_1,clock)::(flagid(clock)=\text{false},clock)::(x_s.items.idx=(x_s.idx+1)\%n,clock)}\end{array}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x=fbyn(se_1,n,se_2)\ \text{reg}\ (x_s,x_h,x_f),clock)>\mapsto<\mathcal{T}'',\mathcal{L}'>} \quad \text{(规则 S7)}$$

$$\frac{\begin{array}{c}\tau=type(se_1)\quad type(x_s)=x_h\quad \text{typedef struct}\{idx:\text{int};\ items:\text{array}[n]\ of\ \tau;\}\ x_h\\ \mathcal{G},\mathcal{T}\vdash flagid(clock)\Downarrow \text{false}\quad \mathcal{G},\mathcal{T}\vdash x_s.items[idx]\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}\end{array}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x=fbyn(se_1,n,se_2)\ \text{reg}\ (x_s,x_h,x_f),clock)>\mapsto<\mathcal{T},\mathcal{L}'>} \quad \text{(规则 S8)}$$

$$\frac{\mathcal{G},\mathcal{T}\vdash flagid(clock)\Downarrow \text{true}\quad \mathcal{G},\mathcal{L}\vdash se_1\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x=\text{arrow}\ (se_1,se_2),clock)>\mapsto<\mathcal{T},\mathcal{L}'>} \quad \text{(规则 S9)}$$

$$\frac{\mathcal{G},\mathcal{T}\vdash flagid(clock)\Downarrow \text{false}\quad \mathcal{G},\mathcal{L}\vdash se_2\Downarrow v\quad \mathcal{L}'=\mathcal{L}\|_{x\leftarrow v}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x=\text{arrow}\ (se_1,se_2),clock)>\mapsto<\mathcal{T},\mathcal{L}'>} \quad \text{(规则 S10)}$$

$$\frac{\begin{array}{c}\mathcal{G},\mathcal{L}\vdash se_i\Downarrow v_i\ (i=1,2,\ldots,m)\quad \mathcal{G}\vdash<\mathcal{T}_{inst\_id},op(v_1,v_2,\ldots,v_m)>\mapsto<\mathcal{T}'_{inst\_id},(r_1,r_2,\ldots,r_n)>\\ \mathcal{T}'=\mathcal{T}\|_{\text{update}}\ \mathcal{T}_{inst\_id}\ \text{to}\ \mathcal{T}'_{inst\_id}\quad \mathcal{L}'=\mathcal{L}\|_{x_i\leftarrow r_i\ (i=1,2,\ldots,n)}\end{array}}{\mathcal{G}\vdash<\mathcal{T},\mathcal{L},(x_1,x_2,\ldots,x_n)=op(se_1,se_2,\ldots,se_m)\ \text{inst}\ inst\_id>\mapsto<\mathcal{T}',\mathcal{L}'>} \quad \text{(规则 S11)}$$

$$\frac{\begin{array}{c}\mathcal{P}\vdash_{\text{node}}(op(args)\ \text{returns}\ (rets)\ \text{var}\ vars\ \text{let}\ eqs\ \text{tel})\quad \mathcal{L}'=\mathcal{L}\|_{args\leftarrow\overrightarrow{in}}\\ \mathcal{G}\vdash<\mathcal{T},\mathcal{L}',eqs>\mapsto<\mathcal{T}',\mathcal{L}''>\ \mathcal{G}\vdash<\mathcal{T}',\mathcal{L}'',eqt(eqs)>\mapsto<\mathcal{T}'',\mathcal{L}''>\quad \overrightarrow{out}=\mathcal{L}''(rets)\end{array}}{\mathcal{G}\vdash<\mathcal{T},op\left(\overrightarrow{in}\right)>\mapsto<\mathcal{T}'',\overrightarrow{out}>} \quad \text{(规则 S12)}$$

由于在生成 LustreS 之前已经进行过拓扑排序, 故本文所涉及的语义定义均指串行操作语义. 其中, 我们用到语义环境 $\mathcal{G}$、$\mathcal{T}$ 和 $\mathcal{L}$, 以及如下几类断言.

- $\mathcal{G},\mathcal{T}\vdash se\Downarrow v$, $\mathcal{G},\mathcal{L}\vdash se\Downarrow v$ $se\Downarrow v$ 含义为: 在 $\vdash$ 左侧给定的环境下, 简单表达式 se 在当前周期的求值结果为 v.
- $\mathcal{G}\vdash<\mathcal{T},\mathcal{L},s>\mapsto<\mathcal{T}',\mathcal{L}'>$ 含义为: 当前周期中语句 (或等式) s 的执行, 会将环境 $\mathcal{T}$ 修改为 $\mathcal{T}'$, 以及将环境 $\mathcal{L}$ 修改为 $\mathcal{L}'$.
- $\mathcal{G}\vdash<\mathcal{T},op(\overrightarrow{in})>\mapsto<\mathcal{T}',\overrightarrow{out}>$ 含义为: 当前周期中具有输入 $\overrightarrow{in}$ 的节点 op 的实例调用将返回 $\overrightarrow{out}$, 并且环境 $\mathcal{T}$ 将更新为 $\mathcal{T}'$.

全局环境 $\mathcal{G}$ 将常量、类型和节点的标识符映射到它们的值或定义. $\mathcal{T}$ 在本文中称为时态环境, 将状态 (state) 变量 (x_s) 和标志 (flag) 变量 (x_f) 映射到它们的流式取值, 且将 $\mathcal{T}$ 中的节点实例标识符映射到为其静态分配的存储

数据状态. 局部环境 $\mathcal{L}$ 将局部变量或参数映射到当前周期动态分配给它们的存储数据值.

