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

Препринты ИПМ • Препринт № 114 за 2019 г.

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

Краснов М.М.

Применение функционального программирования при решении численных задач

Рекомендуемая форма библиографической ссылки: 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
URL: http://library.keldysh.ru/preprint.asp?id=2019-114

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 функционального программирования при решении численных задач

Москва— 2019

Краснов М. М.

Применение функционального программирования при решении численных задач

Излагаются основы функционального программирования на примере языков Haskell и C++. Объясняется, как функциональное программирование может быть применено при решении численных задач для упрощения записи алгоритмов и даже для ускорения работы программ. Излагаются возможности новых стандартов языка C++, касающиеся как функционального программирования, так и распараллеливания на многоядерных процессорах. Показывается применение функционального программирования к разработанному автором сеточно-операторному подходу к программированию для ещё большего упрощения записи математических формул в текстах программ.

Ключевые слова: Функциональное программирование, Haskell, C++, каррирование, функторы, монады

Mikhail Mikhailovich Krasnov

The use of functional programming in solving numerical problems

The basics of functional programming are presented using the Haskell and C++ languages as an example. It is explained how functional programming can be applied in solving numerical problems to simplify the writing of algorithms and even to speed up the work of programs. The possibilities of new C++ language standards (C++11 and higher) are described, regarding both functional programming and parallelization on multi-core processors. It is shown how functional programming can be applied to the grid-operator approach to programming developed by the author to further simplify the recording of mathematical formulas in program sources.

Key words: Functional programming, Haskell, C++, currying, monads

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

Оглавление

Введение

Функциональное программирование широко используется при решении алгоритмически сложных задач, так как часто позволяет записывать алгоритмы в краткой и в то же время понятной форме, что уменьшает вероятность возникновения ошибок в текстах программ и упрощает их отладку. В качестве примера таких алгоритмически сложных задач можно привести грамматический разбор текстов (формул, конфигурационных файлов или даже текстов программ). В конце 60-х годов прошлого века В. Ф. Турчиным был разработан язык функционального программирования Рефал (см. [1]) специально для написания на нём компиляторов. Текст программы на этом языке по сути представляет собой описание грамматических правил (синтаксиса и семантики) языка, с которого делается компиляция. Написать компилятор с простого языка (например, калькулятор) на Рефале можно буквально в несколько десятков строк.

В задачах математической физики алгоритмы часто тоже бывают весьма непростыми. Запись этих сложных алгоритмов выглядит очень громоздко, часто разобраться в этих формулах бывает тяжело. Особенно это касается отладки программ, найти ошибку в таких записях бывает непросто. Поэтому проблема упрощения записи алгоритмов без потери производительности является актуальной. Одним из способов упрощения записи алгоритмов является функциональный подход. Он подразумевает широкое комбинирование простых вычислений (функций), при этом каждую простую функцию отладить несложно, в результате весь текст получается кратким и понятным. Аргумент против применения такого подхода в задачах математической физики часто состоит в том, что вызов большого числа маленьких функций замедляет работу программ, что критично в численных методах. При использовании современных компиляторов с языка C++ этот аргумент не совсем справедлив, так как простые функции можно сделать инлайновыми, при этом замедления скорости из-за вызовов функций не происходит.

Дальнейшее изложение будет вестись на примере двух языков программирования: Haskell (см. [2]) – одного из ярких современных представителей семейства функциональных языков программирования и C++ – языка, на котором пишется большинство программ, реализующих численные методы. При этом Haskell рассматривается как некоторый образец, идеал, к которому хотелось бы по возможности приблизиться.

Язык C++ в последние годы очень активно развивается. Новые стандарты языка выходят регулярно каждые 3 года, начиная с 2011 года. Новейший стандарт C++20 вышел в этом году, и уже появляются компиляторы, поддерживающие его. Изменения в языке в основном

направлены на повышение эффективности программирования, причём особое внимание уделяется производительности. Часть этих изменений связаны с возможностями программирования в «функциональном стиле», что напрямую связано с рассматриваемой темой. Поэтому мы будем подразумевать, что язык C++, который мы рассматриваем, имеет стандарт не ниже C++11.

Автором написана библиотека функционального программирования для языка C++ (funcprog, см. [3]), позволяющая писать программы на этом языке в стиле, близком к стилю программ на языке Haskell. В частности, с использованием этой библиотеки функциональный стиль программирования становится применим к разработанному автором сеточно-операторному подходу к программированию (см. [4, 5]).

Функциональное программирование в C++

Все языки программирования можно разделить на императивные и декларативные. В императивных языках программирования, к которым относятся большинство языков, на которых реализуются численные методы, программист должен описать с помощью инструкций (команд) на данном языке программирования алгоритм решения задачи. В отличие от этого подхода, в декларативных языках (к которым относятся и функциональные языки) задаётся спецификация решения задачи, то есть описывается, *что* представляют собой проблема и ожидаемый результат. Алгоритм решения поставленной задачи остаётся за компилятором. Яркий пример – язык Рефал, программа на нём описывает некоторую грамматику, а алгоритм компиляции (преобразование входной строки в выходную), сам по себе весьма непростой, является внутренним делом компилятора Рефала, и программист, пишущий на Рефале, не обязан этот алгоритм знать.

Конечно же, математик-программист, реализующий некоторый численный метод, должен знать алгоритм этого метода, никто за него его реализовать не сможет. Речь идёт о другом. Как правило, этот алгоритм сводится к обходу ячеек сетки (сеточной функции) и обработки значений некоторых переменных в каждой ячейке. Эта обработка может также состоять в обходе, например, всех ячеек, соседних с данной ячейкой, и обработке значений переменных в этих ячейках. Обход бывает двух типов: преобразование и свёртка. При преобразовании на основании одной сеточной функции заполняется другая сеточная функция, а при свёртке вычисляется некоторая интегральная характеристика, например, норма, или экстремальное значение (минимум или максимум). Таким образом, типичный алгоритм сводится к двум стандартным операциям обхода (преобразованию и свёртке) и нестандартным элементарным

действиям с одной ячейкой.

Эти рассуждения настолько характерны, что для обхода коллекций (а сеточная функция – типичная коллекция) в стандартной библиотеке многих языков (например, языков Haskell и C++) есть готовые функции. Стандартная коллекция, имеющаяся во многих функциональных языках, – это список.

Начнём с языка Haskell. Вначале обратим внимание на ряд особенностей чисто функциональных языков программирования, к которым относится Haskell. Самое важное – это то, благодаря чему функциональные языки так называются. Функция в этих языках – полноправный участник вычислений. Это означает, что функция может быть передана как параметр в другую функцию, а также может быть возвращена как результат работы другой функции. В языках C и C++ это не так, вернее, не совсем так. В языке C есть понятие «указателя на функцию», который можно передать как параметр в другую функцию, и в этом языки C и C++ похожи на Haskell. Но с возвращением функции в качестве результата есть проблемы. Конечно, указатель на функцию вернуть можно, но только на уже существующую. Породить новую функцию нельзя. А в языка Haskell можно. И это отличие очень важное. Современный язык C++ отчасти эту проблему решает. В нём (начиная ещё со «старого» стандарта 1998 года) есть понятие «функционального объекта». Это объект класса, в котором есть переопределённый «функциональный» оператор (). Во все функции стандартной библиотеки языка C++ (но не C), в которые можно передать указатель на функцию, можно также передать (вместо указателя на функцию) и функциональный объект. Некоторое неудобство с функциональными объектами состояло в том, что нужно было заранее реализовать тот самый класс, и это «размывало» код по исходному тексту. В функциональных языках для решения этой проблемы имеются т.н. «лямбда»-выражения, результатом которых являются функции, и в качестве функции в другую функцию можно передать лямбда-выражение. В стандарте C++11 также появились лямбда-выражения, и это одно из самых существенных нововведений этого стандарта. Лямбда-выражение в C++ возвращает «анонимный» функциональный объект, т.е. объект анонимного класса, имеющего функциональный оператор. Этому объекту можно предоставить доступ к переменным из текущего контекста исполнения (функции или метода класса) по ссылке или по копии. Это полностью решает проблему «размывания» кода, но вернуть такой объект из функции по-прежнему нельзя. Это связано с тем, что, как уже говорилось, это объект анонимного класса, а значит, невозможно указать тип значения, возвращаемого такой функцией.

