软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2224−2245 [doi: 10.13328/j.cnki.jos.006579]

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

基于 Coq 的矩阵代码生成技术*

麻莹莹, 陈 钢

(南京航空航天大学 计算机科学与技术学院, 江苏 南京 211106)

通信作者: 陈钢, E-mail: gangchensh@nuaa.edu.cn

摘 要: 矩阵程序在智能系统中扮演着越来越重要的角色. 随着矩阵应用的复杂性日益增加, 生成正确矩阵代码的难度也在不断变大. 并行硬件能够极大地提高矩阵运算的速度, 然而, 使用并行硬件进行编程以实现并行运算, 需要编程人员在程序中描述功能以及如何利用硬件资源来交付结果. 这些程序通常是命令式语言, 难以推理并且重构, 以尝试不同的并行化策略. 在 Coq 中实现了由高级矩阵算子到 C 代码的矩阵表达式代码生成技术, 其能够将带有执行策略的函数式矩阵代码转换为高效低级命令式代码. 未来, 将把矩阵的形式化同矩阵代码自动生成融合在一起, 对矩阵代码转换的过程进行形式化验证, 以保障生成的矩阵代码的可靠性, 为实现基于矩阵形式化方法的高可靠性深度学习编译器的研制打下基础.

关键词: 定理证明; 矩阵代码生成; 形式化工程数学; 高阶定理证明; Coq
中图法分类号: TP311

中文引用格式: 麻莹莹, 陈钢. 基于 Coq 的矩阵代码生成技术. 软件学报, 2022, 33(6): 2224–2245. http://www.jos.org.cn/1000-9825/6579.htm

英文引用格式: Ma YY, Chen G. Coq-based Matrix Code Generation Technology. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2224–2245 (in Chinese). http://www.jos.org.cn/1000-9825/6579.htm

Coq-based Matrix Code Generation Technology

MA Ying-Ying, CHEN Gang

(College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

Abstract: Matrix programs are taking increasing important role in the intelligent systems. As the complexity of matrix applications grows, the difficulty of producing correct matrix code does the same. Parallel hardware can greatly increase the speed of matrix operations; nevertheless, using parallel hardware for programming to achieve parallel operations requires programmers to describe functions in the program and to manage how to use hardware resources to deliver results. These programs are usually written in imperative languages that are difficult to reason about and refactor for different parallelization strategies. A matrix expression code generation technology has been implemented from high-level matrix operators to C code in Coq, which can convert functional matrix code with execution strategies into efficient low-level imperative code. In the future, the formal verification of the matrix will be integrated with the automatic generation of matrix code, and formal verification of the matrix code conversion process will be performed to ensure the reliability of the generated matrix code, laying the foundationfor the development of a high-reliability deep learning compiler based on the matrix formal method.

Key words: theorem proving; matrix code generation; formalized engineering mathematics; higher order theorem proving; Coq

矩阵形式化的现实需求之一是矩阵代码的安全性, 它也是软件安全领域中的一个重要研究课题之一. 矩阵计算中存在一些难以彻底解决的安全隐患, 比如下标越界. 此外, 对矩阵计算性能的需求使得矩阵代码日益复杂, 这增加了发生代码错误的风险. 为了挖掘矩阵代码中的缺陷, 人们使用了多种形式化验证技术[1−3], 比如模型检查、符号执行以及基于霍尔逻辑的形式验证等等. 这些形式验证技术的优点是可以自动运行、适

*本文由“定理证明理论与应用”专题特约编辑曹钦翔副教授、詹博华副研究员、赵永望教授推荐.
收稿时间: 2021-09-05; 修改时间: 2021-10-14; 采用时间: 2022-01-04; jos 在线出版时间: 2022-01-28

用面较广, 但这些技术也存在一些问题: 一是它们受到矩阵规模的限制; 二是无法全面验证矩阵性质. 在智能系统工程实践中, 经常会遇到需要证明两个矩阵表达式等价的问题, 这通常是定理证明之外的其他形式化方法难以奏效的[4]. 基于通用定理证明器的验证技术与此不同, 虽然它们不能自动进行矩阵性质验证, 但是它们可以在人机交互的方式下完整地证明矩阵的各种性质, 甚至可以证明矩阵代码的完全正确性.

来自智能计算的需求增加了矩阵计算的复杂性, 由此也产生了更强烈的矩阵性质形式验证的需求. 谷歌公司出品的 Switch Transformer, 其参数规模达到了万亿级别[5]. 深度学习的计算量主要消耗在矩阵运算中, 尤其是矩阵乘法. 为了应对大规模的矩阵计算的需求, 各种硬件设备被开发出来, 如 GPU 和各种 AI 专用处理器, 它们的主要功能是通过并行计算来提高矩阵运算效率. 为了充分利用这些新型硬件的能力, 矩阵代码就需要作相应的改变, 比如用分块矩阵乘法取代矩阵乘法, 以便能够更充分地利用多处理器或 GPU 的并行计算能力. 因此, 对优化前后的矩阵代码等价性形式验证就显得格外重要, 比如关于分块矩阵乘法与矩阵乘法等价性的证明. 对于命令式语言的矩阵乘法代码, 这样的证明是极其困难的; 而在 Coq 中, 这样的证明虽然困难也比较大, 但能够完成. 为了说明这一点, 我们暂时离开矩阵问题, 看一个简单的程序验证问题, 通过这个问题来说明函数式语言为什么比命令式语言更适合作形式化验证.

考察一个 1 到 n 的求和问题. 定义两个函数 $f(n)=1+2+\ldots+n$ 和 $g(n)=n(n+1)$. 如果用命令式语言(C 语言), 具体实现如下.

int f(*int n*) {*int a*=0; **for** (*int i*=1; *i*≤*n*; *i*+=1) {*a*=*a*+*i*;} **return** *a*;}

int g(*int n*) {*int a*=*n*(*n*+1); **return** *a*;}

众所周知, 对于任意的 n 都能得到 $2*f(n)=g(n)$. 然而, 对于这两个命令式函数, 这个性质的形式化证明就较为复杂, 或者使用霍尔逻辑[6], 或者使用指称语义. 下面函数式程序实现的两个函数, 可以直接在 Coq 定理证明器内对自然数 n 进行归纳证明.

Fixpoint *f*(*n*: *nat*): =match *n* with |*O*⩾*O*|*S n*′⇒*f*(*n*′)+*S n*′ **end**.

Fixpoint *g*(*n*): =*n*(*n*+1).

Lemma *fg_eq*: **forall** (*n*: *nat*), 2**f n*=*g n*. 显然, 函数式语言的形式验证更加直接. 事实上, 基于指称语义的命令式语义验证方法相当于把命令式代码转换到函数式代码, 然后再对函数式代码进行形式验证, 而针对函数式代码的证明就绕过了指称语义这一步.

高性能矩阵代码都是命令式代码, 但在定理证明器中建立的矩阵模型通常是函数式的. 在矩阵代码开发过程中, 函数式代码和命令式代码各有优势. 矩阵的函数式描述适合于矩阵数学性质的证明以及矩阵表达式的优化, 神经网络编译器的中间表示形式也具有很强的函数式风格[7,8]. 但函数式语言的矩阵代码效率不高. 高效的矩阵代码依然是命令式语言代码, 主要是 C 代码. Atkey 等人[9]提出了一种把函数式矩阵描述转换到命令式代码的方法, 该方法的原理来自 continuation passing style translation 概念. 我们把这个方法移植到定理证明器 Coq 中, 实现了基于重写的从函数式代码到命令式代码的转换. 在 Coq 中实现该转换, 有助于未来对该转换过程进行形式化验证.

本文的主要贡献是: 在 Coq 中利用重写技术实现了由函数式矩阵代码到 Open MP C 代码的转换技术, 其中包括对高级函数式原语、中间级命令式宏、低级命令式原语以及转换规则的类型定义; 解决了代码生成中变量重名的问题; 扩展了原语以及规则以实现不同矩阵代码的生成; 分析了该技术的形式化验证方案并对点积运算的代码生成过程进行验证. 第 1 节介绍代码生成的相关工作以及技术. 第 2 节详细介绍如何在 Coq 中实现矩阵代码生成技术, 并且对生成的代码进行相应测试. 第 3 节介绍该代码生成技术的形式化验证方案. 第 4 节是工作的总结以及未来展望与挑战.

本文源代码下载地址: https://gitee.com/yingyingma/Matrix/tree/dpiacoq/.

1 相关工作

本节对相关工作进行介绍, 一方面是基于定理证明的编译验证相关工作, 这类工作说明了基于交互式定

理证明器实现大型软件形式化验证的可行性; 另一方面是代码生成技术的相关工作, 这类工作展示了代码生成的工作原理.

1.1 基于定理证明的编译验证工作

Coq[10]是法国国家信息与自动研究所(INRIA)开发的高阶定理证明器, 它使用形式化语言来描述规范和程序, 证明程序的正确性. 它不但表达能力强, 而且具备多个具有实用价值的证明机制, 比如可扩充的重写机制、基于模块的形式化理论构建机制以及程序抽取机制等等. 它是目前国际上主流交互式定理证明工具之一, 具备在高阶逻辑系统中全面验证软件性质的强大能力.

基于定理证明的形式化方法在系统开发方面取得了一系列重要的突破. 产生了以编译器和操作系统内核验证为代表的诸多优秀工作, 例如 CompCert[11]、seL4[12]等. CompCert 编译器是经过形式化验证的可信编译器的杰出代表, 其在高阶定理证明器内部证明了编译多个阶段的翻译规则, 严格证明了目标代码与源程序的等价性. 清华大学的 L2C 项目[13]从高级建模语言(类 Lustre 的同步数据流语言)到 C 语言的可信翻译, 并与 CompCert 编译器衔接, 形成了从建模语言到汇编语言的可信编译工作. 这些研究工作充分展示了基于交互式定理证明器实现大型软件形式化验证的可行性. Yang 等人[14]报告了他们用 6 年时间检查各种在用编译器(包括 VC、GCC 等)错误的工作, 只有当时版本的 CompCert 验证过的部分未发现错误, 其他广泛使用的编译器都发现了错误, 这也说明了利用形式化方法开发的软件的可信度非常高.

Tanaka 提出了一种带有部分评估的、将 Gallina 程序翻译为 C 程序的编译技术[15], 它能够提高执行效率和消除多态性和依赖类型. 另外, 该技术的编译过程在 Coq 中得到了验证. 但该技术中使用表类型表示矩阵, 翻译的矩阵算法使用 C 语言中的链表结构实现, 因此运算效率不高.

1.2 函数式程序到命令式程序的代码生成技术

并行硬件为矩阵计算的性能带来了很大的提升, 使用并行硬件的程序通常用命令式语言来编写, 但命令式程序难以进行完整的形式化验证, 并且程序难以通过等价变换来尝试不同的并行化策略. 近年来, 出现了许多“智能编译器”, 它们能够使用由高阶函数(抽象组合子)组成的高级语言指定所需要的程序, 通过编译器生成高效的低级命令式代码. 这类技术可以将执行策略与函数式代码一起使用或者直接在函数式代码内部使用. Halide[16]是一个将策略与程序区分开来的例子, 它首次引入将描述程序优化的调度与算法相分离的概念, 许多其他框架在机器学习(TVM)[7]、图应用(Graphlt)[17]和多面体编译(Tiramisu)[18]等领域采用了这一概念, Lift[19]和 Data Parallel Idealized Algol (DPIA)[9]是将策略嵌入函数式代码中的例子, RISE[20]在 Lift 的基础上发展并提出了比 TVM 更为简洁、有效的策略优化技术. DPIA 是一个纯函数式的由高级函数式代码生成高效并行低级代码的技术. 将策略嵌入于函数式代码中的好处在于, 我们可以更好地掌握对于生成代码的优化. Clement Pit-Claudel 提出了在定理证明器 Coq 中开发的一种提取和编译在嵌入式特定领域语言的方法, 该方法提供了从 SQL 风格的关系程序到可执行汇编代码的生成证明的自动转换[21]. CertiCoq 是经过机械验证的 Coq 优化编译器, 它弥合了经过认证的高级程序与其机器语言翻译之间的差距. CertiCoq 是 Gallina 的编译器, 它以 Clight(C 语言的子集)为目标, 可以使用任何 C 编译器进行编译, 包括 CompCert 验证的编译器[22].

选择 DPIA 作为实现模型的原因在于: 它是由纯函数式语言实现的, 有利于在定理证明器中进行形式化验证. DPIA 是对 Syntactic Control of Inference Revisited[23]的扩展, 其中包括索引类型、数组、元组数据类型等等. DPIA 是一个策略保留的并行函数式代码编译方法, 它描述了一种从带有并行策略的高级函数式代码到并行命令式代码的转换, 具有很强的灵活性, 可以通过不断扩展后端来产生针对不同并行硬件的高效代码.

DPIA 的代码生成技术主要分为 3 个步骤:

高级函数式原语→中间级命令式宏→低级命令式原语→低级并行代码.

