Применение функционального программирования для численных методов

М.М. Краснов, ИПМ им. М.В. Келдыша РАН

В последние 10-15 лет большое распространение получили графические ускорители (GPU), позволяющие существенно ускорить решение численных задач (которыми активно занимается наш институт). В нашем институте есть несколько кластеров, оснащённых графическими ускорителями (К-10, К-100, К-60). На многих клиентских рабочих станциях также установлены графические ускорители с вычислительными возможностями. Но по-прежнему остаётся открытым вопрос о том, как переносить существующие программы, реализующие численные методы, на GPU, или, для новых программ, как из нужно писать так, чтобы программы работали и на CPU, и на GPU.

Имеется несколько технологий программирования для GPU. Это CUDA, OpenCL, OpenACC. Каждая из этих технологий имеет свои плюсы и минусы. Технологии CUDA и OpenACC имеют важное преимущество в том, что они позволяют иметь единый исходный текст для CPU и для GPU, который является частью основной программы и компилируется вместе с ней. Основной недостаток CUDA состоит в том, что он рассчитан только на GPU компании NVIDIA и не работает, например, на GPU от компании AMD или Intel. Для нас этот недостаток оказывается неважным, так как на практике мы имеем дело только с GPU от NVIDIA. Основным недостатком технологии OpenACC является отсутствие (у нас) компиляторов с поддержкой этой технологии).

Ещё одним преимуществом технологии CUDA является относительная простота отладки. Во-первых, непосредственно из кода, работающего на GPU, можно выводить (например, отладочную информацию) в стандартный вывод с помощью функции printf. Во-вторых, NVIDIA предоставляет возможность отладки кода на GPU из стандартных средств разработки программ (например, из Microsoft Visual Studio) с контрольными точками, просмотром значений переменных и другими удобствами отладки.

Технология OpenCL является открытым стандартом и реализована практически для всех типов GPU (от NVIDIA, AMD и Intel). С помощью OpenCL можно программировать и для CPU. Но при этом вам предоставляется возможность писать только функции на языке C (а не C++, к которому все уже привыкли) без доступа к глобальным переменным и без объектно-ориентированного программирования (ООП), к которому тоже все уже успели привыкнуть. Всё, что требуется этим функциям для работы, нужно передавать

через параметры простых типов (числа или указатели на массивы данных). Исходный текст для OpenCL компилируется при исполнении основного кода программы (а не вместе с ней), отладка этого кода стандартными средствами отладки невозможна. Поэтому численные методы для CPU пишутся, как правило, для обычных компиляторов и отлаживаются стандартными средствами. Затем этот уже отлаженный код переносится в функции (ядра, kernel) для OpenCL и запускается на счёт на GPU. Приходится иметь две версии исходного кода (для CPU и для GPU), и при изменениях в алгоритме вносить правки в оба текста, что неудобно.

Если рассмотреть ситуацию в мире, то технология OpenACC в перспективе представляется основной. Она является логическим расширением технологии OpenMP, которая позволяла распараллеливать вычисления между ядрами CPU. В OpenACC та же парадигма переносится на потоки GPU. Проблемы, конечно же, имеются. Потоки на CPU имеют доступ к глобальным переменным, которые часто активно используются. Из GPU доступа к глобальным переменным нет, поэтому переписывание из OpenMP в OpenACC будет непростым. Современный компилятор gcc (начиная с версии 12.6) имеет поддержку OpenACC (для этого нужно установить к нему специальное бесплатное расширение) и работает с GPU как от NVIDIA, так и от AMD. Надо будет это внимательнее поисследовать. В качестве основного недостатка можно упомянуть то, что для многих (для меня в том числе) основным средством разработки является Microsoft Visual Studio под Microsoft Windows, для которого OpenACC пока нет. Наши коды до сих пор были свободно переносимы между разными операционными системами, терять это свойство пока не хочется.

Работы по облегчению переноса программного кода, реализующего численные методы, ведутся давно разными коллективами разработчиков, в том числе и в нашем институте. Тут можно упомянуть язык программирования Норма (https://www.keldysh.ru/pages/norma/) и систему DVM (http://dvm-system.org/ru/). И язык Норма, и система DVM были написаны достаточно давно (в конце прошлого века) и в последние годы, после появления графических ускорителей, были адаптированы под CUDA. Основным недостатком языка Норма является то, что существующую программу нужно не просто переписать под новую технологию, но переписать на совершенно другой и непривычный язык программирования (в настоящее время численные методы, как правило, пишутся на языке C++). Это часто может оказаться неприемлемым.

