МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
Прикладная математика и информатика

Кафедра информатики и вычислительного эксперимента

МЕТАПРОГРАММНАЯ РЕАЛИЗАЦИЯ БИБЛИОТЕКИ АРИФМЕТИКИ В ПРОСТЫХ КОНЕЧНЫХ ПОЛЯХ И ИХ РАСШИРЕНИЯХ

Выпускная квалификационная работа
на степень бакалавра

студентки 4 курса
А. С. Мышко

Научный руководитель:
асс. каф. ИВЭ А. М. Пеленицын

Ростов-на-Дону
2015

Содержание

Введение 3

1. Метапрограммирование 3
   1.1. Общая характеристика 3
      1.1.1. Генерация кода 4
      1.1.2. Самомодифицирующийся код 6
   1.2. Метапрограммирование на шаблонах C++ 8

2. Арифметика в конечных полях и их расширениях 11

3. Реализация 12
   3.1. Арифметика в простых полях 12
   3.2. Проверка на простоту 14
   3.3. Многочлены 18

Заключение 22

Список литературы 23

Введение

В последние десятилетия в связи с потребностями теории кодирования и криптографии большое значение имеет эффективная реализация а также высокая повторная используемость и гибкость программного кода, представляющего понятия теории чисел. В настоящее время язык C++ является одним из основных языков реализации задач, в том числе в этой области. В связи с развитием языка и разработкой новых его стандартов возникает вопрос применения новых методов для создания переносимых программ, ориентированных на современные концепции C++-дизайна. Одной из таких концепций является метапрограммирование.

1. Метапрограммирование

1.1. Общая характеристика

Метапрограммирование – это парадигма программирования, основанная на изменении поведения или структуры кода информационной системы в зависимости от данных, действий пользователя или взаимодействия с другими системами. Техника метапрограммирования подразумевает создание программ, порождающих другие программы как результат своей работы [1], либо изменяющих самих себя во время выполнения (самомодифицирующийся код).

Как и большинство других методов программирования, метапрограммирование используется для достижения больших функциональных возможностей ценой меньших затрат, которые измеряются объемом кода, временем выполнения, усилиями на сопровождение и т.п. Характерной особенностью метапрограммирования является то, что определенная часть необходимых вычислений выполняется на этапе трансляции программы. Его применение часто объясняется повышением производительности (вычисление, которое проис-

ходит во время трансляции, легче оптимизировать) или упрощением структуры кода (как правило, метапрограмма короче, чем программа, генерируемая во время её работы), а зачастую – обеими причинами.

1.1.1. Генерация кода

При этом подходе вместо кода, который производит некоторые действия после компиляции на этапе выполнения программы, создается код, генерирующий другой код.

Рассмотрим пример на языке C:

Листинг 1.1. Объявление макроса в C

```
#define MIN(x,y) (((x) > (y)) ? (x) : (y))
```

Макросы языка C могут быть рассмотрены как метапрограммные функции (метафункции), которые, принимая некоторые параметры, генерируют код C.

Листинг 1.2. Использование макроса

```
int main() {
    int a = 7;
    int b = 42;
    int c = MIN(a,b);
}
```

При использовании макроса (листинг 1.2) препроцессор C проводит его синтаксический анализ, интерпретирует аргументы и возвращает код `(((a)> (b))? (a): (b))`. Таким образом, результирующим кодом становится:

Листинг 1.3. Генерация кода

```
int main(){
    int a = 7;
    int b = 42;
    int c = (((a) > (b)) ? (a) : (b));
}
```

Кроме текстовых макросов C/C++ для кодогенерации могут использоваться синтаксические макросы (Lisp, Nemerle, Scala). В качестве примера рассмотрим макросы в языке Lisp.

Описание макроса в Lisp выглядит как описание функции и включает в себя его имя, список имен аргументов и тело. Тело макроса возвращает форму, которая должна быть вычислена. От функций макросы отличаются способом вычисления, которое проходит в два этапа: получение нового выражения, называемое развёрткой макроса (macroexpansion), и вычисление этого выражения. Развёртка макроса порождает код, который на этапе исполнения выполняется так, как если бы он был непосредственно написан в месте вызова. Другим отличием макросов от обычных функций является то, что их аргументы по умолчанию не вычисляются.

Для примера определим макрос и функцию с одинаковыми телами:

Листинг 1.4. Объявление функции и макроса в Lisp

```
(defun foo1 (x)
  (list 'exp x))

(defmacro foo2 (x)
  (list 'exp x))
```

При их вычислении становится видно, что результатом вычисления функции является новый список, а результатом вычисления макроса - число:

```
> (foo1 1)
(EXP 1)
> (foo2 1)
2.7182817
```

Язык Lisp предоставляет широкие возможности для метапрограммирования, так как устроен таким образом, что данные и код в нём — одно и то же. Объявления и вызовы функций, объявление и развёртка макросов в Lisp представляют собой просто списки (возможно, вложенные друг в друга). Это делает доступным весь функционал языка на этапе компиляции.

В случаях, когда языковых средств недостаточно для реализации функциональной парадигмы, могут применяться предметно-ориентированные языки (domain-specific languages). Довольно распространённый пример их использования задействует обобщенное метапрограммирование: генераторы синтаксических и лексических анализаторов (lex, yacc) позволяют описать язык с помощью регулярных выражений и контекстно-свободных грамматик и встраивать сложные алгоритмы, необходимые для эффективного анализа языка. Язык генератора составляется таким образом, чтобы реализовывать правила парадигмы или необходимые специальные функции автоматически или с минимальными усилиями со стороны программиста. Фактически, это более высокоуровневый язык программирования, а генератор — не что иное, как транслятор. Как правило, генераторы пишутся для создания специализированных программ, в которых очень значительная часть стереотипна, либо для реализации сложных парадигм.

