ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 47–59

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 519.6

БИБЛИОТЕКА ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ ДЛЯ ЯЗЫКА C++

© 2020 г. М. М. Краснов[a,*]

[a] *Институт прикладной математики им. М.В. Келдыша РАН*
125047 Москва, Миусская пл., 4, Россия

******E-mail: kmm@kiam.ru*

Поступила в редакцию 26.07.2019 г.
После доработки 30.07.2019 г.
Принята к публикации 08.11.2019 г.

Современные функциональные языки программирования, такие, как Haskell, Scala, ML, F# обладают свойствами, позволяющими с их помощью относительно легко реализовывать логически сложные алгоритмы. К таким свойствам можно отнести композицию функций, каррирование, метафункции (функции над функциями) и ряд других. Это позволяет путем комбинирования простых функций получать более сложные функции. В качестве примера сложного алгоритма упомянем разбор строки (парсер). В языке Haskell имеется библиотека Parsec, представляющая из себя набор элементарных парсеров, комбинируя которые, можно создать более сложные парсеры. Это делает ее относительно простой и в то же время мощной. В языке C++ большинство из этих возможностей в явном виде отсутствуют. В то же время язык C++ достаточно мощный и позволяет реализовать многие из свойств функциональных языков. В предлагаемой библиотеке делается попытка по мере возможности решить данную задачу.

DOI: 10.31857/S0132347420050040

1. ВВЕДЕНИЕ

Решение написать данную библиотеку функционального программирования (funcprog) было принято под впечатлением от языка Haskell [1] в процессе его изучения. Его мощная система типов, функциональная “чистота”, “ленивость”, бесконечные списки, поддержка каррирования функций и их композиции, монады и трансформеры монад, а также многое другое, делают этот язык мощным инструментом для написания логически сложных алгоритмов. Другие современные языки функционального программирования (Scala [2], ML [3], F# [4]) также обладают аналогичными свойствами. При изучении языка Haskell ставилась задача понять, как на нем можно было бы максимально легко реализовать парсер математических формул (калькулятор). При этом выяснилось, что парсер общего назначения в библиотеке языка уже есть, и реализовать на нем парсер формул оказалось несложной задачей.

В то же время основным языком, которым пользуются многие математики, в том числе и автор данной статьи, для решения повседневных задач (в основном для реализации численных методов газовой динамики), является язык C++. При решении таких задач логически сложные алгоритмы тоже часто встречаются (например, тот же разбор формул), и использовать возможности функциональных языков программирования для их реализации кажется весьма привлекательным. Но совместить в одной программе два языка представляется сложной задачей, поэтому встал вопрос – а можно ли, оставаясь в рамках языка C++, писать на нем в стиле языка Haskell, используя многие из его свойств, включая каррирование функций и их композицию, монады, “ленивые” вычисления и т.д. Речь, по сути, идет о реализации в виде библиотеки классов и функций предметно-ориентированного языка (DSL, Domain Specific Language) функционального программирования.

Основное свойство всех функциональных языков – то, что функция в них является полноправным участником вычислений, т.е. функцию можно передать в качестве параметра функции и вернуть как результат вычислений. В частности, во всех функциональных языках есть понятие лямбда-выражения, результатом которого является функция, которую можно передать как параметр в другую функцию или вернуть как результат работы. В современном языке C++ такое понятие также есть, но появилось оно только в стандарте языка от 2011 года. Так как без поддерживаемых языком лямбда-выражений реализовать библио-

теку функционального программирования представляется сложным, да и не нужным (компилятор с поддержкой стандарта 2011 года есть почти на всех системах), то было принято решение не поддерживать более ранние версии языка. Библиотек функционального программирования для языка C++ много (см., например, [5, 6]). Особенностью данной библиотеки является, во-первых, то, что в ней делается попытка максимально полно реализовать функционал языка Haskell и, во-вторых, максимально используются возможности современного стандарта языка C++ от 2011 года и более поздних. При реализации библиотеки активно используется метапрограммирование шаблонов языка C++ (см. [7–9]). Далее дается описание модулей, реализованных в рамках библиотеки funcprog.

