软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(2):332−347 [doi: 10.13328/j.cnki.jos.004785]

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

同步数据流语言高阶运算消去的可信翻译*

刘 洋[1], 甘元科[1], 王生原[1], 董 渊[1], 杨 斐[1], 石 刚[1,2], 闫 鑫[1,3]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](新疆大学 信息科学与工程学院,新疆 乌鲁木齐 830046)

[3](太原理工大学 计算机学院,山西 太原 030021)

通讯作者: 刘洋, E-mail: 13893759002@139.com

摘 要: Lustre 是一种广泛应用于工业界核心安全级控制系统的同步数据流语言,采用形式化验证的方法实现 Lustre 到 C 的编译器可以有效地提高编译器的可信度.基于这种方法,开展了从 Lustre*(一种类 Lustre 语言)到 C 子集 Clight 的可信编译器的研究.由于 Lustre*与 Clight 之间巨大的语言差异,整个编译过程划分为多个层次,每个层次完成特定的翻译工作.阐述了其中高阶运算消去的翻译算法,翻译过程采用辅助定理证明工具 Coq 实现,并进行严格的正确性证明.

关键词: 同步数据流语言;形式化验证;高阶运算;定理证明

中图法分类号: TP314

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/4785.htm

Trustworthy Translation for Eliminating High-Order Operation of a Synchronous Dataflow Language

LIU Yang[1], GAN Yuan-Ke[1], WANG Sheng-Yuan[1], DONG Yuan[1], YANG Fei[1], SHI Gang[1,2], YAN Xin[1,3]

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

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

[3](School of Computer Science, Taiyuan University of Technology, Taiyuan 030021, China)

Abstract: Lustre is a synchronous dataflow language widely used in safety critical industrial control system. Formal verification is efficient to improve the reliability of the compiler which translates Lustre to C. Based on this approach, this paper conducts a research on the trustworthy compiler for translating Lustre*(a Lustre-like language) to Clight (a subset language of C). The entire compiling process is divided into different stages to tackle the large difference between Lustre* and Clight. Each stage performs a specific translation task. This paper describes a translation algorithm which eliminates high-order operations. The implementation of translation process is assisted by theorem proving tool Coq, and a strict proof of correctness of the process is also provided.

Key words: synchronous dataflow language; formal verification; high-order operation; theorem proving

同步数据流语言[1,2](例如 Lustre[3],Signal 等)广泛应用于工业界的核心安全级控制系统,如航空、核电等高安全等级的关键领域,与语言相关的软件的安全性也越来越受到人们的关注,特别是一些基础软件,如操作系统、编译器等.确认这些软件的安全可靠非常重要,同时,随着软件系统的复杂度的提高,软件的安全性保证也变

* 基金项目: 国家自然科学基金(61272086); 国家科技支撑计划(2013BAB05B05)
收稿时间: 2014-07-05; 修改时间: 2014-10-31; 定稿时间: 2014-11-26

得越来越困难,依靠传统的测试、代码审核和过程管控等方法来保证软件的安全性是远远不够的.近年来,形式化验证方法已成功地应用于可信编译器的实现中,CompCert[4,5]是其中的杰出代表.形式化验证方法从数学角度对软件系统进行描述,从逻辑上对软件系统的正确运行进行验证,能够充分地保证软件系统可信,可以最大程度地提高软件系统的可信度.

我们项目组基于形式化验证方法开展了一项从 Lustre*语言到 C 子集 Clight 的可信翻译器的研究工作,称为 L2C 项目[6].图 1 是该项目的整体框架.

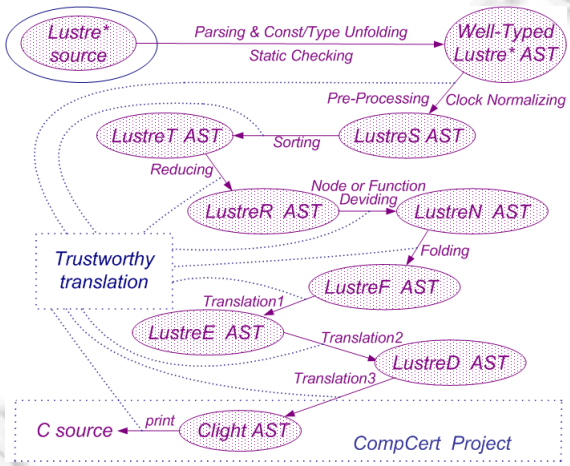


Fig.1 Current architecture of the trustworthy compiler L2C

图 1 当前 L2C 项目的可信编译器总体结构

Lustre*是一种类 Lustre[7]语言,它是以 Lustre V6[8]的核心语言为基础,并加上一些类似于 Scade[9,10]的扩展.Lustre*语言中除了通常的算术和逻辑表达式之外,还包含了 struct,list,switch 和 array 等表达式,这些表达式的执行规则类似于 C 语言中的相应表达式.此外,Lustre*从 Scade 中继承了一些高阶运算和对于数组和结构体的操作.Clight[11]来源于 CompCert 项目,是一种兼容于关键嵌入式软件推荐使用的较大的 C 语言子集.

L2C 项目的开发采用辅助证明工具 Coq 实现,集程序、性质和证明于一体,最终代码被抽取为 Ocaml 代码,与前端的 Ocaml 代码集成,得到完整编译过程的代码.通过图 1 的最后一步,Lustre*源程序最终被翻译到 CompCert 项目的 Clight AST,Clight 语言的语法语义采用 CompCert 的定义.

同步数据流语言 Lustre*与 Clight 相比有着巨大的差异,Lustre*具有时钟同步、数据流、并发及流数据对象等特征,而 Clight 则具有顺序控制流特征,语言跨度较大.直接进行翻译及证明将会非常复杂,因此,我们将项目分为多个层次,保证每个层次语法和语义跨度不大,每个层次完成特定的工作,比如时钟归一化[12]、拓扑排序[13]、时态算子消除[14]等工作.本文的研究工作高阶消去翻译及正确性证明,包含于如图 1 所示的 *LustreT* 到 *LustreR* 层的 Reducing 过程中.

1 Lustre*的高阶运算

Lustre*语言是一种同步数据流语言,主要用于设计实时的反应式系统,其主要应用领域有自动化控制以及信号处理系统等.Lustre*具有时钟特性,在 Lustre*中,任何变量和表达式都代表着一个流(stream)数据,我们将一个流数据看作由两部分组成:一个给定类型的值序列和一个时钟.由于在 Well-Typed Lustre* AST 层到 LustreS AST 层的翻译过程中已进行了时钟归一化处理,所有数据流都统一到全局时钟,因此,在后续的翻译过程中不再需要考虑时钟.

Lustre*语言作为一种同步数据流语言,在实际程序中也有控制流的需求,Lustre*中控制流可以直接从两个方面体现:一是进行周期控制,针对数据流的时钟特性,每个周期处理当前周期的数据;二是通过 if 表达式进行条

件控制.然而,在面对数组的处理时,这两个方面都无能为力.因此,在 Lustre*中增加丰富的高阶运算以满足数组处理的需求.

Lustre*中的高阶运算包括 *map* 和 *fold* 两类算子.*map* 运算形式为:$v_1,...,v_k$=(*map op* ⟨⟨*size*⟩⟩)($a_1,...,a_n$),a_1 到 a_n 是 n 个输入数组,*size* 为数组大小;op 为一个操作或一个节点 *node* 或函数 *function* 调用,其输入参数为 n 个数组;v_1 到 v_k 为输出的数组结果,例如:arr_1=(*map add* ⟨⟨10⟩⟩)(arr_2,arr_3),则 $arr_1[i]=arr_2[i]+arr_3[i]$,$0\leqslant i<10$.

fold 运算形式为 *acc*=(*fold op* ⟨⟨*size*⟩⟩)(*init*,$a_1,...,a_n$),a_1~a_n 是 n 个输入数组,*size* 为数组大小,*op* 为一个操作或一个函数调用 *function*,其输入参数为 n+1 个参数:n 个数组和一个 *acc* 的初值 *init*,输入参数列表必须非空,$acc[0]=init$,$acc[i+1]=op(acc[i],a_1[i],\ldots,a_n[i])$,$0\leqslant i<size$.