此外, 规则 S1–S12 中还用到几个语义函数.

flagid(*clock*) 表示为 *clock* 生成的 flag 变量, 针对每个不同时钟将返回唯一标识符.

$\mathcal{L}\|_{x\leftarrow v}$ 返回一个新的本地环境, 除了 x 所指向的内存块有更新 (基于 x 的类型对 v 的内容进行强制转换) 之外, 该环境与 $\mathcal{L}$ 相同. $\mathcal{T}\|_{\ldots\leftarrow\ldots}$ 也是类似.

规则 S11) 中包含另一个可对时态环境进行更新的函数, 因环境具有树状结构, 其操作较为复杂. 节点实例调用会将被调用者 (callee) 的时态环境 $\mathcal{T}_{inst\_id}$ 更改为新的时态环境 $\mathcal{T}'_{inst\_id}$. 因为被调用者的环境嵌套在调用者 (caller) 的环境中, 所以调用者的时态环境也会同时更新.

我们用 *eqt*(*eqs*) 返回从节点内等式列表 *eqs* 生成的一段 LustreS 代码, 这段代码将作为当前节点实例所对应的尾声阶段代码来执行, 如规则 S12 所示.

基于 LustreS 操作语义定义的节点实例单时钟周期语义, 可以给出相应的多时钟周期语义. 对于所有 $i>0$, 节点实例第 i 个激活周期的语义计算定义为:

$$\begin{cases} \mathcal{T}_0 = \mathcal{T}\|_{x_f\leftarrow \text{true for } \forall x_f\in flags} \\ \mathcal{G}\vdash op\left(\overrightarrow{in_i},\overrightarrow{out_i}\right), \quad \text{if } \mathcal{G}\vdash <\mathcal{T}_{i-1}, op\left(\overrightarrow{in_i}\right)>\mapsto<\mathcal{T}_i,\overrightarrow{out}_i> \end{cases},$$

其中, $\mathcal{G}$ 为全局环境, $\mathcal{T}$ 为时态环境, $\mathcal{T}_i$ 为其第 i 个周期 (cycle) 的分量. 系统重启或执行 reset 操作时重置 $\mathcal{T}$ 为 $\mathcal{T}_0$, 将每个有状态量的初始化标志 (x_f) 复位为“尚未初始化”, 之后在前一周期环境 ($\mathcal{T}_{i-1}$) 下执行 *op* 后得到当前周期下的环境 ($\mathcal{T}_i$). 第 i 个周期执行 *op*, 从系统获得输入参数序列, 得到输出结果序列, 同时改变环境 $\mathcal{T}$ (状态流中多了 $\mathcal{T}_i$ 分量).

3.1.2 中间语言 LustreR 语义

如下所示给出定义 LustreR 操作语义的部分语义规则 (覆盖 3 个主要的时态算子), 可用于 LustreR 节点实例在单个时钟周期的语义计算. 其中用到语义环境 $\mathcal{G}_R$、$\mathcal{T}_R$ 和 $\mathcal{L}_R$, 含义与第 3.1.1 节的 $\mathcal{G}$、$\mathcal{T}$ 和 $\mathcal{L}$ 类似, 分别是全局环境、时态环境和局部环境, 以及二者相关断言也相似.

$$\frac{\mathcal{G}_R,\mathcal{L}_R\vdash se\Downarrow v' \quad v=sem\_unary\_operation(uop,v')}{\mathcal{G}_R,\mathcal{L}_R\vdash uop\ se\Downarrow v} \tag{规则 R1}$$

$$\frac{}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,\text{skip}>\mapsto<\mathcal{T}_R,\mathcal{L}_R>} \tag{规则 R2}$$

$$\frac{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,s_1>\mapsto<\mathcal{T}'_R,\mathcal{L}'_R> \mathcal{G}_R\vdash <\mathcal{T}'_R,\mathcal{L}'_R,s_2>\mapsto<\mathcal{T}''_R,\mathcal{L}''_R>}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,\ \text{seq}\ (s_1,s_2)>\mapsto<\mathcal{T}''_R,\mathcal{L}''_R>} \tag{规则 R3}$$

$$\frac{\mathcal{G}_R,\mathcal{L}_R\vdash se\Downarrow v \quad \mathcal{L}'_R=\mathcal{L}_R\|_{x\leftarrow v}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=se>\mapsto<\mathcal{T}_R,\mathcal{L}'_R>} \tag{规则 R4}$$

$$\frac{\mathcal{G}_R,\mathcal{T}_R\vdash x_f\Downarrow \text{true} \quad \mathcal{G}_R,\mathcal{L}_R\vdash se_2\Downarrow v \quad \mathcal{L}'_R=\mathcal{L}_R\|_{x\leftarrow v}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=fby(se_1,se_2)\ \text{reg}\ (x_s,x_f)>\mapsto<\mathcal{T}_R,\mathcal{L}'_R>} \tag{规则 R5}$$

$$\frac{\mathcal{G}_R,\mathcal{T}_R\vdash x_f\Downarrow \text{false} \quad \mathcal{G}_R,\mathcal{T}_R\vdash x_s\Downarrow v \quad \mathcal{L}'_R=\mathcal{L}_R\|_{x\leftarrow v}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=fby(se_1,se_2)\ \text{reg}(x_s,x_f)>\mapsto<\mathcal{T}_R,\mathcal{L}'_R>} \tag{规则 R6}$$