Система DVM, на первый взгляд, выглядит гораздо привлекательнее. Компиляторы языков C-DVMH и Fortran-DVMH преобразуют входную программу в параллельную программу, использующую стандартные

технологии программирования MPI, OpenMP и CUDA. В состав DVM-системы входят средства функциональной отладки и отладки эффективности DVMH-программ. Программа на DVM выглядит похоже на программу, использующую OpenMP или OpenACC. Управление компиляцией осуществляется с помощью прагм (#pragma dvm …). По всей видимости, непросто будет с использованием глобальных переменных (которые в существующих программных кодах активно используются). Очевидно, переписывание существующего кода не будет простым. Надо разбираться.

Мною в 2017 году была защищена кандидатская диссертация (в нашем диссертационном совете) на тему «Сеточно-операторный подход к программированию задач математической физики», в которой я предложил метод программирования, основанный на сеточных операторах, и который подразумевал для прикладного программиста единый исходный код, работающий и на CPU, и на GPU. Эта работа также была нацелена на решение проблемы переноса программного кода на GPU.

Настоящая работа является, в некотором смысле, продолжением той работы, но на более высоком уровне. Функциональное программирование представляется магистральным направлением развития технологии программирования, в «старых» языках программирования, таких, как C++ и java, появляется всё больше языковых и библиотечных «фич» из мира функционального программирования (например, лямбда-выражения и поддержка монадных вычислений). Настоящая работа идёт в русле этого магистрального направления.

Существующие перспективные технологии (прежде всего OpenACC и система DVM) главным образом направлены на автоматическое распараллеливание циклов на GPU, но не отвечают на вопрос: откуда тело цикла возьмёт необходимые данные? Помимо массивов, с которыми ведутся вычисления (и к которым внутри цикла доступ есть) часто используется дополнительная информация, которая берётся из глобальных или локальных переменных, или, при использовании ООП, из переменных-членов класса. Существующие программы, написанные с использованием OpenMP, таких проблем не имеют. При переносе же на GPU (независимо от того, используется OpenACC или DVM) всё меняется. Доступ к глобальным и локальным переменным и переменным-членам класса пропадает. Решением является создание функций, которым вся нужная информация передаётся в виде параметров, и которые будут вызываться из тела параллельного цикла. Именно так работает OpenCL, но там для этого создаётся особый исходный текст (ядро, kernel). Именно так предлагаю работать и я, но мои функции работают и на CPU, и на GPU, и их можно отлаживать стандартными средствами отладки.

Далее я излагаю основные результаты данной работы, которые основываются, на теории функционального программирования и, в частности, на теории категорий. Основные факты из теории категорий и функционального программирования излагаются ниже в приложении.

Мною была написана библиотека функционального программирования funcprog для языка C++ [1], позволяющая писать на языке C++ в стиле, очень близким к стилю программирования на языке Haskell. В языке Haskell очень мощная система вывода типов, благодаря которой запись на этом языке получается очень элегантной и краткой. В языке C++ система вывода типов не такая мощная (хотя разработчики языка стараются и прогресс на лицо), поэтому запись аналогичных алгоритмов на C++ получается более громоздкой (часто приходится явно указывать типы там, где Haskell их выводит сам), но всё равно получается относительно неплохо.

В первоначальной версии этой библиотеки функцией назывался любой объект класса std::function из стандартной библиотеки языка C++. В объект этого класса можно преобразовать любую обычную функцию или любой функциональный объект (в частности, результат лямбда-выражения). Для объектов класса std::function (функций в понимании библиотеки funcprog) была определена операция композиции функций (через оператор &), а также оператор применения функции к одному (первому) аргументу (через оператор <<). Этот оператор применения функции к аргументу позволил в полной мере реализовать каррирование и рассматривать все функции как одноаргументные (как в языке Haskell). То, что в языке Haskell можно было записать как

```
f x y
```

на языке C++ с помощью библиотеки funcprog можно записать так:

```
f << x << y
```

В обоих случаях, если, например, функция f имела 4 параметра, в результате подучится функция с двумя параметрами.

С помощью библиотеки funcprog можно организовать ленивые вычисления, при которых основным вычисляемым объектом будет функция. Преимущество ленивых вычислений перед обычными в том, что в конечном итоге фактически будут вызываться только те функции, результат работы которых будет в дальнейшем использоваться.