2. БАЗОВЫЕ КОМПОНЕНТЫ БИБЛИОТЕКИ

В данной библиотеке максимально используются имеющиеся возможности, во-первых, стандартной библиотеки языка C++, и, во-вторых, библиотеки boost [10]. Библиотека boost является полигоном для развития языка C++, прежде всего – его стандартной библиотеки. Многое из boost-а уже перекочевало в нее, что-то ждет своего часа. В частности, под функцией в библиотеке funcprog подразумевается объект класса *std::function* (в библиотеке boost это класс *boost::function*). Объект этого класса может быть обвязкой обычной функции, лямбда-функции (функции, порожденной лямбда-выражением) или функционального объекта (объекта, имеющего функциональный оператор). Таким образом, если функция принимает аргумент-функцию, то это значит (в рамках данной библиотеки), что она принимает аргумент класса *std::function*, а если функция возвращает функцию, то это значит, что она возвращает объект класса *std::function*. Шаблон класса *std::function* имеет один аргумент вида *Ret*(*Args*), где *Ret* – тип возвращаемого значения, а *Args* – список типов аргументов. Например, функция, принимающая аргументы типа *int* и *double* и возвращающая строку, имеет следующее описание:

```
std::function<std::string (int, double )>f;
```

Также в библиотеке активно используется переменное число параметров шаблона функции (variadic templates). Например, функция, принимающая первый параметр типа int и, возможно, еще какие-то параметры и возвращающая результат типа double, может быть описана так:

```
template<typename ... Types>
std::function<double (int , Types ... )> f;
```

Для реализации класса *Maybe* (который может содержать или нет значение некоторого типа) используется класс *boost::optional*, а если используется C++ стандарта 2017 г., то *std::optional*, а для класса *Either* (который содержит значение одного из двух типов) используется класс *boost::variant*, а если используется C++ стандарта 2017 г., то *std::variant*.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

Как уже писалось выше, при написании библиотеки ставилась задача как можно полнее реализовать функционал языка Haskell. Но языки C++ и Haskell отличаются очень сильно, и поэтому может возникнуть законный вопрос: в какой степени возможности языка Haskell реализуемы на C++? Неполный список нетривиальных для реализации вещей включает ленивые вычисления, каррирование функций, η (эта)-редукцию (или η-преобразование), бесконечные списки, мощную систему вывода типов, конструкторы типов, полиморфные функции и много другое. Кое-что из перечисленного реализовать не удалось (например, бесконечные списки и η-редукцию), что-то – с определенными ограничениями (полиморфные функции), что-то реализовать удалось достаточно полно, но выглядит это менее элегантно, чем в Haskell-е. Особенно следует упомянуть переопределенные операторы. В языке Haskell почти любую последовательность символов можно определить как оператор, и в языке и его библиотеке это активно используется. С другой стороны, в языке C++ мы ограничены только тем набором операторов, которые изначально встроены в язык. Мы их можем переопределять для наших типов, но добавить новый оператор язык не позволяет. При реализации библиотеки те операторы языка Haskell, для которых есть аналоги в C++, сохранялись, другим (наиболее популярным) подыскивалась замена из числа имеющихся, часть операторов или заменялась на функции, или просто оставалась без реализации.

3.1. Функции

Как уже говорилось выше, под функцией в библиотеке подразумевается объект класса *std::function* (по умолчанию) или *boost::function*. Конкретный класс, реализующий функцию, скрыт за типом *function_t*:

```
template<typename F>
using function_t = std::function<F>;
```

В дальнейшем, чтобы избегать неоднозначности в термине “функция”, словом “функция” будем обозначать объект класса *function_t*, а про обычную функцию так всегда и будем писать – “обычная функция”. Любая обычная функция или функциональный объект (в том числе результат лямбда-выражения) могут быть преобразованы в функцию (к классу *function_t*). Для автоматического такого преобразования в библиотеке имеется специальная обычная функция с односимвольным именем _ (подчерк):

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

Эта обычная функция принимает в качестве параметра или указатель на обычную функцию, или ссылку на функциональный объект (в т.ч. на результат лямбда-выражения). Для преобразования типа параметра в тип функции используется метафункция *function_type*:

```
template<typename Ret , typename ... Args>
struct function_type {
using type = function_t <Ret (Args ... ) >;};

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

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

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

// Common functions
template<typename Ret , typename ... Args>
struct function_type <Ret ( * ) (Args ... )> :
 function_type_<Ret , Args ... > { } ;

template<typename F>
using function_type_t =
   typename function_type <F>: : type ;
```