下面是一个向量点积的例子, 向量 *xs* 和 *ys* 的点积运算可以用基于组合子的函数式代码表示为

$$\textbf{\textit{reduce}}\ (+)\ 0\ (\textbf{\textit{map}}(\lambda x.\textbf{\textit{Fst}}\ x * \textbf{\textit{Snd}}\ x)\ (\textbf{\textit{zip}}\ xs\ ys)) \tag{1}$$

其中, ***reduce***、***map***、***zip*** 是对向量的基本操作. *zip* 是将两个向量组合成一个元素为对偶的向量, 如 ***zip*** [1;2;3] [4;

5;6]=[(1,4);(2,5);(3,6)]; ***map*** 是将一个函数作用于向量中的每个元素上, 如 ***map***(+1) [1;2;3]=[1+1;2+1;3+1]=[2;3;4]; ***reduce*** 是根据一个初始值将向量中的每个元素通过一个函数累积作用, 如 ***reduce*** 0 (+) [1;2;3]=0+1+2+3. 这一表示形式不适用于像 OpenMP 或 OpenCL 这样的编译目标, 因为它们的表达语言中既不包括高阶函数也不包括组合子函数. 点积计算可以通过并行执行, 一种执行方式是先并行执行 *xs* 和 *ys* 的成对乘法, 然后顺序累加中间结果.

$$\textbf{reduceSeq}\ (+)\ 0\ (\textbf{mapPar}\ (\lambda x.\textbf{Fst}\ x * \textbf{Snd}\ x)\ (\textbf{zip}\ xs\ ys)) \tag{2}$$

其中, ***reduceSeq*** 表示顺序执行的 ***reduce***, ***mapPar*** 表示并行执行的 ***map***. 这段程序表示先对两个数组实现 ***zip*** 操作形成一个以对偶为元素的数组, 然后对该数组进行并行的 ***map*** 映射, 最后对映射完的结果进行顺序执行的 ***reduce*** 操作. 虽然这并不是执行点积运算最好的策略, 但通过这个例子我们可以看到, 如何在函数式代码中将并行化策略翻译为相应的命令式代码. 函数式的矩阵代码通常以组合子嵌套的形式呈现, 组合子按照外层后、内层先的顺序执行. 然而, 命令式的矩阵代码通常是按照命令顺序执行, 那么如何将函数式的矩阵代码转换到命令式的矩阵代码? Continuation Passing Style (CPS)的函数式程序能够有效解决这个转换问题, CPS 会在函数 *f* 接受的原有参数的基础上, 多接受一个函数 *g* 以λ抽象形式呈现, 这个函数 *g* 在函数 *f* 计算完成后被调用.

当输入用带有执行策略的函数式矩阵程序后, 该技术能够通过上述 3 个转换步骤生成相应执行策略的命令式矩阵程序. 第 1 个步骤的实现主要依赖于 Acceptor-passing 和 Continuation-passing 这两个翻译器以及与其相关的翻译规则, 在翻译规则中, 这两个翻译器是相互定义的.

下面使用 ***A*** 表示 Acceptor-passing, ***C*** 表示 Continuation-passing. $\textbf{A}(e)_{\delta}(out)$和 $\textbf{C}(e)_{\delta}(c)$是命令式程序, $\textbf{A}(e)_{\delta}(out)$中, δ为数据类型, *e* 为δ类型的表达式, *out* 为赋值对象, 该语句等价于赋值语句 *out*: =*e*; $\textbf{C}(e)_{\delta}(c)$中, δ 为数据类型, *e* 为 δ 类型的表达式, *c* 为作用于该表达式上的后续执行命令, 该语句等价于*c e*. 下面是对上述公式(2)点积的一个部分翻译示例, 该翻译将待翻译的函数式矩阵表达式和输出变量作为输入传送至翻译器 ***A*** 中. 待翻译的矩阵表达式为 ***reduceSeq*** (+) 0 (***mapPar*** (λ*x*.***Fst*** *x*****Snd*** *x*) (***zip*** *xs ys*)), 这个表达式先对两个数组 *xs* 和 *ys* 进行 ***zip*** 操作, 然后将得到的结果进行 ***mapPar*** 操作, 最后进行 ***reduceSeq*** 操作. 翻译的过程分为 3 步: 首先, 将每个组合子依次翻译为由中间级命令式宏组成的程序, 中间级命令式宏通过低级命令式原语实现; 其次, 通过展开利用低级命令式原语实现的中间级命令式宏得到函数式结构描述的命令式代码; 最后, 将其通过显式翻译转换成命令式代码. 下面主要分析前两个步骤的翻译.

$$\textbf{A}(\textbf{reduceSeq}\ (+)\ 0\ (\textbf{mapPar}\ (\lambda x.\textbf{Fst}\ x * \textbf{Snd}\ x)\ (\textbf{zip}\ xs\ ys)))_{\textbf{num}}(out)$$

$$=\textbf{C}(\textbf{mapPar}\ (\lambda x.\textbf{Fst}\ x * \textbf{Snd}\ x)\ (\textbf{zip}\ xs\ ys))_{\textbf{n.num}}$$

$$(\lambda x.\textbf{C}(0)_{\textbf{num}}(\lambda y.\textbf{reduceSeql}\ n(\lambda x\ y\ o.\textbf{A}(x+y)_{\textbf{num}}(o))\ y\ x(\lambda r.\ \textbf{A}(r)_{\textbf{num}}(out)))) \tag{a.9}$$

这一步翻译使用了(a.9)翻译规则, 该规则将最后执行的 ***reduceSeq*** 组合子翻译至后续调用函数之中, 并以中间命令式宏 ***reduceSeql*** 来实现, 其中, ***reduceSeql*** 是利用低级命令式原语实现的 ***reduce*** 操作, 该翻译步骤将高级函数式原语 ***reduceSeq*** 翻译为中间级命令式宏 ***reduceSeql***.

$$=\textbf{new}_{(\textbf{n.num})}(\lambda tmp.\textbf{A}(\textbf{mapPar}\ (\lambda x.\textbf{Fst}\ x * \textbf{Snd}\ x)\ (\textbf{zip}\ xs\ ys))_{\textbf{n.num}}(tmp.1);$$

$$\textbf{C}(0)_{\textbf{num}}(\lambda y.\textbf{reduceSeql}\ n(\lambda x\ y\ o.\textbf{A}(x+y)_{\textbf{num}}(o))\ y\ tmp.2(\lambda r.\textbf{A}(r)_{\textbf{num}}(out)))) \tag{c.5}$$

该步骤的翻译使用了(c.5)翻译规则, *new* 声明了一个长度为 *n* 的临时数组变量, 其中存储着 ***map*** (λ*x*.***Fst*** *x*****Snd*** *x*) (***zip*** *xs ys*)的结果, 该结果将后续在 ***reduceSeql*** 中调用. 接着, 我们需要将高级函数式原语 ***mapPar*** 翻译为中间级命令式宏 ***mapParl***. 我们对该程序内部的翻译分为两部分, 下面是第 1 部分程序的翻译.

$$\textbf{A}(\textbf{mapPar}\ (\lambda x.\textbf{Fst}\ x * \textbf{Snd}\ x)\ (\textbf{zip}\ xs\ ys))_{\textbf{n.num}}(tmp.1)$$

$$=\textbf{C}(\textbf{zip}\ xs\ ys)_{\textbf{n.num*num}}(\lambda x.\textbf{mapParl}\ n(\lambda y\ o.\textbf{A}(\textbf{Fst}\ y * \textbf{Snd}\ y)(o))\ x\ tmp.1) \tag{a.8}$$

$$=\textbf{C}(xs)_{\textbf{n.num}}(\lambda x1.\textbf{C}(ys)_{\textbf{n.num}}(\lambda y1.(\lambda x.\textbf{mapParl}\ n(\lambda y\ o.\textbf{A}(\textbf{Fst}\ y * \textbf{Snd}\ y)_{\textbf{num}}\ (o))\ x\ tmp.1)\ (\textbf{zip}\ x1\ y1)) \tag{c.8}$$

$$=\textbf{mapParl}\ n(\lambda y\ o.\textbf{A}(\textbf{Fst}\ y * \textbf{Snd}\ y)_{\textbf{num}}(o))\ (\textbf{zip}\ xs\ ys)\ tmp.1 \tag{c.1}$$

这一步骤先后使用了(a.8)、(c.8)翻译规则, 将 ***mapPar*** 组合子翻译至后续调用函数之中, 并以中间命令式

宏 ***mapParl*** 来实现, 其中, ***mapParl*** 是利用低级命令式原语实现的 ***map*** 操作, 然后将 ***zip*** 组合子翻译至后续调用函数之中, 最后, 将 ***C*** 翻译器所执行的命令依次作用于表达式上. 其中, $\boldsymbol{A}(\boldsymbol{Fst}\ y*\boldsymbol{Snd}\ y)_{\boldsymbol{num}}(o)$的翻译如下.

$\boldsymbol{A}(\boldsymbol{Fst}\ y*\boldsymbol{Snd}\ y)_{\boldsymbol{num}}(o)$

$=\boldsymbol{C}(\boldsymbol{Fst}\ y)(\lambda x2.\boldsymbol{C}(\boldsymbol{Snd}\ y)(\lambda y2.o: =x2+y2))$ (a.6)

$=\boldsymbol{C}(y)\ (\lambda x3.(\lambda x2.C(\boldsymbol{Snd}\ y)(\lambda y2.o: =x2+y2))\ (\boldsymbol{Fst}\ x3))$ (c.12)

$=\boldsymbol{C}(y)\ (\lambda x3.(\lambda x2.C(y)(\lambda y3.(\lambda y2.o: =x2+y2)\ \boldsymbol{Snd}\ y3))\ (\boldsymbol{Fst}\ x3))$ (c.13)

$=o: =\boldsymbol{Fst}\ y*\boldsymbol{Snd}\ y$ (c.1)

这一步骤使用了(a.6)、(c.12)、(c.13)和(c.1)转换规则, 将布局操作函数 ***Fst*** 和 ***Snd*** 翻译至后续调用函数中, 然后利用 $\boldsymbol{C}(e)_{\&}(c)=c\ e$ 原理将后续作用函数依次作用于表达式上, 最后生成了一个赋值语句.

$\boldsymbol{A}(\boldsymbol{mapPar}\ (\lambda x.\boldsymbol{Fst}\ x*\boldsymbol{Snd}\ x)\ (\boldsymbol{zip}\ xs\ ys))_{\boldsymbol{n.num}}(tmp.1)$

$=\boldsymbol{mapParl}\ n(\lambda y\ o.o: =\boldsymbol{Fst}\ y*\boldsymbol{Snd}\ y)\ (\boldsymbol{zip}\ xs\ ys)\ tmp.1$

第 1 部分的翻译已完成, 该部分的程序主要由中间命令式原语以及命令构成.

下面是第 2 部分的翻译.

$\boldsymbol{C}(0)_{\boldsymbol{num}}(\lambda y.\boldsymbol{reduceSeql}\ n(\lambda x\ y\ o.\boldsymbol{A}(x+y)_{\boldsymbol{num}}(o))\ y\ tmp.2(\lambda r.\boldsymbol{A}(r)_{\boldsymbol{num}}(out)))$

$=\boldsymbol{reduceSeql}\ n(\lambda x\ y\ o.\boldsymbol{A}(x+y)_{\boldsymbol{num}}(o))\ 0\ tmp.2(\lambda r.\boldsymbol{A}(\mathrm{r})_{\boldsymbol{num}}(out))$

$\boldsymbol{A}(x+y)_{\boldsymbol{num}}(o)=\boldsymbol{C}(x)_{\boldsymbol{num}}(\lambda x1.\boldsymbol{C}(y)_{\boldsymbol{num}}(\lambda y1.o: =x1+y1))=o: =x+y$ (a.5) (c.1)

$\boldsymbol{A}(r)_{\boldsymbol{num}}(out)=out: =r$ (a.1)

$\boldsymbol{C}(0)_{\boldsymbol{num}}(\lambda y.\boldsymbol{reduceSeql}\ n(\lambda x\ y\ o.\boldsymbol{A}(x+y)_{\boldsymbol{num}}(o))\ y\ tmp.2(\lambda r.\boldsymbol{A}(r)_{\boldsymbol{num}}(out)))$

$=\boldsymbol{reduceSeql}\ n(\lambda x\ y\ o.\ o: =x+y)\ 0\ tmp.2(\lambda r.out: =r)$ (c.1)

经过上述翻译, 程序已经转换为中间命令式宏的实现.

$\boldsymbol{A}(\boldsymbol{reduceSeq}\ (+)\ 0\ (\boldsymbol{mapPar}\ (\lambda x.\boldsymbol{Fst}\ x*\boldsymbol{Snd}\ x)\ (\boldsymbol{zip}\ xs\ ys)))_{\boldsymbol{num}}(out)$

$=\boldsymbol{new}_{\boldsymbol{(n.num)}}(\lambda tmp.\boldsymbol{mapParl}\ n(\lambda y\ o.\ o: =\boldsymbol{Fst}\ y*\boldsymbol{Snd}\ y)\ (\boldsymbol{zip}\ xs\ ys)\ tmp.1;$

$\boldsymbol{reduceSeql}\ n(\lambda x\ y\ o.\ o: =x+y)\ 0\ tmp.2(\lambda r.out:=r))$

这段程序的整体含义分为两部分: 第 1 部分是对向量 *xs*、*ys* 进行 ***zip*** 操作, 并将其结果做一个映射(映射函数为对偶分量相乘), 然后将该数组结果赋值于局部数组变量 *tmp* 中; 第 2 部分是对局部数组变量 *tmp* 进行一个 ***reduce*** 操作, 并将最终计算结果赋值给 *out*. 下面是第 2 步骤的翻译, ***reduceSeql*** 和 ***mapParl*** 是由低级命令式宏实现的中间命令式宏, 因此可以直接将其定义展开(具体定义详见第 2.1 节).