В рамках библиотеки были реализованы списки (на основе класса std::vector) и тип Maybe (на основе класса std::optional). В полной мере были реализованы функторы, аппликативы и монады, а также полугруппы, моноиды, альтернативы и некоторые другие структуры. Были реализованы трансформеры монад. Заметим, что функторная операция fmap в библиотеке

реализована с помощью оператора деления, а аппликативная (apply) с помощью операции умножения. Монадная операция, как и в языке Haskell, реализована с помощью оператора >>= (такой оператор в языке C++ есть). Единственное отличие C++ от Haskell состоит в том, что в Haskell эта операция левоассоциативна, а в языке C++ правоассоциативна, поэтому для правильно исполнения приходится явно ставить скобки. Например:

```
(pure(5.) >>= safe_sqrt) >>= safe_log;
```

Из библиотеки языка Haskell достаточно полно был перенесён в библиотеку funcprog парсер Parsec, с помощью которого на C++ был реализован полноценный калькулятор со скобками, старшинством операций, левоассоциативностью арифметических операций. Исходный текст калькулятора на этом парсере – пара десятков строк кода на C++.

Наша группа (Жуков В.Т., Феодоритова О.Б., Новикова Н.Д., Краснов М.М.) разрабатывает программный код MCFL для расчёта течения многокомпонентных газов в рамках программного комплекса NOISEtte совместно с группой Горобца А.В. В какой-то момент появилось желание перенести этот код на графические ускорители. Основной программный код NOISEtte также был перенесён на GPU, причём для этого была использована технология OpenCL. Основным недостатком этой технологии является то, что приходится иметь два экземпляра исходных текстов, для CPU и для GPU. При изменениях в алгоритме программы правки приходится вносить в два программных кода вместо одного. Альтернативой этой технологии является CUDA, позволяющая иметь единый исходный текст для CPU и для GPU. Недостатком технологии CUDA является то, что она работает только на GPU от компании NVIDIA, а OpenCL работает на всех GPU, включая GPU от компании AMD. Но в реальной жизни нам пока встречаются только GPU от NVIDIA (в том числе на кластерах K-10, K-100, K-60 нашего института), поэтому этот недостаток по жизни не очень актуален.

В процессе размышлений о том, как переносить код на GPU возникла идея использовать для этой цели разработанную ранее библиотеку функционального программирования. Библиотеку пришлось весьма сильно переписать, она теперь даже называется по-другому, funcprog2. Основная переделка была связана с тем, что объект класса std::function не будет работать на CUDA, так как он для своей работы использует виртуальные функции, которые на CUDA непереносимы. Вместо этого в рамках библиотеки был написан класс function2, работающий без виртуальных методов. Вместо этого тип, реализующий функцию, передаётся в этот класс как дополнительный параметр шаблона класса function2. Отсюда возникло дополнительное ограничение – в объект класса function2 нельзя преобразовать обычную

функцию. Это может быть только функциональный объект (имеющий тип), в частности, лямбда-выражение.

Что касается численных методов, то в численных методах используются сеточные функции, определённые на элементах сетки (ячейках или узлах), которые по индексу элемента возвращают значения переменных в этом элементе. Было введено понятие сеточного выражения (сеточная функция – его частный случай), основное свойство которого – возвращать значение по индексу. Сеточная функция эти значения хранит в памяти, другие сеточные выражения могут значения вычислять. Например, сумма двух сеточных функций есть сеточное выражение, хранящее ссылки на свои аргументы и вычисляющее своё значение по индексу как сумму значений своих аргументов.

Мы можем рассматривать сеточное выражение как некоторый контейнер (сравните со списком или типом Maybe, описанным ранее). Соответственно, сеточное выражения было сделано функтором, аппликативом и монадой. Были доказаны три теоремы о том, что их реализации для сеточных выражений корректны, то есть удовлетворяют всем соответствующим законам. Покажем, как были определены функторные, аппликативные и монадные операции для сеточных выражений. Пусть gexp – некоторое сеточное выражение, gexp_f – сеточное выражение, возвращающее функцию. Тогда

Функтор:

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

Аппликатив:

(pure val)[i] = val

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

Монада:

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

Для примера докажем, что первый функторный закон для сеточных выражений выполнен. Остальные законы доказываются аналогично. Доказательство будет вестись методом эквациональных рассуждений (англ. equational reasoning). В этом методе правая и левая части равенства, которое нужно доказать, путём эквивалентных преобразований приводятся к одинаковому выражению.