Другой особенностью чисто функциональных языков программирования является ограничение некоторых императивных

возможностей, например, то, что, хотя в них и есть переменные, но повторное присваивание значения переменной запрещено. Это как если бы в C++ все переменные были бы *const*. Как следствие, коллекцию (список) нельзя изменять «на месте». Вместо этого функции, преобразующие коллекцию, возвращают новую коллекцию как результат своей работы.

Преобразования

Итак, вернёмся к преобразованиям и свёрткам. Для преобразований списков а языке Haskell имеется функция *map*. Она имеет два параметра: пользовательская функция с одним аргументом, осуществляющая преобразование одного значения, и список. Возвращается список того же размера, что и исходный, состоящий из преобразованных (с помощью пользовательской функции) элементов исходного списка. Естественно, в качестве функции можно передать лямбда-выражение. Например:

```
map (\x->x*x) [1,2,3]
```

Возвращается список квадратов чисел [1,4,9].

Язык C++ императивный, и проблем с повторным присваиванием в нём нет. Кроме того, важнейшая задача – скорость. Поэтому все правки делаются «на месте». Аналогичная функция в библиотеке C++ называется transform. Она, как и все подобные функции в стандартной библиотеке языка C++, не работает с коллекциями напрямую. Вместо этого она работает с итераторами. Это даёт возможность работать не только с коллекциями из стандартной библиотеки языка, но и с обычными массивами. В этом случае в качестве итератора рассматривается указатель. Есть две модификации функции transform: с унарной функцией преобразования и с бинарной. Они имеют следующие прототипы:

```
template<class InputIterator, class OutputIterator, class UnaryOper>
OutputIterator transform(InputIterator first1, InputIterator last1,
  OutputIterator result, UnaryOper unary_op);

template<class InputIterator1, class InputIterator2,
  class OutputIterator, class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, OutputIterator result,
  BinaryOperation binary_op);
```

Первая модификация принимает итераторы начала и конца входной коллекции, итератор начала выходной коллекции и унарную операцию (указатель на функцию или функциональный объект). Вторая модификация работает с двумя входными коллекциями, соответственно, она принимает итераторы начала и конца первой входной коллекции, итератор начала второй входной коллекции, итератор начала выходной коллекции и бинарную операцию (указатель на функцию

или функциональный объект). Ничто не мешает тому, чтобы выходная коллекция совпадала со входной. Обе модификации возвращают итератор конца выходной коллекции. Вот как выглядит пример с квадратами чисел на C++:

```
  const int inp[3] = { 1, 2, 3 };
  int outp[3];
  transform(inp, inp + 3, outp, [](int x){ return x * x; });
```

А вот пример суммирования двух массивов:

```
  const int inp1[3] = { 1, 2, 3 }, inp2[3] = { 4, 5, 6 };
  int outp[3];
  transform(inp1, inp1 + 3, inp2, outp,
    [](int x, int y){ return x + y; });
```

Оба примера в качестве операции передают лямбда-выражение.

Каждый новый стандарт языка C++ добавлял новую важную функциональность в язык. Как уже говорилось, C++11 добавил лямбда-выражения и ещё многое другое, в стандарте C++20 появились долгожданные «концепции» (concepts), улучшающие диагностику ошибок при компиляции. А стандарт C++17 добавил для алгоритмов из стандартной библиотеки распараллеливание на многоядерных процессорах (какими являются практически все современные процессоры). Теперь (если компилятор поддерживает этот стандарт) вычисление квадратов можно записать так:

```
#include <execution>
...
  const int inp[3] = { 1, 2, 3 };
  int outp[3];
  transform(execution::par, inp, inp + 3, outp,
    [](int x){ return x * x; });
```

Конечно, преобразовать коллекцию можно и «по старинке» простым циклом. Использование «функционального» стиля имеет следующие преимущества:

1. более краткая запись;
2. более понятный функционал;
3. возможность распараллеливания.

Естественно, замедления по сравнению с простым циклом практически нет (проверено). И ещё. Поддержка OpenMP для такого распараллеливания не требуется и работает чуть быстрее (порядка 5%), чем простой цикл с OpenMP.

Свёртка

Свёртка (fold), как уже говорилось, вычисляет некоторую интегральную характеристику коллекции. Типичная функция свёртки принимает в качестве параметра бинарную свёрточную функцию, начальное значение и коллекцию. Если коллекция пустая, то возвращается начальное значение. Свёртки бывают левые и правые. Левая свёртка, как следует из названия, сворачивает коллекцию слева направо, при этом начальное значение подставляется как самый левый элемент. Правая свёртка, соответственно, сворачивает справа налево, и начальное значение подставляется как самый правый элемент. Поясним сказанное. Пусть исходный список l состоит из трёх элементов l1, l2 и l3, а начальное значение равно z. Тогда левая и правая свёртки дадут следующие результаты:

```
foldl(f, z, l) == f(f(f(z, l1), l2), l3);
foldr(f, z, l) == f(l1, f(l2, f(l3, z)));
```

Результаты левой и правой свёрток могут быть разными в двух случаях: если операция некоммутативна (например, умножение матриц), или если операция неассоциативна (например, сложение чисел с плавающей точкой). Какую свёртку применять в том или ином случае, решает программист. Пример применения свёртки на языке Haskell:

```
foldl (+) 0 [1,2,3]
```

Результат – 6. В стандартной библиотеке языка C++ есть функции (левой) свёртки двух типов: функции min_element и max_element возвращают элемент коллекции (итератор элемента) с указанным свойством, а функция accumulate возвращает сумму элементов. Все функции принимают итераторы начала и конца входной коллекции. Функции min_element и max_element могут также принимать предикат (бинарную операцию, возвращающую булевское значение), а функция accumulate дополнительно принимает начальное значение и, возможно, бинарную операцию свёртки (по умолчанию это операция сложения). Функции min_element и max_element возвращают итератор соответствующего элемента коллекции или итератор конца коллекции, если коллекция пустая (итераторы начала и конца коллекции совпадают). Это даёт возможность узнать не только значение экстремального элемента, но и его индекс. Для вычисления индекса элемента по его итератору можно вызвать функцию distance, передав ей итератор начала коллекции и итератор элемента. Важное замечание по поводу функции accumulate: тип возвращаемого значения (результата свёртки) совпадает с типом начального значения. Поясним важность этого замечания на следующем примере. Пусть мы хотим просуммировать элементы массива типа double:

```
const double a[3] = { 1.2, 3.4, 5.6 };
const double result = accumulate(a, a + 3, 0);
```

Результат будет равен 9, а не 10.2. Это связано с тем, что в качестве начального значения передано *целое* число 0, значит, после каждого сложения дробная часть результата будет отбрасываться. Чтобы такой ошибки не было, нужно явно передавать вещественный ноль:

```
const double result = accumulate(a, a + 3, 0.);
```

Так же как и для преобразования (transform), в стандартной библиотеке языка C++ имеется функция для свёртки двух коллекций. Она называется inner_product (внутреннее, или скалярное произведение). Полная версия принимает итераторы начала и конца первой коллекции, итератор начала второй коллекции, начальное значение и две бинарных операции. В сокращённой версии последние два параметра (бинарные функции) отсутствуют, в качестве первой подразумевается сложение, а в качестве второй – умножение. Тип возвращаемого значения, как и для функции accumulate, совпадает с типом начального значения. Пример:

```
const int a[3] = { 1, 2, 3 }, b[3] = { 4, 5, 6 };
const int result = inner_product(a, a + 3, b, 0);
```

Результат – 32.