除了 *map*,*fold* 算子以外,Lustre*中支持的其他高阶算子还有 *mapi*,*foldi*,*mapfold*,*mapw*,*foldw*,*mapwi* 和 *foldwi* 等,简单介绍如下:

- *mapi* 运算类似于 *map*,只是其第 1 个输入不是由外部输入,而是取固定的数组[$v_0,v_1,\ldots$].
- *foldi* 运算类似于 *fold*,只是其第 1 个输入数组为固定的数组[$v_0,v_1,\ldots$].
- *mapfold* 运算相当于 *map* 运算和 *fold* 运算的一个组合,即同时进行 *map* 和 *fold* 运算.
- *mapw* 运算是一个条件迭代高阶运算,它的执行过程与 *map* 类似,但是当条件为 false 时终止执行,并记录终止时的数组索引.
- *foldw* 也是一个条件迭代高阶运算,其执行过程与 *fold* 类似,但是当条件为 false 时停止执行,停止执行时的索引值记为一个整型值.
- *mapwi* 运算类似于 *mapw* 只是其第 1 个输入不是由外部输入,而是取固定的数组[$v_0,v_1,\ldots$].
- *foldwi* 类似于 *foldw*,只是其第 1 个输入数组 a_1 是取固定的[$v_0,v_1,\ldots$].

通过对高阶运算的介绍我们可以看出,高阶运算的特点很鲜明:以一个操作或函数调用为输入,使用 *map* 或 *fold* 等算子构成一个新的算子,作用在数组或参数列表上进行循环运算.这样的一种运算在 C 中并没有直接与之对应的操作,因此我们在翻译到 Clight 的过程中必须消去高阶运算,我们将这项工作集中在 *LustreT* 到 *LustreR* 层的翻译阶段完成.

基于高阶运算的特点,我们最终将高阶运算翻译到 C 语言中 for 循环语句中,采用 for 循环来实现高阶消去.因此,我们在 *LustreR* 层中定义一个类 for 的语句,称为 *Rfor* 语句.下面给出一个 *fold* 运算的 Lustre 代码以及高阶消去后的代码(如图 2 所示).

```
node add(a,b: int) returns (s: int);
let
   s=a+b;
tel
node fold_array(acc0: int, arr1: int^5) returns (res: int)
let
   res=(fold add ⟨⟨5⟩⟩) (acc0, arr1);
tel
```

(a) 包含高阶运算的代码

```
node add(a,b: int) returns (s: int);
let
   s=a+b;
tel
node fold_array(acc0: int, arr1: int^5) returns (res: int)
var
   acc: int^5;
let
   loopid=0;
   init=acc0;
   loop_add=loopid+1;
   loop_cond=loopid<5;
   acc[0]=init;
   Rfor(init,loop_cond,loop_add)
   {
   acc[loopid+1]=add(acc[loopid],arr1[loopid]);
   }
   res=acc[loopid+1];
tel
```

(b) 高阶消去后的代码

Fig.2 An example of fold program for eliminating high-order operation

图 2 高阶运算消去的 fold 程序示例

图 2(a)是一段包含 *fold* 高阶运算的 Lustre*代码,*b* 为高阶消去后的 *LustreR* 层代码,在节点 *fold_array* 中,采用 *Rfor* 实现高阶消去,*init* 为初始状态,*loop_add* 为循环增量,*loop_cond* 为循环条件.具体的消去算法后文将做详细的介绍.

2 相关工作

为了实现编译器的可信编译,人们提出了多种方法,主要有大量测试、人工代码审查以及开发过程的严格管控等,Scade 通过严格的 V&V 过程管控实现了从 Lustre 到 C 的可靠翻译.然而,这些方法并不能够完全杜绝误编译的发生.形式化验证方法近年来成功地应用于可信编译器的构造中,编译器的形式化验证主要有两种方式:

- 一种是 Pnueli 等人[15]提出的翻译确认方法.它通过证明源代码和目标代码的语义等价性来验证编译器的正确性,不用对编译器自身进行验证,具有很好的可重用性,然而会有"虚报"的可能性.采用这种方法,Pnueli 等人实现了对同步流语言 Signal 到 C 的翻译过程的确认[16].
- 另一种是对编译器本身进行证明.经过严密的证明,可杜绝误编译的发生.CompCert 编译器是其中的代表.CompCert 将 Clight 到 PowerPC 汇编代码的编译过程划分为多个阶段,每个阶段独立地定义语法语义,完成特定的工作,并对每个阶段的翻译正确性进行证明,从而保证整个编译器的可信.对编译器本身进行验证是一种理想的方法,但其可重用性较差,因而开发成本较高,适用于为了安全性不惜代价的领域.

我们借鉴 CompCert 的成功经验,采用对编译器本身进行证明的方法构造从 Lustre*到 C 的可信编译器.据我们所知,目前仅有两个项目组规模性开展了类似的工作.除了本项目组以外,几年前,M. Pouzet 和他的同事还启动了一项与我们目标相近的研究工作[17].目前,两个项目组都实现了相应的原型系统:M. Pouzet 项目组在其最新的工作中完成了一个精简 Lustre 语言的经过验证的编译器[18];在 L2C 一期项目中,我们实现了一个单时钟的可信编译器原型[19].前者支持多时钟特征;后者根据项目甲方的需求,目前只支持统一的默认时钟(单一时钟).除了多时钟特征以外,后者还支持更大的 Lustre 子集.这两个可信编译器的设计框架完全不同,前者的许多核心翻译都集中在同一阶段,而后者分为多个阶段.相比之下,后者具有更好的可扩展性.根据我们的实践经验,这或许是扩展到实用语言编译器的必由之路.

目前,L2C 项目已进入二期开发阶段.在二期工作中,我们将增加对数组和高阶运算以及其他在一期中未涉及的语言特征;同时,针对一期工作中的不足之处以及新的需求,对系统设计框架进行改进.

本文第 3 节介绍 Lustre*高阶语法及语义.第 4 节介绍高阶消去的算法.第 5 节介绍高阶消去的证明.第 6 节介绍实现情况.第 7 节进行总结.

3 Lustre*高阶语法及语义

3.1 高阶运算的语法

从 Lustre*到 Clight 的整个翻译过程分为多个中间层次,每个中间层次都定义了各自的语法语义,相邻两层之间的语法语义较为接近,差异较大的部分很大程度上取决于当前翻译阶段的主体目标.例如,对于 *LustreT* 层到 *LustreR* 层的翻译阶段,我们定义了许多共有的语法元素,主要不同之处是:*LustreT* 层有高阶运算;而在 *LustreR* 层,它将对应于循环语句.

