ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 51 за 2022 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

М.М. Краснов, О.Б. Феодоритова

Применение библиотеки функционального программирования для распараллеливания вычислений на графических ускорителях с технологией CUDA

Рекомендуемая форма библиографической ссылки: Краснов М.М., Феодоритова О.Б. Применение библиотеки функционального программирования для распараллеливания вычислений на графических ускорителях с технологией CUDA // Препринты ИПМ им. М.В.Келдыша. 2022. № 51. 36 с. https://doi.org/10.20948/prepr-2022-51
https://library.keldysh.ru/preprint.asp?id=2022-51

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

М. М. Краснов, О. Б. Феодоритова

Применение библиотеки функционального программирования для распараллеливания вычислений на графических ускорителях с технологией CUDA

Москва— 2022

М. М. Краснов, О. Б. Феодоритова

Применение библиотеки функционального программирования для распараллеливания вычислений на графических ускорителях с технологией CUDA

Современные графические ускорители (GPU) позволяют существенно ускорить выполнение численных задач. Однако перенос программ на графические ускорители является непростой задачей, иногда требующей практически полного их переписывания. Графические ускорители CUDA, благодаря разработанной компанией NVIDIA технологии, позволяют иметь единый исходный код как для обычных процессоров (CPU), так и для CUDA. Однако распараллеливание на общей памяти всё равно делается по-разному и его нужно указывать явно. Применение разработанной авторами библиотеки функционального программирования позволяет скрыть использование того или иного механизма распараллеливания на общей памяти внутри библиотеки и сделать пользовательский исходный код полностью независимым от используемого вычислительного устройства (CPU или CUDA). В настоящей статье показывается, как это можно сделать.

Ключевые слова: C++; библиотека функционального программирования; CUDA; OpenMP; OpenCL; OpenACC

Mikhail Mikhailovich Krasnov, Olga Borisovna Feodoritova

The use of functional programming library to parallelize on graphics accelerators with CUDA technology

Modern graphics accelerators (GPUs) can significantly speed up the execution of numerical tasks. However, porting programs to graphics accelerators is not an easy task, sometimes requiring their almost complete rewriting. CUDA graphics accelerators, thanks to technology developed by NVIDIA, allow you to have a single source code for both conventional processors (CPUs) and CUDA. However, in this single source code, you need to somehow tell the compiler which parts of this code to parallelize on shared memory. The use of the functional programming library developed by the authors allows you to hide the use of one or another parallelization mechanism on shared memory within the library and make the user source code completely independent of the computing device used (CPU or CUDA). This article shows how this can be done.

Key words: C++; functional programming library library; CUDA; OpenMP; OpenCL; OpenACC

Работа выполнена в рамках госзадания ИПМ им. М.В. Келдыша.

Оглавление

Введение

В последние годы всё большее распространение получают графические ускорители (GPU), используемые в качестве вычислительных устройств для численных расчётов. Такие ускорители устанавливаются на многих вычислительных кластерах. Хотя в списке TOP500 самых производительных суперкомпьютеров от июня 2022 г. (JUNE 2022) [1] графические ускорители от компании NVIDIA [2] уступили пальму первенства технологиям от других производителей, в течение многих лет они были в числе первых. Что касается самых высокопроизводительных суперкомпьютеров России, по состоянию на март 2022 г. [3] практически все они оснащены ускорителями от NVIDIA. Скорость численных расчётов на таких ускорителях может быть во много раз выше, чем на CPU (по опыту авторов, ускорение может достигать 10-20 раз), поэтому перенос на графические ускорители программ, реализующих численные методы, является чрезвычайно актуальной задачей.

Однако перенос существующей программы на GPU является непростой задачей. Возможно, идеальный вариант – сразу писать программу так, чтобы она могла считаться на любых вычислителях. В любом случае главным встаёт вопрос – какую технологию работы на GPU использовать? В настоящий момент существует три основных технологии – это OpenCL (открытый стандарт для гетерогенных систем) [4], OpenACC [5] и CUDA [6] – разработка компании NVIDIA для своих графических ускорителей. Каждая из этих технологий имеет свои преимущества и недостатки. Главное преимущество OpenCL – открытый стандарт. Программа, использующая OpenCL, будет работать на любом вычислительном устройстве, поддерживающем этот стандарт, в том числе на GPU от NVIDIA и от AMD, процессорах Intel Xeon Phi с технологией Intel MIC и даже на обычных CPU. Главным недостатком этой технологии является то, что исходный код программы часто возникает в двух экземплярах: для CPU, который компилируется обычным компилятором и является частью основной программы, и текст для OpenCL в отдельных

файлах. При изменениях в алгоритмах правки надо будет вносить в оба места. Преимущества и недостатки технологии CUDA являются зеркальным отражением недостатков и преимуществ OpenCL. CUDA работает только на GPU от NVIDIA. С другой стороны, в CUDA мы имеем единый исходный текст, который компилируется предварительно и является частью основной программы (в том числе и код, который будет исполняться на GPU). Главный недостаток технологии OpenACC в том, что она пока ещё недостаточно распространена. Компилятор, поддерживающий эту технологию, установлен далеко не на всех кластерах с графическими ускорителями. Кроме того, OpenACC подразумевает, что имеются две копии данных: на основном (host) процессоре и на GPU, и OpenACC сама синхронизует эти данные (копирует их в ту или другую сторону). Мы считаем, что это не нужно. Все вычисления должны проводиться на GPU, и копирование данных должно быть только на старте (например, чтобы закачать начальные данные из файла), в конце, чтобы сохранить результаты расчётов, и, возможно, в процессе расчётов для обмена данными между узлами при использовании технологии MPI. Причём, если данные инициализируются программно, то копирование данных в начале необязательно.

Мы выбираем технологию CUDA. Наш главный аргумент состоит в том, что в (нашей) реальной жизни мы сталкиваемся исключительно с устройствами от NVIDIA. GPU от AMD и процессоры Intel Xeon Phi достаточно экзотичны, и, хотя нам и встречались, реально неактуальны. Поэтому недостаток CUDA недостатком для нас не является, а её преимущество остаётся.

Следующая проблема состоит в том, что распараллеливание на общей памяти на CPU и на GPU делается совершенно по-разному. Если мы хотим получить единый текст, который должен компилироваться и для CPU и для CUDA, то в тех местах, где должно быть распараллеливание, придётся писать разный код (например, с помощью конструкции #ifdef), что неудобно. И тут возникла идея воспользоваться ранее написанной одним из авторов статьи библиотекой функционального программирования для языка C++ [7]. При использовании этой библиотеки всю специфику вычислительного устройства (CPU или CUDA) можно поместить внутрь библиотеки, и пользовательский исходный код останется платформонезависимым. С другими аналогичными работами авторов можно также ознакомиться в работах [8] и [9].

Настоящая работа состоит из трёх основных частей: краткое введение в функциональное программирование (в объёме, необходимом для понимания остального текста), краткое описание библиотеки функционального программирования funcprog и описание применения этой библиотеки для решения численных задач.

Краткое введение в функциональное программирование

В функциональном программировании центральным объектом является (как это и следует из названия) функция. Функции являются полноправными участниками вычислительного процесса, такими же, какими при обычных вычислениях являются числа. Это значит, что функция может быть передана как параметр другой функции и может быть возвращена как результат работы функции (иногда функции, принимающие в качестве параметров другие функции, называют функциями высшего порядка). Функцию можно вычислить так же, как при обычных вычислениях можно вычислить число. Простой пример – композиция двух одноместных функций, которая возвращает новую одноместную функцию, вызывающую последовательно обе функции. В специализированных функциональных языках программирования (таких как Haskell [10]) такие возможности встроены в язык, в то время как реализация композиции функций на языке C++ является нетривиальной задачей, требующей специальных ухищрений. Примеры будут приводиться на языке Haskell, так как этот язык позволяет записывать многие вещи максимально кратко и в то же время понятно. В качестве первоначального «ликбеза» по языку Haskell укажем формат определения и вызова функции. Определение функции f с параметрами a и b имеет вид:

```
f a b = expression
```

Например, функция, принимающая координаты двумерного вектора и возвращающая его длину, выглядит так:

```
veclen x y = sqrt(x * x + y * y)
```

Обратите внимание, что оператор return отсутствует, он не нужен, так как неявно подразумевается. Любая функция должна вернуть результат. При вызове функции скобки и запятые не указываются, разделителем параметров служит пробел. Например:

```
len = veclen 1.23 4.56
```

Параметр функции можно заключить в круглые скобки, но только для того, чтобы указать приоритет операций (как в определении функции veclen). Чтобы избавиться от вложенных скобок, последний параметр при вызове функции можно отделить символом $ (оператор применения функции к параметру). Например, вместо

```
x = f a (g b (h c))
```

можно написать:

```
x = f a $ g b $ h c
```

В частности, определение функции veclen можно переписать так:

```
veclen x y = sqrt $ x * x + y * y
```

Принципы функционального программирования

Функциональное программирование имеет ряд особенностей по сравнению с императивным программированием, которые можно сформулировать в виде нескольких принципов. Некоторые из этих принципов являются обязательными для функционального программирования и поддерживаются всеми языками и библиотеками функционального программирования, а другие – опциональными, то есть в некоторых языках и библиотеках функционального программирования могут отсутствовать. По тому, насколько полно эти принципы реализованы в языке или библиотеке функционального программирования, можно судить о степени её «функциональности».