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] = -- по определению функции fmap
  id gexp[i] =          -- по определению функции id
  gexp[i]
```

то есть, действительно,

```
fmap id gexp = gexp
```

Первое равенство доказано.

Для параллельного (на общей памяти) исполнения кода в библиотеке имеется функция par_execute, принимающая два аргумента. Первый – число итераций параллельного цикла. Второй аргумент – функция с одним аргументом (номером элемента сетки). Параллельное исполнение цикла делается по-разному в зависимости от компилятора. Если программа компилируется с помощью обычного компилятора, то распараллеливание делается с помощью OpenMP, а если компилятором для CUDA (nvcc), то средствами CUDA. Таким образом, пользовательский код становится полностью платформо-независимым. Единый исходный текст компилируется и работает и на CPU, и на GPU. Приведём очень простой (но не рабочий) пример:

```
par_execute(N, _([] __DEVICE __HOST (size_t i){
  ...
}));
```

Макросы __DEVICE и __HOST говорят о том, что код (в данном случае тело лямбда-выражения) будет исполняться на вычислительном устройстве и на хосте (CPU). При компиляции для CPU (обычным компилятором) никакого специального вычислительного устройства нет, а есть только обычный процессор, и макросы расширяются в пустые строки. При компиляции для CUDA они расширяются, соответственно, в __device__ и __host__. Чтобы функция могла сделать что-то полезное, ей нужна дополнительная информация. Приведём работающий пример – функцию axpy из библиотеки BLAS. Функцию оформим в виде лямбда-выражения:

```
auto const axpy = _([] __DEVICE __HOST
  (double xi, double a, double &yi, size_t /*i*/){
    yi += a * xi;
  });
```

Мы ввели дополнительный неиспользуемый параметр – индекс цикла, он нам пригодится. Теперь мы можем написать:

```
grid_function<double> x(N), y(N);
... // Заполняем их как-то
```

```
double const a = 1.23;
#pragma omp parallel for
for(size_t i = 0; i < N; ++i)
  axpy(x[i], a, y[i], i);
```

Этот цикл замечательно работает на CPU и даже распараллеливает вычисления между потоками с помощью OpenMP. Но если мы захотим эту программу запустить на GPU, то нам придётся её полностью переписать.

Предлагаемый функциональный подход направлен на решение именно этой проблемы – предлагается способ написания программы, при котором программа будет распараллеливаться (без изменений) и на CPU, и на GPU. Он основан на том, что сеточное выражение (и сеточная функция в частности) является функтором, аппликативом и монадой. Применим функцию axpy к сеточной функции x как к функтору:

```
auto const ge1 = axpy / x;
```

Мы получим сеточное выражение (grid expression) ge1, которое для каждого индекса i вернёт функцию с тремя параметрами (double a, double &yi, size_t i):

```
ge1[i] = axpy << x[i];
```

Теперь применим это сеточное выражение как аппликатив к pure(a), получим:

```
auto const ge2 = ge1 * pure(a) /* = axpy / x * pure(a)*/;
```

Сеточное выражение ge2 для каждого индекса i вернёт функцию с двумя параметрами (double &yi, size_t i):

```
ge2[i] = ge1[i] << a = axpy << x[i] << a;
```

И, наконец, применим это сеточное выражение как аппликатив к сеточной функции y. Получим сеточное выражение, которое для каждого индекса вернёт функцию с одним параметром (индекс цикла), которую уже можно передать в функцию par_execute:

```
auto const ge3 = ge2 * y /* = axpy / x * pure(a) * y */;
ge3[i] = ge2[i] << y[i] = axpy << x[i] << a << y[i];
```

Теперь мы можем написать:

```
par_execute(N, ge3);
```

Или, короче:

```
par_execute(N, axpy / x * pure(a) * y);
```

Можно объединить всё вместе:

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

Этот цикл будет исполняться параллельно и на CPU, и на CUDA, причём пользовательский код получился платформо-независимым. Есть второй, более короткий способ записи этого выражения. Сеточной функции можно присвоить сеточное выражение, возвращающее для каждого индекса функцию с двумя параметрами:

```
y = _([] __DEVICE __HOST
  (double xi, double a, double &yi, size_t /*i*/){
    yi += a * xi;
  }) / x * pure(a);
```

Этот оператор присваивания применяет переданное сеточное выражение к сеточной функции y как к аппликативу и вызывает функцию par_execute, передавая ей размер сеточной функции в качестве числа итераций параллельного цикла и полученное новое сеточное выражение, то есть делает то же самое, что и предыдущий оператор.

Рассмотрим теперь, как работают монады с сеточными выражениями. Как уже говорилось, наша цель – цепочки монадных вычислений. Пусть у нас число a не является константой, а для каждого индекса вычисляется в зависимости от значения x[i]. Напишем сеточное выражение calc_a:

```
auto const calc_a = _([] __DEVICE __HOST (double xi, size_t /*i*/){
  return xi < 0 ? 1.23 : 2.34;
}) / x;
```

Тогда мы можем написать:

```
y = calc_a >>= _([] __DEVICE __HOST
  (double xi, double a, double &yi, size_t /*i*/){
    yi += a * xi;
  }) / x;
```

Сеточное выражение calc_a мы можем использовать многократно. Если имеется несколько констант, то сеточные выражения, их вычисляющие, можно объединить (с помощью оператора ^). Например:

```
auto const calc_b = _([] __DEVICE __HOST (double xi, size_t /*i*/){
  return sin(xi);
}) / x;