一个 *LustreT*/*LustreR* 程序包括类型定义块、全局常量块以及节点/函数块(含一个主节点/函数).图 3 描述了 *LustreT*/*LustreR* 程序的抽象语法.

prog	::=	(*td**,*cd**,*fd**,*main_id*)	program
td	::=	*id*=τ	type declaration
cd	::=	*id*=*c*	const declaration
main_id	::=	*id*	main node identifier

Fig.3 Abstract syntax of *LustreT*/*LustreR* program

图 3 *LustreT*/*LustreR* 程序的抽象语法

在 *LustreT*/*LustreR* 程序中,节点和函数的定义是一样的,二者的区别在于:*node* 中有时态操作时需要的历史信息,而 *function* 中没有时态操作.节点/函数的抽象语法如图 4 所示,其中,$vars_1$ 为节点的输入参数列表,$vars_2$ 为节点的输出参数列表,节点的 *body* 由语句列表组成.

stmt 描述了等式(或语句)抽象语法,分别定义了赋值语句、节点调用语句及高阶运算语句,如图 5 所示.表达式的抽象语法如图 6 所示.*hstmt* 的抽象语法如图 7 所示.

fd	::=	$id(vars_1)$ **returns** $(vars_2)$ *body*	node/function declaration
body	::=	***var*** *vars* **let** *stmt** **tel**	node/function body
vars	::=	(id,τ)*	parameter/variable list
τ	::=	*TTint* | *Tfloat*	type
	|	*Tarray*(τ,*num*)	array type

Fig.4 Abstract syntax of node/function

图 4 节点/函数的抽象语法

stmt	::=	(id,τ)=*expr*	assignment
	|	$lhs=id_1(expr^*)$ *instance* id_2	node/function call
	|	*Highorder*(*hstmt*,*sexpr*)	high-order operate
lhs	::=	*lh**	left hand identifiers
lh	::=	*id*	identifier
	|	_	anonymous symbol

Fig.5 Abstract syntax of *LustreT* equation/statement

图 5 *LustreT* 等式/语句的抽象语法

sexpr	::=	(c,τ)	constant expression
	|	(id,τ)	variable expression
	|	$(unop\ sexpr_1,\tau)$	unary operation expression
	|	$(sexpr_1\ binop\ sexpr_2,\tau)$	binary operation expression
	|	$(aryacc\ sexpr_1\ sexpr_2,\tau)$	access array expression
expr	::=	*sexpr*	
	|	if $expr_1$ then $expr_2$ else $expr_3$	if expression
	|	*fby* ($expr_1$,*n*,$expr_2$)	fby expression
	|	$expr_1 \rightarrow expr_2$	arrow expression
	|	*nil*	empty list
	|	$expr_1$++$expr_2$	combination of list expression
binop	::=	+|−|*|/	
	|	*div* |mod|and|or|*xor*	
	|	=|!=|<|>|≤|≥	
Unop	::=	not | *neg*	

Fig.6 Abstract syntax of *LustreT*/*LustreR* expression

图 6 *LustreT*/*LustreR* 表达式的抽象语法

Hstmt	::=	(id,τ)	=*mapbinary binop* $sexpr_1$ $sexpr_2$	mapbinary expression
	|	(id,τ)	=*mapbinaryi binop sexpr*	mapbinaryi expression
	|	(id,τ)	=*mapunary unop sexpr*	mapunary expression
	|	(id,τ)	=*mapunaryi unop*	mapunaryi expression
	|	(id,τ)	=*foldbinary binop* $sexpr_1$ $sexpr_2$	foldbinary expression
	|	(id,τ)	=*foldbinaryi binop sexpr*	foldbinaryi expression
	|	(id,τ)	=*foldunaryi unop*	foldunary expression
	|	*lhs*	=*mapcall* id_1(*expr**) *instance* id_2	mapcall expression
	|	(id,τ)	=*foldcall* id_1(*sexpr*,*expr**) *instance* id_2	foldcall expression
	|	*lhs*	=*mapcallw* id_1(*sexpr*,*expr**,*expr**) *instance* id_2	mapcallw expression
	|	(id_1,τ), (id_2,τ)	=*foldcallw* id_1(*sexpr*$_1$,*sexpr*$_2$,*expr**) *instance* id_2	foldcallw expression
	|	*lhs*	=*mapcalli* id_1(*expr**) *instance* id_2	mapcalli expression
	|	(id,τ)	=*foldcalli* id_1(*sexpr*,*expr**) *instance* id_2	foldcalli expression
	|	*lhs*	=*mapcallwi* id_1(*sexpr*,*expr**,*expr**) *instance* id_2	mapcallwi expression
	|	(id_1,τ), (id_2,τ)	=*foldcallwi* id_1(*sexpr*$_1$,*sexpr*$_2$,*expr**) *instance* id_2	foldcallwi expression
	|	(id,τ), *lhs*	=*mapfoldcall* id_1(*sexpr*,*expr**) *instance* id_2	mapfoldcall expression

Fig.7 *LustreT* high-order operation syntax

图 7 *LustreT* 高阶运算语法

与 *LustreT* 层相比,*LustreR* 层的语法中没有高阶语句 *Highorder* 和 *hstmt* 的定义,增加了 *Rfor* 语句,引入了显式的语句顺序复合算子 *seq* 以及 *skip* 语句,如图 8 所示.

$$
\begin{aligned}
Stmt \;::=\;& (id,\tau)=expr\\
\mid\;& lhs=id_1(expr^*)\ instance\ id_2\\
\mid\;& Rfor(stmt_1,expr,stmt_2,stmt_3)\\
\mid\;& seq(stmt_1,stmt_2)\\
\mid\;& skip
\end{aligned}
$$

Fig.8 Abstract syntax of *LustreR* equation/statement

图 8 *Lustre R* 层等式/语句的抽象语法

3.2 语义环境介绍

在介绍高阶运算的语义之前,首先对 Lustre*语义环境和语义值进行介绍.Lustre*语义环境定义如下:

ge	::=	(F,ρ)	global environment
F	::=	$(id\rightarrow fd)$	function map
ρ	::=	$(id,\tau)\rightarrow v$	constant map
le	::=	$(id,\tau)\rightarrow v$	local environment in current cycle
te	::=	le^*	temporal environment
e	::=	$(te,id\rightarrow e^*)$	node environment

语义环境由全局环境(*ge*)和局部环境(*e*)组成,全局环境建立常量 ID 与相应的常量值和类型的映射,节点/函数 ID 与节点/函数定义之间的映射.

局部环境由 3 层构成:

- 当前周期的节点局部环境 *le*:每个时钟周期每个节点实例都将生成一个局部环境 *le*,建立当前周期下变量 ID 到变量值之间的映射关系.
- 时态环境 *te*:Lustre*语言中,有 *pre*,*fby* 以及 *arrow* 等时态操作,在进行时态操作中,需要用到流数据的历史信息,因此,我们构造了时态环境 *te* 来记录每个节点实例的历史信息.一个节点实例的时态环境 *te* 是一个局部环境 *le* 列表,记录当前节点在所有时钟周期的执行情况,第 *n* 元素表示节点第 *n* 个时钟周期的 *le*.
- 节点顶层环境 *e*:一个节点实例的顶层环境 *e* 是一个复杂的树状结构,其数据域 *te* 维护了该节点实例中局部变量、输入和输出参量的历史取值,其每个子树本身又是新节点实例的顶层环境,第 *n* 个子树代表该节点中第 *n* 个调用的子节点实例的顶层环境.

Lustre*语义值包括整型值、浮点型值以及两个逻辑 *bool* 量,如图 9 所示.

v	::=	$Vint(i)$	integer value
	|	$Vfloat(f)$	float value
	|	$Vtrue$	logical true value
	|	$Vfalse$	logical false value

Fig.9 Semantic values

图 9 语义值

3.3 高阶运算的操作语义

3.3.1 基本语义规则

基于以上的语义环境定义,我们有如下的基本执行规则:

- $ge,le\vdash sexpr\Rightarrow v$:在全局环境 *ge* 和当前周期的节点局部环境 *le* 中,执行表达式 *sexpr* 得到值 *v*.
- $ge,te\vdash(expr,\tau)\Rightarrow v$:在全局环境 *ge* 和时态环境 *te* 中,执行表达式 *expr* 得到值 *v*.
- $ge\vdash(e,fd,vargs)\Rightarrow(e',vrets)$:在全局环境 *ge* 中,输入参数列表为 *vargs* 的节点 *fd* 执行后返回值列表 *vrets*,

局部环境由 e 变为 e'.