$$\frac{\begin{array}{c}\tau=type(se_1) \quad type(x_s)=x_h \quad \text{typedef struct}\{idx:\text{int};\ items:\text{array}[n]of\ \tau;\}\ x_h \\ \mathcal{G}_R,\mathcal{T}_R\vdash x_f\Downarrow\text{true} \quad \mathcal{G}_R,\mathcal{L}_R\vdash se_2\Downarrow v \quad \mathcal{T}'_R=\mathcal{T}_R\left\|_{x_s.items[i]\leftarrow v\ (i=0,1,\ldots,n-1),\ x_s.idx\Leftarrow 0}\right. \quad \mathcal{L}'_R=\mathcal{L}_R\left\|_{x\leftarrow v}\right.\end{array}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=fbyn(se_1,n,se_2)\ \text{reg}\ (x_s,x_h,x_f)>\mapsto<\mathcal{T}'_R,\mathcal{L}'_R>} \tag{规则 R7}$$

$$\frac{\begin{array}{c}\tau=type(se_1) \quad type(x_s)=x_h \quad \text{typedef struct}\{idx:\text{int};\ items:\text{array}[n]\ \text{of}\ \tau;\}\ x_h \\ \mathcal{G}_R,\mathcal{T}_R\vdash x_f\Downarrow\text{false} \quad \mathcal{G}_R,\mathcal{T}_R\vdash x_s.items[idx]\Downarrow v \quad \mathcal{L}'_R=\mathcal{L}_R\|_{x\leftarrow v}\end{array}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=fbyn(\text{se}_1,n,\text{se}_2)\ \text{reg}\ (x_s,x_h,x_f)>\mapsto<\mathcal{T}_R,\mathcal{L}'_R>} \tag{规则 R8}$$

$$\frac{\mathcal{G}_R,\mathcal{T}_R\vdash x_f\Downarrow \text{true} \quad \mathcal{G}_R,\mathcal{L}_R\vdash se_1\Downarrow v \quad \mathcal{L}'_R=\mathcal{L}_R\|_{x\leftarrow v}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=\text{arrow}\ (se_1,se_2)\ \text{reg}\ x_f>\mapsto<\mathcal{T}_R,\mathcal{L}'_R>} \tag{规则 R9}$$

$$\frac{\mathcal{G}_R,\mathcal{T}_R\vdash x_f\Downarrow \text{false}\quad \mathcal{G}_R,\mathcal{L}_R\vdash se_2\Downarrow v\quad \mathcal{L}'_R=\mathcal{L}_R\|_{x\leftarrow v}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,x=\text{arrow }(se_1,se_2)\text{ reg }x_f>\mapsto<\mathcal{T}_R,\mathcal{L}'_R>}\quad \text{(规则 R10)}$$

$$\frac{\begin{array}{c}\mathcal{G}_R,\mathcal{L}_R\vdash se_i\Downarrow v_i\ (i=1,2,\ldots,m)\quad \mathcal{G}_R\vdash <\mathcal{T}_{R_{inst\_id}},op(v_1,v_2,\ldots,v_m)>\mapsto<\mathcal{T}'_{R_{inst\_id}},(r_1,r_2,\ldots,r_n)>\\ \mathcal{T}'_R=\mathcal{T}_R\|_{\text{update}}\ \mathcal{T}_{R_{inst\_id}}\text{ to }\mathcal{T}'_{R_{inst\_id}}\quad \mathcal{L}'_R=\mathcal{L}_R\|_{x_i\leftarrow r_i\ (i=1,2,\ldots,n)}\end{array}}{\mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}_R,(x_1,x_2,\ldots,x_n)=op(se_1,se_2,\ldots,se_m)\text{ inst }inst\_id>\mapsto<\mathcal{T}'_R,\mathcal{L}'_R>}\quad \text{(规则 R11)}$$

$$\frac{\begin{array}{c}\mathcal{P}\vdash_{\text{node}}(op(args)\text{ returns }(rets)\text{ var }vars\text{ let }s\text{ tel},flags,svars,\overrightarrow{eqt})\ \ \mathcal{L}'_R=\mathcal{L}_R\|_{args\leftarrow\overrightarrow{in}}\\ \mathcal{G}_R\vdash <\mathcal{T}_R,\mathcal{L}'_R,s>\mapsto<\mathcal{T}'_R,\mathcal{L}''_R>\quad \mathcal{G}_R\vdash <\mathcal{T}'_R,\mathcal{L}''_R,estmt\left(\overrightarrow{eqt}\right)>\mapsto<\mathcal{T}''_R,\mathcal{L}''_R>\quad \overrightarrow{out}=\mathcal{L}''_R(rets)\end{array}}{\mathcal{G}_R\vdash <\mathcal{T}_R,op\left(\overrightarrow{in}\right)>\mapsto<\mathcal{T}''_R,\overrightarrow{out}>}\quad \text{(规则 R12)}$$

LustreR 的语义规则中, 也用到一些语义函数, 多数同 LustreS 中的一致. 规则 R12 中的语义函数 *estmt*, 类似于第 2.4 节 LustreFGen 核心翻译函数中使用的 *trans_pegs* 函数.

值得注意的是, 规则 R5–R8 中 *fby* 或 *fbyn* 的第 1 个操作数 se_1 没有用于语义定义, 说明与源语言中的 *fby* 或 *fbyn* 已经拥有不一致的语义.

在规则 R1 中, 语义函数 *sem_unary_operation* 是指通常的一元运算.

同样, 基于 LustreR 操作语义定义的节点实例单时钟周期语义, 可以给出相应的多时钟周期语义. 对于所有 $i>0$, 节点实例第 i 个激活周期的语义计算定义为:

$$\begin{cases}\mathcal{T}_{R_0}=\mathcal{T}_R\|_{x_f\leftarrow\text{true for }\forall x_f\in flags}\\ \mathcal{G}_R\vdash op\left(\overrightarrow{in}_i,\overrightarrow{out}_i\right),\quad \text{if }\mathcal{G}_R\vdash <\mathcal{T}_{R_{i-1}},op\left(\overrightarrow{in}_i\right)>\mapsto<\mathcal{T}_{R_i},\overrightarrow{out}_i>\end{cases}.$$

这里, 每当系统重启或执行 reset 操作时, 就会重新设置 $\mathcal{T}_{R_0}$.

