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

Препринты ИПМ • Препринт № 2 за 2024 г.

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

М.М. Краснов

Применение монадных вычислений при решении численных задач

Рекомендуемая форма библиографической ссылки: Краснов М.М. Применение монадных вычислений при решении численных задач // Препринты ИПМ им. М.В.Келдыша. 2024. № 2. 24 с.
https://doi.org/10.20948/prepr-2024-2
https://library.keldysh.ru/preprint.asp?id=2024-2

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

Применение монадных вычислений при решении численных задач

Москва— 2024

Краснов М. М.

Применение монадных вычислений при решении численных задач

Данная работа является дальнейшим развитием исследований по применению функционального программирования для численных методов. В частности, функциональное программирование может помочь в перенесении программ на графические ускорители с технологией CUDA. В предыдущих работах основной упор делался на функторах (и аппликативных функторах). Теоретические основы монадных вычислений излагались, но в настоящей работе основной упор делается на их практическом применении. Один из базовых принципов функционального программирования – композиция функций, которая позволяет строить из простых функций сложные и, таким образом, упрощает написание сложных программ. Монадные вычисления позволяют строить цепочки сложных вычислений. Такие цепочки – это тоже, в некотором смысле, композиция функций, но на более высоком, монадном уровне (монадная композиция).

Ключевые слова: C++; функциональное программирование; функторы; монады; численные методы; CUDA

Mikhail Mikhailovich Krasnov

Application of monadic calculations in solving numerical problems

This work is a further development of research on the use of functional programming for numerical methods. In particular, functional programming can help port programs to graphics accelerators with CUDA technology. Previous work has focused on functors (and applicative functors). The theoretical foundations of monadic computing have been outlined, but this paper focuses on its practical applications. One of the basic principles of functional programming is function composition, which allows you to build complex functions from simple ones and, thus, simplifies the writing of complex programs. Monad calculations allow you to build chains of complex calculations. Such chains are also, in a sense, a composition of functions, but at a higher, monadic level (monadic composition).

Key words: C++; Functional programming; Functors; Monads; Numerical methods; CUDA

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

Оглавление

Введение

В работе [1] рассказывается о том, как функциональное программирование можно применять для переноса программ, реализующих численные методы, на графические ускорители с технологией CUDA. Эта работа базируется на написанной ранее библиотеке функционального программирования для языка C++ [2], в ней реализованы многие основные понятия из мира функционального программирования, и она позволяет писать на языке C++ (на котором, как правило, в наше время пишутся численные методы) в стиле, близком к стилю языка функционального программирования Haskell [3]. В частности, в библиотеке реализованы функторы и монады. На эти понятия можно взглянуть с двух точек зрения – программистской и математической. То, что видно с этих двух точек зрения, несколько отличается друг от друга, но взаимосвязано, и полезно видеть и понимать эту взаимосвязь. В основе функционального программирования лежит математическая теория – теория категорий, с основами которой можно ознакомиться, например, в работах [4, 5]. В работе [1] также даётся краткое введение в теорию категорий.

Хотя в работе [1] давались основные понятия функционального программирования (функторов и монад), с практической точки зрения основной упор делался на функторах. Монады остались как бы «за кадром». Настоящая работа в какой-то степени восполняет этот пробел. Чтобы работа оставалась читаемой без необходимости привлекать дополнительные источники, в начале работы делается (очень) краткое введение в теорию категорий. Затем излагается взгляд на функторы и монады с точки зрения программирования. Далее приводятся основные результаты, касающиеся функторов, из предыдущих работ. В конце приводятся основные новые результаты практического применения монад для численных методов.

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 [3]) 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
```

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 [4] и [5]. **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. Пример такой совокупности – совокупность (класс) всех множеств.

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.

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

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

и для двух морфизмов $f\colon A \rightarrow B$ и $g\colon B \rightarrow C$ 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 $\eta_X\colon F(X) \rightarrow G(X)$ в категории **D**, называемый компонентой η в X, так, что для любого морфизма $f\colon X \rightarrow Y$ в категории **C** диаграмма (в категори **D**), изображённая ниже, коммутативна.

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

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

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

Таким образом, естественное преобразование задаёт отображение объектов в категории **C** в морфизмы в категории **D**.

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

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):

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

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 (высшего порядка) с одним параметром, принимающим (простую) функцию, принимающую и возвращающую обычные значения (например, числа), 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 – полиморфная 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
```

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.