Первый и главный обязательный принцип, уже упоминавшийся выше – это то, что функция является полноценным участником вычислительного процесса и может быть как передана в качестве параметра, так и возвращена как результат работы некоторой функции. Другой обязательный принцип – наличие лямбда-выражений. Лямбда-выражение – это такое выражение в языке, результатом которого является функция. Собственно, первый принцип без второго практически невозможен. Как правило, функция, возвращающая в качестве результата функцию, фактически возвращает лямбда-выражение. В частности, в современном стандарте языка C++ (начиная с версии C++11) лямбда-выражения имеются. Остальные принципы функционального программирования не столь важны и часто отсутствуют, но их наличие существенно повышает возможности языка или библиотеки. Опишем основные из них.

«Чистые» функции. Под «чистотой» функции в функциональном программировании подразумевается отсутствие у функции побочных эффектов. Это значит, что результат, возвращаемый функцией, зависит только от переданных аргументов и больше ни от чего.

Следующий принцип функционального программирования – **неизменяемые (immutable) переменные**. Это как если бы в Вашей программе на C++ все переменные имели бы модификатор const (int const i = 5;). Переменные есть, но им что-то присвоить можно только один раз при создании. Именно с такими переменными работают все функциональные языки программирования (Haskell, Lisp). Этот принцип позволяет гарантировать «чистоту» функции. Функция не

меняет значение ни одной переменной (потому что не может), значит, она «чистая».

Каррирование. Названо по фамилии американского математика и логика Хаскелла Карри (а по его имени назван язык программирования Haskell). Принцип каррирования состоит в том, что при неполном указании параметров функции ошибки не происходит, а вместо этого генерируется функция с меньшим (равным числу недостающих параметров) числом параметров. Реализовано во многих современных функциональных языках программирования (в частности, в языке Haskell). Каррирование позволяет функцию с несколькими аргументами рассматривать как набор функций с одним аргументом.

Ленивые вычисления. Этот принцип в языке Haskell является прямым следствием принципа каррирования. В языке Haskell каррирование «идёт до конца». Это значит, что даже при передаче всех параметров функции фактического вызова не происходит. При применении каждого следующего параметра порождается функция с меньшим на единицу числом параметров, и после применения всех параметров получается функция без параметров. Именно эта функция без параметров передаётся в качестве аргумента. То есть фактически любые вычисления в языке Haskell – это вычисления функций.

η-редукция (эта редукция, или η-преобразование). Начнём с примера. Пусть мы хотим написать функцию с одним параметром – списком чисел, возвращающую список синусов этих чисел. Текст этой функции на языке Haskell очевидный:

```
mapsin lst = map sin lst
```

Здесь map – функция, принимающая в качестве параметров одноместную функцию и список и возвращающая новый список, элементы которого получаются из элементов исходного в результате применения функции, передаваемой в качестве первого параметра. Из принципа каррирования следует, что если мы опустим второй параметр при вызове функции map sin lst, то есть напишем просто map sin, то мы получим функцию с одним параметром, принимающим в качестве этого параметра список чисел и возвращающую список синусов этих чисел, то есть фактически функцию mapsin. То есть mapsin эквивалентно map sin. Принцип η-редукции гласит, что в подобных случаях последний параметр в определении функции можно опускать, то есть определение функции mapsin можно записать короче:

```
mapsin = map sin
```

Композиция функций. Композиция функций настолько важна в функциональном программировании, что в языке Haskell эта операция максимально упрощена. Обычно рассматривают композицию

одноместных функций (назовём их f и g), в языке Haskell она записывается так:

```
(f . g) x = f (g x)
map (exp . sin) [1,2,3] -- example
```

Математические основы функционального программирования

В основе функционального программирования (в частности языка Haskell) лежит современная математическая теория – теория категорий. Подробно с этой теорией можно ознакомиться, например, по источникам [11] и [12]. **Категорией** называется совокупность объектов, снабжённых стрелками (морфизмами) между ними (некоторыми из них). Между двумя заданными объектами может быть много стрелок. Совокупность стрелок из объекта *A* в объект *B* в категории **C** обозначается $Hom_{\mathbf{C}}(A, B)$ или просто $Hom(A, B)$, если конкретная категория подразумевается. Если совокупность всех объектов категории и совокупность всех стрелок между объектами образуют множества (возможно, пустые, конечные, счётные или несчётные), то такая категория называется «малой» (small), иначе она называется «большой» (large). Если для любых объектов *A* и *B* в категории **C** совокупность стрелок $Hom_{\mathbf{C}}(A, B)$ образует множество, то такая категория называется «локально малой» (locally small). В частности, любая малая категория является локально малой. Напомним, что совокупность объектов, «бóльшая», чем множество, называется классом объектов.

Для стрелок имеются два обязательных условия. Во-первых, из каждого объекта должна существовать стрелка в него самого («единичная» стрелка, или тождественный морфизм). Тождественный морфизм объекта *A* обозначается id_A. Таким образом, для любого объекта A $Hom(A, A)$ непусто (всегда имеется тождественный морфизм). Во-вторых, для любых двух стрелок $f \in Hom(A, B)$ и $g \in Hom(B, C)$ должна существовать их композиция $g \circ f \in Hom(A, C)$. Для существования композиции морфизмов важно, чтобы конец первой (правой) стрелки f совпадал с началом второй (левой) стрелки g. Обратите внимание, что первая стрелка указывается справа от оператора композиции, а вторая – слева. Композицию стрелок $g \circ f$ можно читать как «g после f».

Морфизм $f \in Hom(A, B)$ называется **изоморфизмом**, если существует такой морфизм $g \in Hom(B, A)$, что $g \circ f = id_A$ и $f \circ g = id_B$. Два объекта, между которыми существует изоморфизм, называются изоморфными. В частности, тождественный морфизм является изоморфизмом, поэтому любой объект изоморфен сам себе.

Морфизмы, в которых начало и конец совпадают (морфизмы из некоторого объекта в него самого), называются **эндоморфизмами**.

Множество эндоморфизмов объекта A: $End(A) = Hom(A, A)$ является **моноидом** относительно операции композиции с единичным элементом id_A. Напомним, что моноидом называется множество с определённой на нём бинарной ассоциативной операцией и нейтральным (единичным) элементом относительно этой операции. Например, натуральные числа образуют моноид относительно операции умножения с единицей в качестве нейтрального элемента, а неотрицательные целые числа образуют моноид относительно операции сложения с нулём в качестве нейтрального элемента.

Приведём примеры категорий:

- **0** – пустая категория (нет объектов и нет морфизмов);
- **1** – категория, состоящая из одного объекта и одного морфизма (единичного морфизма у единственного объекта);
- **2** – категория, состоящая из двух объектов (назовём их первый и второй) и трёх морфизмов: двух обязательных единичных и морфизма из первого объекта во второй.
- **Set** – категория множеств. Объектами в этой категории являются множества, морфизмами — отображения множеств. Это пример «большой» категории;
- **Grp** – категория групп. Объектами являются группы, морфизмами — отображения, сохраняющие групповую структуру (гомоморфизмы групп);
- **Top** – категория топологических пространств. Морфизмы — непрерывные отображения;

Функторами называются отображения категорий, сохраняющие структуру исходной категории. Точнее, функтор $F\colon$ **C** $\to$ **D** ставит в соответствие каждому объекту в категории **C** объект в категории **D** и каждому морфизму $f\colon A \to B$ в категории **C** морфизм $F(f)\colon F(A) \to F(B)$ в категории **D** так, что

- $F(id_A) = id_{F(A)}$

и для двух морфизмов $f\colon A \to B$ и $g\colon B \to C$ в категории **C**

- $F(g) \circ F(f) = F(g \circ f)$.

Функторы из категории **C** в категорию **D** также образуют категорию (категорию функторов), морфизмами в которой являются так называемые **естественные преобразования**. Естественное преобразование предоставляет способ перевести один функтор в другой, сохраняя внутреннюю структуру (например, композиции морфизмов).

Пусть F и G – функторы из категории **C** в категорию **D**. Тогда естественное преобразование $\eta\colon F \Rightarrow G$ сопоставляет каждому объекту X в категории **C** морфизм $\eta_X\colon F(X) \to G(X)$ в категории **D**, называемый компонентой η в X, так, что для любого морфизма $f\colon X \to Y$ в категории **C** диаграмма (в категори **D**), изображённая на рисунке ниже, коммутативна.

$$\begin{array}{ccc} X & F(X) & \xrightarrow{\eta_X} G(X) \\ {\scriptstyle f}\downarrow & {\scriptstyle F(f)}\downarrow & \downarrow{\scriptstyle G(f)} \\ Y & F(Y) & \xrightarrow[\eta_Y]{} G(Y) \end{array}$$

Коммутативность данной диаграммы означает, что из $F(X)$ в $G(Y)$ можно прийти двумя разными путями или что выполнено следующее равенство:

$$\eta_Y \circ F(f) = G(f) \circ \eta_X$$

Два функтора называются **естественно изоморфными**, если между ними существует естественное преобразование η такое, что η_X – изоморфизм для любого X.

Для естественных преобразований можно задать два вида композиции: вертикальную и горизонтальную.