Другие численные алгоритмы

Функции свёртки accumlate и inner_product содержатся в заголовочном файле <numeric>, в котором определены алгоритмы над числовыми коллекциями. Наряду с этими функциями имеются ещё три. Кратко опишем их.

partial_sum. Вычисляет частичные суммы элементов входной коллекции. Принимает итераторы начала и конца входной коллекции и итератор начала выходной коллекции. На выходе каждый элемент выходной коллекции равен сумме соответствующего и всех предыдущих элементов входной коллекции. Возвращает итератор конца выходной коллекции. Есть модификация этой функции, принимающая дополнительный параметр – бинарную операцию (вместо сложения).

adjacent_difference. Параметры те же, что и у функции partial_sum. На выходе каждый элемент выходной коллекции равен разности значений соответствующего и предыдущего элементов входной коллекции, кроме первого элемента, который просто равен значению первого элемента входной коллекции. Также есть модификация, принимающая дополнительный параметр – бинарную операцию (вместо вычитания).

iota. Принимает итераторы начала и конца коллекции и начальное значение. Заполняет коллекцию последовательными значениями,

начиная с начального. Следующее значение получается из предыдущего с помощью префиксного оператора ++. Удобный генератор для начального заполнения коллекции.

Другие алгоритмы

Множество других (нечисленных) алгоритмов определены в заголовочном файле <algorithm>. С полным списком функций можно ознакомится в документации по языку C++, например, на сетевом ресурсе [13]. Опишем их вкратце. Все функции работают с коллекциями (последовательностями) элементов, принимая в качестве параметров итераторы начала и конца последовательности. Их условно можно разбить на несколько групп:

- немодифицирующие операции;
- модифицирующие операции;
- разбиение;
- сортировка;
- бинарный поиск;
- слияние;
- куча (очередь с приоритетами);
- мин/макс;
- другие.

Кратко опишем функции в каждой из этих групп. Вначале введём несколько терминов. Будем называть диапазоном (range) набор элементов, задаваемый итераторами начала и конца коллекции, а последовательностью (sequence) – набор элементов, задаваемый итератором начала коллекции и длиной. Если функция принимает два диапазона одинаковой длины, то для второго диапазона передаётся только итератор его начала.

Немодифицирующие операции

all_of, *any_of*, *none_of*. Каждая из этих функций принимает диапазон (итераторы начала и конца коллекции) и унарный предикат. Функции возвращают булевский результат.

Функция *for_each* вызывает указанную функцию для каждого элемента диапазона. Эту Функцию можно использовать и для модификации

элементов, если передаваемая функция будет принимать значение по ссылке.

Функции *find*, *find_if* и *find_if_not* осуществляют линейный поиск в указанном диапазоне.

Функция *find_end* принимает два диапазона. В первом из них ищется последняя подколлекция (подстрока), совпадающая со значениями из второго диапазона.

Функция *find_first_of* также принимает два диапазона. Возвращает итератор первого элемента первого диапазона, равного одному из элементов второго диапазона, или итератор конца первого диапазона, если поиск оказался неуспешным.

Функция *adjacent_find* ищет указанном диапазоне одинаковые соседние элементы и возвращает итератор первого из них, или итератор конца диапазона, если поиск оказался неуспешным.

Функции *count* и *count_if* подсчитывают число элементов диапазона, равных указанному значению (*count*), или тех, для которых унарный предикат вернул значение «истина».

Функция *mismatch* принимает два диапазона одинаковой длины и возвращает пару итераторов первых несовпадающих элементов.

Функция *equal* проверяет два диапазона одинаковой длины на равенство.

Функция *is_permutation* проверяет, является ли один диапазон перестановкой другого, т.е. то, что для каждого элемента из каждого диапазона в другом найдётся равный ему.

Функция *search* ищет в первом диапазоне первое вхождение второго.

Функция *search_n* ищет в указанном диапазоне цепочку из *n* одинаковых элементов, равных указанному значению, или тех, для которых унарный предикат вернул значение «истина».

Модифицирующие операции

copy копирует диапазон значений.

copy_n копирует последовательность значений заданной длины.

copy_if копирует часть значений из диапазона (для которых унарный предикат вернул значение «истина»).

copy_backward копирует диапазон элементов в обратном порядке. Принимает итератор конца (а не начала) второй коллекции.

move перемещает значения в указанном диапазоне. Для простых типов то же самое, что и копирование (*copy*), имеет смысл только для сложных типов (например, коллекций или строк).

move_backward перемещает диапазон элементов в обратном порядке.

swap меняет местами значения двух переменных.

swap_ranges меняет местами значения двух диапазонов.

iter_swap меняет местами значения двух объектов, на которые

указывают итераторы.

transform преобразует диапазон с помощью унарной функции или пару диапазонов с помощью бинарной функции.

replace, *replace_if* заменяют в указанном диапазоне значение элементов, равных указанному значению, или тех, для которых предикат вернул значение «истина», на заданное значение.

replace_copy, *replace_copy_if* копируют один диапазон в другой с заменой значений.

fill заполняет диапазон указанным значением.

fill_n заполняет последовательность указанным значением.

generate, *generate_n* заполняют диапазон или последовательность значениями, сгенерированными функцией.

remove, *remove_if* удаляют из диапазона элементы, равные указанному значению, или тех, для значений которых указанный предикат вернул значение «истина». Сдвигает остальные элементы диапазона на место удалённых. Возвращает итератор, следующий за последним неудалённым элементом.

remove_copy, *remove_copy_if* копируют указанный диапазон с удалением элементов, равных указанному значению или тех, для значений которых указанный предикат вернул значение «истина».

unique оставляет в указанном диапазоне в каждой группе идущих подряд одинаковых элементов только первый. Сдвигает остальные элементы диапазона на место удалённых. Возвращает итератор, следующий за последним неудалённым элементом.

unique_copy аналогична функции *unique*, но копирует результат в выходную коллекцию. Возвращает итератор конца выходной коллекции.

reverse меняет порядок элементов в указанном диапазоне на обратный.

reverse_copy копирует указанный диапазон в обратном порядке.

rotate принимает итераторы начала, середины и конца коллекции. Переставляет циклически элементы так, чтобы элемент, на который указывает итератор середины, стал первым.

rotate_copy похожа на предыдущую, но элементы копируются в выходную коллекцию. Возвращает итератор конца выходной коллекции.

random_shuffle случайным образом переставляет элементы в указанном диапазоне.

shuffle случайным образом переставляет элементы в указанном диапазоне. Помимо диапазона принимает равномерный генератор случайных чисел (Uniform Random Number Generator, URNG), такой, как один из стандартных генераторов, определённых в заголовочном файле <random>.

Разбиение

is_partitioned проверяет, что указанный диапазон разбит, т.е. что все элементы, для которых указанный унарный предикат возвращает значение «истина», предшествуют элементам, для которых возвращается значение «ложь».

partition переставляет элементы в указанном диапазоне так, чтобы все элементы, для которых указанный унарный предикат возвращает значение «истина», предшествовали элементам, для которых возвращается значение «ложь». Возвращает итератор первого элемента второй группы или итератор конца коллекции, если коллекция пуста.

stable_partition похожа на предыдущую функцию, но сохраняет относительное расположение элементов внутри каждой группы. Обычно реализация этой функции использует внутренний временный буфер.

partition_copy копирует элементы входного диапазона в один из двух выходных диапазонов в зависимости от значения предиката для элемента.

partition_point возвращает итератор «точки разбиения», т.е. первого элемента, для которого предикат возвращает значение «ложь».