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 = f <$> m -- fmap
  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 функции pure и 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

При реализации библиотеки функционального программирования funcprog2 для языка C++ ставилась задача написать библиотеку, с помощью которой на языке C++ можно было бы писать в стиле, близком к стилю языка Haskell. Под функцией в этой библиотеке подразумевается объект класса function2 (подробности – в работе [1]).

Реализация функторов, аппликативов и монад

Реализация функторов, аппликативов и монад в библиотеке funcprog2 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 funcprog2 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 mbind. 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 (вместо (<$>)) используется оператор деления, а в качестве аппликативного (вместо (<*>)) – оператор умножения.

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 funcprog2 так, как это описывается далее.

Сеточные выражения и сеточные функции

Введём понятие сеточного выражения. Это объект, определённый на всех элементах сетки, то есть у 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) [6], при котором в базовый класс в качестве параметра шаблона передаётся конечный класс. Про этот шаблон и другие методы метапрограммирования можно прочитать в книгах [7], [8],

[10]. Про шаблоны выражений можно также прочитать в [11].

Итак, у любого сеточного выражения должен быть определён следующий оператор (назовём его итерационным оператором):

```
value_type operator[](size_t i) const {
        ... // operator body
}
```

где i – индекс сетки. Для разных значений этого индекса вызов этого оператора может быть произведён одновременно (параллельно), нельзя делать никаких допущений относительно порядка вызова этого оператора с разными значениями индекса сетки. Тип возвращаемого этим оператором значения (value_type) у каждого сеточного выражения может быть свой.

Запуск параллельных вычислений осуществляется с помощью функции par_execute из библиотеки funcprog2. Эта функция принимает два параметра. Первый параметр (типа size_t) – число итераций параллельного цикла. Обычно это число равно количеству элементов сетки (её размеру). В качестве второго параметра передаётся сеточное выражение, итерационный оператор которого в качестве результата выполнения возвращает одноместную функцию, также принимающую индекс цикла. Именно эта функция выполняет все вычисления для каждого индекса сетки (параллельно). Пример вызова функции par_exeecute будет приведён ниже. Метод распараллеливания цикла по элементам сетки выбирается 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 библиотеки.

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.

Сеточные выражения как функторы, аппликативы и монады

Сеточные выражения можно рассматривать как контейнеры (особенно это справедливо для сеточных функций). В библиотеке funcprog2 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)
  calculate(f_old[i], f_new[i], i);
```

здесь calculate – функция, вычисляющая новое значение в ячейке по-старому. Ей в качестве параметра передаётся старое значение в ячейке и индекс сетки. В новом подходе мы хотим, чтобы в этом случае можно было написать что-то типа:

```
par_execute(N, _(calculate) / f_old * f_new);
```

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

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

мы могли бы написать:

```
par_execute(N, _(calculate) / f_old * f2 * f3 * f_new);
```

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 чтобы вместо

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

можно было бы написать

```
par_execute(N, _(calculate) / pure(some_value) * f_old * f_new);
```

Обратим внимание на то, что функция par_execute всегда последним параметром передаёт в функцию сеточный индекс, то есть этот параметр как бы «не считается». Например, когда мы говорим про одноместные функции (для функтора), в реальности это должна быть двухместная функция. Аналогично для аппликатива. Если мы хотим передать в функцию N сеточных выражений, то у функции должно быть N+1 параметров (плюс один последний параметр для сеточного индекса).

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 индексный 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 являются сеточными выражениями.

Покажем, как реализованы функторы, аппликативы и монады для сеточных выражений. 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]
```

Или на языке C++:

```
auto operator[](size_t i) const {
    return f << gexp[i];
}
```

Здесь оператор << – это оператор применения функции к параметру. В результате получается функция, у которой на один параметр меньше, чем у функции f, и которая при вызове вызывает функцию f, передавая ей в качестве первого параметра gexp[i]. В функциональном программировании это называется *каррированием*, в стандартной библиотеке языка C++ это делет функция bind.

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

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 функцию, а второй (назовём его gexp) – некоторые значения (параметры этой функции). Определим сеточное выражение функции *apply* следующим образом:

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

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

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 (О монаде)**.** *Определёные выше функции return и bind удовлетворяет монадным законам.*

В работе [1] все три теоремы (о функторе, аппликативе и монаде) доказываются.