Вертикальная композиция. Если $\eta\colon F \Rightarrow G$ и $\epsilon\colon G \Rightarrow H$ – естественные преобразования между функторами $F, G, H\colon$ **C** $\to$ **D**, то мы можем определить композицию между ними и получить естественное преобразование $\epsilon \circ \eta\colon F \Rightarrow H$. Оно определяется покомпонентно: $(\epsilon \circ \eta)_X = \epsilon_X \circ \eta_X$.

$$\begin{array}{ccccc} F(X) & \xrightarrow{\eta_X} & G(X) & \xrightarrow{\epsilon_X} & H(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} & & \downarrow{\scriptstyle H(f)} \\ F(Y) & \xrightarrow{\eta_Y} & G(Y) & \xrightarrow{\epsilon_Y} & H(Y) \end{array}$$

Вертикальная композиция естественных преобразований ассоциативна и имеет единицу. Она позволяет рассматривать набор всех функторов **C** $\to$ **D** в свою очередь как категорию (категорию функторов). Единичное естественное преобразование id_F на функторе F имеет компоненты $(id_F)_X = id_{F(X)}$.

Для $\eta\colon F \Rightarrow G, id_G \circ \eta = \eta = \eta \circ id_F$.

Горизонтальная композиция. Если $\eta\colon F \Rightarrow G$ – естественное преобразование между функторами $F, G\colon \mathbf{C} \to \mathbf{D}$ и $\epsilon\colon J \Rightarrow K$ – естественное преобразование между функторами $J, K\colon \mathbf{D} \to \mathbf{E}$, то композиция функторов позволяет делать композицию естественных преобразований $\epsilon * \eta\colon J \circ F \Rightarrow K \circ G$ с компонентами

$$(\epsilon * \eta)_X = \epsilon_{G(X)} \circ J(\eta_X) = K(\eta_X) \circ \epsilon_{F(X)}$$

Используя wiskering (см. ниже), мы можем записать:

$$(\epsilon * \eta)_X = (\epsilon G)_X \circ (J\eta)_X = (K\eta)_X \circ (\epsilon F)_X.$$

Таким образом,

$$\epsilon * \eta = \epsilon G \circ J\eta = K\eta \circ \epsilon F.$$

$$\begin{array}{ccccc} J(F(X)) & \xrightarrow{J(\eta_X)} & J(G(X)) & \xrightarrow{\epsilon_{G(X)}} & K(G(X)) \\ {\scriptstyle J(F(f))}\downarrow & & \downarrow{\scriptstyle J(G(f))} & & \downarrow{\scriptstyle K(G(f))} \\ J(F(Y)) & \xrightarrow{J(\eta_Y)} & J(G(Y)) & \xrightarrow{\epsilon_{G(Y)}} & K(G(Y)) \end{array}$$

Горизонтальная композиция также ассоциативна и имеет единицу. Эта единица является единичным естественным преобразованием над единичным функтором, то есть естественным преобразованием, которое ставит в соответствие каждому объекту его тождественный морфизм: для каждого объекта X в категории $\mathbf{C}$ $(id_{id_{\mathbf{C}}})_X = id_{id_{\mathbf{C}}(X)} = id_X$.

Для $\eta\colon F \Rightarrow G$, где $F, G\colon \mathbf{C} \to \mathbf{D}$, $id_{id_{\mathbf{D}}} * \eta = \eta = \eta * id_{id_{\mathbf{C}}}$.

Так как единичные функторы $id_{\mathbf{C}}$ и $id_{\mathbf{D}}$ – это функторы, то единица для горизонтальной композиции также является единицей и для вертикальной композиции, но не наоборот.

Можно определить внешнюю (external) бинарную операцию между функторами и естественными преобразованиями (в английской терминологии **whiskering**). Пусть **B**, **C**, **D**, **E** – категории, $K\colon \mathbf{B} \to \mathbf{C}, F, G\colon \mathbf{C} \to \mathbf{D}, H\colon \mathbf{D} \to \mathbf{E}$ – функторы, $\eta\colon F \Rightarrow G$ – естественное преобразование. Тогда мы можем определить естественное преобразование $H\eta\colon H \circ F \Rightarrow H \circ G$, задав

$$(H\eta)_X = H(\eta_X), X \in \mathbf{C}.$$

Здесь задаётся равенство морфизмов в категории **E**. Аналогично мы

можем определить естественное преобразование $\eta K\colon F \circ K \Rightarrow G \circ K$, задав

$$(\eta K)_X = \eta_{K(X)}, X \in \mathbf{B}.$$

Здесь задаётся равенство морфизмов в категории **D**.

Эта операция является также горизонтальной композицией, где одно из естественных преобразований является тождественным естественным преобразованием:

$$H\eta = id_H * \eta \text{ и } \eta K = \eta * id_K.$$

Обратите внимание, что id_H (соответственно id_K) не является единицей горизонтальной композиции * (в общем случае $H\eta \neq \eta$ и $\eta K \neq \eta$), кроме случаев, когда H (соответственно K) является единичным функтором категории **D** (соответственно **C**).

Перестановочный закон. Вертикальная и горизонтальная композиции связаны тождеством, которое меняет их местами. Если имеются четыре естественных преобразования $\alpha, \alpha', \beta, \beta'$, как показано на рисунке справа, то выполнено следующее равенство:

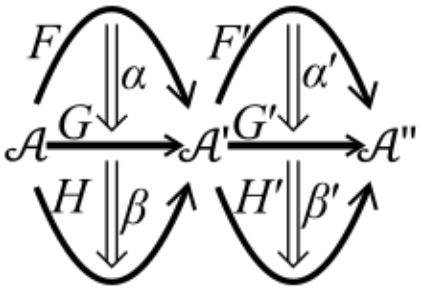

Рис. 1. Горизонтальная и вертикальная композиция естественных преобразований

$$(\beta' \circ \alpha') * (\beta \circ \alpha) = (\beta' * \beta) \circ (\alpha' * \alpha).$$

Вертикальные и горизонтальные композиции также связаны тождественными естественными преобразованиями:

Для $F\colon \mathbf{C} \to \mathbf{D}$ и $G\colon \mathbf{D} \to \mathbf{E}, id_G * id_F = id_{G \circ F}$.

Так как wiskering является горизонтальной композицией с тождественным естественным преобразованием, перестановочный закон немедленно даёт компактную формулу для горизонтальной композиции $\eta\colon F \Rightarrow G$ и $\epsilon\colon J \Rightarrow K$ без необходимости анализировать компоненты и коммутативные диаграммы:

$$\begin{aligned}\epsilon * \eta &= (\epsilon \circ id_J) * (id_G \circ \eta) = (\epsilon * id_G) \circ (id_J * \eta) = \epsilon G \circ J\eta \\ &= (id_K \circ \epsilon) * (\eta \circ id_F) = (id_K * \eta) \circ (\epsilon * id_F) = K\eta \circ \epsilon F.\end{aligned}$$

Функторы из категории в неё саму называются **эндофункторами**. Эндофункторы играют важную роль в теории категорий. Между любыми двумя эндофункторами в некоторой категории **C** определена композиция (так как начало и конец совпадают), причём эта композиция ассоциативна, а также существует единичный эндофунктор (обозначим его как $1_{\mathbf{C}}$),

оставляющий все объекты и морфизмы в категории **C** без изменения. Эндофункторы в некоторой категории **C** образуют свою категорию с естественными преобразованиями в качестве морфизмов.

Определим теперь теоретико-категорное понятие монады. **Монадой** в теории категорий называется тройка (T, η, μ), где

- T – эндофунктор в некоторой категории **K** ($T\colon \mathbf{K} \to \mathbf{K}$);
- $\eta\colon 1_{\mathbf{K}} \to T$ – естественное преобразование;
- $\mu\colon T^2 \to T$ – естественное преобразование;
- следующая диаграмма коммутативна (ассоциативность):

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

- следующая диаграмма коммутативна (двухсторонняя единица):

$$\begin{array}{ccc} T & \xrightarrow{\eta T} & T^2 \\ {\scriptstyle T\eta}\downarrow & & {\scriptstyle \mu}\downarrow \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

Из определения моноида (см. выше) видно, что монада является моноидом в категории эндофункторов $\mathbf{End}(\mathbf{K})$.