- $ge\vdash(e,stmts)\Rightarrow e'$:在全局环境 ge 中,执行等式/语句列表 $stmts$,局部环境由 e 变为 e'.
- $ge\vdash(e,vass,prog,n,maxn)\Rightarrow vrss$:在全局环境 ge 中,程序 $prog$ 从第 n 周期到 $maxn$ 周期周期性地执行,$vass$ 和 $vrss$ 分别为输入和输出流.

我们基于以上基本执行规则定义大步操作语义,限于篇幅,我们只列举一些典型的语义规则(均适用于 *LustreT* 和 *LustreR*).

表达式 *sexpr* 的语义执行规则:

$$\frac{}{ge,le\vdash(i,Tint)\Rightarrow Vint(i)} \tag{1}$$

$$\left.\begin{array}{c} vclassify(id)=Lid \\ le(id,ty)=v \end{array}\right\} \Big/ \; ge,le\vdash(id,ty)\Rightarrow v \tag{2}$$

$$\left.\begin{array}{c} vclassify(id)=Gid \\ ge.\rho(id,ty)=v \end{array}\right\} \Big/ \; ge,le\vdash(id,ty)\Rightarrow v \tag{3}$$

注:L2C 项目中,采用整型值作为变量 id,所有变量 id 进行统一管理,并区分为不同类别.$vclassify(id)$获取 id 的类别,Lid 意为 id 用于节点内部,Gid 意为 id 是一个全局常量的标识符.

$$\left.\begin{array}{c} ge,le\vdash(a_1,ty)\Rightarrow v_1, ge,le\vdash(a_2,ty)\Rightarrow v_2 \\ sem\_binary\_operation(binop,v_1,v_2,ty)=v \end{array}\right\} \Big/ \; ge,le\vdash(a_1\ binop\ a_2,ty)\Rightarrow v \tag{4}$$

注:*sem_binary_operation* 是用于二元操作 *binop* 的语义函数.

赋值语句 *assign* 语义执行规则:

$$\left.\begin{array}{c} ge,(eh::et)\vdash(expr,ty)\Rightarrow v \\ locenv\_setvars(eh,lh::nil,v::nil)=eh' \end{array}\right\} \Big/ \; ge\vdash((eh::et),se,(id,ty)=expr)\Rightarrow((eh'::et),se) \tag{5}$$

locenv_setvars(*eh*,*vass*,*vrss*)=*eh'*意为在当前周期的节点局部环境 *eh* 中,用值列表 *vrss* 修改变量列表 *vars* 中所有变量的值,当前周期的局部环境变为 *eh'*,*vars* 和 *vrss* 具有相同的列表长度.

3.3.2 *LustreT* 层高阶运算语义

高阶运算是一个对数组或输入参数列表进行循环计算的过程,前面已经阐述过,针对高阶运算的这一特点,我们的高阶消去算法实现为将 *LustreT* 层的高阶运算翻译为 *LustreR* 层的 *Rfor* 语句.循环运算分 3 种状态进行归纳定义:初始状态、循环处理状态以及循环终止状态.我们在定义 *LustreT* 层的高阶运算的语义时,抽象地将其构造成 3 种形态:高阶初始状态、高阶运算主体以及高阶运算终止状态,分别对应于 *Rfor* 循环的初始状态、循环处理状态以及循环终止状态.这样的语义定义有利于证明过程中进行归纳证明.

LustreT 层高阶运算的 *eval_Highorder_start* 语义规则:

$$\left.\begin{array}{c} ge\vdash(e,(loopid,Tint)=(0,Tint))\Rightarrow e_1 \\ ge\vdash(e_1,initstmtsof(hstmt))\Rightarrow e_2 \\ ge\vdash(e_2,Highorder(hstmt,j))\Rightarrow e_3 \end{array}\right\} \Big/ \; ge\vdash(e,Highorder(hstmt,j))\Rightarrow e_3 \tag{6}$$

eval_Highorder_start 的语义规则为:在全局环境 ge 中,将循环变量 *loopid* 赋值 0,局部环境由 e 变为 e_1. *initstmtsof*(*hstmt*)表示从高阶运算表达式 *hstmt* 中解析出高阶运算的初始化操作,局部环境由 e_1 变为 e_2;接着执行完高阶语句,局部环境由 e_2 变为 e_3.

LustreT 层高阶运算的 *eval_Highorder_stop* 语义规则:

$$\frac{ge,eh\vdash loopid<j\Rightarrow Vfalse}{ge\vdash(((eh::et),se),Highorder(hstmt,j))\Rightarrow((eh::et),se)} \tag{7}$$

loopid<*j* 为条件表达式,*j* 为数组上界.*eval_Highorder_stop* 语义执行规则为:当在全局环境 *ge* 和局部环境 *eh* 下,如果 *loopid*<*j* 为 *Vfalse*,则高阶运算终止,在高阶运算终止计算中不改变节点的局部环境.以图 2 中 *fold* 程序为例,当 *loopid*<5 为 *Vfalse* 时,高阶运算终止.

LustreT 层高阶运算的 *eval_Highorder_body* 语义规则:

$$\left.\begin{array}{c} ge,eh\vdash loopid<j\Rightarrow Vtrue \\ ge\vdash(((eh::et),se),eval\_htmt(hstmt,loopid))\Rightarrow e_1 \\ ge\vdash(e_1,loop\_add)\Rightarrow e_2 \\ ge\vdash(e_2,Highorder(hstmt,j))\Rightarrow e_3 \end{array}\right\} \Big/ ge\vdash(((eh::et),se),Highorder(hstmt,j))\Rightarrow e_3 \tag{8}$$

eval_Highorder_body 语义的执行规则为:当 *loopid*<*j* 为 *Vtrue* 时执行高阶运算,*eval_htmt* 语义函数表示计算高阶表达式 *hstmt*,局部环境变为 e_1;然后执行增量语句 *loop_add*,局部环境变为 e_2;继续执行完高阶语句,局部环境由 e_2 变为 e_3.

eval_htmt(*hstmt*,*loopid*)的执行语义将归纳于 *hstmt* 的具体高阶算子进行定义,限于篇幅,我们不列举对应于所有这些高阶算子的语义规则,而是代表性地展示其中几条:

- 对应于 *mapbinary* 算子的语义规则:

$$\left.\begin{array}{c} ge,eh\vdash(loopid,Tint)=Vint(i) \\ ge,eh\vdash(lid,Tarray(ty,num))\Rightarrow arr \\ ge,eh\vdash((aryacc\ a_1\ i\ ty)\ binop\ (aryacc\ a_2\ i\ ty),ty)\Rightarrow v \\ locenv\_setlvar(eh,(aryacc\ arr\ i\ ty),v)=eh_1 \end{array}\right\} \Big/ ge\vdash(((eh::et),se),eval\_htmt((lid,Tarray(ty,num))=mapbinary\ binop\ a_1\ a_2,loopid))\Rightarrow((eh_1::et),se) \tag{9}$$

mapbinary 运算功能为对两个数组中所有元素执行二元操作,返回一个执行结果数组.*mapbinary* 的语义执行规则为:首先,计算当前数组元素的索引 *i*;然后,通过 *aryacc* 运算取出数组 a_1 和 a_2 中第 *i* 个元素,并进行二元运算得到值 *v*;最后,通过 *locenv_setlvar* 在局部环境 *eh* 中将值 *v* 赋予一个新的数组中第 *i* 个元素,局部环境变为 eh_1.