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){
  par_execute(y.size(), _([](T a, T xi, T &yi, size_t /*i*/){
    yi += a * xi;
  }) / pure(a) * x * y;
}
```

Лямбда-выражение в теле этой функции принимает в качестве параметров константу *a*, *i* элемент сеточной функции *x*, по ссылке *i*

элемент сеточной функции *y* и текущий индекс цикла *i* (который мы игнорируем). Таким образом, функция (лямбда-выражение) имеет 4 параметра, первым трём соответствуют сеточные выражения, передаваемые первое как функтор (через оператор /, в нашем случае это *pure(a)*), а следующие два – как аппликатив (через оператор *, это сеточные функции *x* и *y*). Последний, четвёртый, обязательный параметр – это сеточный индекс. Вот пример обращения к функции *axpy*:

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

Пример использования монад

Как уже говорилось выше, монады позволяют осуществлять высокоуровневую (монадную) композицию функций. Простейший пример такой композиции – последовательное исполнение монадных функций. Для этого в библиотеке funcprog2 имеется оператор >>. Рассмотрим в качестве примера вычисление параметров компонент смеси газов. В данном примере последовательно вычисляются коэффициенты вязкости отдельных компонент, вязкость смеси, коэффициенты теплопроводности отдельных компонент и смеси в целом, а также коэффициенты диффузии отдельных компонент. Вот как выглядит код для обычного процессора (CPU), использующий для распараллеливания на общей памяти OpenMP:

```
#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]);
  }
}
```

У этого исходного текста есть две особенности, на которые хочется обратить внимание. Во-первых, используются глобальные переменные (Nc, Ncomp 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:

```
par_execute(Nc, _([] __DEVICE __HOST (
  // constants
  tVector_proxy<specie_proxy> const specieThermo,
  mcfl_problem_params const _MCFLParams,
  // input
  const double *UA_ic, const double *USYmWm_ic, const double *TT,
  // output
  double *TempBuf_ic, double &viscous_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);
})
  // constants
  / pure(SpecieThermo) * pure(MCFLParams)
  // input
  * UA * USYmWm * Tmix
  // output
  * TempBuf * viscous * LMix * Dm;
```

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

Эти вычисления можно разбить на несколько частей, вызываемых последовательно (но в одном параллельном цикле). Преимущество такого подхода состоит в том, что эти отдельные части можно использовать многократно и в других местах. Вот как это можно сделать:

```
auto const calc_ViscCoef = _([] __DEVICE __HOST (
  // constants
  tVector_proxy<specie_proxy> const specieThermo,
  double flag_mu,
  // input
  const double *USYmWm_ic, const double *TT,
  // output
  double *MuComp, size_t /*ic*/)
{
  F_CalcComponent_ViscCoef(specieThermo, flag_mu, TT[0], MuComp);
})
  // constants
  / pure(SpecieThermo) * pure(MCFLParams.flag_mu)
  // input
  * USYmWm * Tmix
  // output
  * TempBuf;

auto const calc_viscous = _([] __DEVICE __HOST (
  // constants
  int ncomp,
  // input
  const double *MuComp, const double *USYmWm_ic,
  // output
  double &viscous_ic, size_t /*ic*/)
{
  viscous_ic = F_CalculateMixFunction(ncomp, MuComp, USYmWm_ic);
})
  // constants
  / pure((int)SpecieThermo.size())
  // input
  * TempBuf * USYmWm
  // output
  * viscous;

auto const calc_ConductCoef =  _([] __DEVICE __HOST (
  // constants
  tVector_proxy<specie_proxy> const specieThermo,
  mcfl_problem_params const _MCFLParams,
  // input
  const double *USYmWm_ic, const double *TT,
  // output
  double *TempBuf_ic, size_t /*ic*/)
{
  F_CalcComponent_ConductCoef(specieThermo, _MCFLParams, USYmWm_ic,
    TempBuf_ic, TT[0], TempBuf_ic + specieThermo.size());
```

```
})
  // constants
  / pure(SpecieThermo) * pure(MCFLParams)
  // input
  * USYmWm * Tmix
  // output
  * TempBuf;

auto const calc_MixFunction = _([] __DEVICE __HOST (
  // constants
  int ncomp,
  // input
  const double *TempBuf_ic, const double *USYmWm_ic,
  // output
  double &LMix_ic, size_t /*ic*/)
{
  LMix_ic = F_CalculateMixFunction(ncomp, TempBuf_ic + ncomp,
     USYmWm_ic);
})
  // constants
  / pure((int)SpecieThermo.size())
  // input
  * TempBuf * USYmWm
  // output
  * LMix;