3.1.3 从 LustreS 到 LustreR 的翻译正确性

设 LustreRGen 中的翻译函数为 *tranS2R*, *tranS2R* 中的核心翻译函数是 *trans_node*.

引理 1. LustreRGen 的语义保持性 (一般情形). 对于每个满足 $tranS2R(\mathcal{P})=OK(\mathcal{P}_{\mathcal{R}})$、$f\in\mathcal{P}$、$f'\in\mathcal{P}_{\mathcal{R}}$ 和 $trans\_node(f)=f'$ 的良构的 LustreS 程序 $\mathcal{P}$, 良构的 LustreR 程序 $\mathcal{P}_{\mathcal{R}}$, 节点 f 和 f', 我们有:

$$\forall x,\vec{y},\vec{x'},\vec{y'}.\left(\mathcal{G}\vdash f(\vec{x_i}\vec{y_i})\wedge\mathcal{G}_R\vdash f'\left(\vec{x'_i},\vec{y'_i}\right)\wedge\vec{x_i}=\vec{x'_i}\Rightarrow\vec{y_i}\simeq\vec{y'_i}\right)_{i=1}^{\infty},$$

其中, $\vec{y_i}\simeq\vec{y'_i}$ 是一种通用的表示, 因为完整编译过程中有若干翻译会新增输出参数信息以及改变输出参数格式 (如利用结构体包装输出参数信息), 但翻译前的输出参数值 ($\vec{y_i}$) 总是在翻译后的输出参数值 ($\vec{y'}$) 中可以找到对应的表示, $\vec{y_i}\simeq\vec{y'_i}$ 表示这二者是求值等价的. 而对于输入参数而言, 翻译前后的信息可以直接对应, 故我们用 $\vec{x_i}=\vec{x'_i}$ 表示对应输入参数值之间的相等. 其实, 就本文所涉及的两个翻译过程 (LustreRGen 和 LustreFGen) 而言, 将 $\vec{y_i}\simeq\vec{y'_i}$ 改为 $\vec{y_i}=\vec{y'_i}$ 也是可以的.

这一引理的 Coq 证明中, 我们将 $\mathcal{G}\vdash op(\vec{x_i},\vec{y_i})$ $(i=1,2,\ldots,\infty)$ 扩展为 $\mathcal{G}\vdash_n^{maxn} op(\vec{x},\vec{y})$, 其中 $\vec{x}$ 和 $\vec{y}$ 分别代表 $\vec{x_i}$ $(i=1,2,\ldots,\infty)$ 和 $\vec{y_i}$ $(i=1,2,\ldots,\infty)$. $\mathcal{G}\vdash_n^{maxn} op(\vec{x},\vec{y})$ 定义为:

$$\begin{cases}\text{true}, & \text{if } n>maxn\\ \mathcal{G}\vdash_{n+1}^{maxn} op(tl(\vec{x}),tl(\vec{y}))\wedge\mathcal{G}\vdash <\mathcal{T}_{n-1},op(hd(\vec{x}))>\mapsto<\mathcal{T}_n,hd(\vec{y})>, & \text{if } n\leqslant maxn\end{cases}.$$

这样, 该引理中的公式在 Coq 证明中被替代为证明:

$$\forall n\forall maxn\left(\mathcal{G}\vdash_n^{maxn} op(\vec{x},\vec{y})\wedge\mathcal{G}_R\vdash_n^{maxn} op'\left(\vec{x'},\vec{y'}\right)\wedge hd(\vec{x})=hd\left(\vec{x'}\right)\ arrow\ hd(\vec{y})\simeq hd\left(\vec{y'}\right)\right).$$

基于 LustreRGen 中翻译函数的定义 (本文未给出形式定义, 其非形式解释参见第 2.3 节), LustreS 和 LustreR 的语义定义 (参见第 3.1.1 和 3.1.2 节), 逐步分解为子目标并逐一实施 (互) 归纳证明, 最终完成该引理的证明.

也可以选择采用 CoInductive 描述上述引理并给出相应的证明, 看起来会与原始同步数据流语义更加切近, 但一直以来这一可选工作并非本文项目所急需的.

从引理 1 出发, 可以推论出从 LustreS 到 LustreR 的翻译正确性 (语义保持性).

推论 1. LustreRGen 的语义保持性. 对于每个满足 $tranS2R(\mathcal{P})=OK(\mathcal{P}_{\mathcal{R}})$, f_{main} 是 $\mathcal{P}$ 的主节点, f'_{main} 是 $\mathcal{P}_{\mathcal{R}}$ 的主节点, 和 $trans\_node(f_{\text{main}})=f'_{\text{main}}$ 的良构的 LustreS 程序 $\mathcal{P}$ 以及良构的 LustreR 程序 $\mathcal{P}_{\mathcal{R}}$, 我们有:

$$\forall \vec{x},\vec{y},\vec{x},\vec{y}'.\left(\mathcal{G}\vdash f_{\text{main}}(\vec{x}_i,\vec{y}_i)\wedge \mathcal{G}_R\vdash f'_{\text{main}}\left(\vec{x}'_i,\vec{y}'_i\right)\wedge \vec{x}_i=\vec{x}'_i\Rightarrow \vec{y}_i\simeq \vec{y}'_i=\right)_{i=1}^{\infty}.$$

其他翻译过程的正确性证明与此相似, 比如第 3.2.2 节从 LustreR 到 LustreF 的翻译正确性. 在各阶段翻译过程的正确性证明的基础上, 可以得出完整编译后端正确性的证明.