Сортировка

sort сортирует элементы указанного диапазона по возрастанию с помощью оператора < или с помощью бинарного предиката.

stable_sort аналогична функции sort, но сохраняет относительное расположение равных элементов.

partial_sort осуществляет частичную сортировку элементов в указанном диапазона с помощью оператора < или с помощью бинарного предиката. Принимает итераторы начала (first), середины (middle) и конца (last) коллекции, дополнительно может принимать бинарный предикат. Переставляет элементы в диапазоне [first,last) так, что перед итератором middle оказываются наименьшие элементы всего диапазона, упорядоченные по возрастанию (неубыванию), а остальные элементы остаются в случайном порядке.

is_sorted проверяет, что указанный диапазон отсортирован.

is_sorted_until возвращает итератор первого элемента указанного диапазона, нарушающего сортировку, или итератор конца диапазона.

nth_element переставляет элементы в указанном диапазоне так, что в указанной позиции оказывается элемент, который бы там оказался в отсортированной последовательности. До этой позиции будут только элементы, не превышающие этот элемент, а после этой позиции – не меньшие.

Двоичный поиск

Все функции из этого раздела работают с отсортированными последовательностями или, по крайней мере, разделёнными (partitioned) относительно указанного значения. Все функции принимают итераторы

начала и конца диапазона, значение, которое ищется, и, возможно, двоичный предикат сравнения элементов. Если двоичный предикат не указан, то сравнение осуществляется с помощью оператора <.

lower_bound возвращает итератор первого элемента, значение которого больше или равно указанного значения.

upper_bound аналогична функции *lower_bound*, но возвращает итератор первого элемента, значение которого строго больше указанного.

equal_range объединяет результаты двух предыдущих функций. Возвращает пару итераторов нижней и верхней границы.

binary_search проверяет, имеется ли в указанном диапазоне элемент, значение которого равно указанному.

Слияние

Все функции данного раздела работают с отсортированными последовательностями. Могут дополнительно принимать бинарный предикат сравнения элементов.

merge сливает два отсортированных диапазона так, что результат также получается отсортированным.

inplace_merge сливает два диапазона «на месте».

includes проверяет, что первый диапазон содержит все элементы второго диапазона.

set_union объединяет два диапазона, рассматривая их как множества.

set_intersection строит пересечение двух множеств.

set_difference строит разность двух множеств.

set_symmetric_difference строит симметричную разность двух множеств.

Куча (очередь с приоритетами)

Куча (очередь с приоритетами) – это такой способ организации элементов, который позволяет быстро удалять элемент с наибольшим значением в любой момент (с помощью функции *pop_heap*), в том числе повторно, позволяя в то же время быстро добавлять новые элементы (с помощью функции *push_heap*). Наибольший элемент всегда первый в диапазоне. Порядок остальных элементов зависит от реализации, но он согласован между всеми функциями из данного заголовочного файла. Все функции могут дополнительно принимать бинарный предикат сравнения элементов. Если двоичный предикат не указан, то сравнение осуществляется с помощью оператора <.

make_heap переупорядочивает элементы указанного диапазона так, чтобы они образовали кучу.

push_heap добавляет новый элемент в кучу.

pop_heap удаляет максимальный элемент из кучи (уменьшает её длину на единицу).

sort_heap сортирует кучу, чтобы элементы заданного диапазона шли в возрастающем порядке.

test_heap проверяет, что указанный диапазон образует кучу.

is_heap_until ищет первый элемент в диапазоне, находящийся за пределами кучи.

Мин/макс

min, *max* возвращают соответственно минимальный и максимальный элемент из двух заданных.

minmax возвращает пару из минимального и максимального элементов.

min_element, *max_element* возвращают итераторы соответственно минимального и максимального элементов.

minmax_element возвращает пару из итераторов минимального и максимального элементов.

Другие

lexicographical_compare лексикографически сравнивает два диапазона (возможно, разной длины). Возвращает значение «истина», если первый диапазон лексикографически меньше второго, и значение «ложь», если второй диапазон лексикографически меньше первого или если диапазоны равны. Если один из диапазонов короче другого и поддиапазон более длинного длиной, равной длине короткого, равен короткому, то короткий считается лексикографически меньше.

next_permutation преобразует указанный диапазон в следующую (лексикографически большую) перестановку. Всего имеется N! возможных перестановок, где N – длина диапазона. Лексикографически минимальной перестановкой считается та, в которой все элементы упорядочены по возрастанию, а лексикографически максимальной – по убыванию. Функция возвращает значение «истина», если найти следующую лексикографически большую перестановку удалось, т.е. текущая перестановка не является максимальной. В противном случае элементы преобразуются в лексикографически минимальную перестановку (по возрастанию) и возвращается значение «ложь».

prev_permutation преобразует указанный диапазон в предыдущую (лексикографически меньшую) перестановку. Возвращается значение «истина», если текущая перестановка не лексикографически минимальна, иначе диапазон преобразуется в лексикографически максимальную перестановку (по убыванию) и возвращается значение «ложь».

Функциональное программирование в Haskell

В языке Haskell, как и в любом другом языке функционального программирования, функция является полноценным участником вычислений. Функцию можно передать как параметр в другую функцию или вернуть как результат работы функции. Другое общее свойство

всех языков функционального программирования – наличие лямбда-выражений, результатом которых является безымянная функция, которую можно передать как параметр в другую функцию. Как уже говорилось, лямбда выражения появились и в современном языке C++. Но кроме этих общих для всех функциональных языков свойств, Haskell обладает рядом особенностей, делающих его ещё более удобным и мощным. К наиболее интересным свойствам языка можно отнести следующие:

- каррирование;
- ленивые вычисления (следствие каррирования);
- бесконечные списки (следствие ленивых вычислений);
- η (эта) редукция (следствие каррирования);
- композиция функций и применение функции (function application);
- поддержка на уровне базовой библиотеки функторов, аппликативов и монад;
- встроенный в стандартную библиотеку парсер (Parsec).

Остановимся на этих и некоторых других свойствах языка Haskell подробнее.

Каррирование

В большинстве языков программирования при вызове функции нужно указывать все её параметры. В языке C++ у части последних параметров могут быть заданы значения по умолчанию. Эти параметры указывать не обязательно, вместо них будут переданы их значения по умолчанию. В языке Haskell ситуация совершенно другая. Если указать не все параметры, то будет порождена новая функция с меньшим числом параметров, равным числу недостающих параметров. Следующие прототипы функций эквивалентны:

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

Т.е. функцию *f* можно рассматривать как функцию с двумя параметрами типов *a* и *b* и возвращающую результат типа *c*, как функцию с одним параметром типа *a*, возвращающую функцию типа *(b->c)* или как функцию без параметров, возвращающую функцию типа *(a->b->c)*. Именно последняя форма даёт возможность передавать функцию как параметр в другую функцию. Вторая форма даёт возможность передать

функцию с двумя аргументами туда, где требуется функция с одним аргументом, указав первый аргумент. Например:

```
map (5 -) [1,2,3]
map ((-) 5) [1,2,3]
```

Результатом вычислений в обоих случаях будет список [4,3,2]. В качестве первого параметра функции map (где требуется функция с одним аргументом) передается функция с двумя параметрами (-), которой передан только первый параметр, при этом она превратилась в функцию с одним параметром.