3.2. Ленивые вычисления

Допустим, нам надо описать прототип некоторой функции *f*, принимающей аргумент некоторого типа *a* и возвращающей результат некоторого типа *b*. В языке Haskell прототип такой функции записывается так: f::a–>b. При этом фактически в качестве параметра этой функции можно передать или значение (константу) типа *a*, или функцию без параметров, возвращающую результат типа *a*. С точки зрения языка Haskell это одно и то же. В языке C++ этому прототипу можно соотнести три разных прототипа:

```
b f ( a ) ;
b f ( a const&) ;
b f ( function_t<a ( )> const&) ;
```

Первый из этих прототипов принимает параметр по значению, второй – по константной ссылке и третий – функцию без параметров. С точки зрения языка C++ это три разных прототипа. Как же на C++ написать функцию, “готовую” к ленивым вычислениям, т.е. функцию, которой в качестве параметра можно было бы передать как значение нужного типа, как и функцию без параметров, возвращающую значение этого типа? В библиотеке funcprog для этого определяются несколько типов и функций. Во-первых, есть специальный тип для функций без параметров:

```
template<typename T>
using f0 = function_t<T( ) >;
```

Далее, есть функция *value_of*, которая, если ей в качестве параметра передать ссылку на функцию без параметров, вызывает ее и возвращает результат этой функции, в остальных случаях (если параметр не является ссылкой на функцию без параметров), просто возвращает этот параметр (его копию):

```
template<typename T>
T value_of (T const& arg )
{ return arg ; }

template<typename T>
T value_of ( f0<T> const& f )
{ return f ( ) ; }
```

Имеется метафункция *remove_f0*, преобразующая тип функции без параметров в простой тип:

```
template<typename T>
struct remove_f0
{ typedef T type ; } ;

template<typename T>
struct remove_f0<f0<T> >
{ typedef T type ; } ;

template<typename T>
using remove_f0_t =
  typename remove_f0<T>: : type ;
```

Теперь мы можем написать функцию, принимающую как значение некоторого типа, так и функцию без параметров:

```
template<typename Arg>
remove_f0_t<Arg> myfunc (Arg const& a ) {
  const remove_f0_t<Arg> a_ =
    value_of ( a ) ;
  return a_ * a_;
}
```

Обратиться к этой функции можно, например, так:

```
const f0<int> f = [ ] ( ) { return 5 ; } ;
s td : : cout
 << myfunc ( 5 ) << std : : endl
 << myfunc ( f ) << std : : endl ;
```

3.3. Каррирование

Каррирование не является обязательным атрибутом функциональных языков программирования (в отличие, например, от лямбда-выражений, которые есть во всех языках функционального программирования). Например, в одном из первых языков функционального программирования Lisp каррирования нет. Тем не менее, этот механизм очень удобный и современные функциональные языки его, как привило, имеют. Смысл каррирования в следующем. Если мы при вызове функции укажем не все параметры (опустив несколько последних), то результатом такого вызова будет функция с меньшим числом параметров (равным числу опущенных параметров), при вызове которой будет вызвана исходная функция. На самом деле язык Haskell идет еще дальше. Даже если указать все параметры функции, то и в этом случае фактического вызова функции не происходит. Вместо этого создается функция без параметров, т.е. каррирование идет “до конца”. На этом основаны “ленивые” вычисления. Фактический вызов функции откладывается как можно дальше.

Каррирование в библиотеке funcprog реализовано с помощью перегруженных (overloaded) функций. Пусть имеется некоторая функция (например, с именем myfunc) с тремя параметрами. Тогда при определении этой функции будет создано дополнительно еще 2 перегруженных функции с именем _myfunc, имеющих один и два параметра и возвращающих, соответственно, функции, имеющие два и один параметр. Каждая из этих возвращаемых функций при вызове будет обращаться к функции myfunc. Вот как, например, выглядит определение функции liftA2:

```
DEFINE_FUNCTION_3( 3 ,LIFTA2_T(T0 ,T1 ,T2) ,
    l i f t A 2  , f u n c t i o n _ t <T2> const&, f ,
    T1 const&, x , T0 const&, y ,
    return f / x * y ; )
```

Макрос DEFINE_FUNCTION_3 создаст определение функции liftA2:

```
template<typename T0 , typename T1 ,
typename T2> LIFTA2_T(T0 , T1 , T2)
l i f t A 2 ( f u n c t i o n _ t <T2> const& f ,
  T1 const& x , T0 const& y )
{ return f / x * y ; }
```