Функторы и монады в программировании

Функторы. Пусть у нас есть некоторый контейнер, хранящий какое-то количество значений, например, список или объект класса Maybe (хранящий одно значение указанного типа или не хранящий ничего, в стандартной библиотеке C++ этому классу соответствует класс std::optional). Теперь поставим задачу: применить обычную одноместную функцию (например, sin) к значениям в контейнере. Как это сделать в языке Haskell со списками, известно – применить функцию map. Но как это сделать с типом Maybe, и в общем случае – как это сделать с данными в произвольном контейнере? Универсальный подход состоит в том, чтобы доверить это ответственное дело самому контейнеру. Для этого в языке Haskell определён специальный класс Functor, в котором продекларирована функция fmap:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

Оператор (<$>) – синоним функции fmap. Это оператор применения функции к функтору. Он похож на оператор применения функции к обычному значению ($). Прототип функции fmap можно записать в другом эквивалентном виде (это следует из правоассоциативности стрелки вправо):

```
fmap :: (a -> b) -> (f a -> f b)
```

Функтором называется тип, реализующий класс Functor. Таким образом, функцию fmap можно рассматривать как функцию с одним параметром, принимающую функцию, принимающую и возвращающую обычные значения, и преобразующую её в функцию, принимающую и возвращающую функторы. Любой тип данных может объявить себя функтором, реализовав для себя экземпляр класса Functor и функцию fmap. Любая реализация функтора должна удовлетворять двум функторным законам:

```
1. fmap id = id                          -- 1st functor law
2. fmap (g . f) = fmap g . fmap f -- 2nd functor law
```

Здесь id – функция, возвращающая свой аргумент: id x=x. Первый закон гласит: применение функции id к функтору не должно менять функтор, так же, как применение этой функции к обычному значению его не меняет. Второй закон – это распределительный закон функторной операции относительно композиции функций. Для списков и типа данных Maybe функтор реализован так:

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

Аппликативы. Если стоит задача применить функцию с двумя аргументами к двум контейнерам (например, просуммировать два списка), то функционала класса Functor будет недостаточно. Для решения этой задачи предназначен другой класс – аппликативный функтор (аппликатив). Вот определение класса Applicative:

```
class Functor f => Applicative f where
  pure :: a->f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a->b->c)->f a->f b->f c
  liftA2 f x y = f <$> x <*> y
```

Таким образом, в каждом аппликативе должны быть реализованы две основные операции: функция pure, помещающая обычное значение в «чистый» аппликатив, и оператор (<*>), принимающий в качестве первого параметра функцию, помещённую в аппликатив, и второго параметра – значение, помещённое в тот же аппликатив, и возвращающий результат в том же аппликативе.

Если мы посмотрим на прототип и реализацию функции liftA2, то мы увидим, что она передаёт функцию с двумя параметрами в оператор (<$>), который принимает функцию с одним параметром. Но противоречия тут нет, так как функцию с прототипом a->b->c мы можем записать так: a->(b->c), то есть как функцию с одним параметром, возвращающую функцию. Тогда оператор (<$>) нам как раз и вернёт функцию типа b->c, помещённую в функтор, которая затем передаётся в оператор (<*>). По аналогии с функцией fmap прототип функции liftA2 мы можем записать так:

```
liftA2 :: (a->b->c)->(f a->f b->f c)
```

то есть рассматривать её как функцию с одним аргументом, принимающую функцию, работающую с обычными значениями и возвращающую функцию, работающую с функторами, «поднимающую» функцию с двумя аргументами (отсюда и её название) в аппликатив. По аналогии с функцией liftA2 можно написать функцию, «поднимающую» в аппликатив функцию с тремя (и любым другим числом) аргументами:

```
liftA3 :: Applicative f => (a->b->c->d)->f a->f b->f c->f d
liftA3 f x y z = f <$> x <*> y <*> z
```

Любая реализация аппликатива должна удовлетворять аппликативным законам:

```
1. pure id <*> v = v                            -- Identity
2. pure f <*> pure x = pure (f x)               -- Homomorphism
3. u <*> pure y = pure ($ y) <*> u              -- Interchange
4. pure (.) <*> u <*> v <*> x = u <*> (v <*> x) -- Composition
```

Вот как реализованы аппликативны для списков и Maybe:

```
instance Applicative [] where
  pure x    = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]

instance Applicative Maybe where
  pure = Just -- eta-reduction
  Just f  <*> m = fmap f m
  Nothing <*> _ = Nothing
```

Монады. Напишем «безопасные» функции safe_sqrt и safe_log:

```
safe_sqrt x=if x<0 then Nothing else Just(sqrt x)
```

```
safe_log x=if x<=0 then Nothing else Just(log x)
```

Эти функции возвращают результат типа Maybe, причём если аргумент функции принадлежит области её определения, то возвращается значение Just value, а иначе – Nothing. Пусть теперь мы хотим вычислить квадратный корень от логарифма числа. Для обеих операций у нас есть «безопасные» функции, возвращающие результат типа Maybe и проверяющие, что значение аргумента принадлежит области определения функции. Как нам теперь применить функцию safe_sqrt к результату функции safe_log? Функционала функтора и аппликатива для этого недостаточно. Для этой цели служат монады. Монады можно рассматривать как дальнейшее продолжение аппликатива, они предназначены для построения цепочек монадных вычислений.

Определение класса Monad в языке Haskell выглядит так:

```
class Functor m => Monad f where
  return :: a->f a
  (>>=) :: m a -> (a -> m b) -> m b
```

Каждая монада имеет две основных функции: return и bind (в языке Haskell оператор (>>=). Функция return аналогична функции pure для аппликативов (фактически для большинства монад return определяется как pure). Операция bind принимает в качестве параметров монаду и функцию, принимающую обычное (не монадное) значение и возвращающую монадное значение (возможно, другого типа, но в той же монаде). Будем называть такие функции монадными. Функции safe_log и safe_sqrt (а также функция return) – примеры монадных функций.

Функции return и bind должны удовлетворять трём монадным законам. Для того чтобы их сформулировать, введём операцию монадной композиции (оператор (>=>) в языке Haskell). Она определяется следующим образом:

```
(>=>) :: f >=> g = \x -> (f x >>= g)
```

Оба операнда монадной композиции и её результат – монадные функции. Следовательно, монадную композицию можно рассматривать как групповую операцию в пространстве таких функций. В терминах этой групповой операции монадные законы формулируются так:

```
1. return >=> f == f
2. f >=> return == f
3. (f >=> g) >=> h == f >=> (g >=> h)
```

Другими словами, функции return и bind должны быть определены так, чтобы, во-первых, функция return являлась единичным элементом (левым и правым) монадной композиции (первые два закона), и, во-вторых, монадная композиция должна быть ассоциативной (третий закон).

Так как любая монада также является функтором, то для любой монады операцию bind можно определить через операцию fmap следующим образом:

```
x >>= f = join (f <$> x)
```

Функция fmap в данном случае вернёт значение «монады в монаде» (например, список списков). Функция join убирает этот один лишний уровень вложенности. Функции fmap и join можно определить через монадные операции:

```
fmap f m = m >>= (return . f)
join n   = n >>= id
```

Таким образом, по-умолчанию, bind, join и fmap циклически определяются друг через друга. Чтобы разорвать этот замкнутый круг, нужно какие-то из этих операций определить явно. Как правило, явно определяются fmap и bind.

Покажем, как определена монадная операция bind для списков и типа Maybe:

```
instance Monad [] where
  xs >>= f = [y | x <- xs, y <- f x]

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= _ = Nothing
```

Если сравнить математическое и «программистское» определения монады, то можно заметить, что естественному преобразованию η соответствует функция return, а естественному преобразованию μ – функция join.

Возвращаясь к нашим «безопасным» функциям, заметим, что теперь квадратный корень от логарифма можно вычислить так:

```
safe_log   5  >>= safe_sqrt -- Just 1.2686362411795196
safe_log (-5) >>= safe_sqrt -- Nothing
safe_log 0.5  >>= safe_sqrt -- Nothing
```

Если где-то в цепочке вычислений возникает ошибка, то происходит быстрый выход из всей цепочки вычислений, остальные функции фактически не вычисляются. Это чем-то напоминает исключения (exceptions) в императивных языках (таких, как C++).

Библиотека функционального программирования

Общее описание библиотеки

При реализации библиотеки функционального программирования funcprog для языка C++ ставилась задача написать библиотеку, с помощью которой на языке C++ можно было бы писать в стиле, близком к стилю языка Haskell.

Важный вопрос – что такое функция с точки зрения этой библиотеки? В первоначальной версии библиотеки под функцией подразумевался объект класса std::function. Этот вариант нас теперь не устраивает, так как мы хотим, чтобы функция исполнялась на графическом ускорителе, а объект класса std::function может исполняться только на CPU (в частности, потому, что в её реализации используются виртуальные функции, которые на GPU непереносимы). Это не может быть и обычная функция, так как её невозможно передать как параметр из CPU в CUDA, так как обычную функцию можно передать только по адресу, а адреса из CPU в CUDA передавать нельзя. Было принято решение в рамках библиотеки funcprog создать класс function2 и считать функцией любой объект этого класса. В объект класса function2 можно преобразовать любой функциональный объект (имеющий функциональный оператор ()), в частности, это может быть лямбда-выражение. Напомним, что в современном языке C++ (начиная с версии C++11) лямбда-выражение начинается с пары квадратных скобок, внутрь которых могут быть помещены переменные, доступные из лямбда-выражения. Для того чтобы объект можно было передавать в CUDA, функциональный оператор должен быть помечен ключевым словом __device__. Для преобразования функционального объекта в объект класса function2 в библиотеке имеется специальная функция _ (подчерк). Недостатком указанного подхода является то, что в метаданные функции попадает не только её прототип (типы параметров и возвращаемое значение), но и реализация (класс объекта) – такова плата за возможность переноса на CUDA. Вот как реализована функция (в библиотеке класс function2):

```
template<typename FuncType, typename FuncImpl> struct function2;

template<typename Ret, typename... Args, typename FuncImpl>
struct function2<Ret(Args...), FuncImpl> {
  using result_type = Ret;
  function2(FuncImpl const& impl) : impl(impl){}
  result_type operator()(Args... args) const {return impl(args...);}
  private: FuncImpl const impl;
};
```

Видно, что, в отличие от класса std::function, в класс function2 передаются два параметра шаблона. Вот как определена вспомогательная функция _ (подчерк), упомянутая ранее:

```
template<typename FuncImpl, typename Ret, typename... Args>
struct function2_type_ {
  using type = function2<Ret(Args...), FuncImpl>; };

template<class F>  // Common case for functors & lambdas
struct function2_type : function2_type<decltype(&F::operator()) >{};

template< class Cls, typename Ret, typename... Args>
struct function2_type<Ret(Cls::*)(Args...)> :
function2_type_<Cls, Ret, Args...>{};

template< class Cls, typename Ret, typename... Args>
struct function2_type<Ret(Cls::*)(Args...) const> :
function2_type_<Cls, Ret, Args...>{};

template<typename F>
using function2_type_t = typename function2_type<F>::type;

template<typename F>
function2_type_t<F> _(F f){ return f; }
```

Приведём пример работы с этой библиотекой:

```
double d=(_([](double x){ return x*x; }) &
  _([](double x){ return x + 1; }))(5); //36
```

В этом примере мы создали две функции, сделали их композицию (с помощью оператора &) и вызвали получившуюся составную функцию с параметром 5. В результате получили число 36.