Каррирование идёт «до конца». Это значит, что если даже указаны все параметры функции, то вызова всё равно не происходит, вместо этого создаётся функция без параметров (скалярная функция), которая будет вызвана только при фактической необходимости (например, при выводе на экран). Это и есть «ленивые» вычисления. Фактически в нашем примере функция *map* вернёт не список, а функцию без параметров, возвращающую наш список, который будет вычислен при его выводе на экран. Бесконечные списки (типа [1..]) также не вычисляются полностью (для бесконечных списков это невозможно). Будут вычислены только те первые элементы списка, которые требуются (например, для вывода на экран).

η-редукция

Рассмотрим две функции со следующими прототипами:

```
f::Num a=>a->b->c
g::Num a=>a->a->b->c
```

Т.е. f – некоторая функция с двумя параметрами, а g – функция с тремя параметрами. Пусть функция g определена следующим образом:

```
g x y z=f (x+y) z
```

Тогда из соображений каррирования понятно, что следующее определение функции g эквивалентно этому определению:

```
g x y=f (x+y)
```

Т.е. последний параметр функции можно опустить (редуцировать), если в теле функции он стоит последним параметром. Такая редукция называется η-редукцией. Если в новом определении последний параметр функции опять окажется последним параметром в её теле, то и его можно редуцировать, и т.д., сколько возможно. В частности, если мы хотим завести функцию h, полностью эквивалентную некоторой функции f, то достаточно просто написать: h=f, т.е. редуцировать все параметры.

Композиция функций

Пусть имеются две функции f и g, каждая с одним параметром, и пусть мы хотим преобразовать список с помощью функции *map* по правилу g (f x). Тогда мы можем написать это в виде лямбда-выражения:

```
map (\x->g (f x)) [1,2,3]
```

Haskell позволяет написать это гораздо короче и нагляднее:

```
map (g.f) [1,2,3]
```

Здесь . (точка) – это композиция функций. Композиция функций ассоциативна, поэтому композицию из трёх функций можно записать просто как h.g.f. Оператор композиции функций – это бинарная ассоциативная функция, и её можно передать как параметр в префиксной форме (.) (как и все операторы в языке Haskell).

Применение функции

В языке Haskell параметры передаются в функцию просто через пробел. Но это не всегда бывает удобно. Рассмотрим такой пример. Пусть мы хотим вызвать некоторую функцию f от агрумента x+y. Если написать f x+y, то это будет неправильно, т.к., согласно старшинству операций, это исполнится как (f x)+y. Значит, мы должны будем написать так: f (x+y). Но такой стиль написания программ может породить большое количество скобок, как в языках Lisp или Рефал. Вспомним одну из шуточных расшифровок аббревиатуры LISP – Lost In Stupid Parenthesis (потерявшийся в дурацких скобках). Чтобы решить эту проблему, был введён оператор «применения» функции к аргументу $ (доллар). Он определён очень просто:

```
f $ x = f x
```

Этот оператор имеет низкий приоритет, поэтому запись f $ x+y будет правильной. Ещё пример. Запись f x (g (y+z)) можно заменить на f x $ g $ y+z. Если аргумент, к которому применяется функция, не последний, то перед следующим аргументом нужно всю конструкцию взять в скобки: (f $ arg1) arg2.

Функторы, аппликативы, монады

Функторы (Functor) и монады (Monad) в функциональном программировании обычно рассматриваются либо как контейнеры, либо как контексты вычислений. Примеры функторов – списки и класс Maybe. Выход из такого контейнера или контекста, как правило, невозможен, т.е. результатом любых вычислений со значениями внутри функтора или

монады является некоторое значение в том же функторе или монаде (исключение – свёртка). Функтор или монада могут содержать любое количество значений, в частности, могут их не содержать вообще (пример – список). Поэтому извлечь явно значение из функтора или монады невозможно.

Функторы

Если стоит задача применить некоторую функцию (с одним аргументом) к значению, хранящемуся в контейнере, то сделать это может только сам контейнер, только он знает, сколько в нём хранится значений и как их извлечь. Для этого предназначен специальный класс *Functor*. В нем есть специальная функция *fmap*, осуществляющая такое применение. Этой же цели служит оператор (<$>), синоним функции *fmap*. Каждый контейнер может реализовать свою специализацию класса *Functor* со своей реализацией функции *fmap*. Вот определение класса *Functor*:

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

Прототип функции *fmap* можно записать в другом эквивалентном виде:

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

Таким образом, функцию *fmap* можно рассматривать как функцию с одним параметром, принимающим функцию, принимающую и возвращающую обычные значения и преобразующую её в функцию, принимающую и возвращающую функторы.

Любая реализация функтора должна удовлетворять двум функторным законам:

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

Вот так выглядит специализация класса *Functor* для списков и для класса *Maybe*:

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs

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

Результатом применения функции к списку является список из преобразованных значений (возможно, другого типа). Аналогично, результатом применения функции к объекту класса *Maybe* будет объект класса *Maybe*, но, возможно, со значением другого типа.

Аппликативные функторы

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 теперь посмотрим на прототип и реализацию функции *liftA2*, то мы увидим, что она передаёт функцию с двумя параметрами в оператор (<$>), который принимает функцию с одним параметром. Но противоречия тут нет, так как функцию с прототипом a->b->c мы можем записать так: a->(b->c), т.е как функцию с одним параметром, возвращающую функцию. Тогда оператор (<$>) нам как раз и вернёт функцию (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)
```

т.е. 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
```

С помощью функции *pure* можно написать функцию, «поднимающую» в аппликатив функцию с одним параметром:

```
  liftA::Applicative f => (a -> b) -> f a -> f b
  liftA f x = pure f <*> x
```

Любая реализация аппликатива должна удовлетворять аппликативным законам:

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

Монады

Монады можно рассматривать как дальнейшее продолжение аппликатива, они предназначены для построения цепочек монадных вычислений. Каждая монада имеет две основных функции: *mreturn* (в языке Haskell *return*) и *mbind* (в языке Haskell оператор >>=). Функция *mreturn* аналогична функции *pure* для аппликативов (фактически для большинства монад *mreturn* определяется как *pure*), а операция *mbind* (>>=) имеет следующее определение:

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

Она принимает в качестве параметров монаду и функцию, принимающую обычное (не монадное) значение и возвращающую монадное значение (возможно, другого типа, но в той же монаде). Если исходная монада пустая, то возвращается также пустая монада, иначе значение извлекается из монады, к нему применяется функция и возвращается её результат.

Для каждой монады две монадные функции должны удовлетворять трём т.н. «монадным законам». Для того чтобы их сформулировать, введём операцию монадной композиции функций (*mcompose* или оператор (>=>) в языке Haskell). Она определяется следующим образом:

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

Оба операнда – т.н. «монадные функции», принимающие обычные значения и возвращающие монадные. Результатом монадной композиции (монадных) функций также является монадная функция. Следовательно, операцию монадной композиции можно рассматривать как групповую операцию в пространстве монадных функций. В терминах этой групповой операции монадные законы формулируются так:

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

Другими словами, монадные функции *mreturn* и *mbind* должны быть определены так, чтобы, во-первых, функция *mreturn* являлась единичным элементом (левым и правым) монадной композиции (первые два закона),

и, во-вторых, монадная композиция должна быть ассоциативной (третий закон).

Ещё одна очень широко применяемая при работе с монадами конструкция – т.н. *do*-нотация. Она позволяет существенно сократить запись и сделать её более наглядной. Смысл её в следующем. Пусть *m* – некоторое монадное значение, а *action* – некоторое вычисление, которое нужно провести со значением, хранящимся в *m* (и которое, естественно, возвращает монадное значение в той же монаде). Это можно было бы записать на языке Haskell следующим образом:

```
m >>= \x -> action
```

С помощью *do*-нотации то же самое записывается так:

```
do x <- m; action
```

Если нужно извлечь значения из двух монад, то разница ещё существеннее. Вместо

```
m1 >>= \x -> m2 >>= \y -> action
```

можно написать:

```
do x <- m1; y <- m2; action
```

Полугруппы и моноиды

Полугруппа (Semigroup) в общей алгебре – множество с заданной на нём ассоциативной бинарной операцией. В языке Haskell эта операция задаётся оператором (<>). Моноид – это полугруппа с единичным элементом. Единичный элемент моноида возвращается функцией *mempty*. Моноидами в языке Haskell являются списки и, в частности, строки.