а также определение двух перегруженных функций (с одним первым и с двумя первыми параметрами) с именем _liftA2. Вот как, например, будет выглядеть определение функции с двумя первыми параметрами:

```
template<typename T0 , typename T1 ,
  typename T2> f u n c t i o n _ t <
  LIFTA2_T(T0 ,T1 ,T2) (T0 const&)
> _ l i f t A 2 ( f u n c t i o n _ t <T2> const& f ,
    T1 const& x ) {
  return [ f , x ] (T0 const& y ) {
    return l i f t A 2 ( f , x , y ) ; } ;
}
```

3.4. Применение функции

Применение функции к значению для функций с одним параметром в языке C++ – это, по сути, просто вызов функции и, как известно, делается с помощью выражения типа f(x). В языке Haskell скобки опускаются и это выглядит как f x,

или, с использованием оператора $ (специального оператора применения функции к параметру), как f $ x. Кроме того, как уже говорилось выше, в языке Haskell фактического вызова функции при этом не происходит, вместо этого порождается функция без параметров.

В случае же функции с несколькими параметрами ситуация иная. В языке C++ мы не можем указать только часть параметров и при вызове функции должны всегда указывать все параметры, так что применение функции с несколькими параметрами к одному параметру невозможно. В языке Haskell с этим никаких проблем нет, в этом случае просто порождается функция с меньшим на единицу числом параметров. Отказ от скобок при вызове функций в языке Haskell имеет глубокий смысл. Он позволяет интерпретировать применение функции к нескольким параметрам как применение функции последовательно к каждому параметру по одному: $f\ x\ y \equiv (f\ x)y$.

Таким образом, важной является возможность применения функции к одному (первому) параметру. При этом должна порождаться функция с меньшим на единицу числом параметров, которую, в свою очередь, можно применить к ее первому параметру (второму параметру исходной функции) и т.д. В библиотеке funcprog для этой цели был выбран оператор <<. Он является весьма точным аналогом оператора $ языка Haskell. В частности, если применяемая функция имеет один параметр, то в результате получится функция без параметров. Тип значения, к которому применяется функция, должен соответствовать типу первого параметра функции. Оператор << определен следующим образом:

```
template<typename Ret , typename Arg0 , typename . . . Args>
function_t<Ret (Args . . . )> operator<<(
  function_t<Ret (Arg0 , Args . . . )>const& f ,
  fdecay<Arg0> const& arg0 )
{
  return [ f , arg0 ] ( Args . . . args ) {
    return f ( arg0 , args . . . ) ; } ;
}
```

Это определение относительно несложное. По сути, делается привязка (bind) первого параметра функции. В стандартной библиотеке языка C++ есть функция std::bind, делающая практически то же самое. Проблема с этой функцией состоит в том, что ее результат (функциональный объект) трудно преобразовать в объект std::function из-за того, что функциональный оператор этого объекта шаблонный.

3.5. Композиция функций

В языке Haskell для композиции функций применяется оператор . (точка). В библиотеке funcprog для этого был выбран оператор &:

```
( g & f ) (x , a r g s . . . ) =
  g ( i n v o k e _ f 0 ( f << x ) , a r g s . . . )
```

Здесь x – первый параметр функции f, а *args* – дополнительные параметры функции g. Тип результата функции f должен соответствовать (быть преобразуем к) типу первого параметра функции g. Результатом композиции функций будет функция, имеющая то же число параметров, что и функция g, но тип первого параметра этой функции будет тот же, что и тип первого параметра функции f. Если функция f имеет более одного параметра, то функция g должна первым параметром принимать функцию (результат применения функции f к ее первому параметру).

4. ФУНКТОРЫ, АППЛИКАТИВНЫЕ ФУНКТОРЫ, МОНАДЫ

4.1. Функторы

Функторы и монады 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. 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. 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 нем 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 (⟨$⟩) (в библиотеке funcprog – operator/), 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 ее в функцию, принимающую и возвращающую функторы.

Вот так выглядит специализация класса *Functor* для списков:

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

Результатом применения функции к списку является список из преобразованных значений (возможно, другого типа).