- 对应于 *foldbinary* 算子的语义规则:

$$\left.\begin{array}{c} ge,e\vdash(loopid,Tint)=Vint(i),ge,e\vdash(lid,ty)\Rightarrow v \\ ge\vdash(e,v=(v\ binop\ (aryacc\ a_1\ i\ ty)),ty)\Rightarrow e_1 \end{array}\right\} \Big/ ge\vdash(e,eval\_htmt((lid,ty)=foldbinary\ binop\ v\ a_1,loopid))\Rightarrow e_1 \tag{10}$$

foldbinary 运算功能为对一个数组和一个初值进行折叠处理,返回一个处理结果.*foldbinary* 的语义执行规则为:对变量 *v* 和数组 a_1 中第 *i* 个元素进行二元操作,并将操作结果赋于变量 *v*,节点局部环境变为 e_1.

- 对应于 *mapcall* 算子的语义规则:

$$\left.\begin{array}{c} ge,eh\vdash(loopid,Tint)=Vint(i) \\ ge.F(id_1)=fd \\ fd=id_1(args)\ return\ (rets)\ body \\ locenv\_getarys(eh,i,args)=vargs \\ call\_env(fd,id_2,se,se_1,ef,ef') \\ ge\vdash(ef,fd,vargs)\Rightarrow(ef',vrets) \\ locenv\_setarys(eh,lhs,vrets)=eh_1 \end{array}\right\} \Big/ ge\vdash(((eh::et),se),eval\_htmt(lhs=mapcall\ id_1(args)\ instance\ id_2,loopid))\Rightarrow((eh_1::et),se_1) \tag{11}$$

mapcall 运算功能为调用节点对参数列表进行处理,返回一个结果列表.*locenv_getarys*(*eh*,*args*)=*vargs* 意为从当前周期的局部环境 *eh* 中获取变量列表 *args* 的值列表 *vargs*,*call_env*(*fd*,id_2,*se*,se_1,*ef*,*ef'*)意为调用节点 *fd* 时,实例 id_2 对应的子环境由 *se* 变为 se_1,局部环境由 *ef* 变为 *ef'*.*mapcall* 的语义执行规则为:从全局环境中通过节点

id_1 找到对应的节点定义 *fd*,从局部环境 *eh* 中获取输入参数变量列表的取值 *vargs*,调用节点对输入参数列表的取值进行处理,局部环境由 *ef* 变为 *ef'*,得到输出值列表 *vres*.最后通过 *locenv_setarys* 在当前周期的局部环境 *eh* 中将 *vres* 赋于变量列表 *lhs*,当前周期的局部环境变为 eh_1.

以上述 3 个高阶算子语义规则为参照,对 *LustreT* 层其他高阶算子语义规则简要概括介绍如下:

- *mapbinaryi*:语义与 *mapbinary* 语义类似,区别在于二元运算的第 1 个参数不是数组 a_1 中的元素,而是当前 *loopid* 值;
- *mapunary*:语义与 *mapbinary* 语义类似,区别在于输入为一个数组 a_1,操作算子为一元操作.
- *mapunaryi*:语义与 *mapunary* 语义类似,区别在于输入为当前 *loopid* 值.
- *foldbinaryi*:语义与 *foldbinary* 语义类似,区别在于输入为当前 *loopid* 值.
- *foldcall*:语义规则为在 *foldbinary* 语义的基础上,以函数调用 *call* 的语义代替二元运算的语义,构成 *foldcall* 的语义规则.
- *mapcallw*:*mapcallw* 在 *mapcall* 的运算中增加了一个 *bool* 变量 *mapwid*,当 *mapwid* 为 *Vtrue* 时,其后续执行规则与 *mapcall* 相同;当为 *Vfalse* 时,其语义规则为

$$\left.\begin{array}{c} ge,eh\vdash(MAPWID,Tbool)\Rightarrow Vfalse \\ ge\vdash(eh,ad)\Rightarrow vrs \\ locenv\_setlarys(eh,lhs,vrs)=eh_1 \end{array}\right\} \over ge\vdash(((eh::et),se),eval\_htmt(lhs=mapcallw\ id_1(ac,ad,args)\ instance\ id_2))\Rightarrow((eh_1::et),se) \tag{12}$$

- *foldcallw*:*foldcallw* 在 *foldcall* 的运算中增加了一个 *bool* 变量 *mapwid*,当 *mapwid* 为 *Vtrue* 时,其后续执行规则与 *foldcall* 相同;当为 *Vfalse* 时,其语义规则为

$$\frac{ge,eh\vdash(MAPWID,Tbool)\Rightarrow Vfalse}{ge\vdash(((eh::et),se),eval\_htmt((id_1,Tint),(id_2,ty)=foldcallw\ id_1(ac,ai,args)\ instance\ id_2))\Rightarrow((eh::et),se)} \tag{13}$$

- *mapcalli*:语义与 *mapcall* 语义类似,区别在于节点计算时第 1 个输入参数为当前列表索引,在语义规则中表示如下:*mapcall* 为 $ge\vdash(ef,fd,vargs)\Rightarrow(ef',vrets)$,*mapcalli* 为 $ge\vdash(ef,fd,(i::vargs))\Rightarrow(ef',vrets)$.
- *foldcalli*:语义与 *foldcall* 语义类似,区别在于节点计算时第 1 个输入参数为当前列表索引,在语义规则中表示如下:*mapcall* 为 $ge\vdash(ef,fd,(v::vargs))\Rightarrow(ef',(vrets::nil))$,*foldcalli* 为 $ge\vdash(ef,fd,(i::v::vargs))\Rightarrow(ef',(vrets::nil))$.
- *mapcallwi*:语义与 *mapcallw* 语义类似,这二者的区别与 *mapcalli* 和 *mapcall* 的区别相同.
- *foldcallwi*:语义与 *foldcallw* 语义类似,这二者的区别与 *foldcalli* 和 *foldcall* 的区别相同.
- *mapfoldcall*:*mapfoldcall* 的作用相当于 *mapcall* 和 *foldcall* 的一个综合体,其语义也是这二者的一个综合体.

高阶运算具有复杂的语义,其复杂之处主要在于环境变化的复杂性,每次计算都会引起局部环境的变化.当经过高阶消去翻译到 *LustreR* 层时,如何体现这种环境的变化以及如何证明这两层间环境的匹配是主要的难点.

3.3.3 *LustreR* 层循环语义

LustreT 层高阶运算经过高阶消去后翻译到 *LustreR* 层,对应的是 *Rfor* 语句,*Rfor* 语句包括 3 种状态,分别是 *Rfor* 初始状态、*Rfor* 循环状态以及 *Rfor* 终止状态.

在介绍 *Rfor* 语义规则之前,先介绍 *LustreR* 层另外两个语句 *skip* 和 *seq* 的语义规则:

- *skip* 语句的语义规则:

$$\frac{}{ge\vdash(e,skip)\Rightarrow e} \tag{14}$$

- *seq* 语句的语义规则:

$$\left.\begin{array}{c} ge\vdash(e,s_1)\Rightarrow e_1 \\ ge\vdash(e_1,s_2)\Rightarrow e_2 \end{array}\right\} \over ge\vdash(e,seq\ s_1\ s_2)\Rightarrow e_2 \tag{15}$$

skip 语句的语义执行规则为:在局部环境 *e* 中执行语句 *skip*,局部环境不变.*seq* 语句的语义执行规则为:如果

在全局环境 *ge* 中执行语句 s_1 后,局部环境由 *e* 变为 e_1,执行语句 s_2 后局部环境由 e_1 变为 e_2,则在局部环境 *e* 中执行语句 *seq* s_1 s_2 后,局部环境变为 e_2.

LustreR 层 *Rfor* 语句的 *eval_Rfor_start* 语义规则:

$$\left.\begin{array}{c} ge\vdash(e,a_1)\Rightarrow e_1 \\ ge\vdash(e_1,Rfor(skip,a_2,a_3,s))\Rightarrow e_2 \end{array}\right\} \over ge\vdash(e,Rfor(a_1,a_2,a_3,s))\Rightarrow e_2 \tag{16}$$

eval_Rfor_start 的语义规则为:在全局环境 *ge* 中,从局部环境 *e* 中计算置初值语句 a_1,局部环境变为 e_1;接着执行完整个 *Rfor* 语句,局部环境变为 e_2.