$=\boldsymbol{new}_{\boldsymbol{(n.num)}}(\lambda tmp.\boldsymbol{parfor}\ n\ tmp.1(\lambda i\ o.o: =\boldsymbol{Fst}(\boldsymbol{fun\_idx}(\boldsymbol{zip}\ xs\ ys)\ i)*\boldsymbol{Snd}(\boldsymbol{fun\_idx}(\boldsymbol{zip}\ xs\ ys)\ i));$

$\boldsymbol{new}_{\boldsymbol{num}}(\lambda accum.accum.1: =0;\ \boldsymbol{seqfor}\ n(\lambda i.accum.1: =accum.2+idx\ tmp.2\ i);\ out: =accum.2))$

这是命令式程序的函数式表示, 其中, ***parfor*** 是并行执行的 for 循环命令, ***seqfor*** 是顺序执行的 for 循环命令, ***fun_idx*** 为获取数组表达式元素的原语, *accum*.1 与 *accum*.2 分别为 *accum* 左值和右值. 此时我们可以看到: 程序已经基本转换为函数式描述的命令式程序, 该程序首先声明一个长度为 *n* 的临时数组变量, 用于存放用并行 for 循环实现的 ***map*** 和 ***zip*** 操作的计算结果; 然后声明一个数的临时变量, 作为用 for 循环实现的 ***reduce*** 操作的临时变量; 最后使用第 3 步骤的重写规则, 将这些低级命令式原语直接翻译为命令式程序(OpenMP C 代码). 具体的翻译过程在第 2.3 节详细加以介绍. 其中, #pragma omp parallel for 表示并行执行的 for 循环.

```
float tmp[n];
#pragma omp parallel for
for (int i=0; i<n; i+=1)
  tmp[i]=xs[i]*ys[i];
float accum=0;
for (int i=0; i<N; i+=1)
  accum=accum+tmp[i];
```

out=accum;

2 在 Coq 中实现的矩阵代码生成技术

本节详细介绍了如何在 Coq 定理证明器内实现矩阵代码生成技术, 分析了一个代码生成例子的过程, 并对相应规则进行了扩展.

2.1 类型、原语的定义

首先, 我们需要在 Coq 中定义类型部分, 主要分为数据、短语、命令这 3 种类型. 数据类型具有 4 种, 分别是数值(***num***)、索引(***idx***)、数组(***ary***)以及对偶(***tensor***)类型. 短语类型分为两种, 分别是左值(***acc***)和右值(***exp***). DPIA 技术中的类型设计较为复杂, 出于逻辑一致性的需要, Coq 中定义的函数都必须是可终止的, 这对归纳数据结构有特殊的要求和限制, 使得一些归纳结构无法实现. 参考 DPIA 的类型设计原理并对其进行修改, 我们的类型以如下方式实现(如图 1 所示).

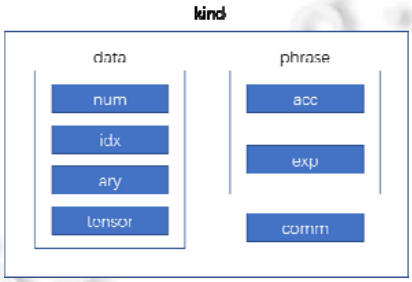

图 1 类型定义

Definition ***kinds***: *Type*: =*Type*.
Inductive ***data***: *kinds*: =
|***num***: *data*
|***idx***: *nat*→*data*
|***ary***: *nat*→*data*→*data*
|***tensor***: *data*→*data*→*data*.
Definition ***phrase***: *kinds*: =*Type*.
Parameter ***exp***: *data*→*phrase*.
Parameter ***acc***: *data*→*phrase*.

其中, ***exp*** 为表达式类型, ***acc*** 为接受器类型.

函数式原语分为 5 类.

- 第 1 类是标量操作原语, 主要用于对标量的计算;
- 第 2 类是不带执行策略的高级函数式原语, 用于处理数组数据, 如 ***map*** 和 ***reduce***, 该类原语不用作描述函数式矩阵表达式的输入;
- 第 3 类是带有执行策略的高级函数式原语, 用于处理数组数据, 其中, ***mapSeq*** 和 ***mapPar*** 分别是 ***map*** 的顺序执行和并行执行的函数式原语, ***reduceSeq*** 为 ***reduce*** 顺序执行的函数式原语. 带执行策略的函数式原语在语义上与不带执行策略的高级函数式原语等价, 但在翻译中, 翻译者通过输入带执行策略的函数式原语描述的函数式矩阵程序来得到相应的命令式矩阵程序. 例如: 使用 ***mapSeq*** 描述的数组映射表达式, 则会翻译得到顺序 for 循环实现的数组映射命令式程序; 使用 ***mapPar***, 则会通过翻译得到并行 for 循环实现的命令式程序;
- 第 4 类是对数据布局的函数式原语, ***zip*** 将两个长度相等的数组转换为元素为对偶的数组, ***split*** 和 ***join***

分别是对数组的切割和合并操作, ***fun_idx*** 为在数组表达式中查找一个值. 下文中, 在数组表达式 *a* 中获取第 *i* 个元素记作 *a*[*i*], ***Pai*** 构造对偶, ***Fst*** 和 ***Snd*** 用于解构对偶;

- 第 5 类是数据布局函数式原语的变体, 主要用于将接收器通过显式索引计算转换为左值, 其作用于接收器类型之上.

标量操作原语:

neg: (*exp num*)→(*exp num*).

add mul: (*exp num*)→(*exp num*)→(*exp num*).

zero one: *exp num*.

不带执行策略的高级函数式原语:

map: **forall** (*n*: *nat*) (*s t*: *data*), ((*exp s*)→(*exp t*))→(*exp*(*ary n s*))→(*exp*(*ary n t*)).

reduce: **forall** (*n*: *nat*) (*s t*: *data*), ((*exp s*)→(*exp t*)→(*exp t*))→(*exp t*)→(*exp*(*ary n s*))→(*exp t*).

带有执行策略的高级函数式原语:

mapSeq: **forall** (*n*: *nat*) (*s t*: *data*), ((*exp s*)→(*exp t*))→(*exp*(*ary n s*))→(*exp*(*ary n t*)).

mapPar: **forall** (*n*: *nat*) (*s t*: *data*), ((*exp s*)→(*exp t*))→(*exp*(*ary n s*))→(*exp*(*ary n t*)).

reduceSeq: **forall** (*n*: *nat*) (*s t*: *data*), ((*exp s*)→(*exp t*)→(*exp t*))→(*exp t*)→(*exp*(*ary n s*))→(*exp t*).

数据布局的函数式原语:

zip: **forall** (*n*: *nat*) (*s t*: *data*), (*exp*(*ary n s*))→(*exp*(*ary n t*))→(*exp*(*ary n*(*tensor s t*))).

split: **forall** (*n m*: *nat*) (*t*: *data*), (*exp*(*ary* (*n***m*) *t*))→(*exp*(*ary m*(*ary n t*))).

join: **forall** (*n m*: *nat*) (*t*: *data*), (*exp*(*ary m*(*ary n t*)))→(*exp*(*ary*(*n***m*) *t*)).

fun_idx: **forall** (*n*: *nat*) (*t*: *data*), (*exp*(*ary n t*))→(*exp*(*idx n*))→(*exp t*).

Pair: **forall** {*s t*: *data*}, (*exp s*)→(*exp t*)→(*exp*(*tensor s t*)).

Fst: **forall** {*s t*: *data*}, (*exp*(*tensor s t*))→(*exp s*).

Snd: **forall** {*s t*: *data*}, (*exp*(*tensor s t*))→(*exp t*).

数据布局函数式原语变体:

zipAcc1: **forall** (*n*: *nat*) (*s t*: *data*), (*acc*(*ary n*(*tensor s t*)))→*acc*(*ary n s*).

zipAcc2: **forall** (*n*: *nat*) (*s t*: *data*), (*acc*(*ary n*(*tensor s t*)))→*acc*(*ary n t*).

idxAcc: **forall** (*n*: *nat*) (*t*: *data*), (*acc*(*ary n t*))→(*exp*(*idx n*))→(*acc t*).

splitAcc: **forall** (*n m*: *nat*) (*t*: *data*), (*acc*(*ary m*(*ary n t*)))→(*acc*(*ary*(*n***m*) *t*)).

joinAcc: **forall** (*n m*: *nat*) (*t*: *data*), (*acc*(*ary*(*n***m*) *t*))→(*acc*(*ary m*(*ary n t*))).

pairAcc1: **forall** (*s t*: *data*), (*acc*(*tensor s t*))→(*acc s*).

pairAcc2: **forall** (*s t*: *data*), (*acc*(*tensor s t*))→(*acc t*).

其中, ***idxAcc*** 为获取数组接收器的某个元素, 在下文中, 数组接收器 *a* 的第 *i* 个元素记作 *a*{*i*}.

命令式原语中包含的命令如下:

Inductive ***comm***: *Type*: =

|***seq***: *comm***comm*→*comm*

|***skip***: *comm*

|***assign***: **forall** (*d*: *data*), (*acc d*)*(*exp d*)→*comm*

|***new***: **forall** (*t*: *data*), ((*acc t*)*(*exp t*)→*comm*)→*comm*

|***seqfor***: **forall** (*n*: *nat*), ((*exp*(*idx n*))→*comm*)→*comm*

|***parfor***: **forall** (*n*: *nat*) (*t*: *data*), (*acc*(*ary n t*))→((*exp*(*idx n*))→(*acc t*)→*comm*)→*comm*.

其中, ***seq***(;)为顺序执行命令, ***skip*** 不做任何操作, ***new*** 用于变量声明, ***assign***(: =)是赋值语句. 循环由顺序循环 ***seqfor*** 和并行循环 ***parfor*** 构建而成, 其中, 并行 for 循环 ***parfor*** 与通常的顺序 **for** 循环的不同在于: 它需

要声明一个数组型变量, 该变量用于每次迭代的输出, 这能够保证 ***parfor*** 的实现是安全的.

在完成命令式原语定义的基础上定义中间级命令式宏, 主要有 3 个, 分别是 ***mapParl***、***mapSeql*** 和 ***reduceSeql***, 它们的定义通过低级的 for 循环实现.

Definition ***mapParl*** (*n*: *nat*) (*s t*: *data*) (*f*: (*exp s*)→(*acc t*)→*comm*) (*xs*: (*exp* (*ary n s*))) (*out*: (*acc* (*ary n t*))): = ***parfor*** *n t out*(λ*i o*.⇒*f*(*xs*[*i*]) *o*).

Definition ***mapSeql*** (*n*: *nat*) (*s t*: *data*) (*f*: (*exp s*)→(*acc t*)→*comm*) (*xs*: (*exp*(*ary n s*))) (*out*: (*acc*(*ary n t*))): =
seqfor *n*(λ*i*.*f*(*xs*[*i*]) (*out*{*i*})).

并行和顺序执行的 *map* 函数定义的区别在于, 其中使用的循环分别是并行和顺序的 for 循环.

Definition ***reduceSeql*** (*n*: *nat*) (*d*1 *d*2: *data*) (*f*: (*exp d*1)→(*exp d*2)→(*acc d*2)→*comm*)(*init*: *exp d*2)
(*xs*: *exp*(*ary n d*1)) (*c*: *exp d*2→*comm*): =
new *d*2(λ*ac*.let(*vw*,*vr*): =*ac* in *vw*: =$_{d2}$ *init*; ***seqfor*** *n*(λ*i*.*f*(*xs*[*i*]) *vr vw*); *c vr*).

顺序执行的 *reduce* 操作函数定义接受一个操作函数 *f*、初始值 *init*、数组 *xs* 和后续命令 *c*, 首先声明一个局部变量 *ac*, 将初始值 *init* 赋值于 *ac*, 然后通过一个 for 循环来实现数组每个变量与 *ac* 之间的操作, 最后将后续命令 *c* 作用于 *ac* 值上.

2.2 转换规则的定义

翻译步骤主要分为 3 个步骤: 第 1 步是将高级函数式原语翻译为中间命令式宏, 第 2 步是将中间命令式宏翻译为低级命令式原语组成的程序(通过展开中间命令式宏的定义即可完成), 第 3 步是将低级命令式原语进行显式翻译. 翻译规则主要分为两类: 第 1 类为由 Acceptor-passing 和 Continuation-passing 相互定义的转换规则, 主要用以第 1 步的翻译将函数式表达式转换为命令式代码; 第 2 类是命令式程序生成的翻译规则, 主要用于第 3 步的翻译. 与 DPIA 不同的是: 本次实现通过修改命令显式生成规则解决了代码生成中变量重名的问题, 扩展了原语以及相应规则以实现不同矩阵代码的生成.

1. 第 1 步的翻译规则

Acceptor-passing 翻译器 $\boldsymbol{A}(e)_{\delta}(a)$等价于一个赋值语句, 其中, *e* 为赋值的表达式; *a* 为被赋值的对象, 即接收器.

Acceptor-passing 规则.