Но 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
```

Оператор (⟨*⟩) (в библиотеке funcprog operator*) принимает в качестве первого параметра функцию, помещенную в функтор и второго параметра – значение, помещенное в тот же функтор и возвращает результат в том же функторе. Если мы теперь посмотрим на прототип и реализацию функции *liftA2*, то мы увидим, что она передает 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 вернет функцию (b->c), помещенную в функтор, которая затем передается 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 ее как функцию с одним аргументом, принимающую функцию, работающую с обычными значениями и возвращающую функцию, работающую с функторами, “поднимающую” функцию с двумя аргументами (отсюда и ее 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
```

В классе *Applicative* есть также функция *pure*, помещающая обычное значение в “чистый” аппликатив. С ее помощью можно, например, написать функцию, “поднимающую” в аппликатив функцию с одним параметром:

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

4.2. Монады

Монады можно рассматривать как дальнейшее продолжение аппликатива, они предназначены для построения цепочек монадных вычислений. Каждая монада имеет две основных операции: *mreturn* (в языке Haskell *return*) и *mbind* (в языке Haskell и в библиотеке funcprog оператор ⩾=). Операция *mreturn* аналогична операции *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 ее результат.

Для каждой монады две монадные операции должны удовлетворять трем т.н. “монадным законам”. Для того, чтобы их сформулировать, введем операцию монадной композиции функций (*mcompose* или оператор (>=>)). Она определяется следующим образом:

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

Оба операнда – функции, принимающие обычные значения и возвращающие монадные. Результатом монадной композиции функций также является функция, принимающая обычное значение и возвращающее монадное. Следовательно, опера-

цию монадной композиции можно рассматривать как групповую операцию в пространстве таких функций (принимающих обычное значение и возвращающих монадное в некоторой монаде). 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 операции *mreturn* и *mbind* должны быть определены так, чтобы, во-первых, операция *mreturn* являлась единичным элементом (левым и правым) монадной композиции (первые два закона), и, во-вторых, монадная композиция должна быть ассоциативной (третий закон).

Еще одна очень широко применяемая при работе с монадами конструкция – т.н. *do*-нотация. Она позволяет существенно сократить запись и сделать ее более наглядной. Смысл ее 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 еще существеннее. Вместо

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

можно написать:

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

Библиотека funcprog также поддерживает *do*-нотацию. Она реализована с помощью макроса *_do*:

```
#define _do( var , mexpr , expr ) \
  ( (mexpr ) >>= _( [=] ( typename \
    std::decay_t<decltype (mexpr )> \
   ::value_type const& var ) { expr }) )
```

С помощью этого макроса описанное выше вычисление можно записать так:

```
_do(x , m, action )
```

Для большего числа переменных (от двух до пяти) есть макросы от *_do2* до *_do5*, реализованные через цепочку обращений к макросу *_do*. Вот текст реализации макросов *_do2* и *_do3*:

```
#define _do2( v1 ,m1 , v2 ,m2, expr ) \
  _do( v1 ,m1 , return _do( v2 ,m2, expr ) ; )

#define _do3( v1 ,m1, v2 ,m2 , v3 ,m3, expr ) \
 _do2( v1 ,m1, v2 ,m2,
   return _do( v3 ,m3, expr ) ; )
```

Соответственно, пример с двумя монадами можно записать так:

```
_do2(x , m1, y , m2, action )
```

4.3. Реализация функторов и монад

С помощью библиотеки funcprog любой подходящий класс (в том числе и внешний) можно сделать функтором и монадой. В сам класс никаких изменений вносить не требуется. В рамках самой библиотеки монадами являются, к примеру, классы *Identity*, *Maybe* и *List*. Рассмотрим общие принципы реализации. Любой функтор или монада являются типом с одним параметром. В языке C++ типы с параметром реализуются с помощью шаблонов классов. Например, шаблон класса *Maybe* определен так:

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

Но сам шаблон класса типом не является, и его нельзя передать как параметр шаблона другого класса, что бывает иногда нужно. Чтобы это можно было сделать, определяется еще один класс (без шаблона) с именем *_Maybe* (с подчерком впереди). Это уже настоящий класс, его можно передавать как параметр шаблона. В нем определена метафункция *type*, возвращающая класс *Maybe*:

```
struct _Maybe {
  template<typename A>
  using type = Maybe<A>;
};
```

Шаблон класса *Maybe* наследуется от этого класса:

```
template<typename A>
struct Maybe : _Maybe , optional_ t <A>{
  ...
};
```