3.2 翻译过程 LustreFGen (含时态算子第 2 阶段编译) 正确性验证

3.2.1 中间语言 LustreF 语义

如下所示给出定义 LustreF 操作语义的部分语义规则, 可用于 LustreF 节点实例在单个时钟周期的语义计算. 我们仍然使用 3 个语义环境, 全局环境 $\mathcal{G}_F$、时态环境 $\mathcal{T}_F$ 和局部环境 $\mathcal{L}_E$, 含义与第 3.1.2 节的 $\mathcal{G}_R$、$\mathcal{T}_R$ 和 $\mathcal{L}_R$ 类似, 相关断言也相似.

$$\frac{\mathcal{G}_F,\mathcal{T}_F,\mathcal{L}_F\vdash se_1\Downarrow v_1\quad \mathcal{G}_F,\mathcal{T}_F,\mathcal{L}_F\vdash se_2\Downarrow v_2\quad v=sem\_binary\_operation(bop,v_1,v_2)}{\mathcal{G}_F,\mathcal{T}_F,\mathcal{L}_F\vdash se_1\ bop\ se_2\Downarrow v} \tag{规则 F1}$$

$$\frac{\mathcal{G}_F,\mathcal{T}_F,\mathcal{L}_F\vdash se\Downarrow v\quad \mathcal{L}'_F=\mathcal{L}_F\left\|_{x\leftarrow v\ (x\in dom(\mathcal{L}_F))}\quad \mathcal{T}'_F=\mathcal{T}_F\right\|_{x\leftarrow v\ (x\in dom(\mathcal{T}_F))}}{\mathcal{G}_F\vdash <\mathcal{T}_F,\mathcal{L}_F,x=se>\mapsto <\mathcal{T}'_F,\mathcal{L}'_F>} \tag{规则 F2}$$

$$\frac{\begin{array}{c}\mathcal{G}_F,\mathcal{L}_F\vdash se_i\Downarrow v_i\ (i=1,2,\ldots,m)\quad \mathcal{G}_F\vdash <\mathcal{T}_{F_{inst\_id}},op(v_1,v_2,\ldots,v_m)>\mapsto <\mathcal{T}'_{F_{inst\_id}},(r_1,r_2,\ldots,r_n)>\\ \mathcal{T}'_F=\mathcal{T}_F\|_{\text{update}}\mathcal{T}_{F_{inst\_id}}\ \text{to}\ \mathcal{T}'_{F_{inst\_id}}\quad \mathcal{L}'_F=\mathcal{L}_F\|_{x_i\leftarrow r_i}\ (i=1,2,\ldots,n)\end{array}}{\mathcal{G}_F\vdash <\mathcal{T}_F,\mathcal{L}_F,(x_1,x_2,\ldots,x_n)=op(se_1,se_2,\ldots,se_m)\ \text{inst}\ inst\_id>\mapsto <\mathcal{T}'_F,\mathcal{L}'_F>} \tag{规则 F3}$$

$$\frac{\begin{array}{c}\mathcal{P}\vdash_{\text{node}}(op\ (args)\ \text{returns}\ (rets)\ \text{var}\ vars\ \text{let}\ s\ \text{tel},flags,svars)\\ \mathcal{L}'_F=\mathcal{L}_F\|_{args\leftarrow\vec{in}}\quad \mathcal{G}_F\vdash <\mathcal{T}_F,\mathcal{L}'_F,s>\mapsto <\mathcal{T}'_F,\mathcal{L}''_F>\ \overrightarrow{out}=\mathcal{L}''_F(rets)\end{array}}{\mathcal{G}_F\vdash <\mathcal{T}_F,op\left(\vec{in}\right)>\mapsto <\mathcal{T}'_F,\overrightarrow{out}>} \tag{规则 F4}$$

类似地, 对于所有 $i>0$, LustreF 节点实例第 i 个激活周期的语义计算定义为:

$$\begin{cases}\mathcal{T}_{F_0}=\mathcal{T}_F\|_{x_f\leftarrow \text{true for}\ \forall x_f\in flags}\\ \mathcal{G}_F\vdash op\left(\vec{in}_i,\overrightarrow{out}_i\right),\quad \text{if}\ \mathcal{G}_F\vdash <\mathcal{T}_{F_{i-1}},op\left(\vec{in}_i\right)>\mapsto <\mathcal{T}_{F_i},\overrightarrow{out}_i>\end{cases}.$$

3.2.2 从 LustreR 到 LustreF 的翻译正确性

对于 LustreFGen 语义保持性的描述, 可参考第 3.1.3 节中 LustreRGen 的语义保持性, 只要将第 3.1.3 节中的 LustreRGen 替换为 LustreFGen、LustreS 替换为 LustreR、LustreR 替换为 LustreF、$\mathcal{G}$ 替换为 $\mathcal{G}_R$ 以及 $\mathcal{G}_R$ 替换为 $\mathcal{G}_F$.

LustreFGen 中翻译函数的部分定义参见第 2.4 节, 其中包含核心翻译函数 *trans_node*.

4 相关工作

目前从事类 Scade 可信编译器形式化验证的长线项目主要有两个[7,28,31], 除本课题组的 L2C 可信编译器项目[8,24,30–33]外, 另一个是法国 Inria M. Pouzet 教授项目组的工作[6,7,25–29], 其中 Vélus 可信编译器[6,7,23,28,29]代表该项目组的最新进展.

表 3 给出几种类 Lustre 语言编译器所支持的时态与时钟算子特性对比. 相比 Vélus 可信编译器, 本文工作 (对应 Open-L2C 可信编译器) 所支持的时态算子特性更接近于 Scade, 而在 L2C 的当前版本里, 又新增了类 Scade 状态机与条件块等同步控制流特性以及其他重要的 Scade 特性 (参见附录 A).