(a.1) $\boldsymbol{A}(e)_{num}(a)=a: =_{num}e$

(a.2) $\boldsymbol{A}(e)_{n.\delta}(a)=\boldsymbol{mapParl}_{n\ \delta\ \delta}(\lambda x\ a.a: =_{\delta}x)\ e\ a$

(a.3) $\boldsymbol{A}(e)_{\delta 1\times\delta 2}(a)=\boldsymbol{pairAcc1}\ a: =_{\delta 1}\boldsymbol{Fst}\ e; \boldsymbol{pairAcc2}\ a: =_{\delta 2}\boldsymbol{Snd}\ e$

(a.4) $\boldsymbol{A}(\boldsymbol{neg}\ e)_{num}(a)=\boldsymbol{C}(e)_{num}(\lambda x.a: =_{num}(\boldsymbol{neg}\ x))$

(a.5) $\boldsymbol{A}(e1+e2)_{num}(a)=\boldsymbol{C}(e1)_{num}(\lambda x.\boldsymbol{C}(e2)_{num}(\lambda y.a: =_{num}(x+y))$

(a.6) $\boldsymbol{A}(e1*e2)_{num}(a)=\boldsymbol{C}(e1)_{num}(\lambda x.\boldsymbol{C}(e2)_{num}(\lambda y.a: =_{num}(x*y))$

(a.7) $\boldsymbol{A}(\boldsymbol{mapSeq}_{n,\delta 1,\delta 2}\ f\ e)_{n.\delta 2}(a)=\boldsymbol{C}(e)_{n.\delta 1}(\lambda x.\boldsymbol{mapSeql}_{n,\delta 1,\delta 2}(\lambda y.(\lambda o.\boldsymbol{A}(f\ y)_{\delta 2}))\ x\ a)$

(a.8) $\boldsymbol{A}(\boldsymbol{mapPar}_{n,\delta 1,\delta 2}\ f\ e)_{n.\delta 2}(a)=\boldsymbol{C}(e)_{n.\delta 1}(\lambda x.\boldsymbol{mapSeql}_{n,\delta 1,\delta 2}(\lambda y.(\lambda o.\boldsymbol{A}(f\ y)_{\delta 2}))\ x\ a)$

(a.9) $\boldsymbol{A}(\boldsymbol{reduceSeq}_{n,\delta 1,\delta 2}\ f\ i\ e)_{n.\delta 2}(a)$
$=\boldsymbol{C}(e)_{n.\delta 1}(\lambda x.\boldsymbol{C}(i)\ (\lambda y.\boldsymbol{reduceSeql}_{n,\delta 1,\delta 2}(\lambda x'.(\lambda y'.(\lambda o.\boldsymbol{A}(f\ x'\ y')_{\delta 2}))\ y\ x(\lambda r.\boldsymbol{A}(r)_{\delta 2}(a)))$

(a.10) $\boldsymbol{A}(\boldsymbol{zip}\ e1\ e2)_{n,\delta 1,\delta 2}(a)=\boldsymbol{A}(e1)_{n.\delta 1}(\boldsymbol{zipAcc1}\ a); \boldsymbol{A}(e2)_{n.\delta 1}(\boldsymbol{zipAcc2}\ a)$

(a.11) $\boldsymbol{A}(\boldsymbol{split}_{n,m,\delta}\ e)_{n.m.\delta}(a)=\boldsymbol{A}(e)_{n*m.\delta}(\boldsymbol{splitAcc}_{n,m,\delta}\ a)$

(a.12) $\boldsymbol{A}(\boldsymbol{join}_{n,m,\delta}\ e)_{n*m.\delta}(a)=\boldsymbol{A}(e)_{n.m.\delta}(\boldsymbol{joinAcc}_{n,m,\delta}\ a)$

(a.13) $\boldsymbol{A}(\boldsymbol{Pair}_{\delta 1,\delta 2}\ e1\ e2)_{\delta 1\times\delta 2}(a)=\boldsymbol{A}(e1)_{\delta 1}(\boldsymbol{pairAcc1}_{\delta 1,\delta 2}\ a); \boldsymbol{A}(e2)_{\delta 2}(\boldsymbol{pairAcc2}_{\delta 1,\delta 2}\ a)$

(a.14) $\boldsymbol{A}(\boldsymbol{Fst}_{\delta 1,\delta 2}\ e)_{\delta 1}(a)=\boldsymbol{C}(e)_{\delta 1\times\delta 2}(\lambda x.a: =_{\delta 1}\boldsymbol{Fst}_{\delta 1,\delta 2}\ x)$

(a.15) $\boldsymbol{A}(\boldsymbol{Snd}_{\delta 1,\delta 2}\ e)_{\delta 2}(a)=\boldsymbol{C}(e)_{\delta 1\times\delta 2}(\lambda x.a: =_{\delta 2}\boldsymbol{Snd}_{\delta 1,\delta 2}\ x)$

Continuation-passing 规则.

(c.1) $\boldsymbol{C}(e)_{\delta}(c)=c\ e$

(c.2) $\boldsymbol{C}(\boldsymbol{neg}\ e)_{num}c=\boldsymbol{C}(e)_{num}(\lambda x.c(\boldsymbol{neg}\ x))$

(c.3) $\boldsymbol{C}(e1+e2)_{num}c=\boldsymbol{C}(e1)_{num}(\lambda x.\boldsymbol{C}(e2)_{num}(\lambda y.c(x+y))))$

(c.4) $\boldsymbol{C}(e1*e2)_{num}c=\boldsymbol{C}(e1)_{num}(\lambda x.\boldsymbol{C}(e2)_{num}(\lambda y.c(x*y))))$

(c.5) $\boldsymbol{C}(\boldsymbol{mapPar}_{n,\delta1,\delta2}\ f\ e)_{n.\delta2}(c)=\boldsymbol{new}_{n.\delta2}(\lambda tmp.\boldsymbol{A}(\boldsymbol{mapPar}_{n,\delta1,\delta2}\ f\ e)_{n.\delta2}(\boldsymbol{Fst}\ tmp);\ c(\boldsymbol{Snd}\ tmp)))$

(c.6) $\boldsymbol{C}(\boldsymbol{mapSeq}_{n,\delta1,\delta2}\ f\ e)_{n.\delta2}(c)=\boldsymbol{new}_{n.\delta2}(\lambda tmp.\boldsymbol{A}(\boldsymbol{mapSeq}_{n,\delta1,\delta2}\ f\ e)_{n.\delta2}(\boldsymbol{Fst}\ tmp);\ c(\boldsymbol{Snd}\ tmp)))$

(c.7) $\boldsymbol{C}(\boldsymbol{reduceSeq}_{n,\delta1,\delta2}\ f\ i\ e)_{\delta2}(c)$

$=\boldsymbol{C}(e1)_{n.\delta1}(\lambda x.\boldsymbol{C}(i)_{\delta2}(\lambda y.\boldsymbol{reduceSeql}_{n,\delta1,\delta2}(\lambda x'\ y'\ o.\boldsymbol{A}(f\ x'\ y')_{\delta2}(o))\ y\ x\ c))$

(c.8) $\boldsymbol{C}(\boldsymbol{zip}_{n,\delta1,\delta2}\ e1\ e2)_{n.\delta1\times\delta2}(c)=\boldsymbol{C}(e1)_{n.\delta1}(\lambda x.\boldsymbol{C}(e2)_{n.\delta2}(\lambda y.c(\boldsymbol{zip}_{n,\delta1,\delta2}\ x\ y)))$

(c.9) $\boldsymbol{C}(\boldsymbol{split}_{n,m,\delta}\ e)_{n.m.\delta}(c)=\boldsymbol{C}(e)_{n*m.\delta}(\lambda x.c(\boldsymbol{split}_{n,m,\delta}\ x))$

(c.10) $\boldsymbol{C}(\boldsymbol{join}_{n,m,\delta}\ e)_{n*m.\delta}(c)=\boldsymbol{C}(e)_{n,m,\delta}(\lambda x.c(\boldsymbol{join}_{n,m,\delta}\ x))$

(c.11) $\boldsymbol{C}(\boldsymbol{Pair}_{n,\delta1,\delta2}\ e1\ e2)_{\delta1\times\delta2}(c)=\boldsymbol{C}(e1)_{n.\delta1}(\lambda x.\boldsymbol{C}(e2)_{n.\delta1}(\lambda y.c(\boldsymbol{Pair}_{n,\delta1,\delta2}\ x\ y)))$

(c.12) $\boldsymbol{C}(\boldsymbol{Fst}_{\delta1\times\delta2}\ e)_{\delta2}(c)=\boldsymbol{C}(e)_{\delta1\times\delta2}(\lambda x.c(\boldsymbol{Fst}_{\delta1\times\delta2}\ x))$

(c.13) $\boldsymbol{C}(\boldsymbol{Snd}_{\delta1\times\delta2}\ e)_{\delta2}(c)=\boldsymbol{C}(e)_{\delta1\times\delta2}(\lambda x.c(\boldsymbol{Snd}_{\delta1\times\delta2}\ x))$

2. 第 3 步的转换规则

第 3 步的转换规则主要用于将命令、左值、右值转换成相应的命令式代码. 代码生成中对于左值、右值的翻译中包含对访问路径、变量名的生成, 访问路径分为数组索引与对偶分量, 路径要与需要翻译的左值相一致. 在对该部分的转换规则实现之前, 首先介绍一下关于访问路径、变量名生成相关的技术实现.

(1) 访问路径的形式化定义

```
Inductive path: =Aidx (i: nat)|Sacc (p: string).
Definition pslist: =list path.
```

上面是访问路径的实现方式, *Aidx* 和 *Sacc* 分别为数组和对偶分量的路径表示. 变量的显式翻译, 我们通过一个函数作用于相应的变量名以及变量路径来实现, 具体定义如下.

```
Fixpoint applyps (v: string) (ps: pslist): string: =
  match ps with
    |nil⇒v
    |(Aidx i):: tl⇒applyps (v++ "["++ (writeNat i)++ "]") tl
    |(Sacc p):: tl⇒applyps (v++ p) tl
  end.
```

例如, 数组 *v* 第 3 个元素的第 1 个分量的显式翻译为 *applyps* "*v*" [*Aidx* 3; *Sacc* ".*x*1"], 输出为"*v* [3].*x*1". 由这个例子可以看到, 访问路径的翻译顺序是先入后出, 这种实现正与接收器与表达式的翻译规则中访问路径的加入顺序相一致.

(2) 变量名翻译

变量名的翻译主要通过将 DPIA 标识符输出为生成代码标识符来实现, 下面定义了存储该映射的环境以及翻译函数.

```
Definition env: =list(string*string).
Fixpoint getvar (s: string) (eta: env): string: =
  match eta with
    |(x,y):: tl⇒if string_dec s x then y else getvar s tl
    |nil⇒"."
  end.
```

例如, *getvar* "*x*" [("*x*","*X*")]="*X*"可以将 DPIA 标识符"*x*"翻译为"*X*".

(3) 临时变量名自动生成

DPIA 在代码生成部分的翻译规则的设计中忽略了新变量的自动命名(其中包括临时变量以及循环变量), 因此我们将整个程序变量声明的个数作为命令代码翻译规则的参数, 并根据不同命令产生相应变化, 该参数作为新变量命名的编号, 以保证不产生重复命名的情况.

首先, 我们定义了变量名生成函数 *vstr* 与 *istr*, 分别用于临时变量以及循环变量的变量名生成, 具体如下.

Definition *vstr* (*cnt*: *nat*): *string*: =“*v*”++ (*writeNat cnt*).

Definition *istr* (*cnt*: *nat*): *string*: =“*i*”++ (*writeNat cnt*).

其次, 我们需要定义一个函数来计算每个命令所产生的临时变量声明的个数, 主要用生成程序中变量名的编号. ***skip*** 命令不做任何操作, 因此变量声明个数为 0; ***seq*** 用于顺序执行两个命令, 变量声明的个数则为两个命令之和; ***assign*** 赋值语句变量声明个数为 0; ***new*** 用于声明一个新变量并执行带参数的命令 *p*, 则其变量声明个数为其中 *p* 中变量声明的个数加 1; ***seqfor*** 与 ***parfor*** 中都含有循环变量的声明, 因此变量声明个数为循环体 *p* 中的变量声明个数+1. 具体定义如下.

```
Fixpoint cnt_var(c: comm): =
  match c with
  |skip⇒0
  |seq (c1,c2)⇒cnt_var c1+cnt_var c2
  |assign ac⇒0
  |@new t p⇒1+cnt_var(p(vwt "·",vrd "·"))
  |seqfor n p⇒1+cnt_var(p(vrd "·"))
  |parfor n t a p⇒1+cnt_var(p(vrd "·")(vwt "·"))
  end.
```

在完成该函数的定义后, 临时变量名的自动生成并未全部实现, 还需要通过在命令显式翻译中加入自然数参数作为命令中临时变量生成的规则, 具体在步骤(4)中命令的翻译规则来实现.

(4) 翻译器以及翻译规则实现

在定义翻译规则之前, 还需要定义 3 个翻译器, 分别用于命令、左值和右值的翻译.

$\mathbf{CodeGen}_{comm}$: *env*→*comm*→*nat*→*string*.

$\mathbf{CodeGen}_{acc}$: **forall** (*p*: *phrase*), *env*→*pslist*→*p*→*string*.

$\mathbf{CodeGen}_{exp}$: **forall** (*p*: *phrase*), *env*→*pslist*→*p*→*string*.

env 为环境类型, 其中存放着 DPIA 标识符与生成程序中的变量标识符. 命令的翻译规则如下.

(cgc.1) $\mathbf{CodeGen}_{comm}(\mathbf{skip}, \eta, cnt)$=/*$\mathbf{skip}$*/

(cgc.2) $\mathbf{CodeGen}_{comm}(p1;p2, \eta, cnt)=\mathbf{CodeGen}_{comm}(p1, \eta, cnt)\ \mathbf{CodeGen}_{comm}(p2, \eta, \mathbf{cnt\_var}\ p1+cnt)$

(cgc.3) $\mathbf{CodeGen}_{comm}(a: =_{num} e, \eta, cnt)=\mathbf{CodeGen}_{acc}(a, \eta, [\cdot])\text{“=”}\mathbf{CodeGen}_{exp}(e, \eta, [\cdot])$;

(cgc.4) $\mathbf{CodeGen}_{comm}(\mathbf{new}_{\delta}(\lambda v.p), \eta, cnt)$=let *v*: =*vstr cnt* in

$\mathbf{CodeGen}_{data[\delta]}(v)$; $\mathbf{CodeGen}_{comm}(p(\mathbf{vwt}_{\delta}\ v, \mathbf{vrd}_{\delta}\ v), v::\ \eta), (cnt+1))$

(cgc.5) $\mathbf{CodeGen}_{comm}(\mathbf{seqfor}\ n(\lambda i.p), \eta, cnt)$=let *i*: =$\mathbf{istr}$ *cnt* in

for (*int i*=0; *i*<*n*; *i*+=1) {$\mathbf{CodeGen}_{comm}(p, i::\ \eta, cnt+1)$}

(cgc.6) $\mathbf{CodeGen}_{comm}(\mathbf{parfor}\ n\ \delta\ a(\lambda i\ o.p), \eta, cnt)$=let *i*: =$\mathbf{istr}$ *cnt* in

#pragma *omp* parallel for

for (*int i*=0; *i*<*n*; *i*+=1) {$\mathbf{CodeGen}_{comm}(p(\mathbf{vrd}_{idx[n]}\ i)\ a\{i\}, i::\ \eta, cnt+1)$}

左值的翻译规则如下.