В библиотеке funcprog достаточно полно реализовано каррирование функций. Для этого в библиотеке реализован оператор применения аргумента к функции с помощью оператора сдвига влево <<. При этом создаётся новая функция, имеющая на один параметр меньше. В частности, если исходная функция имела единственный параметр, то создастся функция без параметров. В библиотеке имеется также функция invoke_f0, которой можно передать любую функцию. Если переданная функция без параметров, то она будет вызвана и вернётся результат её исполнения. Если же переданная функция имеет параметры (один или больше), то будет просто возвращена эта функция. В библиотеке имеется также метафункция remove_f0, принимающая в качестве параметра шаблона тип функции. Если функция без параметров, то в переменной type вернётся тип возвращаемого функцией значения, а если параметры есть, то тип самой функции.

Реализация функторов, аппликативов и монад

Реализация функторов, аппликативов и монад в библиотеке funcprog в чём-то похожа на реализацию этих понятий в языке Haskell. Любой класс может объявить себя функтором, аппликативом или монадой. Для этого достаточно для этого класса реализовать специализацию классов соответственно Functor, Applicative и Monad. В сам класс никаких изменений вносить не требуется.

Любой функтор или монада являются типом с одним параметром. В языке C++ типы с параметром реализуются с помощью шаблонов классов. Рассмотрим определение функтора на примере класса Maybe. Класс Maybe в библиотеке funcprog определён так:

```
template<typename A> struct Maybe;
```

Шаблон класса типом не является, и его нельзя передать как параметр шаблона другого класса. Поэтому определяется ещё один класс (без шаблона) с именем _Maybe (с подчерком впереди). Это уже настоящий класс, его можно передавать как параметр шаблона:

```
struct _Maybe {};
```

Шаблон класса Maybe наследуется от этого класса:

```
template<typename A>
struct Maybe : std::optional<A>, _Maybe {
        ...
};
```

Специализации классов Functor, Applicative и Monad пишутся именно от этого класса _Maybe:

```
template<> struct Functor<_Maybe>{
        ...
};
```

Внутри специализации класса Functor нужно определить статическую функцию fmap. Специализации классов Applicative и Monad определяются аналогично. Для аппликатива методы называются pure и apply, а для монады – mreturn и bind. При реализации статических методов этих классов нужно не забывать про выполнение функторных, аппликативных и монадных законов.

Заметим также, что в библиотеке funcprog в качестве функторного оператора используется оператор деления, а в качестве аппликативного – умножение.

Применение библиотеки для численных методов

Технологически любая задача численного моделирования начинается с построения сетки в области расчета. Причем в областях сложной формы эта сетка, как правило, неструктурная. Интересующие исследователя сеточные функции могут быть заданы как в узлах ячеек, так и в их центрах.

На данном этапе исследования мы рассматриваем только нестационарные задачи и считаем, что для интегрирования по времени используются различные явные схемы, например, в программном комплексе, который мы использовали для перевода на GPU, это явная классическая схема Рунге–Кутты четвертого порядка. Неявные схемы, предполагающие решение линейных систем уравнений, в настоящий момент не рассматривались. Использование описанного подхода для решения задач на установление с использованием неявных схем требует дополнительных разработок, расширяющих возможности представленной библиотеки.

Если метод явный, то вычислять значения сеточных функций в разных точках сетки можно независимо друг от друга, и, следовательно, эти вычисления можно вести параллельно. Таким образом, каждый явный шаг (цикл по индексу сеточной функции) можно распараллеливать на общей памяти. Заметим, что MPI-параллелизация также возможна, но пока не рассматривается. При расчётах на CPU распараллеливание циклов делается, как правило, с помощью OpenMP. При расчётах на CUDA методы распараллеливания свои, и они сильно отличаются от OpenMP. Чтобы скрыть метод распараллеливания от прикладного программиста-математика, реализующего численный метод, предлагается использовать библиотеку функционального программирования funcprog так, как это описывается далее.

Сеточные выражения и сеточные функции

Введём понятие сеточного выражения. Это объект, определённый на всех элементах сетки, то есть для любого объекта, являющегося сеточным выражением, можно узнать, чему равно его значение для любого индекса сетки. Частным случаем сеточного выражения является сеточная функция, которая свои значения просто хранит в памяти и их, если нужно, возвращает. Для сеточных выражений определяется шаблон класса grid_expression, от которого должны наследоваться все классы объектов, являющихся сеточными выражениями (в частности, класс grid_function также пронаследован от класса grid_expression). Таким образом, фраза «объект является сеточным выражением» означает, что

класс этого объекта пронаследован от класса grid_expression. При таком наследовании используется шаблон проектирования CRTP (Curiously Recurring Template Pattern) [13], при котором в базовый класс в качестве параметра шаблона передаётся конечный класс. Про этот шаблон и другие методы метапрограммирования можно прочитать в книгах [14], [15], [17]. Про шаблоны выражений можно также прочитать в [18].

Любое сеточное выражение можно присвоить сеточной функции. Этот оператор присваивания проходит по всем индексам сеточной функции, которой присваивается сеточное выражение, для каждого индекса запрашивает у сеточного выражения его значение и присваивает это значение сеточной функции по данному индексу. Этот оператор присваивания подразумевает, что значения для разных индексов можно вычислять независимо друг от друга, и, значит, их можно вычислять параллельно. Именно в этом операторе присваивания выполняется тот самый внутренний цикл по элементам сеточной функции. Метод распараллеливания этого цикла выбирается оператором присваивания в зависимости от того, каким компилятором компилируется программа. Если это компилятор для CUDA (определена переменная препроцессора __CUDACC__), то распараллеливание осуществляется с помощью CUDA, иначе – с помощью OpenMP. Таким образом, метод распараллеливания скрыт от прикладного программиста внутри этого оператора присваивания. Покажем, как примерно реализован этот оператор присваивания для CPU:

```
template<class GEXP>
void operator=(grid_expression<GEXP, typename GEXP::proxy_type>
    const& gexp0)
{
  GEXP const& gexp = gexp0();
  #pragma omp parallel for
  for(size_t i = 0; i < size(); ++i)
    (*this)[i] = gexp[i];
}
```

Говоря про сеточные функции, нужно упомянуть ещё один аспект. GPU может работать только со своей памятью, это значит, что при работе на GPU сеточная функция должна память под свои данные запрашивать в памяти CUDA. С этим также проблем нет. Сеточные функции устроены так, что при компиляции на CUDA они запрашивают память в CUDA, иначе – в памяти CPU.

Объекты-заместители (proxy)

Шаблон класса сеточного выражения grid_expression определён следующим образом:

```
template<class E, class _Proxy = E>
struct grid_expression;
```

Здесь E – конечный класс, а _Proxy – класс заместителя. По умолчанию (если он не указан) класс заместителя совпадает с конечным классом. Он нужен для создания копии объекта. Дело в том, что единственный способ передачи параметров из памяти основного процессора в память CUDA – это передача по значению (то есть делается копия параметра). Передача по адресу и ссылке невозможна. По значению можно передавать только переменные простых типов (числа и указатели) и объекты классов, содержащие только простые типы. Кроме того, эти объекты не должны содержать виртуальных методов, и вызываемые в них методы должны быть доступны из CUDA. Далеко не все объекты удовлетворяют всем этим требованиям. Если же такой объект всё-таки нужно передать из CPU в CUDA, то для него можно создать объект-заместитель, удовлетворяющий перечисленным требованиям и хранящий все основные данные из основного объекта. Есть и другая причина того, почему не всегда можно хранить ссылки и указатели на объекты даже на CPU. Дело в том, что в сложных выражениях могут появляться временные безымянные переменные, у которых нет постоянной памяти и ссылки и указатели на которые сохранять нельзя. Такие объекты необходимо копировать. Всегда копировать объекты тоже нельзя, так как бывают «большие» объекты (например, вектора данных), которые при копировании делают копию этих данных. Общее правило следующее. Если объект «маленький» и не имеет виртуальных методов, то его, как правило, можно копировать, и класс-заместитель не нужен, иначе такой класс необходим.