LustreR 层 *Rfor* 语句的 *eval_Rfor_stop* 语义规则:

$$\frac{ge\vdash(eh,a_2)\Rightarrow Vfalse}{ge\vdash(((eh::et),se),Rfor(skip,a_2,a_3,s))\Rightarrow((eh::et),se)} \tag{17}$$

eval_Rfor_stop 的语义规则为:在全局环境 *ge* 中,在局部环境 *eh* 中计算循环控制表达式 a_2 为 *Vfalse*,则结束处理.

LustreR 层 *Rfor* 语句的 *eval_Rfor_loop* 语义规则:

$$\left.\begin{array}{c} ge\vdash(eh,a_2)\Rightarrow Vtrue \\ ge\vdash(((eh::et),se),s)\Rightarrow e_1 \\ ge\vdash(e_1,a_3)\Rightarrow e_2 \\ ge\vdash(e_2,Rfor(skip,a_2,a_3,s))\Rightarrow e_3 \end{array}\right\} \over ge,te\vdash(((eh::et),se),Rfor(skip,a_2,a_3,s))\Rightarrow e_3 \tag{18}$$

eval_Rfor_loop 的语义规则为:在全局环境 *ge* 中,在局部环境 *eh* 中计算循环控制表达式 a_2 为 *Vtrue*.在环境((*eh*::*et*),*se*)等式/语句 *s*,局部环境变为 e_1.在局部环境 e_1 中计算语句 a_3,局部环境变为 e_2.处理完毕后,在局部环境 e_2 中继续执行完整个 *Rfor* 语句,局部环境变为 e_3.

4 高阶消去翻译

如上所述,鉴于高阶运算的特点,我们将 *LustreT* 层高阶运算翻译为 *LustreR* 层 *Rfor* 循环运算,从而实现高阶消去.*Rfor* 语句的语法为 *Rfor*(*stmt*$_1$,*expr*,*stmt*$_2$,*stmt*$_3$),其中,*stmt*$_1$ 为置初值语句,*stmt*$_2$ 为增量语句,*stmt*$_3$ 为主循环体,*expr* 为循环条件表达式.根据 *Rfor* 的语法语义定义,我们需要从高阶语句 *hstmt* 中分别提取出这 4 个部分.我们以图 2 中的 *fold* 程序为例阐述整个翻译算法(代码片段为辅助定理工具 Coq[12]中编写的代码段).

- 置初值语句

在从高价语句中翻译置初值语句时,*map* 运算与 *fold* 运算有所不同:

- *map* 运算的初始状态为循环变量 *loopid* 的初值,定义为

  Definition loop_init:=*assign*(*loopid*,*Tint*32)((*Cint Int.zero*),*Tint*32).

  即,初始状态为循环变量 *loopid*=0,
- 而 *fold* 运算除此之外,还必包含一个输入的初始值,需要从高阶运算表达式中提取:

```
Definition trans_finit(s:hstmt):stmt:=
  match s with
    |Foldbinary lh op init a⇒assign lh init
    |…
```

注:|…表示当前定义还包括对 *s* 的其他构造子的相应处理,限于篇幅,在此不再详述,下同.

assign lh init 意为将初值 *init* 赋于变量 *lh*.

因此,*fold* 运算完整的初始状态为

init:=*seq loop_init*(*trans_finit htmt*).

- 循环条件表达式

对于高阶运算,其循环条件表达式为判断 *loopid* 是否小于数组长度,因此定义为

Definition loop_cond:=*binop Olt*(*loopid*,*Tint*32)((*Cint j*),*Tint*32) *Tbool*.

意为 *loop_cond*(*j*):=*loopid*<*j*.

例如,图 2 中 *fold* 程序的条件语句为 *loop_cond*(5):=*loopid*<5.

- 增量语句:

高阶运算的增量语句是数组或参数列表的索引 *loopid* 每进行一次运算就加 1,因此,增量语句定义为

Definition loop_add:=*assign*(*loopid*,*Tint*32)(*binop Oadd*(*loopid*,*Tint*32)((*Cint Int.one*),*Tint*32)*Tint*32).

意为 *loop_add*:=*loopid*=*loopid*+1.

- 主循环体

foldbinary 运算是对数组进行折叠运算最终生成一个单值,因此,*foldbinary* 的翻译函数为

Definition trans_hstmt(*s*:*hstmt*):*res stmt*:=
  match s with
    |*Foldbinary lh op init a*⇒
      let *es*:=*binop op lh*(*aryacc a*(*loopid*,*Tint*32)(*typeof lh*)) (*typeof lh*) in
    OK (*assign lh es*)
  |…

aryacc 表达式为获取数组 *a* 中第 *loopid* 个元素,将该值与变量 *lh* 进行二元操作 *op*,得到结果 *es*,将结果 *es* 赋予变量 *lh*.

综上所述,高阶消去翻译函数为

Definition trans_stmt(*s*:*LustreT.stmt*):*res stmt*:=
|*Highorder fs j*⇒
  do fs_1←*trans_hstmt fs*;
  let *init*:=*seq loop_init*(*trans_finit fs*) in
  OK (*Rfor init*(*loop_cond j*) *loop_add* fs_1).

5 高阶消去翻译证明

5.1 高阶消去翻译的证明框架

正如语法语义的定义采用的是一种层次结构,证明工作相应的也是一个层次化归纳的证明过程.整个证明框架如图 10 所示.

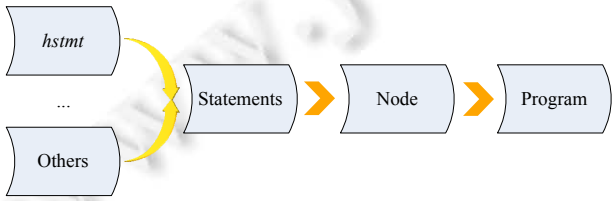


Fig.10 Hierarchy figure of proving

图 10 证明层次图

因此,我们提出了如下定理:

程序翻译正确性定理. *exec_prog_correct_general*(从 *LustreT* 到 *LustreR* 程序的语义等价性):

$$\forall ge,e,mainT,vass,vrss,n,maxn,$$
$$exec\_progT(progT,ge,eval\_nodeT(mainT,e,n,maxn,vass,vrss)) \wedge trans\_node(mainT)=mainR \rightarrow$$
$$exec\_progR(progR,ge,eval\_nodeR(mainR,e,n,maxn,vass,vrss)).$$

其中,

- *progT* 为 *LustreT* 层程序;
- *progR* 为经过 *progT* 翻译到 *LustreR* 层的程序;
- *mainT* 为程序 *progT* 的的主节点;
- *mainR* 为节点 *mainT* 翻译到 *LustreR* 层的节点;
- *trans_node* 函数将节点 *mainT* 翻译到 *LustreR* 层得到节点 *mainR*;
- *eval_nodeT* 和 *eval_nodeR* 分别为 *LustreT* 层和 *LustreR* 层的节点计算函数.

该定理语义等价性的含义是:如果程序 *progT* 在全局环境 *ge* 下对输入实参流 *vass* 的执行结果为流 *vrss*,那么 *progR* 在全局环境 *ge* 下对输入实参流 *vass* 的执行结果同样是流 *vrss*,且 *progT* 和 *progR* 中的局部环境 *e* 在 *n* 个周期中始终保持匹配关系.