(cga.1) $\mathbf{CodeGen}_{acc[\delta]}(x, \eta, p)=applyps(getvar\ x\ \eta)\ p$

(cga.2) $\mathbf{CodeGen}_{acc[\delta 1]}(\mathbf{zipAcc1}_{\delta 1,\delta 2}\ a, \eta, i::\ p)=\mathbf{CodeGen}_{acc[n.(\delta 1\times\delta 2)]}(a, \eta, i::\ .x1::\ ps)$

(cga.3) $\boldsymbol{CodeGen}_{acc[\delta 2]}(\boldsymbol{zipAcc2}_{\delta 1,\delta 2}\ a,\eta,i::p)=\boldsymbol{CodeGen}_{acc[n.(\delta 1\times\delta 2)]}(a,\eta,i::.x2::ps)$

(cga.4) $\boldsymbol{CodeGen}_{acc[n*m.\delta]}(\boldsymbol{splitAcc}_{n,m,\delta}\ a,\eta,i::p)=\boldsymbol{CodeGen}_{acc[n.m.\delta]}(a,\eta,i/m::i\%m::p)$

(cga.5) $\boldsymbol{CodeGen}_{acc[n.m.\delta]}(\boldsymbol{joinAcc}_{n,m,\delta}\ a,\eta,i::j::p)=\boldsymbol{CodeGen}_{acc[n*m.\delta]}(a,\eta,i*m+j::p)$

(cga.6) $\boldsymbol{CodeGen}_{acc[\delta]}(\boldsymbol{idxAcc}_{n,\delta 1}\ a\ i,\eta,p)=\boldsymbol{CodeGen}_{acc[n.\delta]}(a,\eta,\boldsymbol{CodeGen}_{exp[idx[n]]}\ (i,\eta,[\cdot])::ps))$

(cga.7) $\boldsymbol{CodeGen}_{acc[\delta 1]}(\boldsymbol{pairAcc1}_{\delta 1,\delta 2}\ a,\eta,p)=\boldsymbol{CodeGen}_{acc[\delta 1\times\delta 2]}(a,\eta,.x1::ps)$

(cga.8) $\boldsymbol{CodeGen}_{acc[\delta 2]}(\boldsymbol{pairAcc2}_{\delta 1,\delta 2}\ a,\eta,p)=\boldsymbol{CodeGen}_{acc[\delta 1\times\delta 2]}(a,\eta,.x2::ps)$右值的翻译规则.

(cge.1) $\boldsymbol{CodeGen}_{exp[\delta]}(x,\eta,p)=applyps(getvar\ x\ \eta)\ p$

(cge.2) $\boldsymbol{CodeGen}_{exp[num]}(\boldsymbol{neg}\ e,\eta,[\cdot])=(-\boldsymbol{CodeGen}_{exp[num]}(e,\eta,[\cdot]))$

(cge.3) $\boldsymbol{CodeGen}_{exp[num]}(e1+e2,\eta,[\cdot])=(\boldsymbol{CodeGen}_{exp[num]}(e1,\eta,[\cdot])+\boldsymbol{CodeGen}_{exp[num]}(e2,\eta,[\cdot]))$

(cge.4) $\boldsymbol{CodeGen}_{exp[num]}(e1*e2,\eta,[\cdot])=(\boldsymbol{CodeGen}_{exp[num]}(e1,\eta,[\cdot])*\boldsymbol{CodeGen}_{exp[num]}(e2,\eta,[\cdot]))$

(cge.5) $\boldsymbol{CodeGen}_{exp[n.(\delta 1\times\delta 2)]}(\boldsymbol{zip}_{n,\delta 1,\ \delta 2}\ e1\ e2,\eta,i::.x1::p)=\boldsymbol{CodeGen}_{exp[n.\delta 1]}(\boldsymbol{zip}_{n,\delta 1,\delta 2}\ e1\ e2,\eta,i::p)$

(cge.6) $\boldsymbol{CodeGen}_{exp[n.(\delta 1\times\delta 2)]}(\boldsymbol{zip}_{n,\delta 1,\delta 2}\ e1\ e2,\eta,i::.x2::p)=\boldsymbol{CodeGen}_{exp[n.\delta 2]}(\boldsymbol{zip}_{n,\delta 1,\delta 2}\ e1\ e2,\eta,i::p)$

(cge.7) $\boldsymbol{CodeGen}_{exp[\delta]}(\boldsymbol{fun\_idx}_{n,\delta}\ e\ i,\eta,p)=\boldsymbol{CodeGen}_{exp[n.\delta]}(e,\eta,\boldsymbol{CodeGen}_{exp[idx[n]]}(i,\eta,[\cdot]))::p)$

(cge.8) $\boldsymbol{CodeGen}_{exp[n.m.\delta]}(\boldsymbol{split}_{n,m,\delta}\ e,\eta,i::j::p)=\boldsymbol{CodeGen}_{exp[n*m.\delta]}(e,\eta,i*n+j::p)$

(cge.9) $\boldsymbol{CodeGen}_{exp[n*m.\delta]}(\boldsymbol{join}_{n,m,\delta}\ e,\eta,i::p)=\boldsymbol{CodeGen}_{exp[n.m.\delta]}(e,\eta,i/n::i\%n::p)$

(cge.10) $\boldsymbol{CodeGen}_{exp[\delta 1\times\delta 2]}(\boldsymbol{Pair}_{\delta 1,\delta 2}\ e1\ e2,\eta,.x1::p)=\boldsymbol{CodeGen}_{exp[\delta 1]}(e1,\eta,p)$

(cge.11) $\boldsymbol{CodeGen}_{exp[\delta 1\times\delta 2]}(\boldsymbol{Pair}_{\delta 1,\delta 2}\ e1\ e2,\eta,.x2::p)=\boldsymbol{CodeGen}_{exp[\delta 2]}(e2,\eta,p)$

(cge.12) $\boldsymbol{CodeGen}_{exp[\delta 1]}(\boldsymbol{Fst}_{\delta 1,\delta 2}\ e,\eta,p)=\boldsymbol{CodeGen}_{exp[\delta 1\times\delta 2]}(e,\eta,.x1::p)$

(cge.13) $\boldsymbol{CodeGen}_{exp[\delta 2]}(\boldsymbol{Snd}_{\delta 1,\delta 2}\ e,\eta,p)=\boldsymbol{CodeGen}_{exp[\delta 1\times\delta 2]}(e,\eta,.x2::p)$点积代码生成例子

第 1 节中的点积例子经过第 1 步、第 2 步的翻译后, 得到以下低级命令式原语组成的程序.

$=\boldsymbol{new}_{(n.num)}(\lambda tmp.\boldsymbol{parfor}\ n\ tmp.1(\lambda i\ o.o:=\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)\ [i]));$

$\boldsymbol{new}_{num}(accum.accum.1:=_{num}0;\ \boldsymbol{seqfor}\ n(\lambda i.accum.1:=accum.2+tmp.2[i]);\ out:=accum.2))$

利用 $\boldsymbol{CodeGen}_{comm}$ 以及第 2.2 节中的规则将其进行化简.

$\boldsymbol{CodeGen}_{comm}(\boldsymbol{new}_{(n.num)}(\lambda tmp.\boldsymbol{parfor}\ n\ tmp.1(\lambda i\ o.o:=\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i]));$

$\boldsymbol{new}_{num}(accum.accum.1:=0;\ \boldsymbol{seqfor}\ n(\lambda i.accum.1:=accum.2+tmp.2[i]);\ out:=accum.2)),\ [xs;ys],\ 0)$

$=\boldsymbol{CodeGen}_{comm}(\boldsymbol{new}_{(n.num)}(\lambda tmp.\boldsymbol{parfor}\ n\ tmp.1(\lambda i\ o.o:=_{num}\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i])),\ [xs;ys],$ 0) ";"

$\boldsymbol{CodeGen}_{comm}(\boldsymbol{new}_{num}(\lambda accum.accum.1:=0;\ \boldsymbol{seqfor}\ n(\lambda i.accum.1:=accum.2+tmp.2[i]);\ out:=_{num}accum.2)),$ $[xs;ys],\ \boldsymbol{cnt\_var}(\boldsymbol{new}_{(n.num)}(\lambda tmp\ \boldsymbol{parfor}\ n\ tmp.1(\lambda i\ o.o:=_{num}\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i]))))$ (cgc.2)

利用 **seq** 命令的显式翻译规则(cgc.2)将整个程序拆分成两段程序的翻译, 下面是对第 1 段程序进行翻译.

$\boldsymbol{CodeGen}_{comm}(\boldsymbol{new}_{(n.num)}(\lambda tmp.\boldsymbol{parfor}\ n\ tmp.1(\lambda i\ o.o:=_{num}\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i])),\ [xs;ys],\ 0)$

$=\boldsymbol{CodeGen}_{data[num]}(v0);\ \boldsymbol{CodeGen}_{comm}(\boldsymbol{parfor}\ n(vwt_{\delta}\ v0)\ (\lambda i\ o.o:=_{num}\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i])),\ [v0;$ $xs;\ ys],\ 1)$

$=int\ v0[n];$

#pragma *omp* parallel for

for ($int\ i1=0;\ i1<n;\ i1++$)

$\{\boldsymbol{CodeGen}_{comm}(v0\{i1\}:=\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i1])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i1]),\ [i1;\ v0;\ xs;\ ys],\ 2)\}$

下面对循环体内部进行化简.

$\boldsymbol{CodeGen}_{comm}(v0\{i1\}:=\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i1])*\boldsymbol{Snd}(\boldsymbol{idx}(\boldsymbol{zip}\ xs\ ys)[i1]),\ [i1;\ v0;\ xs;\ ys],\ 2)$

$=\boldsymbol{CodeGen}_{acc[num]}(v0\{i1\},\ [i1;\ v0;\ xs;\ ys],\ [\cdot])"="\boldsymbol{CodeGen}_{exp[num]}(\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i1])*\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i1]),\ [i1;$ $v0;\ xs;\ ys],\ [\cdot])$ ";"

$="v0[i1]="\boldsymbol{CodeGen}_{exp[num]}(\boldsymbol{Fst}((\boldsymbol{zip}\ xs\ ys)[i1]),\ [i1;\ v0;\ xs;\ ys],\ [\cdot])"*"\boldsymbol{CodeGen}_{exp[num]}(\boldsymbol{Snd}((\boldsymbol{zip}\ xs\ ys)[i1]),\ [i1;$

*v*0; *xs*; *ys*], [·]) “;”

=“*v*0[*i*1]=”$\boldsymbol{CodeGen_{exp[num]}}$((***zip*** *xs ys*)[*i*1], [*i*1; *v*0; *xs*; *ys*],[.*x*1])“*”$\boldsymbol{CodeGen_{exp[num]}}$((***zip*** *xs ys*)[*i*1], [*i*1; *v*0; *xs*; *ys*], [.*x*2]) “;”

=“*v*0[*i*1]=”$\boldsymbol{CodeGen_{exp[num]}}$(***zip*** *xs ys*, [*i*1; *v*0; *xs*; *ys*], [[*i*1]; .*x*1])“*”$\boldsymbol{CodeGen_{exp[num]}}$(***zip*** *xs ys*, [*i*1; *v*0; *xs*; *ys*], [[*i*1]; .*x*2]) “;”

=“*v*0[*i*1]=”$\boldsymbol{CodeGen_{exp[num]}}$(*xs*, [*i*1; *v*0; *xs*; *ys*], [[*i*1]])“*”$\boldsymbol{CodeGen_{exp[num]}}$(*ys*, [*i*1; *v*0; *xs*; *ys*], [[*i*1]]) “;”

=“*v*0[*i*1]=*xs*[*i*1]**ys*[*i*1]; ”

第 1 段程序的翻译结果如下:

#pragma *omp* parallel for

for (*int i*1=0; *i*1<*n*; *i*1++)

{*v*0[*i*1]=*xs*[*i*1]**ys*[*i*1];}

下面对第 2 段程序进行翻译.

$\boldsymbol{CodeGen_{comm}}$(***new***$_{num}$(*λaccum.accum*.1: =0; ***seqfor*** *n*(*λi.accum*.1: =$_{num}$*accum*.2+*tmp*.2[*i*]); *out*: =$_{num}$*accum*.2)), [*i*1; *v*0; *xs*; *ys*], ***cnt_var***(***new***$_{(n.num)}$(*λtmp*.***parfor*** *n tmp*.1(*λi o.o*: =$_{num}$***Fst***((***zip*** *xs ys*)[*i*])*$\boldsymbol{Snd}$((***zip*** *xs ys*)[*i*]))))

=$\boldsymbol{CodeGen_{comm}}$(***new***$_{num}$(*accum.accum*.1: =0; ***seqfor*** *n*(*λi.accum*.1: =$_{num}$*accum*.2+*tmp*.2[i]); *out*: =$_{num}$*accum*.2)), [*i*1; *v*0; *xs*; *ys*],2)

=$\boldsymbol{CodeGen_{data[num]}}$(*v*2);

$\boldsymbol{CodeGen_{comm}}$(*v*2: =$_{num}$0; ***seqfor*** *n*(*λi.v*2.1: =$_{num}$*v*2.2+*v*0[*i*]); *out*: =$_{num}$*v*2.2, [*v*2; *i*1; *v*0; *xs*; *ys*], 3)

=*int v*2;

$\boldsymbol{CodeGen_{comm}}$(*v*2: =$_{num}$0, [*v*2; *i*1; *v*0; *xs*; *ys*], 3);

$\boldsymbol{CodeGen_{comm}}$(***seqfor*** *n*(*λi.v*2.1: =$_{num}$*v*2.2+*v*0[*i*]), [*v*2; *i*1; *v*0; *xs*; *ys*], 3);

$\boldsymbol{CodeGen_{comm}}$(*out*: =$_{num}$*v*2.2, [*i*3; *v*2; *i*1; *v*0; *xs*; *ys*], 4)

其中, 剩余部分的翻译分别如下.