y = calc_a ^ calc_b >>= _([] __DEVICE __HOST
  (double xi, double a, double b, double &yi, size_t /*i*/){
    yi += a * b * xi;
  }) / x;
```

Список литературы

1. Краснов М.М. Библиотека функционального программирования для языка C++ // Программирование, 2020 г., № 5, с. 47-59. DOI: 10.31875/S0132347420050040.
2. Краснов М.М., Феодоритова О.Б. Применение библиотеки функционального программирования для распараллеливания вычислений на CUDA. // Программирование, 2024 г., № 1
3. Краснов М.М. Применение монадных вычислений при решении численных задач // Препринты ИПМ им. М.В.Келдыша. 2024. № 2. 24 с. DOI: 10.20948/prepr-2024-2 URL: https://library.keldysh.ru/preprint.asp?id=2024-2
4. 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
5. Krasnov, M.M., Feodoritova, O.B.: Functional programming libraries for graphics accelerators. Supercomputing Frontiers and Innovations 9(4), 28–37 (2022). https://doi.org/10.14529/jsfi220403

Приложение. Введение в функциональное программирование

В основе функционального программирования лежит теория категорий. В частности, современный популярный язык функционального программирования Haskell основывается на этой теории. Многие примеры будут приводится на этом языке, так как он позволяет записывать многие вещи одновременно кратко и понятно.

Остановимся кратко на основных положениях этой теории, нужных для понимания дальнейшего материала.

Категорией называется набор (совокупность) объектов и стрелок (морфизмов) между ними. На морфизмы накладываются следующие два условия:

- Для каждого объекта A категории существует так называемый единичный или тождественный морфизм (из объекта в него самого). Обозначается id_A.
- Если существует морфизм f из объекта A в объект B (обозначается f: $A \to B$) и морфизм g из объекта B в объект C (g: $B \to C$), то должен существовать морфизм h из объекта A в объект C (h: $A \to C$). Такой морфизм называется композицией морфизмов и обычно обозначается как $h = g \circ f$ (читается как g после f).

Из одного объекта категории в другой может быть любое количество морфизмов, в том числе нулевое. Исключение – эндоморфизмы (из объекта в него самого). Для любого объекта должен существовать как минимум один эндоморфизм – единичный. Наличие морфизмов из одного объекта в другой не подразумевает наличие морфизмов в обратную сторону (они могут быть, а могут и не быть).

Примеры категорий:

- **0** – категория без объектов и морфизмов;
- **1** – категория с единственным объектом и единственным единичным морфизмом;
- **2** – категория с двумя объектами и тремя морфизмами (кроме двух обязательных единичных морфизмов ещё морфизм из первого объекта во второй);
- **Set** – категория множеств. Морфизмы – отображения множеств (функции);
- **Grp** – категория групп. Морфизмы – гомоморфизмы групп;
- **Top** – категория топологических пространств. Морфизмы – непрерывные отображения;
- $\mathbf{Vect_K}$ – категория векторных пространств над полем K. Морфизмы – линейные отображения.
- **Pos** – категория частично-упорядоченных множеств. Морфизмы – отношение упорядоченности (обычно обозначается как $\leq$);
- Моноид как категория с единственным объектом. Все морфизмы в этой категории – это эндоморфизмы этого единственного объекта. Каждый морфизм соответствует одному объекту моноида, в частности, единичный морфизм соответствует единичному объекту. Моноидной операции соответствует композиция этих морфизмов.

Категория называется *малой*, если совокупности всех объектов и всех морфизмов категории являются множествами. Если морфизмы между двумя любыми объектами категории образуют множество, то такая категория

называется *локально малой*. **Set** – пример большой категории (не существует множества всех множеств, но категория всех множеств существует). Любая малая категория является локально малой.

Применительно к языкам программирования важную роль играет категория типов языка. В этой категории (для языка Haskell он называется **Hask**) в качестве объектов выступают типы (в частности, типы функций), а в качестве морфизмов – одноместные функции (принимающие аргумент исходного типа и возвращающие результат конечного типа). Обратим внимание, что в языке Haskell любую функцию можно рассматривать как одноместную.

Функтором F из категории **C** в категорию **D** (обозначается F: **C**→**D**) называется отображение, сохраняющее структуру исходной категории внутри конечной. А именно, каждому объекту категории **C** ставится в соответствие объект категории **D** и каждому морфизму f: $A \to B$ морфизм F(f): F(A)→ F(B) так, что

- $F(id_A)=id_{F(A)}$ (сохраняется единичный морфизм);
- $F(g \circ f)=F(g) \circ F(f)$ (сохраняется композиция морфизмов).

Примеры функторов:

- Постоянный функтор, отображающий все объекты категории **C** в некоторый фиксированный объект категории **D** и все морфизмы – в единичный морфизм этого объекта.
- «забывающий» функтор, теряющий некоторые свойства объектов в исходной категории, например, «забывающий» функтор **Grp**→**Set**, «забывающий» групповую структуру.

Функторы позволяют определить категорию локально малых категорий (**Cat**), в которой в качестве морфизмов выступают функторы. Такая категория не является локально малой, поэтому не является элементом самой себя.

Пусть есть три категории **C**, **D** и **E** и два функтора: F:**C**→**D** и G:**D**→**E**. Тогда мы можем рассмотреть композицию этих функторов и получить новый функтор H:**C**→**E**. Будем его обозначать так: $H=G \cdot F$.

Важное значение в теории категорий играют **эндофункторы**, отображающие категорию в себя. Простейший эндофунктор для любой категории – единичный функтор, отображающий каждый объект и каждый морфизм в себя. Такой эндофунктор для категории **C** обозначается $I_{\mathbf{C}}$.

Вернёмся к языкам программирования. Как уже говорилось, мы рассматриваем категорию типов, значит, мы рассматриваем только эндофункторы (так как категория у нас единственная). Существуют типы

данных, параметризованные другим типом данных. Рассмотрим для примера два таких типа. Первый – это список значений какого-то другого типа (этот другой тип и есть параметр списка, например, список строк или список чисел типа double). Так как список – это тоже обычный тип данных, то мы можем рассматривать, например, список списков строк или список списков списков чисел. Второй такой тип данных – это тип Maybe (в стандартной библиотеке языка C++ ему соответствует класс optional). Значение этого типа может содержать единственное значение (*Just a*) или не содержать ничего (*Nothing*). Если *a* – это некоторый тип данных, то мы можем рассматривать типы *Maybe a*, *Maybe (Maybe a)* и т.д. В обоих случаях (списки и тип Maybe) мы можем рассматривать структуру как некоторый контейнер данных. Рассмотрим, как содержимое такого контейнера можно обрабатывать некоторым унифицированным образом.

Мы можем рассмотреть отображение любого типа *a* в тип *Maybe a*, или всей категории типов на подкатегорию типов, задаваемую типом *Maybe*. Это отображение будет функтором (эндофунктором), если мы определим действие этого функтора на морфизмы (функции). Функцию, реализующую это действие, назовём *fmap* (именно так называется соответствующая функция в языке Haskell). Итак, пусть у нас есть функция (морфизм) *f:a→b,* где *a* и *b* – некоторые типы. Нам нужно определить новую функцию *fmap(f): Maybe a→Maybe b*. Полностью тип функции *fmap* выглядит так: *fmap: (a→b) →(Maybe a→Maybe b)*. Вторую пару скобок в языке Haskell можно опустить (так как стрелка правоассоциативна) и записать прототип этой функции на языке Haskell так:

fmap:: (a→b) →Maybe a→Maybe b

то есть можно рассматривать функцию fmap как функцию с двумя аргументами: обычной функцией и функтором и возвращающую функтор. В общем случае, если T – некоторый тип, параметризованный другим типом, то прототип функции fmap, делающий тип T функтором, выглядит так:

fmap:: (a→b) →T a→T b

Для того, чтобы определение функции *fmap* было корректным, оно должно сохранять единичный морфизм и сохранять композицию морфизмов (это следует из определения функтора), то есть удовлетворять двум законам:

1. fmap id = id
2. fmap (g . f) = fmap g . fmap f

Для типа Maybe определение функции fmap выглядит так:

fmap::(a→b)→Maybe a→Maybe b
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)

Несложно доказать, обычно это делается методом эквациональных рассуждений (англ. equational reasoning), что оба функторных закона выполняются. Таким образом, тип данных Maybe является функтором.

Для списков всё даже несколько проще:

fmap::(a→b)→[a]→[b]
fmap f xs = [f x | x ← xs]

Также несложно доказать, что определение функции *fmap* корректно, и значит, список – тоже функтор.

Функтор, с программистской точки зрения, позволяет применить одноместную функцию к содержимому контейнера. А можно ли, например, применить двухместную функцию к содержимому двух контейнеров? Например, сложить два списка. Какой список при этом должен получиться? В общем случае, как применить n-местную функцию к содержимому n функторов? Пусть имеется двухместная функция *f::a→b→c*. Запишем её как одноместную, получим: *f::a→(b→c)*. Применим теперь эту функцию к функтору T. Получим:

fmap:: (a→(b→c))→T a→T (b→c)

В результате мы получим в качестве содержимого контейнера функции (например, список функций). Применить такой контейнер к содержимому другого контейнера, пользуясь только функционалом функтора, нельзя. Для этого предназначены аппликативные функторы, или **аппликативы**. Чтобы тип данных сделать аппликативом, для него нужно определить две функции, *pure* и *apply* (в языке Haskell оператор <*>). Вот их определение:

pure :: a→T a
(<>) :: T (a→b)→T a→T b*

Чтобы определения этих функций были корректными, нужно, чтобы для них выполнялись следующие четыре аппликативных закона:

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 эти операции определены для списков и для Maybe:

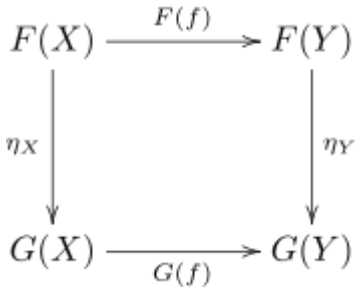
```
instance Applicative [] where
  pure x      = [x]
  fs <*> xs = [f x | f ← fs, x ← xs]