LustreT 层中的程序执行函数 *exec_progT* 和 *LustreR* 层的程序执行函数 *exec_progR* 二者的执行过程是相似的,可以理解为周期性主节点的串行执行过程,因此,程序执行的等价性可以转化为节点执行的环境匹配问题,也即节点翻译正确性问题,所以我们需要证明如下定理:

节点翻译正确性定理. *trans_node_all_correct*(从 *LustreT* 到 *LustreR* 节点的语义等价性):

$$\forall ge,e,e',nodeT,vargs,vrets,$$
$$exec\_nodeT(progT,ge,e,nodeT,vargs)=(e',vrets) \wedge trans\_node(nodeT)=nodeR \rightarrow$$
$$exec\_nodeR(progR,ge,e,nodeR,vargs)=(e',vrets).$$

该定理语义等价性的含义是:*nodeT* 经过节点翻译函数 *trans_node* 翻译到 *LustreR* 层得到节点 *nodeR*,如果节点 *nodeT* 在全局环境 *ge* 下对输入实参流 *vargs* 的执行结果为流 *vrets*,局部环境由 *e* 变为 *e*′,那么 *nodeR* 在全局环境 *ge* 下对输入实参流 *vargs* 的执行结果同样为流 *vrets*,局部环境由 *e* 变为 *e*′.

eval_nodeT 和 *eval_nodeR* 分别为 *LustreT* 层和 *LustreR* 层的节点计算函数,二者的执行过程是相似的.节点执行的过程是节点中表达式串行化执行过程,因此,节点执行的等价性可以转换为相应语句 *stmt* 执行的环境匹配问题.

stmt 采用归纳定义,我们根据 *stmt* 的语义定义进行归纳证明.

按照 *stmt* 的构造子的不同,证明目标细分为普通语句、高阶运算以及节点调用这 3 个子目标分别进行证明.

这些子目标的语义保持性都可以描述为:翻译前后的程序语句执行时,对语义环境的改变是一致的.不同语句的程序等价性的表述形式有所不同,本文重点讨论高阶运算语句的语义等价性.

5.2 高阶消去翻译的语义等价性证明

翻译正确性的证明也就是对翻译前后程序的语义等价性的证明,如果翻译前后的程序语义是等价的,则翻译过程是正确的.因此,基于前面描述的语法语义,我们对高阶消去前的 *LustreT* 程序与高阶消去后的 *LustreR* 程序的语义等价性进行证明.

结合前面语义的描述,语义等价性的证明归结起来本质上是在同一全局环境下,高阶消去前后的程序对环境的改变是匹配的,包括时态环境以及相应节点实例的局部环境.

由前面介绍的语义及翻译算法可知,*LustreT* 层的高阶运算表达式翻译到 *LustreR* 层时将分成 3 个部分:初始状态、主循环体以及循环终止状态.因此,结合语义的定义,我们给出了如下的高阶消去正确性定理:

高阶消去正确性定理. 在任意全局环境 *ge* 中,如果满足以下条件,则高阶消去前后的程序是语义等价的:

- $ge \vdash eval\_Highorder\_start(e_1,e_2) \sim eval\_Rfor\_start(e_1,e_2)$;
- $ge \vdash eval\_Highorder\_body(e_1,e_2) \sim eval\_Rfor\_loop(e_1,e_2)$;
- $ge \vdash eval\_Highorder\_stop \sim eval\_Rfor\_stop$.

符号“~”表示左右两个目标等价.例如,程序 p_1 和 p_2 语义等价,写作 $p_1 \sim p_2$.

相应地,该定理的证明从这 3 个部分进行归纳证明:

在介绍这 3 部分证明之前,先介绍 3 个引理:

赋值等价引理. 在任意全局环境 *ge* 中,*LustreT* 层和 *LustreR* 层的赋值语句是等价的:

$$\frac{LustreT.assign\ lh(Expr\ e)=ta, LustreR.assign\ lh\ e=ra}{ge\vdash eval\_stmtT(progT,nk,e_1,e_2,ta)\sim eval\_stmtR(progR,nk,e_1,e_2,ra)}.$$

初值等价引理. 在任意全局环境 *ge* 中,*LustreT* 层和 *LustreR* 层的循环初始值是等价的:

$$ge\vdash eval\_stmtT(progT,nk,e_1,e_2,LustreT.loop\_init)\sim eval\_stmtR(progR,nk,e_1,e_2,LustreR.loop\_init).$$

循环增量等价引理. 在任意全局环境 *ge* 中,*LustreT* 层和 *LustreR* 层的循环增量是等价的:

$$ge\vdash eval\_stmtT(progT,nk,e_1,e_2,LustreT.loop\_add)\sim eval\_stmtR(progR,nk,e_1,e_2,LustreR.loop\_add).$$

eval_stmtT 为 *LustreT* 中表达式计算的语义,*eval_stmtR* 为 *LustreR* 层中表达式计算的语义.应用前面定义的语义执行规则和 Coq 证明策略,可以轻易地证明以上 3 条引理.以引理 1 的证明为例,其证明思路如下:

假设在 *LustreT* 层中,在全局环境 *ge* 和局部环境 e_1 下,*LustreT* 层程序 *progT* 翻译到 *LustreR* 层为程序 *progR*.我们引入条件 *H*:

$$eval\_stmts(progT,nk,e_1,e_2,ta).$$

意为执行 *progT* 程序中的赋值语句 *ta* 后,局部环境变为 e_2.对条件 *H* 进行连续的逆向展开,条件 *H* 变为:将变量 *var* 的值 *v* 存入局部环境 *eh* 中,局部环境变为 *eh'*.然后,对证明目标应用 *assign* 语义规则,证明目标变为当设置变量 *var* 的值为 *v* 时,局部环境由 *eh* 变为 *eh'*.此时,应用条件 *H* 及环境的构造子即可完成证明.

5.2.1 初始状态的证明

将初始状态的等价性进行展开,我们需要证明如下引理:

引理 *trans_finit_correct*. 在任意全局环境 *ge* 中,*LustreT* 层程序 *progT* 翻译到 *LustreR* 层为程序 *progR*,任意高阶运算表达式 *s* 在程序 *progT* 中计算初值时局部环境由 e_1 变为 e_2,那么表达式 *s* 在 *progR* 中计算初值时局部环境也是由 e_1 变为 e_2,即,翻译前后程序对局部环境的改变是匹配的:

$$\frac{initstmtsof\ s=si, trans\_finit\ s=ri}{ge\vdash eval\_stmtT(progT,nk,e_1,e_2,si)\sim eval\_stmtR(progR,nk,e_1,e_2,ri)}.$$

其中,*si* 为从语句 *s* 中提取的 *LustreT* 层计算初值的语句,*ri* 为将语句 *s* 翻译到 *LustreR* 层后计算初值的语句,*eval_stmtT* 和 *eval_stmtR* 对 *LustreT* 层和 *LustreR* 的语句进行计算.*initstmtsof*,*trans_finit* 分别为 *LustreT* 层和 *LustreR* 层求初值函数,二者的定义是:*initstmtsof* 为 *assign lh*(*init*)::*nil*,*trans_finit* 为 *assign lh init*.我们以图 2 中的 *fold* 程序证明为例介绍整个证明思路.

证明思路是:首先对高阶表达式 *s* 进行分解,同时,采用 *initstmtsof* 和 *trans_finit* 的定义进行化简,证明目标细分为 9 个子目标(注:虽然高阶表达式 *s* 有 16 个构造子,但 *map* 运算如 *mapbinary* 等其初值只是循环变量 *loopid*=0,并不经过 *trans_finit* 处理,所以化简后直接排除了 *mapbinary* 等 7 个构造子).*fold* 高阶运算为其中的 *foldbinary* 构造子化简所得的子目标.对该子目标进行证明:对条件 *H*:*eval_stmtT*(*progT*,*nk*,e_1,e_2,*si*)进行连续的逆向展开,此时的证明目标与赋值等价引理相同,应用此引理即可完成证明.