$\boldsymbol{CodeGen_{comm}}$(*v*2: =$_{num}$0, [*v*2; *i*1; *v*0; *xs*; *ys*], 3)=“*v*2=0”;

$\boldsymbol{CodeGen_{comm}}$(***seqfor*** *n*(*λi.v*2.1: =*v*2.2+***idx*** *v*0 *i*), [*v*2; *i*1; *v*0; *xs*; *ys*], 3)

=**for** (*int i*3; *i*3<*n*; *i*++) {*v*2=*v*2+*v*0[*i*];}

$\boldsymbol{CodeGen_{comm}}$(*out*: =$_{num}$*v*2.2, [*i*3; *v*2; *i*1; *v*0; *xs*; *ys*], 4)=“*out*=*v*2;”

第 2 段程序的翻译结果如下.

*int v*2; *v*2=0;

for (*int i*3; *i*3<*n*; *i*++)

{*v*2=*v*2+*v*0[*i*];}

out=*v*2;

因此, 第 1 节的点积运算程序的翻译结果如下.

*int v*0[*n*];

#pragma *omp* parallel for

for (*int i*1=0; *i*1<*n*; *i*1++)

{*v*0[*i*1]=*xs*[*i*1]**ys*[*i*1];}

*int v*2; *v*2=0;

for (*int i*3; *i*3<*n*; *i*++)

{*v*2=*v*2+*v*0[*i*];}

out=*v*2;

上述点积运算程序的执行过程如图 2 所示. 显然, 这样的并行执行方式并不理想.

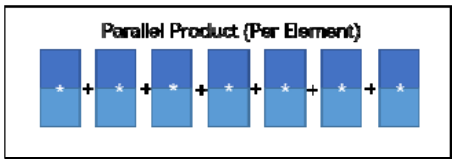

图 2 点积运算过程(元素并行)

对于一个特别长的向量的点积算法，我们可以通过将该向量进行分割，对分割后的子向量实现并行运算，每个点积内部运算为顺序执行，如图 3 所示.

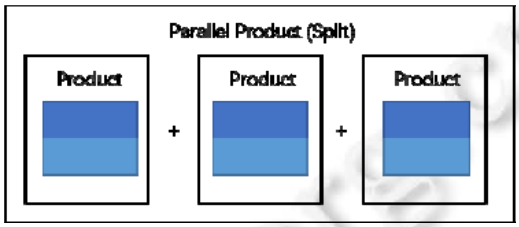

图 3 点积运算过程(分割)

对于这样的点积定义，需要使用到 *split* 原语，具体如下.

reduceSeq (+) 0 (***mapPar*** (*λy*.***reduceSeq*** (*λx a*.((***Fst*** *x*)+(***Snd*** *x*))+*a*) 0 *y*) (***split*** (***zip*** *xs ys*))).

这个点积的具体执行过程是：先将数组 *xs* 和 *ys* 进行 ***zip*** 操作；然后，使用 ***split*** 原语将其分割，此时得到一个二维数组；接着，将并行映射函数 ***map*** 作用于该二维数组之上，映射函数为点积运算函数，即对二维数组中的每个一维数组进行并行点积运算；最后，将这些数组点积的值求和即可得到该数组的点积运算结果. 生成的 C 代码具体如下.

```
float v0[n];
#pragma omp parallel for
for (int i1=0; i1<n; i1+=1)
  {float v2;
    v2=0;
    for (int i3=0; i3<m; i3+=1)
    {v2=((A[i1*m+i3]+B[i1*m+i3])+v2);}
    v0[i1]=v2;}
float v4;
v4=0;
for (int i5=0; i5<n; i5+=1)
{v4=(v0[i5]+v4);}
out=v4;
```

在第 2.5 节，我们将该点积算法与完全顺序执行的点积运算进行了比较测试.

我们可以看到，由高级函数原语描述的高级函数式矩阵程序能够通过该技术进行命令式代码生成. 下面是能够生成命令式代码的函数式程序(见表 1).

表 1 函数式程序

函数作用	函数式矩阵程序
向量内积 *a*b*	*product*
向量内积 *a*b*(*split*)	*product_split*
向量加法 *a+b*	*vec_add*

表 1 函数式程序(续)

函数作用	函数式矩阵程序
向量元素求和	*sum_vec*
向量范数$\|\|a\|\|$	*norm*1, *norm*2
向量数乘 $k*a$	*scale_vec*
向量线性关系 $k*a+b$	*axpy_vec*
向量与矩阵乘法 $a\times A$	*l_mul_dl*
矩阵与向量乘法 $A\times a$	*dl_mul_l*
矩阵乘法 $A\times B$	*dl_mul_dl*
矩阵加法 $A+B$	*mat_add*
矩阵线性关系 $k*A+B$	*axpy_mat*
矩阵数乘 $k*A$	*scale_mat*
矩阵元素求和	*sum_mat*
矩阵转置	*trans*
分块矩阵加法 $bA+bB$	*block_add*
分块矩阵乘法 $bA\times bB$	*block_dl_mul_dl*
分块矩阵数乘 $k*bA$	*scale_block*
分块矩阵线性关系 $k*bA+bB$	*axpy_block*
分块矩阵元素求和	*sum_block*
卷积操作(特殊)	*conv*

其中, 关于向量范数以及分块矩阵的程序需要通过扩展相关规则以达到重写的目的.

2.3 规则扩展

在上述规则中, 我们能够通过该编译技术产生矩阵运算的相应代码; 同时, 该规则具有良好的扩展性, 通过扩展规则可以实现不同向量和矩阵表达式的重写. 下面是我们根据向量运算函数(向量范数、向量绝对值求和、向量赋值、向量逆转、向量赋值和矩阵转置)进行的相应扩展.

原语扩展.

idxpred: **forall** (*n*: *nat*), *exp*(*idx n*).

addidx subidx mulidx dividx: **forall** (*n*: *nat*), (*exp*(*idx n*))→(*exp*(*idx n*))→(*exp*(*idx n*)).

reverse: **forall** [*n*: *nat*] [*s*: *data*], (*exp*(*ary n s*))→(*exp*(*ary n s*)).

reverseSeq: **forall** [*n*: *nat*] [*s*: *data*], (*exp*(*ary n s*))→(*exp*(*ary n s*)).

copy: **forall** [*n*: *nat*] [*s*: *data*], (*exp*(*ary n s*))→(*exp*(*ary n s*)).

copySeq: **forall** [*n*: *nat*] [*s*: *data*], (*exp*(*ary n s*))→(*exp*(*ary n s*)).

transPar: **forall** [*m n*: *nat*] [*s*: *data*], (*exp*(*ary m*(*ary n s*)))→(*exp*(*ary n*(ary *m s*))).

reverseSeql [*n*: *nat*] [*d*1: *data*] (*xs*: *exp*(*ary n d*1)) (*out*: *acc*(*ary n d*1)): =

  seqfor *n*(*fun i*⇒*out*{*i*}: =$_{d1}$*xs*[***subidx*** *n*(***idxpred*** *n*) *i*]).

copySeql [*n*: *nat*] [*d*1: *data*] (*xs*: *exp* (*ary n d*1)) (*out*: *acc*(*ary n d*1)): =

  seqfor *n*(*fun i*⇒*out*{*i*}: =$_{d1}$*xs*[*i*]).

transParl [*m n*: *nat*] [*d*1: *data*] (*xs*: *exp*(*ary m*(*ary n d*1))) (*out*: *acc*(*ary n*(*ary m d*1))): =

  parfor *n*(*ary m d*1) *out*(*fun i*⇒*fun o*⇒***seqfor*** *m*(*fun j*⇒*out*{*i*}{*j*}: =$_{d1}$*xs*[*j*][*i*])).

规则扩展.

A(***sqrt*** *e*)$_{num}$(*a*)=***C***(*e*)$_{num}$(λ*x*.*a*: =$_{num}$(***sqrt*** *x*)) (根号)

A(***abs*** *e*)$_{num}$(*a*)=***C***(*e*)$_{num}$(λ*x*.*a*: =$_{num}$(***abs*** *x*)) (绝对值)

A(***reverseSeq***$_{n,\delta}$ *e*)$_{n.\delta}$(*a*)=***C***(*e*)$_{n.\delta}$(λ*x*.***reverseSeql***$_{n,\delta}$ *x a*)

A(***copySeq***$_{n,\delta}$ *e*)$_{n.\delta}$(*a*)=***C***(*e*)$_{n.\delta}$(λ*x*.***copySeql***$_{n,\delta}$ *x a*)

A(***transPar***$_{m,n,\delta}$ *e*)$_{n.\delta}$(*a*)=***C***(*e*)$_{n,m.\delta}$(λ*x*.***transParl***$_{n,m,\delta}$ *x a*)

C(***sqrt*** *e*)$_{num}$ *c*=***C***(*e*)$_{num}$(λ*x*.*c*(***sqrt*** *x*))

C(***abs*** *e*)$_{num}$ *c*=***C***(*e*)$_{num}$(λ*x*.*c*(***abs*** *x*))

$C(\mathbf{reverseSeq}_{n,\delta}\ e)_{n.\delta}(c)=\mathbf{new}_{n.\delta 2}(\lambda tmp.\mathbf{A}(\mathbf{reverseSeq}_{n,\delta}\ f\ e)_{n.\delta 2}(\mathbf{Fst}\ tmp);\ c(\mathbf{Snd}\ tmp)))$

$C(\mathbf{copySeq}_{n,\delta}\ e)_{n.\delta 2}(c)=\mathbf{new}_{n.\delta}(\lambda tmp.\mathbf{A}(\mathbf{copySeq}_{n,\delta}\ e)_{n.\delta}(\mathbf{Fst}\ tmp);\ c(\mathbf{Snd}\ tmp)))$

$C(\mathbf{transPar}_{m,n,\delta}\ e)_{m.n.\delta 2}(c)=\mathbf{new}_{m.n.\delta}(\lambda tmp.\mathbf{A}(\mathbf{transPar}_{m,n,\delta}\ e)_{m.n.\delta}(\mathbf{Fst}\ tmp);\ c(\mathbf{Snd}\ tmp)))$

$\mathbf{CodeGen}_{exp[num]}(\mathbf{sqrt}\ e,\eta,[\cdot])=(\mathbf{sqrt}\ \mathbf{CodeGen}_{exp[num]}(e,\eta,[\cdot]))$

$\mathbf{CodeGen}_{exp[num]}(\mathbf{abs}\ e,\eta,[\cdot])=(\mathbf{abs}\ \mathbf{CodeGen}_{exp[num]}(e,\eta,[\cdot]))$

$\mathbf{CodeGen}_{exp[idx[n]]}(\mathbf{idxpred}\ n,\eta,[\cdot])=(n-1)$

$\mathbf{CodeGen}_{exp[idx[n]]}(\mathbf{addidx}\ e1\ e2,\eta,[\cdot])=(\mathbf{CodeGen}_{exp[idx[n]]}(e1,\eta,[\cdot])+\mathbf{CodeGen}_{exp[idx[n]]}(e2,\eta,[\cdot]))$

$\mathbf{CodeGen}_{exp[idx[n]]}(\mathbf{subidx}\ e1\ e2,\eta,[\cdot])=(\mathbf{CodeGen}_{exp[idx[n]]}(e1,\eta,[\cdot])-\mathbf{CodeGen}_{exp[idx[n]]}(e2,\eta,[\cdot]))$

$\mathbf{CodeGen}_{exp[idx[n]]}(\mathbf{mulidx}\ e1\ e2,\eta,[\cdot])=(\mathbf{CodeGen}_{exp[idx[n]]}(e1,\eta,[\cdot])*\mathbf{CodeGen}_{exp[idx[n]]}(e2,\eta,[\cdot]))$

最初的重写规则定义无法实现分块矩阵的代码生成，原因是 *reduceSeql* 需要对初始值进行赋值，在向量运算中为 *num* 类型．但分块矩阵在使用 *reduceSeql* 时是对小矩阵进行初始赋值，而在赋值命令 *assign* 的翻译规则中，只存在对 *num* 类型的翻译规则，为此，我们新添加了关于数组类型的规则．

$\mathbf{CodeGen}_{comm}(a:=_{n.\delta e},\eta,cnt)=\mathbf{CodeGen}_{comm}(\mathbf{seqfor}\ n(\lambda i.a\{i\}:=_{\delta e}[i]),\ \eta,\ cnt)$

这条规则是对数组类型的赋值命令的显式翻译，将对于数组类型的赋值转换为顺序执行的 for 循环，其循环体表示为“*a*[*i*]=*e*[*i*];”，语义为依次对数组的元素加以赋值．

添加这条规则之后，即可实现分块矩阵的乘法代码生成，其中两个矩阵的大小分别为 *m*1**n*1 与 *p*1**n*1，内部分块为 *m***n* 与 *p***n*．

首先，我们需要分块矩阵乘法函数进行定义，分块矩阵的函数式代码定义在矩阵的函数式代码之上，而矩阵的函数式代码定义于向量点积运算之上．先定义顺序执行的向量点积函数和向量加法函数，*zero* 表示 0 作为 *reduce* 操作的初始值: Definition ***product_seq*** *zero* {*n*} (*xs ys*: *exp*(*ary n num*)): =***reduceSeq add*** *zero*(***mapSeq*** (***curry mul***) (***zip*** *xs ys*)).

Definition ***vec_add_seq*** {*n*} (*xs ys*: *exp*(*ary n num*)): =***mapSeq*** (***curry*** *add*) (***zip*** *xs ys*).

定义向量与矩阵相乘、矩阵与矩阵相乘和相加的运算函数，其中的执行策略皆为顺序执行．

Definition ***l_mul_dl_ss*** *zero*{*m n*: *nat*} (*dl*: *exp*(*ary m*(*ary n num*))) (*l*: *exp*(*ary n num*))

: =***mapSeq*** (***product_seq*** *zero l*) *dl*.