auto const calc_DiffusionCoef = _([] __DEVICE __HOST (
  // constants
  tVector_proxy<specie_proxy> const specieThermo,
  mcfl_problem_params const _MCFLParams,
  // input
  const double *UA_ic, const double *USYmWm_ic, const double *TT,
  const double *TempBuf_ic,
  // output
  double *Dm_ic, size_t /*ic*/)
{
  F_CalcComponent_DiffusionCoef(specieThermo, _MCFLParams, UA_ic,
    USYmWm_ic, TT[0], TempBuf_ic, TempBuf_ic + specieThermo.size(),
    TempBuf_ic, Dm_ic);
})
  // constants
  / pure(SpecieThermo) * pure(MCFLParams)
  // input
  * UA * USYmWm * Tmix * TempBuf
  // output
  * Dm;

par_execute(Nc, calc_ViscCoef >> calc_viscous >> calc_ConductCoef
  >> calc_MixFunction >> calc_DiffusionCoef);
```

В следующем, более сложном, примере первая монадная функция

передаёт результат вычислений во вторую монадную функцию. Как уже писалось выше, для этой цели служит монадный оператор (>>=). Для примера разобьём монадную функцию calc_ViscCoef на две части (функции). Первая будет вычислять температуру (доставать её из сеточной функции), а вторая будет её принимать непосредственно в качестве своего параметра. Вот как это будет выглядеть:

```
auto const get_T = _([] __DEVICE __HOST (
  const double *TT, size_t /*ic*/){ return TT[0]; }
) / Tmix;

auto const calc_ViscCoef = _([] __DEVICE __HOST (
// constants
  tVector_proxy<specie_proxy> const specieThermo,
  double flag_mu,
  // input
  const double *USYmWm_ic,
  // output
  double *MuComp, double T, size_t /*ic*/)
{
  F_CalcComponent_ViscCoef(specieThermo, flag_mu, T, MuComp);
})
  / pure(SpecieThermo) * pure(MCFLParams.flag_mu)
  * USYmWm * TempBuf;
```

Обратим внимание на то, что дополнительные параметры (для которых нет сеточных выражений, передаваемых как функтор или аппликатив) идут в самом конце списка параметров непосредственно перед индексом сетки. Теперь параллельный цикл будет выглядеть так:

```
par_execute(Nc, (get_T >>= calc_ViscCoef) >> calc_viscous
  >> calc_ConductCoef >> calc_MixFunction >> calc_DiffusionCoef);
```

Про оператор >>= в языке C++ нужно иметь в виду, что он, как и любой другой оператор присваивания, правоассоциативен. Это значит, что если несколько таких операторов идут подряд, то нужно явно указывать скобки. Например:

```
par_execute(N, (f1 >>= f2) >>= f3);
```

Если нужно вычислить более одного параметра, то монадные функции, вычисляющие эти параметры, следует разделять оператором ^. Например:

```
par_execute(N, get_T ^ calc_P >>= _([] __DEVICE __HOST
  (arguments..., double T, double P, size_t i)
  {
    ... // Calculations
  }) / p(first_arg) * other_args...);
```

Заключение

Функторы и монады из мира функциональных языков программирования оказались удобным инструментом, помогающим записывать численные алгоритмы в форме, позволяющей абстрагироваться от способа распараллеливания вычислений на общей памяти (CPU или GPU) и писать программы, компилирующиеся без изменений исходного кода для разных вычислительных платформ (CPU или CUDA). Монады позволяют создавать многократно используемые монадные функции (сеточные выражения), которые затем можно комбинировать друг с другом, создавая, как из кирпичиков, сложные вычислительные модули из простых. Написанная автором библиотека функционального программирования для языка C++ позволяет писать на языке C++ в стиле, близком к стилю функциональных языков программирования, таких, как Haskell. В этой библиотеке сеточные выражения сделаны функторами, аппликативами и монадами, и при работе с сеточными выражениями программисту становится доступным весь арсенал функционального программирования. С дополнительной информацией о языке C++ можно ознакомиться в источниках [12] - [18].

Библиографический список

1. 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. — 10.20948/prepr-2022-51

2. Краснов М. М. Библиотека функционального программирования для языка C++ // Программирование, 2020, №5, с. 47-59. DOI: 10.31875/S0132347420050040.

3. Haskell language. URL: https://www.haskell.org/

4. Маклейн С. Категории для работающего математика / Перевод с англ. под ред. В.А. Артамонова. - М.: ФИЗМАТЛИТ, 2004. - 352 с. - ISBN 5-9221-0400-4.

5. Bartosz Milewski. Category Theory for Programmers. URL: https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v1.3.0/category-theory-for-programmers.pdf

6. J.O. Coplien. Curiously recurring template patterns. C++ Report, February 1995, pp. 24-27.

7. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley. – 2004. 400 c. ISBN 978-0-321-22725-6.

8. 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.

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

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

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

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

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

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

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

16. 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.

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

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