Сеточные выражения как функторы, аппликативы и монады

Сеточные выражения можно рассматривать как контейнеры (особенно это справедливо для сеточных функций). В библиотеке funcprog контейнеры (например списки) являются функторами, аппликативами и монадами. Это даёт возможность применять к значениям, хранящимся в них, обычные функции (свойство функторов). Сделаем сеточное выражение также функтором, аппликативом и монадой, чтобы и к сеточным выражениям можно было применять функции. Чтобы понять, как это можно сделать, рассмотрим типичный цикл, вычисляющий новое значение сеточной функции по старой:

```
for(size_t i = 0; i < N; ++i)
  f_new[i] = calculate(f_old[i]);
```

здесь calculate – функция, вычисляющая новое значение в ячейке по

старому. Ей в качестве параметра передаётся старое значение в ячейке. В новом подходе мы хотим, чтобы в этом случае можно было написать что-то типа:

```
f_new = _(calculate) / f_old;
```

Если же для вычислений требуются ещё несколько сеточных функций (назовём их f2 и f3), то вместо

```
for(size_t i = 0; i < N; ++i)
  f_new[i] = calculate(f_old[i], f2[i], f3[i]);
```

мы могли бы написать:

```
f_new = _(calculate) / f_old * f2 * f3;
```

то есть для первой сеточной функции мы применили свойство функтора, а для последующих – аппликатива. Если мы хотим в функцию передать ещё дополнительно некоторое постоянное значение (не зависящее от индекса цикла), то вместо

```
for(size_t i = 0; i < N; ++i)
  f_new[i] = calculate(f_old[i], some_value);
```

можно было бы написать

```
f_new = _(calculate) / f_old * pure(some_value);
```

Таким образом, результатом применения функции к сеточному выражению (или к нескольким сеточным выражениям в случае аппликатива) должно быть также сеточное выражение, то есть у него можно запросить значение по индексу (должен быть реализован оператор []). Сеточными выражениями, помимо сеточных функций, являются также результаты применения функций к сеточным выражениям как к функторам и монадам. Кроме того, сумма и разность двух сеточных выражений, а также произведение и частное сеточного выражения и числа также являются сеточными выражениями.

Покажем, как реализованы функторы, аппликативы и монады для сеточных выражений и докажем, что эти реализации корректны (удовлетворяют требуемым законам). Доказывать теоремы будем методом эквационального рассуждения (equational reasoning). Этот метод доказательства основан на приведении левой и правой частей доказываемого равенства путём эквивалентных преобразований к одинаковому выражению. Пусть теперь f – применяемая функция, а gexp – сеточное выражение.

Функтор. Функция *fmap* принимает функцию с одним параметром и функтор (в нашем случае сеточное выражение) и возвращает тот же функтор (новое сеточное выражение). Это новое сеточное выражение

запоминает параметры функции *fmap* в своих переменных-членах класса (назовём их f и gexp) и реализует оператор [] следующим образом:

```
(fmap f gexp)[i] = f gexp[i]
```

Теорема 1 (Теорема о функторе)**.** *Определённая выше функция fmap удовлетворяет функторным законам.*

Доказательство. Перепишем первый функторный закон полностью (без η-редукции):

```
fmap id gexp = id gexp
```

или (по определению функции id):

```
fmap id gexp = gexp
```

Далее,

```
(fmap id gexp)[i] = -- defenition of fmap
  id gexp[i] =      -- definition of id
  gexp[i]
```

т.е., действительно, fmap id gexp = gexp. Первый закон доказан. Перепишем второй функторный закон без η-редукции:

```
fmap (g . f) gexp = (fmap g . fmap f) gexp
```

Тогда

```
(fmap (g . f) gexp)[i] = -- definition of fmap
  (g . f) gexp[i] =      -- definition of function composition
  g (f gexp[i])
```

С другой стороны:

```
((fmap g . fmap f) gexp)[i] = -- definition of function composition
  (fmap g (fmap f gexp))[i] = -- definition of fmap
  g (fmap f gexp)[i] =        -- definition of fmap
  g (f gexp[i])
```

т.е. действительно, fmap (g . f) gexp = (fmap g . fmap f) gexp. Теорема доказана. Определение функтора корректно. □

Аппликатив. Функция *pure* принимает некоторое значение и «вносит» его в аппликатив. В нашем случае делает из него сеточное выражение. Определим его оператор [] так, чтобы он для любого индекса возвращал одинаковое значение val:

```
(pure val)[i] = val
```

Функция *apply* (аналог оператора (<*>) в языке Haskell) в нашем случае принимает два сеточных выражения: первый (назовём его gexp_f) возвращает функции, а второй (назовём его gexp) – некоторые значения (параметры этих функций). Определим сеточное выражение функции *apply* следующим образом:

```
(apply gexp_f gexp)[i] = gexp_f[i] gexp[i]
```

Теорема 2 (Теорема об аппликативе)**.** *Определёные выше функции pure и apply удовлетворяют аппликативным законам.*

Доказательство. Перепишем аппликативные законы с использованием функции *apply*:

```
apply (pure id) gexp = gexp                          -- Identity
apply (pure f) (pure x) = pure (f x)                 -- Homomorphism
apply u (pure y) = apply (pure ($ y)) u              -- Interchange
apply (apply (apply (pure (.)) u) v) x =
  apply u (apply v x)                                -- Composition
```

Первый закон (Identity):

```
(apply (pure id) gexp)[i] = -- definition of apply
  (pure id)[i] gexp[i] =    -- definition of pure
  id gexp[i] =              -- definition of id
  gexp[i]
```

т.е. apply (pure id) gexp = gexp. Первый закон доказан.

Второй закон (Homomorphism):

```
(apply (pure f) (pure x))[i] = -- definition of apply
  (pure f)[i] (pure x)[i] =    -- definition of pure (2 times)
  f x
```

С другой стороны:

```
(pure (f x))[i] = -- definition of pure
  f x
```

Второй закон доказан. Третий закон (Interchange):

```
(apply u (pure y))[i] = -- definition of apply
  u[i] (pure y)[i] =    -- definition of pure
  u[i] y
```

С другой стороны:

```
(apply (pure ($ y)) u)[i] = -- definition of apply
  (pure ($ y))[i] u[i] =    -- definition of pure
  ($ y) u[i] =              -- definition of function application
  u[i] y
```

Третий закон доказан. Четвёртый закон (Composition):

```
(apply (apply (apply (pure (.)) u) v) x)[i] = -- definition of apply
  (apply (apply (pure (.)) u) v[i] x[i] =     -- definition of apply
  (apply (pure (.)) u)[i] v[i] x[i] =         -- definition of apply
  (pure (.))[i] u[i] v[i] x[i] =              -- definition of pure
  (.) u[i] v[i] x[i] = -- rewrite function composition in infix form
  (u[i] . v[i]) x[i] = -- definition of function composition
  u[i] (v[i] x[i])
```

С другой стороны:

```
(apply u (apply v x))[i] = -- definition of apply
  u[i] (apply v x)[i] =    -- definition of apply
  u[i] (v[i] x[i])
```

Четвёртый закон доказан. Теорема доказана. Определение аппликатива корректно. □

Монада. Монадная функция *return* определена так же, как и аппликативная функция *pure*:

```
(return val)[i] = val
```

Монадная операция *bind* (в языке Haskell и в библиотеке funcprog оператор >>=) принимает монаду (в нашем случае сеточное выражение, обозначим его переменной gexp) и функцию, принимающую обычное (не монадное) значение и возвращающую монаду (сеточное выражение). Определим операцию *bind* следующим образом:

```
(bind gexp f)[i] = (f gexp[i])[i]
```

Теорема 3 (Теорема о монаде)**.** *Определёные выше функции return и bind удовлетворяет монадным законам.*

Доказательство. Перепишем монадные законы в терминах функций return и bind:

```
1. bind (return x) f = f x
2. bind gexp return = gexp
3. bind (bind gexp f) g = bind gexp (\x->bind (f x) g)
```

Первый закон:

```
(bind (return x) f)[i] = -- definition of bind
  (f (return x)[i])[i] =  -- definition of return
  (f x)[i]
```

Первый закон доказан. Второй закон:

```
(bind gexp return)[i] = -- definition of bind
  (return gexp[i])[i] = -- definition of return
  gexp[i]
```

Второй закон доказан. Третий закон:

```
(bind (bind gexp f) g)[i] = -- definition of bind
  (g (bind gexp f)[i])[i] = -- definition of bind
  (g (f gexp[i])[i])[i]
```

С другой стороны:

```
(bind gexp (\x -> bind (f x) g))[i] = -- definition of bind
  ((\x->bind (f x) g) gexp[i])[i] =   -- beta-reduction, substitute
                                      -- gexp[i] instead of x
  (bind (f gexp[i]) g)[i] =          -- definition of bind
  (g (f gexp[i])[i])[i]
```

Результат получился одинаковый. Третий закон доказан. Теорема доказана. □

Итак, сеточные выражения являются функторами, аппликативами и монадами. Это значит, что любую обычную унарную функцию можно «применить» к сеточному выражению (с помощью функции *fmap*). Такое применение вернёт новое сеточное выражение. Любую бинарную функцию можно «применить» к двум сеточным выражениям (с помощью функции *apply*), а любую функцию с n аргументами можно «применить» к n сеточным выражениям. Это можно сделать одной строчкой. Например, путь есть две сеточных функции f и g. Тогда можно написать так:

```
g = _(sin) / f;
```

Т.к. сеточные выражения являются монадами, то из них можно строить цепочки монадных вычислений вида:

```
g = f >>= f1 >>= f2 >>= f3;
```

Здесь f1, f2, f3 – некоторые функции, принимающие обычные (не монадные) значения и возвращающие сеточные выражения.