Definition ***dl_mul_dl_sss*** *zero*{*m n p*: *nat*} (*dl*1: *exp*(*ary m*(*ary n num*))) (*dl*2: *exp*(*ary p*(ary *n num*)))

: =***mapSeq*** (***l_mul_dl_ss*** *zero dl*2) *dl*1.

Definition ***mat_add_ps*** {*m n*: *nat*} (*xs ys*: *exp*(*ary m*(*ary n num*))): =***mapPar*** (***curry*** (*vec_add_seq*)) (***zip*** *xs ys*).

在定义完上述函数后，定义元素为矩阵的向量“点积”和“加法”运算，*mzero* 为 0 元素二维矩阵，因为向量的元素为矩阵，元素之间的相加、相乘使用的都是矩阵加法和乘法．

Definition ***block_product*** {*m n p n*1: *nat*} *zero*(*mzero*: *exp*(*ary m*(*ary p num*)))

(*xs*: *exp*(*ary n*1(*ary m*(*ary n num*)))) (*ys*: *exp*(*ary n*1(*ary p*(*ary n num*))))

: =***reduceSeq mat_add_ps*** *mzero*(***mapSeq*** (***curry*** (***dl_mul_dl_sss*** *zero*)) (***zip*** *xs ys*)).

Definition ***block_add_vec*** {*m n n*1: *nat*} (*dl*1 *dl*2: *exp*(*ary n*1(*ary m*(*ary n num*))))

: =***mapSeq*** (***curry mat_add_ss***) (***zip*** *dl*1 *dl*2).

接着，定义元素为矩阵的向量与矩阵“相乘”、矩阵之间相加的运算函数．

Definition ***block_l_mul_dl*** {*m n p m*1 *n*1: *nat*} *zero*(*mzero*: *exp*(*ary m*(*ary p num*)))

(*dl*: *exp*(*ary m*1(*ary n*1(*ary p*(*ary n num*))))) (*l*: *exp*(*ary n*1(*ary m*(*ary n num*))))

: =***mapPar*** (***block_product*** *zero mzero l*) *dl*.

Definition **block_add** {*m n m*1 *n*1: *nat*} (*dl*1 *dl*2: *exp*(*ary m*1(*ary n*1(*ary m*(*ary n num*)))))

: =***mapPar*** (***curry block_add_vec***) (***zip*** *dl*1 *dl*2).

最后，定义分块矩阵乘法运算函数．

Definition ***block_dl_mul_dl*** {*m n p m*1 *n*1 *p*1: *nat*} *zero*(*mzero*: *exp*(*ary m*(*ary p num*)))
(*dl*1: *exp*(*ary m*1(*ary n*1(*ary m*(*ary n num*))))) (*dl*2: *exp*(*ary p*1(*ary n*1(*ary p*(*ary n num*)))))
: =***mapPar*** (***block_l_mul_dl*** *zero mzero dl*2) *dl*1.

将分块矩阵函数以及对应参数放入相应的翻译器中, *xs*、*ys* 分别是相乘的分块矩阵, 输出为"*out*":

$$\boldsymbol{CodeGen}_{comm}eta(\boldsymbol{A}(\boldsymbol{block\_dl\_mul\_dl}\ zero\ mzero\ xs\ ys)|(vwt\ "out")) 0$$

分块矩阵的代码生成与点积代码的翻译过程类似, 是对分块矩阵实现的原语组合的翻译, 翻译的过程主要如图 4 所示. 但在 Coq 的具体实现中, 存在λ抽象内无法进行规则重写的问题. 因此, 一段程序的翻译是不断迭代该翻译过程, 直至完全翻译完成来实现.

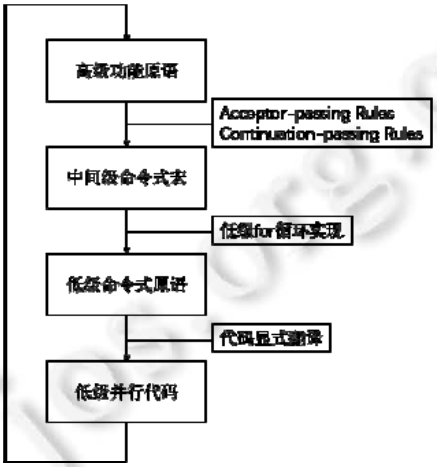

图 4 翻译过程

利用第 2.2 节中的转换规则进行自动化重写, 得到以下命令式代码.

```
#pragma omp parallel for
for (int i0=0; i0<m1; i0+=1)
  {#pragma omp parallel for
for (int i1=0; i1<p1; i1+=1)
  {int v2[n1][m][p];
#pragma omp parallel for
for (int i3=0; i3<n1; i3+=1)
  {#pragma omp parallel for
for (int i4=0; i4<m; i4+=1)
  {#pragma omp parallel for
for (int i5=0; i5<p; i5+=1)
  {int v6[n];
for (int i7=0; i7<n; i7+=1)
  {v6[i7]=(xs[i0][i3][i4][i7]*ys[i1][i3][i5][i7]);}
int v8; v8=0;
for (int i9=0; i9<n; i9+=1)
  {v8=(v6[i9]+v8);}
v2[i3][i4][i5]=v8;}}}
int v10[m][p];
```

```
for (int i11=0; i11<m; i11+=1)
  {for (int i12=0; i12<p; i12+=1)
  {v10[i11][i12]=mzero[i11][i12];}}
for (int i11=0; i11<n1; i11+=1)
  {#pragma omp parallel for
for (int i12=0; i12<m; i12+=1)
  {for (int i13=0; i13<p; i13+=1)
  {v10[i12][i13]=(v2[i11][i12][i13]+v10[i12][i13]);}}}
#pragma omp parallel for
for (int i14=0; i14<m; i14+=1)
  {for (int i15=0; i15<p; i15+=1)
  {out[i0][i1][i14][i15]=v10[i14][i15];}}}}
```

这样一个复杂的命令式程序的正确性很难直接判断，因此，未来我们需要对这样的代码的生成过程进行形式化验证，以保证生成矩阵代码的可靠性.

2.4 生成代码测试

我们对生成的向量和矩阵 C 程序进行了相应的性能测试，本次测试使用的处理器为 Intel i7-8565U(4 核、8 核). 本次测试主要分为生成的顺序执行的 C 代码和带 OpenMP 并行的 C 代码，主要测试结果如下.

长度为 100 000 的向量顺序执行的点积运算时间为 0.002 991 s，加入分割与并行操作后的执行时间为 0.000 998 s(4 核).

由表 2、表 3 我们可以看到，利用该矩阵代码生成技术生成的并行矩阵程序与纯顺序执行的 C 程序的相比在速度上有了大幅度的提升.

表 2 测试结果(4 核)

函数	矩阵大小	顺序执行时长(s)	并行执行时长(s)	分块矩阵并行时长(s)
Scale $k*A$	20000*20000	1.020 283	0.355 051	0.679 184
Axpy $k*A+B$	2000*3000	0.025 117	0.007 960	–
Elements Sum	2000*3000	0.014 960	0.005 456	–
Mul $A\times B$	2000*3000 3000*4000	127.721 608	46.025 829	62.478 652

表 3 测试结果(8 核)

函数	矩阵大小	顺序执行时长(s)	并行执行时长(s)	分块矩阵并行时长(s)
Scale $k*A$	20000*20000	0.142 877	0.130 010	0.131 996
Axpy $k*A+B$	2000*3000	0.007 740	0.009 014	–
Elements Sum	2000*3000	0.006 301	0.002 621	–
Mul $A\times B$	2000*3000 3000*4000	29.818 269	4.283 270	5.499 242

3 形式化验证方案

本节阐述了对矩阵代码生成技术进行形式化验证的具体设计以及方案，具体包括形式化验证框架、类型和原语的语义定义.

3.1 形式化验证框架

对矩阵代码转换过程进行形式化验证需要两部分的工作：一是对所有类型和原语的语义进行形式化定义，类型包括数据类型、短语类型和命令类型，原语包括标量操作原语、数组和对偶的操作原语以及命令原语；二是对代码转换规则进行归纳证明，其中，需要证明的转换规则为 Acceptor-passing、Continuation-passing 规则，关于低级命令显示翻译的规则并不在我们验证的范围之内.

如何证明 Acceptor-passing、Continuation-passing 的相应代码转换规则？在函数式代码中，所有数组的操

作原语通过向量的形式化方法内定义的相应函数来实现; 而在命令式代码中, 所有数组的操作都是通过 for 循环来实现的. 因此, 函数式代码转到命令式代码转换规则的正确性可以归结为向量形式化方法中的函数与 for 循环实现的数组函数的等价性证明. 一方面, 我们需要找到一个合适于该证明的向量形式化方法; 另一方面, 我们需要用函数式语言完成易于证明的命令式原语的实现.

在本次工作中, 我们对点积代码生成的过程进行形式化验证, 具体为对点积代码转换过程使用到的规则(针对点积的具体例子)进行了形式化验证. 我们将在下面介绍形式化的具体方案.

3.2 类型形式化

类型的形式化主要分为短语类型、命令类型.

短语类型(phrase)的实现主要分为接收器类型(*acc*)和表达式类型(*exp*): 接收器类型使用路径类型(*path*)实现, 用于变量的存储, 主要分为变量路径(*Var*)、数组元素路径(*AryIdx*)、对偶元素路径(*PairL*,*PairR*); 表达式类型将数据类型映射至具体类型, 其中分为数值表达式(*exp num*)、数组表达式(*exp*(*ary n d*))以及对偶表达式(*exp* (*tensor d*1 *d*2)). 具体如图 5 所示, 其中, *fin n* 表示小于 *n* 的 *nat* 类型.

```
path:                                        exp:
Var: path → path                             exp num ⇒ R
AryIdx : forall n, fin n → path → path       exp (idx n) ⇒ fin n
PairL : path → path                          exp (ary n d) ⇒ vector n (exp d)
PairR p: path → path                         exp (tensor d1 d2) ⇒ (exp d1, exp d2)
```

图 5 短语类型定义

命令类型(*comm*)的实现, 为不同环境(状态)之间的转换函数. 环境中存储着已声明的变量赋值情况, 环境的实现有多种选择, 我们使用类似 *total_map* 函数的形式: *path*→*exp num*. 至于使用 *exp num* 而不是 *exp d* 的原因在于转换规则中对于数组类型的整体操作拆分成依次对其内部元素的操作, 如下所示:

$$\boldsymbol{A}(\boldsymbol{mapSeq}_{n,\delta 1,\delta 2}\, f\, e)_{n.\delta 2}(a)=\boldsymbol{C}(e)_{n.\delta 1}(\lambda x.\boldsymbol{mapSeql}_{n,\delta 1,\delta 2}(\lambda y.(\lambda o.\boldsymbol{A}(f\, y)_{\delta 2}))\; x\; a)$$

该转换规则将数组(*e*: *exp*(*ary n d*1))的整体映射(*map*)操作转换为对于数组 *e* 中内部元素依次进行映射, 并将每个元素映射的结果分别赋值于 *a* 对应的路径之中. 这在证明的情况下要求我们所实现的环境需要满足既可以对变量整体进行操作, 又可以对变量(如数组)的分量进行操作.

环境的定义: Definition *env*: =*path*→*exp num*.

空环境定义: Definition *env_empty*(*v*: *exp num*): *env*: =*fun*_⇒*v*.

环境变量更新定义:

Definition *update*(*m*: *env*) (*x*: *path*) (*v*: *exp num*): =*fun x*′⇒**if** *path_eqb x x*′ **then** *v* **else** *m x*′.

由于环境的实现为函数, 而无法实现对函数内部进行修改的函数, 因此环境变量删除函数的语义实现通过和变量更新函数的公理实现, 该公理表示对一个环境进行变量更新后, 删除该变量等于环境本身.

Parameter remove: *path*→*env*→*env*.

Axiom *remove_update*: **forall** (*x*: *path*) (*e*: *exp num*) (*m*: *env*), *remove x*(*update m x e*)=*m*.

命令 *comm* 的类型定义在环境类型(*env*)之上则为 *env*→*env*, 其表示为对环境的修改函数的类型.

3.3 原语语义定义

标量操作原语: *R* 域上的操作函数.

negate: =*Ropp*; ***add***: =*Rplus*; ***mul***: *Rmult*; ***zero***: =*R*0; ***one***: =*R*1.

数组操作原语: 定义在数组上的操作函数.

map: **forall** (*n*: *nat*) (*A B*: *Set*), (*A*→*B*)→*vector n A*→*vector n B*.

reduce: **forall** (*n*: *nat*) (*A B*: *Set*), (*A*→*B*→*B*)→*B*→*vector n A*→*B*.

zip: **forall** (*n*: *nat*) (*A B*: *Set*), *vector n A*→*vector n B*→*vector n*(*A***B*).

split: **forall** (*n m*: *nat*) (*A*: *Set*), *vector*(*n***m*) *A*→*vector m*(*vector n A*).

join: **forall** (*n m*: *nat*) (*A*: *Set*), *vector m*(*vector n A*)→*vector*(*n***m*) *A*.

fun_idx: **forall** (*n*: *nat*) (*A*: *Set*), *vector*→*fin n*→(*exp t*).

mapSeq: =***map***; ***mapPar***: =***map***; ***reduceSeq***: =***reduce***.

上述数组操作原语需要根据数组的形式化方法来具体定义, 由于在命令式语言中数组的处理都是通过循环来实现的, 因此数组的形式化方式需要与命令原语的语义定义相统一, 即采用头递归的方法 *vector n A***A*. 不使用尾递归的原因将具体在下一节 ***seqfor*** 语义定义中给出解释.

对偶操作原语: 定义在对偶上的操作函数:

Pair: =*pair*; ***Fst***: =*fst*; ***Snd***: =*snd*. 此处的 *pair*、*fst*、*snd* 都是 Coq 标准库里对偶相应的函数.