Vélus 中使用一个辅助算子 *hold* 来定义二元 *fby* 算子的操作语义. 借助于 *hold*, 二元 *fby* 算子的参数流定会向右滑过无定义且位置保持固定的间隙, 以便允许在值域中显式编码存在和不存在[6]. 在开展 L2C 项目的早期, 课题组成员在实现一种在类 Lustre 语言中合并多重嵌套时钟的可信变换时, 尝试过类似的方法来定义二元 *fby* 算子的语义[8,24], 转换器的设计与验证基于 Coq 实现. 变换后, 所有时钟被合并成基准时钟, 如同本文的 *base* 时钟. 从我们自己的实践认识到, 这种方法很难推广到三元 *fby* 算子的翻译与验证. 相比较, 本文所讨论的“两阶段翻译和验证”

有明显的优势, 无论是二元还是三元 *fby* 算子, 二者被纳入统一的翻译和验证框架, 二者的翻译和验证难度没有本质的区别. 其次, 一般而言, 合理的分阶段翻译和验证本身比集中在单一阶段的翻译和验证会在很大程度上降低复杂性, 所涉及的其他特性的翻译和验证也随之降低了难度.

表 3　4 种类 Lustre 语言编译器所支持的时态与时钟算子特性对比

算子类型	Open-L2C	Vélus	Lustre V6	Scade
时钟算子	when, merge (bool)	when, merge (bool)	when, merge (bool, enum)	when, merge (bool, enum)
时态算子	*pre*, *fby* (二元), arrow (->), *fby* (三元)	*fby* (二元), arrow (->)	*pre*, *fby* (二元)	*pre*, *fby* (二元), arrow (->), *fby* (三元), times

翻译确认 (translation validation)[34]是另外一种风格的编译器验证方法, 也有许多面向同步语言 (如 Signal) 的工作[9,10,12,35], 其中不乏针对时态和时钟算子的翻译确认研究.

在国内, 有不少从事同步语言或其他建模语言的建模、调度以及代码生成等方面的研究工作[12,36–41].

5 总　结

本文采用多阶段分步的方法来实现类 Scade 时态算子的编译及其形式化验证, 其中包含两个最核心的阶段: 第 1 阶段将时态算子的语义划分为计算阶段 (*computation*) 和尾声阶段 (*epilogue*), 第 2 阶段完成后续翻译和验证. 需要说明的是, 这里的计算阶段和尾声阶段是指语义执行的阶段, 应注意与本文所提的时态算子核心处理过程的两阶段翻译和验证进行区分. 这种方法能够简化翻译算法和语义定义, 从而降低编译正确性证明的难度, 使得像 Scade 的三元 *fby* 这样难处理的算子也能够在我们的可信编译器项目中得以支持.

本文工作基于交互式定理证明工具 Coq 实现, 已稳定融入 L2C 可信编译器多年, 后者在国内安全攸关领域得到实际应用. 在研发过程中, 相关测试任务随 L2C 可信编译器开展, 后者在每个稳定版本定型之前均经历过企业级的高强度测试.

如第 2 节所述, 文中的可信编译器架构中时态算子编译的核心翻译步骤 (参见图 2), 是以形式化验证的可信编译器 L2C 的一个开源版本 (Open-L2C[30]) 为例, 以方便有兴趣的读者查阅 Coq 实现代码.

虽然能够满足安全攸关领域实际项目的要求, 但本文的分阶段时态算子编译与验证方法并不够完美, 也刻意回避了理论上的难题. 是否可以找到更理想的解决方案, 既看上去优雅又容易完成机器证明, 仍然是今后需要关注的实际问题.

合理的分阶段翻译和验证相比集中在单一阶段的翻译和验证会在很大程度上降低复杂性, 但同时也引入了额外的语言转换与语义保持负担, 因此需要在工程实践中反复进行权衡. 当前版本的 L2C 可信编译器架构是在日积月累的实践中逐步成型的, 后端部分一直相对稳定, 参见附录 A.

References

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

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

[3] LeGuernic 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]

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

[5] Leroy X. Verified squared: Does critical software deserve verified tools? In: Proc. of the 38th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Austin: ACM, 2011. 1–2. [doi: 10.1145/1926385.1926387]

[6] Bourke T, Brun L, Dagand PÉ, Leroy X, Pouzet M, Rieg L. A formally verified compiler for Lustre. In: Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Barcelona: ACM, 2017. 586–601. [doi: 10.1145/3062341.

3062358]

[7] Bourke T, Brun L, Pouzet M. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In: Proc. of the ACM on Programming Languages, 2019, 4: 44. [doi: 10.1145/3371112]

[8] Shi G, Zhang YC, Shang S, Wang SY, Dong Y, Yew PC. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. Science China Information Sciences, 2019, 62(1): 12801. [doi: 10.1007/s11432-016-9270-0]

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

[10] Ngo VC, Talpi JP, Gautier T. Translation validation for synchronous data-flow specification in the SIGNAL compiler. In: Proc. of the 35th IFIP WG 6.1 Int'l Conf. on Formal Techniques for Distributed Objects, Components, and Systems. Grenoble: Springer, 2015. 66–80. [doi: 10.1007/978-3-319-19195-9_5]

[11] Ryabtsev M, Strichman O. Translation validation: From Simulink to C. In: Proc. of the 21st Int'l Conf. on Computer Aided Verification. Grenoble: Springer, 2009. 696–701. [doi: 10.1007/978-3-642-02658-4_57]

[12] Yang ZB, Bodeveix JP, Filali M, Hu K, Zhao YW, Ma DF. Towards a verified compiler prototype for the synchronous language SIGNAL. Frontiers of Computer Science, 2016, 10(1): 37–53. [doi: 10.1007/s11704-015-4364-y]