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

Несложно проверить, что и для списков, и для Maybe все аппликативные законы выполнены. Теперь мы можем сложить два списка:

```
pure (+) <*> [1,2] <*> [3,4,5] = [4,5,6,5,6,7]
```

В результате каждое значение из первого списка складывается с каждым значением второго. Только такое определение аппликатива для списков корректно.

Естественное преобразование (англ. natural transformation) предоставляет способ перевести один функтор в другой, сохраняя внутреннюю структуру (например, композиции морфизмов). Поэтому естественное преобразование можно понимать как «морфизм функторов». Пусть F и G – функторы из категории **C** в категорию **D**. Тогда естественное преобразование из F в G сопоставляет каждому объекту X категории **C** морфизм η_X: F(X)→G(X) в категории **D**, называемый компонентой η в X, так, что для любого морфизма f: X→Y диаграмма, изображённая на рисунке ниже, коммутативна (или $\eta_Y \circ F(f)=G(f) \circ \eta_X$).



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

Естественные преобразования функторов из категории **C** в категорию **D** позволяют определить категорию функторов из категории **C** в категорию **D**, в которой естественные преобразования выступают в качестве морфизмов. В частности, если категории **C** и **D** совпадают, то мы можем рассмотреть категорию эндофункторов **End**(**C**).