Для того, чтобы класс *Maybe* сделать функтором, нужно написать специализацию шаблонов классов *is_functor* и *is_same_functor* для класса *Maybe* и специализацию шаблона класса *Functor* для класса *_Maybe*. Шаблон класса (метафункции) *is_functor* принимает тип и возвращает в переменной *value* булевское значение, говорящее о том, является этот тип функтором. Шаблон класса (метафункции) *is_same_functor* принимает два типа и возвращает булевское значение, говорящее о том, являются эти типы одним и тем же функтором. По умолчанию обе эти метафункции всегда возвращают значение *false*:

```
template<typename T>
struct is_functor : std::false_type {};
template<typename T1 , typename T2>
struct is_same_functor : std::false_type {};
```

Шаблон класса *Functor* просто продекларирован:

```
template<typename T> struct Functor ;
```

Специализация этих классов для класса *Maybe* выглядит так:

```
template<typename A>
struct is_functor<Maybe<A> > :
  std::true_type {};
template<typename A1 , typename A2>
struct is_same_functor<Maybe<A1>,
  Maybe<A2> > : std::true_type {};
template<> struct Functor<_Maybe>{
  ...
};
```

Внутри класса Functor<_Maybe> требуется определить метафункцию *fmap_result_type* и статическую функцию *fmap*. Метафункция *fmap_result_type* принимает в качестве параметра тип функции – первого параметра функции *fmap* и возвращает в переменной *type* тип возвращаемого функцией *fmap* значения.

Определение аппликатива и монады аналогично определению функтора. Нужно создать специализации шаблонов классов *is_applicative*, *is_same_applicative* и *Applicative* для аппликатива и шаблонов классов *is_monad*, *is_same_monad* и *Monad* для монады. Для аппликатива в специализации шаблона класса *Applicative* нужно определить статические функции *pure* и *ap*, а также метафункцию *ap_result_type*, вычисляющую тип результата функции *ap*. Для монады в специализации шаблона класса *Monad* нужно определить статические функции *mreturn* и *mbind*, а также метафункцию *mbind_result_type*, вычисляющую тип результата функции *mbind*.

5. ПОЛУГРУППЫ, МОНОИДЫ

Полугруппа (Semigroup) в общей алгебре – множество с заданной на нем ассоциативной бинарной операцией. В языке Haskell эта операция задается оператором (⟨ ⟩), а в библиотеке funcprog – оператором %. Моноид – это полугруппа с единичным элементом. Единичный элемент моноида возвращается функцией *mempty*. Аналогично функторам и монадам, для того, чтобы некоторый класс сделать полугруппой и моноидом, нужно создать специализации шаблонов классов *is_semigroup*, *is_same_semigroup* и *Semigroup* для полугруппы и шаблонов классов *is_monoid*, *is_same_monoid* и *Monoid* для моноида. Для полугруппы в специализации класса *Semigroup* нужно создать статические функции *semigroup_op* и *stimes*, а для моноида – *mempty* и *mappend*.

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

6. FOLDABLE И TRAVERSABLE

6.1. Класс Foldable

Класс *Foldable* является обобщением сверточного функционала, первоначально определенного для списков, на произвольные типы данных, поддерживающих свертку. Основные сверточные операции – это правая и левая свертки (*foldr* и *foldl*). Эти функции принимают в качестве параметров функцию, по которой делается свертка, начальное значение и свертываемый объект (реализующий класс *Foldable*). Если тип данных, хранящихся в свертываемом объекте, является моноидом, то для такого объекта в классе *Foldable* определена также функция *fold*, принимающая единственный параметр – свертываемый объект. Этот объект сворачивается в моноид, например, список списков свернется в один список, включающий все объекты из всех списков. В качестве начального значения при свертке берется единичный элемент моноида (возвращаемый функцией *mempty*), а в качестве сверточной функции – функция *mappend*. Так как моноидная операция ассоциативна, то правая и левая свертки дают одинаковый результат.