Примеры

Простейший пример

Рассмотрим функцию axpy из библиотеки BLAS. Эта функция принимает константу a и два вектора (x и y) и модифицирует вектор y по формуле $y[i] += a * x[i]$. Её реализацию для обычного процессора можно записать так:

```
template<typename T>
void axpy(T a, vector<T> const& x, vector &y){
#pragma omp parallel for
  for(int i = 0; i < y.size(); ++i)
    y[i] += a*x[i];
}
```

Эта функция прекрасно распараллеливается, но способ распараллеливания в данной реализации указан явно и для графических ускорителей не подходит. С использованием функционального программирования эту функцию можно было бы переписать следующим образом:

```
template<typename T>
void axpy(T a, grid_function<T> const& x, grid_function<T> &y){
  y = _([](T a, T xi, T &yi, size_t /*i*/){
    yi += a * xi;
  }) / pure(a) * x;
}
```

Лямбда-выражение в теле этой функции принимает в качестве параметров нашу константу a, i-ый элемент сеточной функции x, по ссылке i-ый элемент сеточной функции y и текущий индекс цикла i (который мы игнорируем). Каждому входному параметру (а их у нас два) соответствует один параметр лямбда-выражения, а сеточной функции, которой делается присваивание выражения, соответствует выходной параметр (передаваемый по ссылке) и индекс цикла. Первый параметр (в нашем случае это pure(a)) передаётся как для функтора (через оператор /), а следующие – как для аппликатива (через оператор *). Вот пример обращения к функции axpy:

```
int main(){
  size_t const N = 10;
  math_vector<double> x(N, 2), y(N, 3);
  axpy(5., x, y);
  std::cout << y[0] << std::endl; // 13
  return 0;
}
```

Более сложный пример

В качестве более сложного примера из реальной программы покажем метод, вычисляющий коэффициенты вязкости и теплопроводности смеси газов. Первоначальный исходный текст, использовавший OpenMP, выглядел так:

```
#pragma omp parallel
{
  vector<double> &buf = omp_buffer[omp_get_thread_num()];
  double *MuComp = &buf[0];
  double *LComp = &buf[Ncomp];

  #pragma omp for schedule(static)
  for(int ic = 0; ic < Nc; ic++){
    F_CalcComponent_ViscCoef(ic, MuComp);
    viscous[ic] = CalculateMixFunction(ic, MuComp, USYmWm[ic]);

    CalcComponent_ConductCoef(ic, MuComp, LComp);
    LMix[ic] = CalculateMixFunction(ic, LComp, USYmWm[ic]);

    CalcComponent_DiffusionCoef(ic, MuComp, LComp, Dm[ic]);
```

```
    if(TurbSolver && !(IsNoSlipBC(CellBCType[ic]) && CodeNOISETTE)){
      viscousTurb[ic] = CalcSSTviscousTurb_point(ic);

      double const T = Tmix[ic][0];
      LMix[ic] += viscousTurb[ic] / MCFLParams.Prand_turb *
        F_CalcLoc_Cp(T, USYmWm[ic], Ncomp);

      double const rho_inv = 1 / UA[ic][Var_R];
      for(int icomp = 0; icomp < Ncomp; ++icomp)
        Dm[ic][icomp] += viscousTurb[ic] / MCFLParams.Sc_turb *
          rho_inv;
    }
  }
}
```

У этого исходного текста есть две особенности, на которые хочется обратить внимание. Во-первых, используются глобальные переменные (Nc, Ncomp, TurbSolver и другие), и, во-вторых, для промежуточных вычислений используется вспомогательный буфер (переменная buf), размер которого равен числу потоков OpenMP. При вычислениях на GPU глобальные переменные использовать нельзя, а размер буфера для промежуточных вычислений приходится делать равным размеру сеточной функции. В итоге новый исходный код получается таким:

```
g(Dm) = _([] __DEVICE __HOST (
  // constants
  tVector_proxy<specie_proxy> const specieThermo,
  mcfl_problem_params const _MCFLParams,
  bool turbSolver, bool codeNOISETTE, tArray_proxy<double> const
    Dist_p,
  tArray_proxy<double> const absS_p,
  // input
  const double *UA_ic, const double *USYmWm_ic, const double *TT,
  tBCtype cellBCType,
  // output
  double *TempBuf_ic, double &viscous_ic, double &viscousTurb_ic,
  double &LMix_ic, double *Dm_ic, int ic)
{
  int const ncomp = (int)specieThermo.size();
  double
    *MuComp = TempBuf_ic,
    *LComp = MuComp + ncomp;

  double const T = TT[0];

  F_CalcComponent_ViscCoef(specieThermo, _MCFLParams.flag_mu,
    T, MuComp);
  viscous_ic = F_CalculateMixFunction(ncomp, MuComp, USYmWm_ic);
```

```
  F_CalcComponent_ConductCoef(specieThermo, _MCFLParams, USYmWm_ic,
    MuComp, T, LComp);
  LMix_ic = F_CalculateMixFunction(ncomp, LComp, USYmWm_ic);

  F_CalcComponent_DiffusionCoef(specieThermo, _MCFLParams, UA_ic,
    USYmWm_ic, T, MuComp, LComp, TempBuf_ic, Dm_ic);

  if(turbSolver && !(IsNoSlipBC(cellBCType) && codeNOISETTE)){
    viscousTurb_ic = F_CalcSSTviscousTurb_point(UA_ic, viscous_ic,
       Dist_p[ic], absS_p[ic]);

    LMix_ic += viscousTurb_ic / _MCFLParams.Prand_turb *
      F_CalcLoc_Cp(specieThermo, T, USYmWm_ic);

    double const rho_inv = 1 / UA_ic[Var_R];
    for(int icomp = 0; icomp < ncomp; ++icomp)
      Dm_ic[icomp] += viscousTurb_ic / _MCFLParams.Sc_turb * rho_inv;
  }
})
  // constants
  / pr(SpecieThermo_proxy) * pure(MCFLParams) * pure(TurbSolver)
  * pure(CodeNOISETTE) * pr(Dist) * pr(absS)
  // input
  * g(UA) * g(USYmWm) * g(Tmix) * g(CellBCType)
  // output
  * g(TempBuf) * g(viscous) * g(viscousTurb) * g(LMix);
```

Глобальные константы приходится передавать явно с помощью функции pure, а для промежуточных вычислений используется буфер TempBuf, размер которого равен числу узлов сетки. Обратите также внимание на то, что все параметры передаются по значению (то есть делается их копия).

При вызове используются также вспомогательные функции pr и g. Функция pr преобразует свой аргумент в соответствующий ему объект-заместитель (proxy), как правило это вектор значений, и передаёт его с помощью функции pure. Функция g преобразует вектор значений в сеточное выражение, которое затем можно передавать в функтор.

Ещё одна важная часто возникающая задача – это редукция (например, поиск максимального значения или сумма всех значений). Редукцию также очень важно выполнять параллельно. В исходной версии программы редукция выполнялась средствами OpenMP, но теперь мы этот механизм использовать не можем. К счастью, в современном языке C++ (начиная со стандарта языка C++17) у функций редукции (таких как accumulate и reduce) есть параллельная версия. При работе на CUDA мы используем библиотеку thrust, входящую в состав CUDA Computing Toolkit. В этой библиотеке также имеются параллельные функции редукции, работающие на GPU. Таким образом, при работе на CUDA мы используем

параллельные функции редукции из библиотеки thrust, при работе на CPU, если имеются параллельные версии функций редукции (если компилятор их поддерживает), то используются они, иначе редукуция делается последовательно.

Сравнение эффективности

Для оценки эффективности предлагаемого подхода предлагается задача, моделирующая эксперимент, осуществленный в трубе L2K (Германия) [19]. Осесимметричное тело, состоящее из двух конических и цилиндрической поверхностей, помещается в набегающий поток воздуха со следующими характеристиками: $M_\infty = 4.7$, $p_\infty = 272$ Па, $T_\infty = 764$ К, $Y_{O_2} = 0.245$, $Y_{N_2} = 0.755$. Геометрия задачи представлена на рисунке 2. Твердое тело скомбинировано из двух частей с разными термодинамическими характеристиками: головная часть сделана из материала класса UHTC (Ultra High Temperature Material), а задняя цилиндрическая часть – медная.