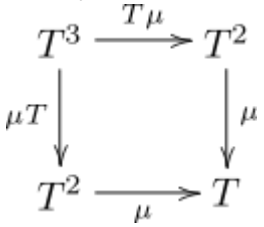
Вернёмся опять к программированию. У нас есть два эндофунктора: Maybe и списки. Попробуем построить естественные преобразования между ними. Из типа Maybe в список естественное преобразование строится легко:

```
maybe2list::Maybe a→[a]
maybe2list Nothing = []
maybe2list (Just x) = [x]
```

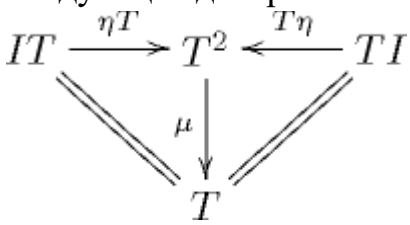
В другую сторону не получается, так как список, содержащий более одного элемента, в тип Maybe корректно (без нарушения функторных законов) преобразовать нельзя.

Следующее важное теоретико-категорное понятие – это понятие **монады**. Пусть имеется некоторая категория **K** и эндофунктор T в этой категории. Кроме того, в категории **K** (как и в любой другой) имеется единичный эндофунктор $I_{\mathbf{K}}$. Так как начальная и конечная категория у эндофункторов совпадают, то мы всегда можем рассматривать композицию эндофункторов, в частности, мы можем рассмотреть функтор $T^2=T\cdot T$ (например, список списков). Монадой в теории категории называется тройка {T,μ,η}, где

- T – эндофунктор в некоторой категории **K**;
- μ – естественное преобразование $\mu: I_{\mathbf{K}} \rightarrow T$;
- η – естественное преобразование $\eta: T^2 \rightarrow T$;
- следующая диаграмма коммутативна (ассоциативность):



- следующая диаграмма коммутативна (двусторонняя единица):



Монаду можно ещё определить как моноид в моноидальной категории эндофункторов **End**(**K**).

В языке Haskell естественному преобразованию μ соответствует функция *pure*, а естественному преобразованию η – функция *join*. Например,

```
join :: Monad m => m (m a) -> m a
join [[1,2],[3,4,5],[6,7,8,9]]=[1,2,3,4,5,6,7,8,9]
```

В языке Haskell монада определяется немного другим, но эквивалентным способом, а именно, через функцию bind (или оператор >>=):

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Смысл этого оператора – построение цепочек монадных вычислений. Поясним на примере. Пусть у нас имеются следующие «безопасные» функции:

```
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 аналогичны стандартным функциям sqrt и log, но они проверяют параметры на область определения соответствующих функций, и возвращают *Nothing*, если аргумент выходит за область определения (аналог исключительной ситуации) и Just, если всё нормально. Как теперь, например, применить функцию safe_log к результату функции safe_sqrt? Функционала функтора и аппликатива для этого недостаточно. Для этой цели как раз и служат монады. Мы можем написать:

```
Just 2 >>= safe_sqrt >>= safe_log
Just 0.3465735902799727

Just 0.4 >>= safe_log
Just (-0.916290731874155)

Just 0.4 >>= safe_log >>= safe_sqrt
Nothing
```

Функции safe_sqrt и safe_log – примеры так называемых «монадных» функций, принимающих обычное (не монадное) значение, а возвращающих значение в монаде (в данном случае Maybe). Другой пример монадной функции – функция pure (монадный эквивалент этой функции называется return). Функция bind (или его синоним оператор >>=) должны удовлетворять трём монадным законам. Чтобы сформулировать эти законы, введём определение монадной композиции (композиции монадных функций), в языке Haskell это оператор >=> (рыбка):

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(>=>) :: f >=> g = \x -> (g x >>= f)
```

Этот оператор принимает в качестве параметров две монадных функции и возвращает в качестве результата также монадную функцию. Монадные законы формулируются в терминах этого оператора:

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

Первые два закона говорят о том, что функция return является левой и правой единицей монадной композиции, а третий закон – что монадная композиция ассоциативна. Вот как определена монадная операция для списков и для Maybe:

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

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

Несложно проверить, что все монадные законы выполняются, а значит, списки и тип Maybe являются монадами.

Эквивалентность теоретико-категорного определения монады через функцию join и программистского через bind легко доказать, показав, что эти функции определяются друг через друга:

```
x >>= f = join (fmap f x)
join n = n >>= id
```

Более того, функцию fmap можно определить через монадную операцию bind:

```
fmap f m = m >>= (return . f)
```

Таким образом, достаточно определить операцию bind, а fmap и join определяются через неё. Тем не менее, обычно fmap тоже определяют явно (например, для большей эффективности).