Еще одна важная функция класса *Foldable* – *foldMap*. Ее 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 свертываемом объекте и возвращающую моноид) и свертываемый объект и возвращающую моноид. Как и функция *fold*, она делает свертку объекта в моноид. Значения, хранящиеся в свертываемом 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: ($idx = x$), или, на C++:

```
template<typename T>
T id (T const& value ) { return value ; }
```

В последнем определении функции *fold* использовалась упоминавшаяся выше η-редукция. Полное определение выглядит, естественно, так: $foldx = foldM\ ap\ id\ x$. Но в языке Haskell параметры, стоящие в самом конце левой и правой частей определения функции, можно опускать, если они совпадают. Библиотека funcprog, к сожалению, η-редукцию не поддерживает, поэтому определение функции fold в библиотеке выглядит так:

```
template<typename F>
typename F::value_type fmap (F const& x ) {
 return foldMap (
   _(_id<typename F::value_type>) , x ) ;
}
```

Смотрится более громоздко, но видно, что написано в принципе то же самое, просто типы приходится указывать явно, что удлиняет запись.

Функция *foldMap* похожа на функцию *fmap* (не случайно 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* – свертывая значения в моноид с помощью функции *mappend* (оператора $\langle\ \rangle$).

6.2. Класс Traversable

Класс *Traversable* играет для аппликативов ту же роль, что класс *Foldable* для моноидов. Основная функция класса *Traversable* – *traverse*. Вот ее 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 функтора запрещен). 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* создает 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 ее прототип и определение на языка Haskell:

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

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

7. READER, WRITER И STATE

7.1. 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 или хранить ее в глобальных переменных, оба подхода часто оказываются неудобными. Монада *Reader* позволяет решить эту проблему, предоставляя общий для всех контекст, в котором любая функция может запросить информацию. Тип хранимого значения передается 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 на языке Haskell:

```
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"). Вот как выглядит та же программа на C++:

```
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!") ;
```

Как обычно, на C++ получилось несколько длиннее, но видно, что делается все то же самое.

7.2. 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 функции класса *Writer*. *Writer* выводит информацию в любой моноид, например, в строку (тип моноида передается 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 $
  half8 >>= half
```

Функция *main* печатает: "I just halved 8!I just halved 4!". Вот тот же самый пример на C++:

```
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!") ;
```

7.3. Монада State

Монада *State* похожа на монаду *Reader* в том, что она также хранит некоторое значение (состояние), его тип передается 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.

8. ТРАНСФОРМЕРЫ МОНАД

В реальных вычислениях обычно бывает нужно использовать не какую-то одну монаду, а одновременно несколько. Например, все три описанные в предыдущей главе монады – явные кандидаты на одновременное использование. Есть еще монады

Maybe и *Either*, обрабатывающие ошибки вычислений, и другие монады. Поэтому встает 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. В настоящий момент в библиотеке funcprog реализованы трансформеры монад *IdentityT*, *MaybeT*, *ReaderT*, *WriterT* и *StateT*. А сами классы *Reader*, *Writer* и *State* реализованы, как и в языке Haskell, как "неподвижные точки" (fixed points) соответствующих трансформеров монад, а именно, каждый из этих классов является соответствующим трансформером монады *Identity*, не меняющей значение. 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)
```

То же самое на языке C++ выглядит так:

```
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 изменении порядка монад *Reader* и *Writer* меняются местами две строки:

```
   divide<_ReaderT<int, _Writer<String>>>
   (8).run(2).run().run()) ==
```

9. КОМБИНАТОРНЫЙ ПАРСЕР PARSEC

В языке Haskell *Parsec* [11] – это простой, хорошо документированный и быстрый парсер. Он определяется как неподвижная точка трансформера монад *ParsecT*. Библиотека *Parsec* представляет из себя набор достаточно простых парсеров с широкими возможностями их комбинирования. Сам класс *ParsecT* является функтором, аппликативом, монадой и моноидом. Соответственно, все возможности, которые предлагают эти классы, для парсеров также доступны. Парсер *Parsec* достаточно полно реализован в библиотеке funcprog.

Главная проблема при его реализации была следующая. Функция *unParser* в определении класса *ParsecT* определена как полиморфная:

```
unParser :: forall b.
      State su
  ... -- Текст опущен
-> m b
```

В терминах языка C++ это означает, что *unParser* – это шаблон функции с параметром шаблона *b*. А такую функцию (*function_t*) описать невозможно. Метод в классе с таким определением нельзя сделать виртуальным. Единственный выход – сделать в классе метод с таким определением и передать тип класса, реализующий этот метод как дополнительный параметр шаблона класса *ParsecT*. Это создало дополнительные трудности при реализации парсера, но в конце концов все получилось. И калькулятор заработал. Опишем вкратце эти трудности.