Аналогично функторам, монадам, полугруппам и моноидам реализованы классы *Alternative* (моноиды над аппликативами) с операциями *empty* и (<|>) и *MonadPlus* – монады с поддержкой выбора и неудачи (failure) с операциями *mempty* и *mplus*.

Класс Foldable

Класс *Foldable* является обобщением свёрточного функционала, первоначально определённого для списков, на произвольные типы данных, поддерживающих свёртку. Основные свёрточные операции – это правая и левая свёртки (*foldr* и *foldl*). Эти функции принимают в качестве параметров функцию, по которой делается свёртка, начальное значение и свёртываемый объект (реализующий класс *Foldable*). Если тип данных, хранящихся в свёртываемом объекте, является моноидом, то для такого объекта в классе *Foldable* определена также функция *fold*,

принимающая единственный параметр – свёртываемый объект. Этот объект сворачивается в моноид, например, список списков свернётся в один список, включающий все объекты из всех списков. В качестве начального значения при такой свёртке берётся единичный элемент моноида (возвращаемый функцией *mempty*), а в качестве свёрточной функции – функция *mappend*. Так как моноидная операция ассоциативна, то правая и левая свёртки дают одинаковый результат.

Ещё одна важная функция класса *Foldable* – *foldMap*. Её прототип на языке Haskell выглядит так:

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

Эта функция принимает в качестве параметров функцию (в свою очередь принимающую в качестве параметра значение того типа, который хранится в свёртываемом объекте и возвращающую моноид) и свёртываемый объект и возвращает моноид. Как и функция *fold*, она делает свёртку объекта в моноид. Значения, хранящиеся в свёртываемом объекте (произвольного типа), предварительно пропускаются через функцию, принимаемую в качестве первого параметра (и, таким образом, преобразуются в моноид). Функцию *fold* можно естественным образом определить через функцию *foldMap*:

```
fold = foldMap id
```

Здесь *id* – функция, возвращающая свой аргумент: (id x = x).

В последнем определении функции *fold* использовалась η-редукция, описанная выше. Полное определение выглядит так: fold x = foldMap id x.

Функция *foldMap* похожа на функцию *fmap* (неслучайно сходство их названий). Это видно из сравнения определений этих функций для списков:

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs

instance Foldable [] where
  foldMap _ []     = mempty
  foldMap f (x:xs) = f x <> foldMap f xs
```

Обе функции проходят по списку, при этом функция *fmap* формирует результат, перестраивая список, а *foldMap* – свёртывая значения в моноид с помощью оператора <> (mappend).

Класс Traversable

Класс *Traversable* играет для аппликативов ту же роль, что класс *Foldable* для моноидов. Основная функция класса *Traversable* – *traverse*. Вот её прототип:

```
traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Функция, передаваемая первым параметром, возвращает значение в аппликативе, значит, и функция *traverse* должна вернуть результат в том же аппликативе (выход за пределы аппликатива запрещён). Определение этой функции для списков выглядит так:

```
instance Traversable [] where
  traverse _ []     = pure []
  traverse f (x:xs) = (:) <$> f x <*> traverse f xs
```

Сравнение этого определения с определением функции *foldMap* класса *Foldable* показывает большое внешнее сходство. Функция *traverse* проходит по элементам списка (в общем случае – по элементам объекта класса *Traversable*), преобразует каждый элемент в аппликатив с помощью функции – первого параметра и формирует новый аппликатив, внутри которого хранится исходная структура объекта, но с новыми значениями. Другими словами, функция *traverse* создаёт новый внешний (аппликативный) уровень для исходного контекста.

Если значения, хранящиеся внутри объекта класса *Traversable*, являются аппликативами, возможен простой обход данного объекта. Для этого предназначена другая функция класса *Traversable* – *sequenceA*. Вот её прототип и определение на языка Haskell:

```
sequenceA :: Applicative f => t (f a) -> f (t a)
sequenceA = traverse id
```

В классе *Traversable* есть ещё две функции – *mapM* и *sequence*. Они аналогичны функциям *traverse* и *sequenceA* (фактически их вызывают), но работают с монадами.

Монады Reader, Writer и State

Монада Reader

Монада *Reader* предназначена для предоставления общей информации (например, общих параметров системы) одновременно нескольким функциям. Альтернативный подход – передавать эту информацию во все функции в качестве параметров, что неудобно. Монада *Reader* позволяет решить эту проблему, предоставляя общий для всех контекст, в котором любая функция может запросить информацию. Тип хранимого значения передаётся как параметр класса Reader. Основные функции класса *Reader* – *ask*, возвращающая значение, *local*, временно меняющая контекст, и *reader*, создающая объект *Reader*. Вот небольшой демонстрационный пример использования этой монады:

```
  hello = reader $ \name -> ("Hello," ++ name ++ "!")
```

```
bye = reader $ \name -> ("Bye," ++ name ++ "!")
convo = reader (const (++)) <*> hello <*> bye
main = print . runReader convo $ "Fred"
```

Функция *main* печатает: "Hello,Fred!Bye,Fred!". Обе функции *hello* и *bye* прочитали переданное значение ("Fred").

Монада Writer

Монада *Writer* предназначена для вывода информации в некоторое общее для всех функций место (например, для логирования). При этом функция не должна заботиться о том, как и куда выводить логи. Если функция оказалась в контексте монады *Writer*, ей становятся доступными её функции. *Writer* выводит информацию в любой моноид, например, в строку (тип моноида передаётся как параметр класса). Вот пример, демонстрирующий использование монады *Writer* на примере функции, делящей число пополам:

```
half x = do
  tell ("I just halved "++(show x)++"!")
  return (x `div` 2)
main = print . execWriter $ half 8 >>= half
```

Функция *main* печатает: "I just halved 8!I just halved 4!".

Монада State

Монада *State* похожа на монаду *Reader* в том, что она также хранит некоторое значение (состояние), его тип передаётся как параметр класса. Отличие же состоит в том, что это хранимое значение можно менять. Эту монаду можно, например, использовать в качестве аналога обычной (изменяемой) переменной в императивных языках программирования. Смысл же этого класса в том, чтобы хранить некоторое общее для всех функций состояние, которое все функции могут запрашивать и, при необходимости, менять. Основные функции класса *State* – *get*, *put* и *modify*, которые, соответственно, запрашивают состояние, устанавливают новое и модифицируют.

Трансформеры монад

В реальных вычислениях обычно бывает нужно использовать не какую-то одну монаду, а одновременно несколько. Например, все три описанные ранее монады – явные кандидаты на одновременное использование. Есть ещё монады *Maybe* и *Either*, обрабатывающие ошибки вычислений, и другие монады. Поэтому встаёт вопрос об объединении монад. Этой цели служат т.н. трансформеры монад, позволяющие вносить

одну монаду внутрь другой. С помощью трансформеров монад строится стек монад, позволяющий вести вычисления одновременно в нескольких монадах. Например, трансформер монад *ReaderT* позволяет построить монаду *Reader* поверх какой-нибудь другой монады. Аналогично работают трансформеры монад *WriterT*, *StateT* и другие. Более того, сами классы *Reader*, *Writer* и *State* в языке Haskell и в библиотеке *funcprog* определяются как «неподвижные точки» (fixed points) соответствующих трансформеров монад, т.е. как соответствующие трансформеры монад над монадой *Identity* (которая является тривиальным контейнером).