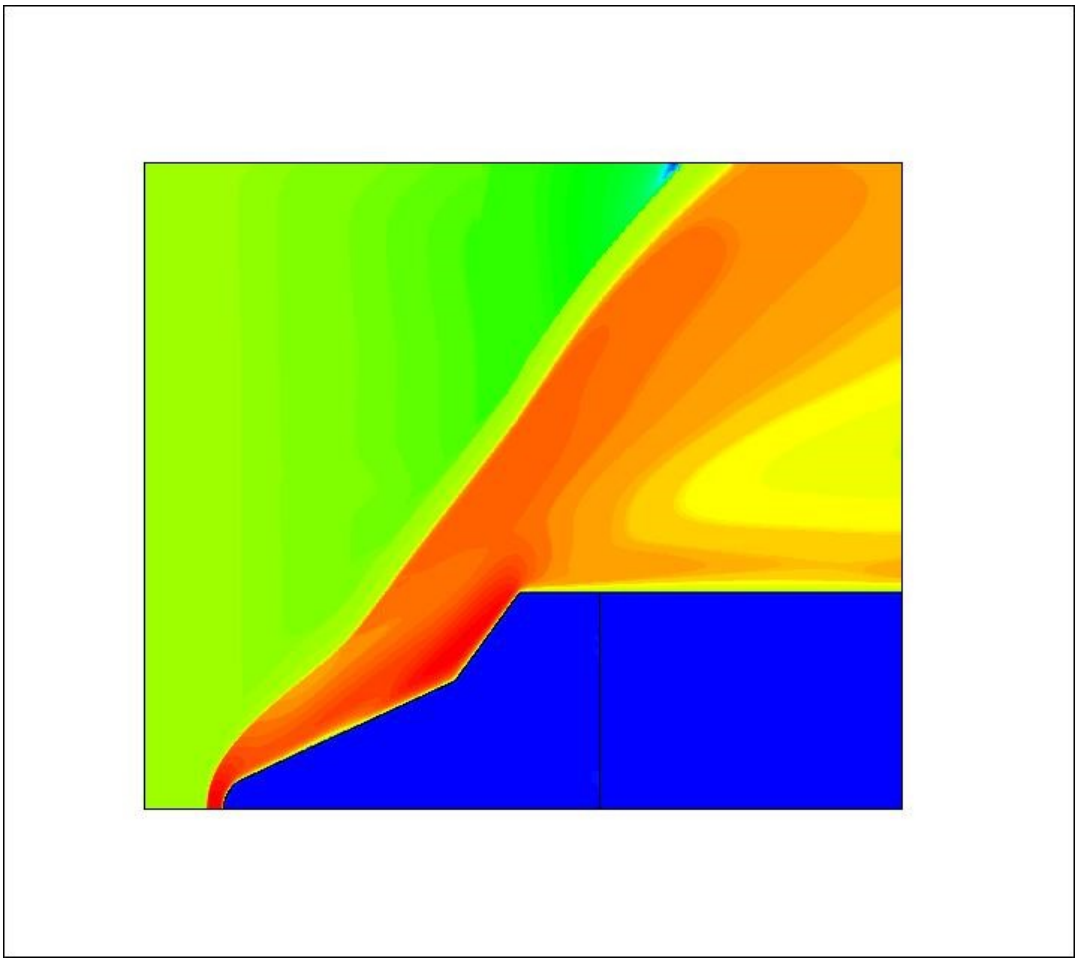

Рис. 2. Геометрия задачи со структурой ударной волны.

Начальная температура тела $T = 300K$. Решается сопряженная задача в двумерной постановке. Моделирование происходит на основе решения уравнений Навье-Стокса в многокомпонентном газе и уравнения теплопроводности в твердом теле с помощью оригинальной методики на основе явных Чебышевских итераций [20]-[22]. Размер сетки 297192 узлов.

Мы решали эту задачу с помощью гибридного суперкомпьютера К-60, установленного в Центре коллективного пользования ИПМ им. М.В. Келдыша РАН [23], на одном узле в трёх режимах: на обычном процессоре (2 x Intel Xeon Gold 6142 v4, 16 ядер) с использованием OpenMP и на GPU (nVidia Volta GV100GL) с использованием OpenCL и с использованием описанного подхода. Ускорение по сравнению с CPU при использовании OpenCL составило около 20 раз, а при использовании нашего подхода – около 12 раз. Ускорение получилось не таким большим, как при использовании OpenCL, но с учётом указанных преимуществ (прежде всего единый исходный код) его можно считать приемлемым.

Заключение

Декларативные языки программирования, к которым относятся и функциональные языки, позволяют, в отличие от императивных языков, к которым относятся большинство языков программирования, на которых реализуются численные методы, кратко и в то же время достаточно ясно записывать желаемый результат, не вдаваясь в подробности реализации. Конкретная реализация может быть скрыта в языке и зависеть от текущего программно-аппаратного окружения. Язык C++ оказался достаточно мощным, чтобы позволить реализовать на нём библиотеку функционального программирования, позволяющую писать программы в стиле, близком к стилю чисто функциональных языков, таких как Haskell. Такие понятия из мира функционального программирования, как функторы и монады, реализованные в библиотеке функционального программирования, оказались очень удобным средством для переноса численных задач на графические ускорители CUDA. Сеточные выражения были определены как функторы, аппликативы и монады, что позволило применять функции к хранящимся в них значениям. Сами эти функции можно строить, комбинируя сложные функции из простых, что является также сильной стороной функционального программирования. Авторам представляется, что за функциональным программированием будущее технологии программирования вообще и решения численных задач в частности. Мы надеемся, что данная работа будем нашим вкладом в этом направлении. С дополнительной информацией о языке C++ можно ознакомиться в источниках [24] - [30].

Библиографический список

1. TOP 500. URL: https://www.top500.org
2. NVIDIA. URL: https://www.nvidia.com
3. TOP 50. URL: http://top50.supercomputers.ru
4. OpenCL. URL: https://www.khronos.org/opencl/
5. OpenACC. URL: https://www.openacc.org
6. CUDA Zone. URL: https://developer.nvidia.com/cuda-zone
7. Краснов М. М. Библиотека функционального программирования для языка C++ // Программирование, 2020, №5, с. 47-59. DOI: 10.31875/S0132347420050040.
8. Краснов М. М. Операторная библиотека для решения трёхмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA // Математическое моделирование, 2015, т.27, № 3, с. 109-120. URL: http://www.mathnet.ru/links/38633e7a627ab2ce1527ae4a092be72f/mm3585.pdf
9. Краснов М. М. Кандидатская диссертация "Сеточно-операторный подход к программированию задач математической физики". Автореферат. URL: http://keldysh.ru/council/1/2017-krasnov/avtoref.pdf
10. Haskell language. URL: https://www.haskell.org/
11. Маклейн С. Категории для работающего математика / Перевод с англ. под ред. В.А. Артамонова. - М.: ФИЗМАТЛИТ, 2004. - 352 с. - ISBN 5-9221-0400-4.
12. Bartosz Milewski. Category Theory for Programmers. URL: https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v1.3.0/category-theory-for-programmers.pdf
13. J.O. Coplien. Curiously recurring template patterns. C++ Report, February 1995, pp. 24-27.
14. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley. – 2004. 400 с. ISBN 978-0-321-22725-6.
15. Краснов М. М. Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им. М.В. Келдыша, 2017. 84 с. DOI: 10.20948/mono-2017-krasnov.

16. Краснов М. М. Применение символьного дифференцирования для решения ряда вычислительных задач // Препринты ИПМ им. М.В. Келдыша. 2017. №4. 24 с. DOI: 10.20948/prepr-2017-4.

17. Краснов М. М. Применение функционального программирования при решении численных задач // Препринты ИПМ им. М.В. Келдыша. 2019. №114. 36 с. DOI: 10.20948/prepr-2019-114.

18. T. Veldhuizen, Expression Templates. C++ Report, Vol. 7 № 5, June 1995, pp. 26-31.

19. C. Murty, P.Manna and D.Chakraborty Conjugate heat transfer analysis in high speed flows. Proceedings of Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering, 2013. DOI: 10.1177/0954410012464920

20. O.B. Feodoritova, M.M. Krasnov and V.T. Zhukov. A Numerical Method for Conjugate Heat Transfer Problems in Multicomponent Flows. //J. Phys.: Conf. Ser., 2021. Vol. 2028 012024, DOI: 10.1088/1742-6596/2028/1/012024.

21. Жуков В.Т., Новикова Н.Д., Феодоритова О.Б, А.П. Дубень. Явно-итерационная схема для интегрирования по времени системы уравнений Навье–Стокса. // Матем. моделирование. 32:4 (2020), 57–74.

22. Жуков В.Т., Новикова Н.Д., Феодоритова О.Б. Об одном подходе к интегрированию по времени системы уравнений Навье–Стокса. // Ж. вычисл. матем. и матем. физ. 2020. Т. 60. № 2. С. 267–280.

23. Вычислительный комплекс К-60.
URL: https://www.kiam.ru/MVS/resourses/k60.html.

24. Bjarne Stroustrup. The C++ Programming Language, Fourth Edition. Addison-Wesley, 2013. ISBN 978-0-321-56384-2, 1368 c.

25. Bjarne Stroustrup. Programming: Principles and Practice Using C++, Second Edition. Addison-Wesley, 2013, 1312 c. ISBN 978-0-321-99278-9.

26. Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley, 1994, 480 c. ISBN 978-0-201-54330-8.

27. Bjarne Stroustrup. A Tour of C++. Addison-Wesley, 2014, 192 c. ISBN 978-0-321-95831-0.

28. Бьерн Страуструп. Программирование: Принципы и практика с использованием C++, второе издание. пер. с англ., Вильямс, 2016, 1328 с. ISBN 978-5-8459-1949-6, 978-0-321-99278-9.

29. Бьерн Страуструп. Дизайн и эволюция языка C++. ДМК Пресс, 2016, 446 с. ISBN 978-5-97060-419-9, 978-0-201-54330-8.

30. The C++ Resources Network. URL: http://www.cplusplus.com/