3.4 命令以及翻译器语义定义

命令原语有 6 种, 分别是不做任何操作(***skip***)、顺序执行(***seq***)、顺序循环(***seqfor***)、并行循环(***parfor***)、赋值(***assign***)以及新变量声明(***new***), 第 2.1 节中声明了这些函数的类型. 下面主要介绍这些函数具体定义的方法.

第 3.2 节中, *comm*: =*env*→*env*, 因此所有命令类型的函数都是以环境为输入, 以处理过的环境作为输出.

skip 函数的作用是不做任何操作, 其接受一个环境 *eta* 然后返回该环境: *fun eta*⇒*eta*.

seq 函数是接受两个命令类型(*p*1 *p*2: *comm*)作为参数, 按顺序执行这两个命令: *fun eta*⇒*p*2 (*p*1 *eta*). 这里先是将 *p*1 作用于原始环境上, *p*1 *eta* 则为 *p*1 作用后的环境, 然后再将 *p*2 作用于该环境上. 简单来说就是先执行 *p*1, 再执行 *p*2.

seqfor 函数是循环的顺序实现, 其接受自然数 *n* 以及循环体(*p*: *fin n*→*comm*), 将 *p*0, *p*1, …, *p*(*n*−1)依次作用于环境之上. 定义的主要递归变量为自然数 *n*, 当 *n*=0 时, ***seqfor*** 不执行任何操作: *skip*. 当 *n*=*S n*′时, 考虑到后续性质验证以及化简, 其定义不适合使用尾递归实现, 如 *n*=*S n*′⇒***seqfor*** *n*′ *p*(*p*0 *eta*). 可以看到: 若对 *n* 进行一次归纳后, 在化简方面没有帮助. 由于 ***seqfor*** 本身实现的特殊性, 先执行 *p*0, *p*1…, 最后执行 *p*(*n*−1), 因此, 我们应采用非尾递归的方式来定义 ***seqfor*** 函数以便于化简, 如 *n*=*S n*⇒*p n*′(***seqfor*** *n*′ *p eta*). 在这种实现方式下, 可以将第 *n* 次的循环化简至最外层. 若是利用 for 循环实现数组的映射, 化简后即可得到数组最后一个元素的映射语句, 这与数组的形式化方法有很大的关联, 若数组采用尾递归的方法以 *A***vector n*′ *A* 定义, 则难以通过化简拆分出其最后一个元素, 这会给未来转换规则的正确性验证带来很大的困难. 因此, 我们需要使用非尾递归的数组形式化方法 *vector n*′ *A***A*, 这与 ***seqfor*** 循环的定义具有一致性. 具体定义如下.

```
Fixpoint seqfor (m: nat): (fin m→comm)→comm: =
  match m with
  |O⇒fun (_:_)(eta: env)⇒eta
  |S m′⇒fun(p: fin(S m′)→comm) (eta: env)⇒
    p(exist_m′(Nat.lt_succ_diag_r m′)) (seqfor m′(fun i⇒p (fin2finS i)) eta)
  end
```

parfor 函数是循环的并行实现, 其定义原理与 ***seqfor*** 大致相同, 不同的是 ***parfor*** 需要提供一个路径数组作为临时变量存储的地址. 具体定义如下.

```
Fixpoint parfor (m: nat): (fin m→path→comm)→vector path m→comm: =
  match m with
  |O⇒fun(p: fin O→path→comm) (v: vector path O) (eta: env)⇒eta
  |S m′⇒fun(p: fin(S m′)→path→comm) (v: vector path(S m′)) (eta: env)⇒
    p(exist_m′(Nat.lt_succ_diag_r m′)) (snd v) (parfor m′(fun i⇒p(fin2finS i)) (fst v) eta)
  end.
```

assign(: =)函数是用于变量的赋值, 其将变量的值存储于环境之中, 对于不同数据类型, 赋值方法也不同. 对于路径为 *p* 的数组类型的表达式(*e*: *exp*(*ary n d*))进行赋值, 需要使用 ***seqfor*** 循环实现: ***seqfor*** *n*(*fun i*⇒ *p*{*i*}: =*e*[*i*]). 具体定义如下.

```
Fixpoint envAdd {d: data}: path→exp d→comm: =
  match d with
  |num⇒fun(a: path) (e: exp num) (eta: env)⇒update eta a e
  |ary n m d1⇒fun(a: path) (e: exp(ary n m d1))⇒
    seqfor m(fun i⇒(envAdd(AryIdx n(nat2fin n i) a) e[i]))
  |tensor d1⇒fun(a: path) (e: exp(tensor d1))⇒
    seq (envAdd(PairL a) (fst e)) (envAdd(PairR a) (snd e))
end.
Definition assign {d: data}: path→(exp d)→comm: =envAdd d.
```

new 函数用于新变量生成, 在这里, 主要是临时变量的声明以及赋值. 但其类型与第 2.1 节中类型声明不同, 主要是在具体实现中需要路径以及初始默认值: **new** (*d*: *data*) (*p*: *path*exp d→comm*) (*var*: *path*exp d*): *comm*: =*p var*; *envRm d*(*fst var*)). **new** 用于临时变量的声明, 在对其赋值后, 该变量会存储于环境之中. 该临时变量只作用于执行程序体 *p* 中, 因此在执行完 *p* 程序后, 该循环变量在环境中的存储需要删除. 这里使用 *envRm* 命令对环境进行修改, *envRm* 主要基于 *remove* 函数的定义. 考虑到对于长度为 *n* 的数组类型表达式 *a* 的赋值是按照从 *a*[0]到 *a*[*n*−1]的顺序依次赋值, 因此为了更易于证明, 在删除变量时, 按照从 *a*[*n*−1]到 *a*[0]的顺序依次删除. 具体定义如下.

```
Fixpoint for_rev(n: nat): (fin n→comm)→comm: =
  match n with
  |O⇒fun(p: fin O→comm) eta⇒eta
  |S n'⇒fun(p: fin(S n')→comm) (eta: env)⇒
    for_rev n'(fun i⇒p(fin2finS i)) (p(exist_n'(Nat.lt_succ_diag_r n')) eta)
  end.
```

for_rev 与 **seqfor** 函数的定义相似, 区别在于其循环体执行是从 *i*=*n*−1 到 0 的顺序执行.

```
Fixpoint envRm (d: data): path→comm: =
  match d with
  |num⇒fun(p: path) (eta: env)⇒remove p eta
  |ary n m d1⇒fun(p: path) (eta: env)⇒
    for_rev m(fun i⇒(envRm d1(AryIdx n(nat2fin n i) p))) eta
  |tensor d1⇒fun(p: path) (eta: env)⇒envRm d1(PairL p) (envRm d1(PairR p) eta)
  end.
```

除了命令原语的语义定义外, 还需要翻译器 ***A*** 和 ***C*** 的语义定义, 在第 2.2 节的规则定义中, 其实包含了这两者的语义定义: $\boldsymbol{A}(e)_{\delta}(a)$: =($a$: =$_{\delta e}$), $\boldsymbol{C}(e)_{\delta}(c)$: =*c e*.

4 总结与展望

本文在 Coq 定理证明器中构建了一个基于重写的从函数式矩阵代码生成 C 代码的方法, 其中包括类型定义、原语类型定义以及翻译规则定义, 解决了重名变量声明问题以及分块矩阵代码生成问题. 该方法支持自定义算子, 并且通过重写技术, 将函数式定义的矩阵与分块矩阵程序转换为命令式代码. 除此之外, 我们对该技术的形式化方案进行了分析, 其中包括类型和原语的语义定义. 接着, 在其基础上对点积运算的转换过程进行形式化验证. 我们已对矩阵进行形式化[24,25], 对矩阵代码生成技术进行形式化验证需要对从类型到原语以及翻译器都进行具体的函数定义, 所有定义的设计都相互关联从而组成一个整体, 因此在函数设计方面尤为重要也非常具有难度, 这能够直接决定证明工作能否完成、证明工作的难易程度以及工作量大小.

未来, 我们期望在定理证明器中将矩阵的形式化与矩阵代码生成相结合, 并对所有的转换规则进行形式

化验证，以保证代码转换的正确性以及生成的矩阵代码的可靠性，为深度学习编译器的形式化验证作铺垫.同时，我们期望该方法能够针对于 GPU、FPGA 等异构设备进行扩展，构建一个从高级可移植程序为各种并行硬件生成高效矩阵代码的技术.但在这些设备中，策略的选择要比传统 CPU 更加复杂，这涉及到对这些并行设备中存在的并行化和内存层次结构进行建模.Henriksen 等人提出的 Futhark 是一种纯函数式数据并行数组语言[26]，它提供了一个机器中立的编程模型和一个为 GPU 生成 OpenCL 代码的优化编译器，这说明了以 GPU 硬件作为后端实现代码生成的可行性.

References:

[1] Chen G, Yu Ly, Qiu Zy, Wang Y. Logic based formal verification methods: Progress and applications. Acta Scientiarum Naturalium Universitatis Pekinensis, 2016, 52(2): 363−373 (in Chinese with English abstract).

[2] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 33−61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[3] Almeida JB, Frade MJ, Pinto JS, *et al*. An overview of formal methods tools and techniques. In: Proc. of the Rigorous Software Development. London: Springer, 2011. 15−44.

[4] Garavel H, Beek MHT, Pol JVD. The 2020 expert survey on formal methods. In: Proc. of the Formal Methods for Industrial Critical Systems. 2020.

[5] Fedus W, Zoph B, Shazeer N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv: 2101.03961, 2021. [doi: 10.48550/arXiv.2101.03961]

[6] Hoare C. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576−580, 583.

[7] Chen TQ, Moreau T, Jiang ZH, *et al*. TVM: An automated end-to-end optimizing compiler for deep learning. arXiv: 1802.04799v3, 2018.

[8] Jia ZH, Padon O, Thomas J, *et al*. TASO: Optimizing deep learning computation with automated generation of graph substitutions. In: Proc. of the SOSP 2019. Ontario, 2019. 47−62.

[9] Atkey R, Steuwer M, Lindley S, *et al*. Strategy preserving compilation for parallel functional code. arXiv: 1710.08332, 2017. [doi: 10.48550/arXiv.1710.08332]

[10] Bertot Y, Casteran P. Interactive theorem proving and program development. In: Coq'Art: The Calculus of Inductive Constructions. Springer, 2004. [doi: 10.1007/978-3-662-07964-5]

[11] Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009.

[12] Klein G, Elphinstone KJ, Heiser G, *et al*. seL4: Formal Verification of an OS Kernel. In: Proc. of the 22nd ACM Symp. on Operating Systems Principles 2009. ACM, 2009.

[13] 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]

[14] Yang X, Yang C, Eide E, *et al*. Finding and understanding bugs in C compilers. ACM SIGPLAN Notices, 2012, 47(6): 283.

[15] Tanaka A. Coq to C translation with partial evaluation. In: Proc. of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2021). New York: Association for Computing Machinery, 2021. 14−31.

[16] Ragan-Kelley JM, Barnes C, Adams A, *et al*. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM, 2013.

[17] Zhang YM, Yang MJ, Baghdadi R, Kamil S, Shun J, P. Amarasinghe S. GraphIt: A high-performance graph DSL. PACMPL 2, OOPSLA, 2018, 121:1−121:30.

[18] Baghdadi R, Ray J, Romdhane MB, Sozzo ED, Akkas A, Zhang YM, Suriana P, Kamil S, P. Amarasinghe S. Tiramisu: A polyhedral compiler for expressing fast and portable code. In: Proc. of the IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO 2019). Washington, 2019. 193−205.

[19] Hagedorn B, Stoltzfus L, Steuwer M, *et al*. High performance stencil code generation with Lift. In: Proc. of the Int'l Symp. 2018.

[20] Bastian H, Johannes L, Thomas K, Qin XY, Sergei G, Michel S. Achieving high-performance the functional way—A functional pearl on expressing high-performance optimizations as rewrite strategies. In: Proc. of the ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP 2020). 2020.

[21] Pit-Claudel C. Compilation using correct-by-construction program synthesis. 2016. https://www.researchgate.net/publication/316074866_Compilation_using_correct-by-construction_program_synthesis

[22] Wang SY, Cao QX, Mohan A, Hobor A. Certifying graph-manipulating c programs via localizations within data structures. In: Proc. of the Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2019). Athens, 2019.

[23] O'Hearn PW, Power AJ, Takeyama M, *et al*. Syntactic control of interference revisited. Theoretical Computer Science, 1999, 228(1-2): 211−252.

[24] Ma ZW, Chen G. Matrix formalization based on Coq record. Computer Science, 2019, 46(7): 139−145 (in Chinese with English abstract).

[25] Ma YY, Ma ZW, Chen G. Formalization of operations of block matrix based on Coq. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1882−1909 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6255.htm [doi: 10.13328/j.cnki.jos.006255]

[26] Henriksen T, Serup N, Elsman M, *et al*. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. ACM SIGPLAN Notices, 2017, 52(6): 556−571.

附中文参考文献:

[1] 陈钢, 于林宇, 裘宗燕, 王颖. 基于逻辑的形式化验证方法: 进展及应用. 北京大学学报(自然科学版), 2016, 52(2): 363−373.

[2] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33−61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[13] 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]

[24] 马振威, 陈钢. 基于 Coq 记录的矩阵形式化方法. 计算机科学, 2019, 46(7): 139−145.

[25] 麻莹莹, 马振威, 陈钢. 基于 Coq 的分块矩阵运算的形式化. 软件学报, 2021, 32(6): 1882−1909. http://www.jos.org.cn/1000-9825/6255.htm [doi: 10.13328/j.cnki.jos.006255]

麻莹莹(1997－), 女, 硕士, CCF 学生会员, 主要研究领域为形式化工程数学, Coq 定理证明, 函数式语言, 形式化方法.

陈钢(1958－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为形式化工程数学, Coq 定理证明, 函数式语言, 类型系统, 形式化方法, 控制系统.