В следующем примере параметр делится на число из монады *Reader*, лог пишется в монаду *Writer*:

```
type Divide=WriterT String (Reader Int)
runApp k n=runReader (runWriterT k) n

divide :: Int -> Divide Int
divide x = do
  n <- ask
  tell $ "Dividing " ++ (show x) ++ " by " ++ (show n) ++ "!"
  return $ x `div` n

main = print . runApp (divide 8) $ 2
```

Функция *main* печатает: (4,"Dividing 8 by 2!"). Здесь 4 – это результат деления. Монады *Reader* и *Writer* можно поменять местами. При этом изменятся только две первых строки:

```
type Divide=ReaderT Int (Writer String)
runApp k n=runWriter (runReaderT k n)
```

Комбинаторный парсер Parsec

В языке Haskell *Parsec* – это простой, хорошо документированный и быстрый парсер. Он определяется как неподвижная точка трансформера монад *ParsecT*. Библиотека *Parsec* представляет собой набор достаточно простых парсеров с широкими возможностями их комбинирования. Сам класс *ParsecT* является функтором, аппликативом, монадой и альтернативой (*Alternative*), а также реализует интерфейс *MonadPlus*. Соответственно, все возможности, которые предлагают эти классы, для парсеров также доступны. Вот как можно написать калькулятор со скобками, с лево-ассоциативностью арифметических операций и с приоритетом операций умножения и деления над операциями сложения и вычитания с помощью этого парсера:

```
{-# LANGUAGE FlexibleContexts #-}
import Prelude hiding (subtract)
import Text.Parsec
```

```
import Text.Parsec.String (Parser)

number :: Parser Float
number = fmap readFloat numberString
  where readFloat = read :: String -> Float
  numberString = many1 digit

plus :: Parser (Float -> Float -> Float)
plus = fmap (const add) plusChar
  where add a b = a + b
        plusChar = char '+'

subtract :: Parser (Float -> Float -> Float)
subtract = fmap (const subtract) subtractChar
  where subtract a b = a - b
        subtractChar = char '-'

divide :: Parser (Float -> Float -> Float)
divide = fmap (const divide) divideChar
  where divide a b = a / b
        divideChar = char '/'

multiply :: Parser (Float -> Float -> Float)
multiply = fmap (const multiply) multiplyChar
  where multiply a b = a * b
        multiplyChar = char '*'

expressionWithParens :: Parser Float
expressionWithParens = between (char '(') (char ')') expression

expressionWithParensOrNumber :: Parser Float
expressionWithParensOrNumber = withWhitespace (expressionWithParens
   <|> number)
  where withWhitespace parser = between spaces spaces parser

chainMultiplyDivide :: Parser Float
chainMultiplyDivide = chainl1 expressionWithParensOrNumber (divide
   <|> multiply)

expression :: Parser Float
expression = chainl1 chainMultiplyDivide (plus <|> subtract)
```

Обратиться к нему можно так:

```
parse expression "(unknown)" "7-1-3*(4+5)"
```

Выдастся результат: Right (-21.0)

Библиотека funcprog

Библиотека funcprog (см. [3]) предназначена для того, чтобы на языке C++ можно было писать в функциональном стиле, близком к стилю программирования на языке Haskell. В ней реализована большая часть из перечисленных выше возможностей языка Haskell, включая каррирование функций, композицию функций, применение функции к аргументу, функторы и монады, классы *Maybe*, *Either*, *List* (на основе класса *std::list*), *Foldable* и *Traversable*, монады *Reader*, *Writer* и *State*, трансформеры монад *IdentityT*, *MaybeT*, *ReaderT*, *WriterT* и *StateT*. Так же, как и в языке Haskell, классы *Reader*, *Writer* и *State* определены как неподвижные точки соответствующих трансформеров монад. Удалось перенести большую часть (но не всё) парсера Parsec.

Под «функцией» в библиотеке *funcprog* подразумевается объект класса *std::function* из стандартной библиотеки языка C++. К объекту этого класса может быть преобразована практически любая обычная функция или функциональный объект, включая лямбда-выражения. Для объектов этого класса переопределены многие операторы языка C++, например, оператор композиции функций реализован через переопределённый оператор & (т.к. бинарного оператора . (точка) в языке C++ нет). В языке Haskell можно почти любую комбинацию символов определить как оператор (функцию), в языке C++ это невозможно, приходится довольствоваться встроенными операторами языка, которых часто не хватает. Например, монадная функция mbind так же, как и в языке Haskell, реализована через оператор >>=, а альтернативная операция <|> через оператор | (что близко).

Покажем, как выглядят некоторые демонстрационные примеры, показанные выше, на языке C++ с использованием библиотеки funcprog. Первый пример – из монады *Reader*:

```
using R = String;
using _Rdr = _Reader<R>;
const Reader<R, String>
  hello = _Rdr::reader(_([](R const& name){
    return String("Hello,"+name+"!"); })),
  bye=_Rdr::reader(_([](R const& name){
    return String("Bye,"+name+"!"); })),
  convo=_Rdr::reader(_const_<String>(_(concat2<char>)))*hello*bye;
BOOST_TEST(convo.run("Fred").run() == "Hello,Fred!Bye,Fred!");
```

Второй – из монады *Writer*:

```
using W = String;
using _Wrt = _Writer<W>;
using mWrt = Monad<_Wrt>;
const function_t<Writer<W, int>(int)>
half = [](int x) { return _Wrt::tell(
  W("I␣just␣halved␣" + eval(x) + "!")) >> mWrt::mreturn(x / 2); };
```

```
BOOST_TEST(half(8).exec().run() == "I␣just␣halved␣8!");
BOOST_TEST((half(8)>>=half).exec().run() ==
  "I␣just␣halved␣8!I␣just␣halved␣4!");
```

Третий пример – из трансформеров монад:

```
template<typename _Divide>
typename _Divide::template type<int>
divide(int x){ return _do( n, MonadReader<int,_Divide>::ask(),
  return MonadWriter<String,_Divide>::tell("Dividing␣" + eval(x) +
    "␣by␣" + eval(n) + "!") >> Monad<_Divide>::mreturn(x / n);
  );
}
BOOST_AUTO_TEST_CASE(test_MonadTrans){
  BOOST_TEST(eval(divide<_WriterT<String,_Reader<int>>>
    (8).run().run(2).run()) == "(4,\"Dividing␣8␣by␣2!\")");
}
```

При сравнении с кодом на языке Haskell видно, что, с одной стороны, получается более громоздко, а с другой стороны, очень похоже. Калькулятор на парсере Parsec также заработал, не без трудностей, но многие сложные места удалось преодолеть.

Сеточно-операторный подход

Сеточно-операторный подход к программированию (см. [4, 5]) предназначен для решения двух задач. Во-первых, он позволяет кратко (сравнимо с текстом в математической литературе) записывать формулы в текстах программ, и, во-вторых, легко, путём простой перекомпиляции, переносить программы на нетрадиционные вычислительные архитектуры, такие как CUDA. В рамках этого подхода введено понятие программного сеточного оператора, аналогичного математическому оператору, который можно применять к вычисляемому объекту (например, к сеточной функции).