其他高阶运算的证明思路类似.

高阶运算初始状态的证明为

$$ge\vdash eval\_Highorder\_start(e_1,e_2)\sim eval\_Rfor\_start(e_1,e_2).$$

借助于引理 *trans_finit_correct*,证明即可轻松完成,证明思路是:

假设在全局环境 *ge* 中,有条件 *eval_Highorder_start*(e_1,e_2),我们应用 *eval_Highorder_start* 和 *eval_Rfor_start* 的语义规则进行展开,然后应用引理 *trans_finit_correct* 即可完成证明.

5.2.2 循环体的证明

循环体的证明包括两部分:

- 一是 *eval_Highorder_body* 和 *eval_Rfor_loop* 的语义等价性;
- 二是 *LustreT* 层和 *LustreR* 层的循环增量的等价性.

后者应用循环增量等价引理可证;第 1 项的证明稍显复杂,仍以图 2 中 *fold* 程序为例阐述整个证明思路.

证明思路:先分别将 *eval_Highorder_body* 和 *eval_Rfor_loop* 按语义展开后,根据高阶运算的归纳定义,原证明目标依照各高阶运算表达式构造子被细分为多个证明子目标,这些子目标为各构造子翻译的正确性证明.此处我们证明 *foldbinary* 构造子对应的子目标:

$$ge\vdash(e_1,e_2,progT,eval\_hstmt((lid,ty)=foldbinary\ op\ v\ a_1,loopid))\sim(e_1,e_2,progR,eval\_stmt(texp,loopid)).$$

progR 为 *progT* 翻译到 *LustreR* 层的程序,*rexp* 是高阶语句$(lid,ty)=foldbinary\ op\ v\ a_1$经过高阶消去后翻译到 *LustreR* 层所得的表达式.

上式的含义是:在全局环境 *ge* 中,如果执行程序 *progT* 中高阶语句后,局部环境由 e_1 变为 e_2,那么执行程序 *progR* 中相应的表达式 *rexp* 后,局部环境也将由 e_1 变为 e_2.

依据前面介绍的高阶翻译函数 *trans_hstmt* 的定义,在全局环境 *ge* 中,存在条件 *H*:

$$trans\_hstmt((lid,ty)=foldbinary\ op\ v\ a_1)=texp.$$

我们在证明目标中,将 *rexp* 通过条件 *H* 和翻译函数 *trans_hstmt* 的定义进行逆向展开,目标将变为 *R* 层的一个数组赋值表达式计算,此时,应用赋值等价引理即可完成证明.

5.2.3 终止状态的证明

终止状态的证明是直接可得的,直接运用 *eval_Highorder_stop* 和 *eval_Rfor_stop* 的语义规则进行展开,即可完成证明.

6 实现情况

高阶消去工作是整个 L2C 项目中的一个重要组成部分,程序的语法语义定义、算法实现以及证明工作都采用辅助定理工具 Coq 实现,整个工作大约 4 500 行代码,其中,

- 基本框架部分约 1 600 行代码,包括约 700 行定义、78 个定理约 900 行代码.
- 高阶消去的语义部分约 1 400 行代码,包括语法定义 300 行、语义定义 600 行、31 个定理约 500 行.
- 翻译算法部分约 400 行代码,包括算法实现 300 行、8 个定理约 100 行.
- 证明部分约 1 100 行,包含 45 个定理或引理.

表 1 展示了整个代码情况.

Table 1

表 1

		行数	比例(%)
基本框架	定义	700	15.6
	定理及证明	900	20
高阶消去语法语义	语法定义	300	6.7
	语义定义	600	13.3
	定理及证明	500	11.1
高阶消去算法	算法实现	300	6.7
	定理	100	2.2
可信证明	定理及证明	1 100	24.4

如表 1 所示,证明部分占更大的比重.

7 总 结

本文介绍了 Lustre*到 C 子集 Clight 编译中高阶运算消去的翻译算法,并对算法正确性进行了形式化证明.虽然 Lustre*与 Clight 之间有着巨大的语言差异,然而在 L2C 项目合理的框架下,高阶消去集中在 *LustreT* 层到

LustreR 层的翻译中,两层之间的语法语义差异较小,这给我们的整个翻译算法以及证明工作带来了极大的好处,良好的语义定义大幅度减轻了证明的难度.然而,高阶运算本身是比较复杂的,这项工作的难度也是显而易见的,需要细致、扎实的工作,语法、语义以及定理的定义也需要一定的技巧,目标是简洁、易理解以及有利于归纳和证明.本文的工作得益于 L2C 可信编译器框架已有较长时间处于稳定状态.本文的工作将随 L2C 编译器得到实际应用.

致谢 在此,我们向对 L2C 项目组所有成员表示感谢.

References:

[1] 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). IEEE Computer Society Washington, 2005. [doi: 10.1109/MEMCOD.2005.1487884]

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

[3] 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). 1987.

[4] CompCert home page. http://compcert.inria.fr/

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

[6] 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].

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

[8] Lustre V6. The lustre v6 reference manual (draft). 2009. http://wwwverimag.imag.fr/DISTTOOLS/SYNCHRONE/Lustre-v6/doc/lv6-refman.pdf

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

[10] Le Sergent T. SCADE: A comprehensive framework for critical system and software engineering. In: Proc. of the Intergrating System and Software Modeling. LNCS 7083, Berlin, Herdelberg: Springer-Verlag, 2012. 2−3.

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

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

[13] 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]

[14] Gan YK, Zhang LB, Shi G, Wang SY, Dong Y, Zhang ZH, Wang YH. A verified sequentializer for synchronous data-flow programs. Computer Applications and Software, 2014,31(5):1−6.

[15] Pnueli A, Siegel M, Singerman E. Translation validation. In: Proc. of the TACAS'98. LNCS 1384, 1998.

[16] Pnueli A, Shtrichman O, Siegel M. Translation validation for synchronous languages. In: Proc. of the ICALP'98. LNCS 1443, 1998. 235C246.

[17] Biernacki D, Cola_co JL, Hamon G, Pouzet M. Clock-Directed modular code generation for synchronous data-flow languages. In: Proc. of the LCTES. 2008. 121−30. [doi: 10.1145/1375657.1375674]

[18] Auger C, Cola_co JL, 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

[19] Zhang LB, Gan YK, Shi G, Wang SY, Dong Y, Zhang ZH, Wang YH. A certified translation for eliminating temporal feature of a synchronize dataflow program. Computer Engineering and Design, 2014,35(1):137−143.

附中文参考文献:

[13] 石刚,王生原,董渊,嵇智源,甘元科,张玲波,张煜承,王蕾,杨斐.同步数据流程序可信编译器的构造.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]

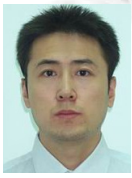
[14] 甘元科,张玲波,石刚,王生原,董渊,张智慧,王沿海.同步数据流程序的可信排序.计算机应用与软件,2014,31(5):1−6.

[19] 张玲波,甘元科,石刚,王生原,董渊,张智慧,王沿海.同步数据流语言时态消去的可信翻译.计算机工程与设计,2014,35(1):137−143.

刘洋(1983－),男,江西新建人,工程师,主要研究领域为编译器形式化验证.

甘元科(1983－),男,硕士,主要研究领域为编译器形式化验证.

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



董渊(1973－),男,博士,副教授,CCF 会员,主要研究领域为嵌入式操作系统,编译系统,基于语言的可信软件.

杨斐(1984－),男,助理工程师,主要研究领域为可信编译器形式化验证.

石刚(1972－),男,博士生,讲师,CCF 学生会员,主要研究领域为系统软件与软件工程,形式化验证,云计算,信息安全.

闫鑫(1989－),男,助理工程师,主要研究领域为形式化验证.