Класс *ParsecT* в языке Haskell имеет 4 параметра: ParsecT⟨S, U, M, A⟩, где *S* – тип входного потока, *U* – пользовательские данные, *M* – монада и *A* – тип выходных данных. Таким образом, два разных (имеющих разную реализацию функции *unParser*) парсера, у которых эти 4 параметра совпадают, имеют один тип, следовательно, некоторая функция, которая должна вернуть парсер, может, в зависимости от некоторого условия, вернуть один из двух этих парсеров (if cond then p1 else p2). В библиотеке funcprog по указанной выше причине к классу *ParsecT* был добавлен еще один параметр *P*, отвечающий как раз за реализацию метода *unParser*. Таким образом, в библиотеке funcprog разные парсеры всегда имеют разные типы и написать подобную конструкцию оказывается невозможным. Если в исходном тексте на языке Haskell встречается такая конструкция, то такую функцию перенести на C++ не удается. К счастью, такие случаи встречаются нечасто.

10. ЗАКЛЮЧЕНИЕ

Функциональное программирование – мощный инструмент при реализации логически сложных алгоритмов. Изящность и красота языков функционального программирования, таких, как Haskell, основаны на возможности комбинирования функций, когда сложное строится из простого. Язык C++, хотя и содержит в себе элементы функционального программирования (например, лямбда-выражения), в полной мере такими возможностями не обладает. Но опыт написания библиотеки funcprog показал, что язык C++ достаточно мощный для того, чтобы средствами библиотеки шаблонов функций и классов реализовать многие из возможностей языка Haskell. В частности, удалось достаточно полно реализовать парсер *Parsec*.

Функциональный стиль программирования позволяет упростить запись логически сложных алгоритмов за счет того, что такой стиль позволяет “поднять” запись алгоритмов на более высокий логический уровень, что делает эту запись более понятной и одновременно менее громоздкой. Функциональное программирование широко используется во многих областях программирования, в частности, для обработки текстов (например, при компиляции). Данная библиотека ставит цель привнести этот стиль программирования в численные методы, в частности, в планах – встроить библиотеку funcprog в разработанный автором сеточно-операторный подход к программированию (см. [12, 13]), что позволит создавать сложные операторы путем комбинирования более простых, легко “превращать” обычные функции в сеточные (свойство функторов) и т.д. Исследование влияния применения этого стиля на быстродействие (что является чрезвычайно важным в численных методах) пока полноценно не проводолись. Ясно, что замедление будет (привнесение нового уровня абстракции к этому приводит неизбежно), но важным является не сам этот факт, а численные оценки. Замедление в разы, конечно же, неприемлемо, но автор надеется, что это будут скорее проценты, и плата за несомненное удобство не будет слишком большой.

СПИСОК ЛИТЕРАТУРЫ

1. Haskell language. https://www.haskell.org/
2. The Scala Programming Language. https://www.scala-lang.org/
3. Standard ML of New Jersey. http://smlnj.org/
4. Visual F#. https://msdn.microsoft.com/ru-ru/visualfsharpdocs/conceptual/visual-fsharp
5. FC++: Functional Programming in C++. https://yanniss.github.io/fc++
6. C++ template library for fans of functional programming. https://github.com/beark/ftl
7. *David Abrahams*, *Aleksey Gurtovoy*. C++ Template Metaprogramming. Addison-Wesley. 2004.

8. *Краснов М.М.* Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им. М.В. Келдыша, 2017. 84 с. DOI: http://keldysh.ru/e-biblio/krasnov https://doi.org/10.20948/mono-2017-krasnov

9. *Краснов М.М., Ладонкина М.Е.* Разрывный метод Галеркина на трехмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++. Программирование, 2017 г., № 3, стр. 41–53.https://elibrary.ru/download/elibrary_29207399_42944924.pdf

10. Boost C++ Libraries. https://www.boost.org/

11. parsec: Monadic parser combinators. http://hackage.haskell.org/package/parsec

12. *Краснов М.М.* Операторная библиотека для решения трехмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA. Математическое моделирование, 2015, т. 27, № 3, стр. 109–120. http://www.mathnet.ru/links/38633e7a627ab2ce1527ae4a092be72f/mm3585.pdf

13. *Краснов М.М.* Кандидатская диссертация “Сеточно-операторный подход к программированию задач математической физики”. Автореферат. http://keldysh.ru/council/1/2017-krasnov/avtoref.pdf