Основными объектами в сеточно-операторном подходе к программированию являются вычисляемый объект (evaluable object) и сеточный оператор (grid operator). Простейший вычисляемый объект – это сеточная функция (grid function), определённая на элементах некоторой сетки (вершинах, рёбрах, гранях или ячейках). Арифметические операции между вычисляемыми объектами или вычисляемым объектом и числом, а также применение сеточного оператора к вычисляемому объекту порождают сеточные вычислители, которые также являются вычисляемыми объектами. Таким образом, вычисляемые объекты определяются рекурсивно (сумма и разность вычисляемых объектов есть вычисляемый объект, результат применения сеточного оператора к вычисляемому объекту есть вычисляемый объект и т.д.), причём рекурсия

завершается на сеточных функциях. Таким образом, вычисляемый объект «запоминает» цепочку вычислений, не производя их. Вычисляемый объект может быть присвоен сеточной функции, именно в этот момент запускаются вычисления в соответствии с запомненной цепочкой. Таким образом, реализуется концепция отложенных вычислений. Основная операция, которая должна быть реализована к любом вычисляемом объекте (концепция вычисляемого объекта) – это оператор [], принимающий индекс элемента сетки (неотрицательное число) и возвращающий значение по этому индексу.

Сеточно-операторный подход к программированию реализован в виде библиотеки классов для языка C++ (*gridmath*). Данная библиотека активно использует метапрограммирование шаблонов языка C++ (см. [6], в частности, шаблоны выражений (см. [14, 15]) для реализации вычисляемых объектов. Методы функционального программирования могут быть эффективно применены к сеточно-операторному подходу к программированию. Покажем, как это можно сделать. Как уже говорилось, одним из основных объектов в сеточно-операторном подходе является вычисляемый объект, в котором должен быть определён оператор [], принимающий индекс и возвращающий значение по этому индексу.

Покажем, что вычисляемый объект можно сделать функтором, аппликативным функтором и монадой. В рамках библиотеки *funcprog* для этого для вычисляемого объекта нужно создать специализации классов *Functor* (с функцией *fmap*), *Applicative* (с функциями *pure* и *apply*) и *Monad* (с функциями *mreturn* и *mbind*), причём функция *fmap* должна удовлетворять функторным законам, функции *pure* и *apply* – аппликативным законам, а функции *mreturn* и *mbind* – монадным законам. Каждая из пяти перечисленных функций должна возвращать вычисляемый объект.

Функтор

Функция *fmap* принимает функцию с одним параметром и функтор (в нашем случае вычисляемый объект) и возвращает тот же функтор (новый вычисляемый объект). Этот новый вычисляемый объект запоминает параметры функции *fmap* в своих переменных-членах класса (назовём их f и eobj) и реализует оператор [] следующим образом:

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

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

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

```
fmap id eobj = id eobj
```

или (по определению функции id):

```
fmap id eobj = eobj
```

Далее,

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

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

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

Тогда

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

С другой стороны:

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

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

Аппликатив

Функция *pure* принимает некоторое значение и «вносит» его в аппликатив. В нашем случае делает из него вычисляемый объект. 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 вычисляемых объекта: первый (назовём его eobj_f) возвращает функции, а второй (назовём его eobj) – некоторые значения (параметры этих функций). Определим вычисляемый объект функции *apply* следующим образом:

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

т.е. apply (pure id) eobj = eobj. 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 *mreturn* (в языке Haskell *return*) определена так же, как и функция *pure*:

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

Монадная функция *mbind* (в языке Haskell и в библиотеке funcprog оператор >>=) принимает монаду (в нашем случае вычисляемый объект, обозначим его переменной eobj) и функцию, принимающую обычное (не монадное) значение и возвращающую монаду (вычисляемый объект). Определим функцию *mbind* следующим образом:

```
(mbind eobj f)[i] = (f eobj[i])[i]
```

Теорема 3 (Теорема о монаде)**.** *Определёные выше функции mreturn и mbind удовлетворяет монадным законам.*

Доказательство. Перепишем монадные законы в терминах функций mreturn и mbind:

```
1. mbind (mreturn x) f = f x
2. mbind eobj mreturn = eobj
3. mbind (mbind eobj f) g = mbind eobj (\x->mbind (f x) g)
```

Первый закон:

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

Первый закон доказан. Второй закон:

```
(mbind eobj mreturn)[i] = -- definition of mbind
  (mreturn eobj[i])[i] =  -- definition of mreturn
  eobj[i]
```

Второй закон доказан. Третий закон:

```
(mbind (mbind eobj f) g)[i] = -- definition of mbind
  (g (mbind eobj f)[i])[i] =  -- definition of mbind
  (g (f eobj[i])[i])[i]
```

С другой стороны:

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

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

Итак, вычисляемые объекты являются функторами, аппликативами и монадами. Это значит, что любую обычную унарную функцию можно «применить» к вычисляемому объекту (с помощью функции *fmap*). Такое применение вернёт новый вычисляемый объект. Любую бинарную функцию можно «применить» к двум вычисляемым объектам (с помощью функции *apply*) и т.д., любую функцию с n аргументами можно «применить» к n вычисляемым объектам. Это можно сделать одной строчкой. Например, путь есть две сеточных функции f и g. Тогда можно написать так:

```
g = fmap(sin, f);
```

Т.к. вычисляемые объекты 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 вычисляемые объекты.

Заключение

Функциональное программирование не противоречит численным методам, а, наоборот, помогает более чётко и кратко сформулировать задачу. Применение функциональных возможностей языка C++ повышает читаемость программ, упрощает их отладку, уменьшает объём кода и за счёт этого уменьшает число возможных ошибок. В новых версиях языка C++ появилась возможность распараллеливать стандартные алгоритмы, что позволяет существенно ускорять программы на многоядерных процессорах, каковыми являются все современные процессоры, а на современных суперкомпьютерных кластерах число ядер на одном узле может составлять десятки.

Написанная автором библиотека функционального программирования *funcprog* позволяет писать на языке C++ в функциональном стиле, близком к стилю программ на функциональном языке Haskell. Это позволяет программировать алгоритмически сложные задачи на языке C++, не сильно теряя в компактности и элегантности, но существенно выигрывая в производительности по сравнению с языком Haskell.

Разработанный автором сеточно-операторный подход к программированию численных задач позволяет кратко записывать формулы в программах, в том числе за счёт введения сеточных

программных операторов, аналогичных математическим операторам. Кроме того, написанные в этом подходе программы можно запускать на вычислительной архитектуре CUDA путём простой перекомпиляции соответствующим компилятором. Соединение функционального стиля и сеточно-операторного подхода позволяет ещё больше упростить текст программ. Вычисляемые объекты, одни из основных объектов в сеточно-операторном подходе, являются функторами, аппликативами и монадами, для вычисляемых объектов реализованы все нужные функции, доказана корректность определения этих функций. Это позволяет применять к сеточно-операторному подходу многие возможности, предоставляемые функциональным программированием и библиотекой *funcprog*.

Библиографический список

1. Содружество «РЕФАЛ/Суперкомпиляция». URL: http://www.refal.ru/
2. Haskell language. URL: https://www.haskell.org/
3. Краснов М. М. Библиотека функционального программирования для языка C++ // Программирование, 2020, №5 (принято в печать).
4. 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
5. Краснов М. М. Кандидатская диссертация "Сеточно-операторный подход к программированию задач математической физики". Автореферат. URL: http://keldysh.ru/council/1/2017-krasnov/avtoref.pdf
6. David Abrahams, Aleksey Gurtovoy. C++ Template Metaprogramming. Addison-Wesley. – 2004. 400 с. ISBN 978-0-321-22725-6.
7. Bjarne Stroustrup. The C++ Programming Language, Fourth Edition. Addison-Wesley, 2013. ISBN 978-0-321-56384-2, 1368 с.
8. Bjarne Stroustrup. Programming: Principles and Practice Using C++, Second Edition. Addison-Wesley, 2013, 1312 с. ISBN 978-0-321-99278-9.
9. Bjarne Stroustrup. A Tour of C++. Addison-Wesley, 2014, 192 с. ISBN 978-0-321-95831-0.
10. Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley, 1994, 480 с. ISBN 978-0-201-54330-8.

11. 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.

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

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

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

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

16. Boost C++ Libraries. URL: http://www.boost.org/

17. 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.

18. 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.