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

[14] 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. Charleston: ACM, 2006. 42–54. [doi: 10.1145/1111037.1111042]

[15] CompCert. 2025. http://compcert.inria.fr

[16] Ansys SCADE suite. 2025. https://www.ansys.com/products/embedded-software/ansys-scade-suite

[17] Colaço JL, Pagano B, Pouzet M. SCADE 6: A formal language for embedded critical software development (invited paper). In: Proc. of the 2017 Int'l Symp. on Theoretical Aspects of Software Engineering (TASE). Sophia Antipolis: IEEE, 2017. 1–11. [doi: 10.1109/TASE.2017.8285623]

[18] Bertot Y, Castéran P. Coq'Art: The calculus of inductive constructions. In: Interactive Theorem Proving and Program Development. Berlin: Springer, 2004. [doi: 10.1007/978-3-662-07964-5]

[19] Explore the Rocq documentation. 2025. https://rocq-prover.org/docs

[20] 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]

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

[22] Lustre V6. 2025. http://www-verimag.imag.fr/Lustre-V6.html

[23] Vélus. 2025. https://velus.inria.fr

[24] Zhang YC. Part of a research project to construct a formally verified compiler for Lustre. 2013. https://github.com/yczhang89/unify-clock

[25] Biernacki D, Colaco JL, Hamon G, Pouzet M. Clock-directed modular code generation for synchronous data-flow languages. ACM SIGPLAN Notices, 2008, 43(7): 121–130. [doi: 10.1145/1379023.1375674]

[26] Gérard L, Guatto A, Pasteur C, Pouzet M. A modular memory optimization for synchronous data-flow languages: Application to arrays in a Lustre compiler. ACM SIGPLAN Notices, 2012, 47(5): 51–60.

[27] Auger C. Compilation Certifiée de Scade/Lustre [Ph.D. Thesis]. Paris: Université Paris Sud, 2013.

[28] Bourke T, Pesin B, Pouzet M. Verified compilation of synchronous dataflow with state machines. ACM Trans. on Embedded Computing Systems, 2023, 22(5S): 137. [doi: 10.1145/3608102]

[29] Bourke T, Jeanmaire P, Pesin B, Pouzet M. Verified Lustre normalization with node subsampling. ACM Trans. on Embedded Computing Systems, 2021, 20(5S): 98. [doi: 10.1145/3477041]

[30] L2C Team. Open source version of L2C (1.0). 2021. https://github.com/wwssyy/Open-L2C/blob/master/open-l2c-1.0.zip

[31] 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): 2003–2017 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5755.htm [doi: 10.13328/j.cnki.jos.005755]

[32] Shi G, Gan YK, Shang S, Wang SY, Dong Y, Yew PC. A formally verified sequentializer for Lustre-like concurrent synchronous data-flow programs. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering Companion (ICSE-C). Buenos Aires: IEEE, 2017. 109–111. [doi: 10.1109/ICSE-C.2017.83]

[33] 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]

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

[35] Ngo VC, Talpi 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. London: Springer, 2015. 171–185. [doi: 10.1007/978-3-662-46675-9_12]

[36] Wang HF, Yang ZB, Zhou Y, Wang XL, Deng WL, Li W. AGVTS: Automated generation and verification of temporal specifications for aeronautics SCADE models. In: Proc. of the 26th Int'l Symp. on Formal Methods. Milan: Springer, 2025. 338–355. [doi: 10.1007/978-3-031-71177-0_21]

[37] Yang ZB, Yuan SH, Xie J, Zhou Y, Chen Z, Xue L, Bodeveix JP, Filali M. Multi-threaded code generation tool for synchronous language. Ruan Jian Xue Bao/Journal of Software, 2019, 30(7): 1980–2002 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5754.htm [doi: 10.13328/j.cnki.jos.005754]

[38] 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). Oslo: Springer, 2015. 551–569. [doi: 10.1007/978-3-319-19249-9_34]

[39] Zou L, Zhan NJ, Wang SL, Fränzle M. Formal verification of simulink/stateflow diagrams. In: Proc. of the 13th Int'l Symp. on Automated Technology for Verification and Analysis. Shanghai: Springer, 2015. 464–481. [doi: 10.1007/978-3-319-24953-7_33]

[40] Zou L, Zhan NJ, Wang SL, Fränzle M, Qin SC. Verifying simulink diagrams via a hybrid Hoare logic prover. In: Proc. of the 2013 Int'l Conf. on Embedded Software (EMSOFT). Montreal: IEEE, 2013. 1–10. [doi: 10.1109/EMSOFT.2013.6658587]

[41] Li TF, Sun JF, Lü XJ, Chen X, Liu J, Sun HY, He JF. SMT-based formal verification of synchronous reactive model for zone controller. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3080–3098 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6861.htm [doi: 10.13328/j.cnki.jos.006861]

附中文参考文献

[31] 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 [doi: 10.13328/j.cnki.jos.005755]

[33] 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]

[37] 杨志斌, 袁胜浩, 谢健, 周勇, 陈哲, 薛垒, Bodeveix JP, Filali M. 一种同步语言多线程代码自动生成工具. 软件学报, 2019, 30(7): 1980–2002. http://www.jos.org.cn/1000-9825/5754.htm [doi: 10.13328/j.cnki.jos.005754]

[41] 李腾飞, 孙军峰, 吕新军, 陈祥, 刘静, 孙海英, 何积丰. 基于 SMT 的区域控制器同步反应式模型的形式化验证. 软件学报, 2023, 34(7): 3080–3098. http://www.jos.org.cn/1000-9825/6861.htm [doi: 10.13328/j.cnki.jos.006861]

附录 A. 现阶段可信编译器完整架构

图 A1 刻画了现阶段 L2C 高可信编译器的完整架构. 该编译器提供了一种基于形式化验证 (交互式定理证明) 的类 Scade 同步建模语言高可信 C 代码生成器的解决方案, 从基本的分层策略与编译架构技术层面上满足对 Scade 同步建模语言核心特性的完整覆盖. 值得注意的是, 本文工作对应于其中时态算子的核心处理过程, 本质上未发生改变, 但时态算子编译和形式化验证的分阶段处理位置有所调整. 当前编译架构的后端起点不再是图 2 中所示的 LustreS, 而是 LustreV (如图 A1 所示), 远期将会过渡到 LustreW. 如果对应到远期 (以 LustreW 作为后端起点), 将本文用来实现类 Scade 时态算子的编译及其形式化验证的核心阶段对应过来, 前期处理和第 1 阶段将集中于 LustreVGen 步骤, 第 2 阶段仍然包含于 LustreFGen 步骤, 如图 A1 所示. 下面, 抛开本文的内容, 仅对现阶段这一可信编译器的完整编译架构进行简要概述.

现阶段 L2C 高可信编译器的源语言 Lustre*是与 Scade 非常接近的同步建模语言, 目标语言是与 CompCert Clight 基本相同的类 C 语言 Ctemp.

基于验证逻辑, 可将图 A1 所示编译架构分为 3 个阶段近 20 个翻译/变换过程. 第 1 阶段是将同步建模语言

Lustre*源程序翻译至抽象语法树 (Lustre* AST), 翻译过程包括词法和语法分析 (Lexing 和 Parsing). 第 2 阶段是类型和时钟检查等静态语义分析/处理工作以及一些负责预处理或规范化的变换 (LustreWGen 和 LustreVGen), 生成中间表示 LustreV. 从 LustreV 开始是一个转折点, 开始第 3 阶段 (我们称之为后端), LustreV 以及此后的每一中间表示都有形式化定义且满足语义确定性的操作语义 (动态语义), 每一步变换过程的正确性 (语义保持性) 都进行证明 (所有定义和证明都在交互式定理证明工具比如 Coq 中实现). 第 3 阶段是该解决方案的核心, 包含了代码生成器构造和验证的大部分工作.

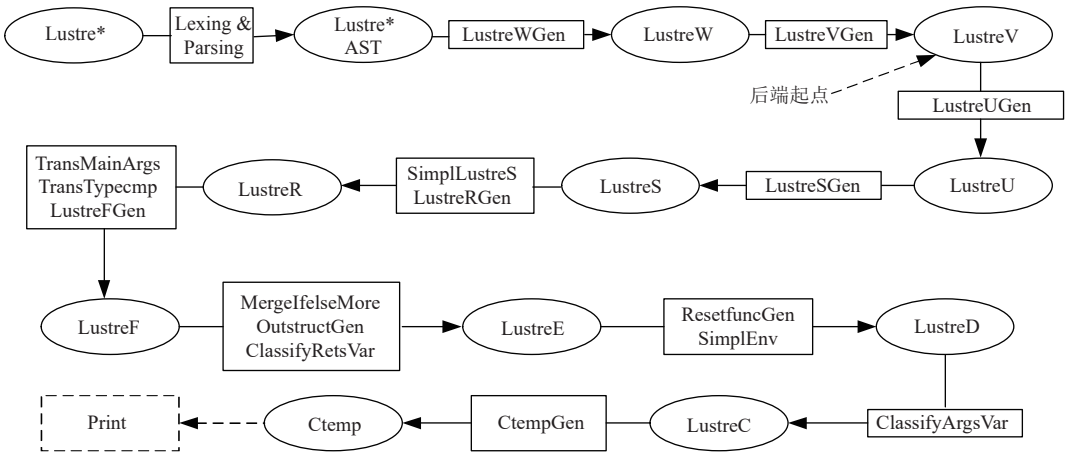

图 A1　可信编译器完整架构 (当前版)

以下进一步解释该解决方案中主要的翻译步骤.

(1) 词法和语法分析. 核心工作包含于如图 A1 所示的 Lexing & Parsing 步骤.

(2) 静态检查及规范化 AST. 核心工作包含于如图 A1 所示的 LustreWGen 和 LustreVGen 步骤. 静态检查的核心工作包含名字空间检查、传统的类型检查、时钟检查、因果性检查以及初始化检查等. 规范化 AST 的核心工作包含各类新增 ID (如 node 实例 ID, *fby* ID 等) 的管理以及表达式或等式相关的一些简单语法层面的变换. 这一步骤包含了本文所述的时态算子编译前期预处理阶段以及第 1 阶段的核心工作.

(3) 状态机 (state machines) 变换. 核心工作是对状态机特性的翻译工作, 如图 A1 所示的步骤 LustreUGen.

(4) 条件块 (conditional blocks) 变换. 核心工作是对条件块 (If 块和 Match 块) 的处理, 如图 A1 所示的步骤 LustreSGen.

(5) 高阶算子 (higher-ordered operators) 的翻译. 核心工作是消去高阶迭代算子, 包含于如图 A1 所示的步骤 LustreRGen.

(6) 时态算子的翻译. 核心工作是消去 *fby* 算子, 参见图 A1 所示的步骤 LustreFGen, 该步骤可对应于时态算子编译的第 2 阶段 (主要完成时态算子的最终消去) 的翻译与验证.

(7) 生成初始化函数. 核心工作是生成 reset 函数, 参见图 A1 所示的步骤 ResetfuncGen.

(8) 生成 C 代码. 核心工作是生成类 C 中间表示 Ctemp 代码, 参见图 A1 所示的步骤 CtempGen.

作者简介

甘元科, 工程师, 主要研究领域为编译器, 形式化验证.

王生原, 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, 编译器, 形式化方法.