1.1.2. Самомодифицирующийся код

Основными методами реализации данного вида метапрограммирования являются интроспекция и интерпретация произвольного кода.

Интроспекция (type introspection) – это способность языка программирования определять структуру, свойства и тип объектов непосредственно в процессе выполнения программы. При этом внутренние структуры языка представляются в виде переменных встроенных типов с возможностью доступа к ним из программы.

Листинг 1.5. Интроспекция в C#

```
class MyClass {
   public static void Main(string[] arg) {
      try { // Get the type of a specified class.
         Type myType1 = Type.GetType("System.Int32");
         Console.WriteLine("The full name is {0}.",
                           myType1.FullName);
      }
   }
}
```

Языки программирования, в которых реализована интроспекция: PHP, C++, Java, C#, Delphi, Perl, Ruby, Smalltalk, JavaScript, ObjectiveC, Python.

Интерпретация произвольного кода, представленного в виде строки, существует естественным образом во множестве интерпретируемых языков, например, `eval()` в PHP:

Листинг 1.6. Интерпретация произвольного кода в PHP

```
<?
php $string = 'cup';
$name = 'coffee';
$str = 'This is a $string with my $name in it.<br>';
echo $str;
eval ("\$str = \"$str\";");
echo $str;
?>
```

Результат работы:

```
This is a $string with my $name in it.
This is a cup with my coffee in it.
```

1.2. Метапрограммирование на шаблонах C++

Первый пример метапрограммы на языке C++ был найден практически случайно в 1994 году [2], [3] и демонстрировал, что механизм инстанцирования шаблонов можно использовать как минималистичный рекурсивный язык, применимый для метапрограммирования. Программа реализовывала поиск простых чисел до некоторого заданного значения, выдавая их в виде последовательности сообщений об ошибках. Вычисления такого вида, проводимые с использованием шаблонов, принято называть метапрограммированием на шаблонах [4].

При таком методе метапрограммирования шаблоны используются для генерации типов данных во время трансляции. Типы данных могут быть определены рекурсивно, а процесс их вычисления может реализовывать произвольный алгоритм, поскольку данная технология является тьюринг-полной [5]. Метапрограммирование с использованием шаблонов C++ использует парадигму функционального программирования, в котором процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Основными чертами функциональных языков программирования, используемыми при метапрограммировании, являются:

- чистые функции — функции, не имеющие побочных эффектов ввода-вывода и памяти, зависящие только от своих параметров и возвращающие только свой результат;
- использование рекурсии. В функциональных языках циклы подразумевают реализацию в виде рекурсии: рекурсивные функции

вызывают сами себя, позволяя операции выполняться снова и снова;

- функции высших порядков — функции, принимающие в качестве аргументов и возвращающие другие функции. Примерами таких функций в математическом анализе являются производная и интеграл.

В качестве примера рассмотрим алгоритм, реализующий возведения числа 3 в заданную степень n. Как было сказано ранее, метапрограммирование на шаблонах C++ предполагает использование рекурсивного механизма их инстанцирования. В первом шаблоне реализовано общее рекурсивное правило; чтобы обеспечить выход из рекурсии, используется полная специализация шаблона для $n = 0$.

Листинг 1.7. Возведение числа 3 в степень n

```
template<int N>
struct Pow3 {
        enum { result = 3 * Pow3<N-1>::result };
};

template<>
struct Pow3<0> {
        enum { result = 1 };
};
```

В устаревших версиях компиляторов C++ значения перечислимого типа (**enum**) были единственной возможностью использовать «истинные» константы (constant-expressions), в объявлении класса или структуры. Позже в процессе стандартизации языка C++ появилась концепция внутриклассовых статических инициализирующих констант (спецификаторы **static const**), которые можно использовать в качестве альтернативы перечисляемому типу. Однако этот способ обладает недостатком.

Пусть объявлена функция

```
void foo(int const&);
```

и в неё передаётся результат выполнения описанной в листинге 1.7 метапрограммы:

```
foo(Pow3<7>::result);
```

В этом случае функции должен быть передан адрес переменной `Pow3<7>::result`, то есть, она должна быть помещена в определённую ячейку памяти при инстанцировании. В результате вычисление выходит за пределы этапа компиляции. Переменные же перечислимого типа не имеют адреса, и при их передаче «по ссылке» статическая память не используется. Ввиду этого их применение остаётся предпочтительным при метапрограммировании [4].

Важным примером метафункций являются числовые метафункции (numerical metafunctions), возвращающие тип-обёртку над значением (возможно, саму себя). Тип-обёртка хранит информацию о заданном значении в константном статическом поле `value`.

Листинг 1.8. Тип-обёртка для значения 5

```
struct five {
        static int const value = 5;
        typedef int value_type;
        //...more declarations...
};
```

Этот дополнительный уровень абстракции может показаться неудобным, но требование для всех метафункций принимать и возвращать типы делает их более единообразными, более полиморфными и более совместимыми [6].

2. Арифметика в конечных полях и их расширениях

Начала теории конечных полей восходят к XVII и XVIII векам и связаны с именами выдающихся математиков Ферма, Эйлера, Лагранжа и Лежандра, которые внесли вклад в структурную теорию простых конечных полей. Общая теория конечных полей началась с работ Гаусса и Галуа, и до последней четверти 20-го века изучалась и находила применение только в алгебре и теории чисел [7]. Однако в последние десятилетия в связи с потребностями теории кодирования и криптографии активно изучаются прикладные вопросы. Ввиду этого большое значение имеет эффективная реализация а также высокая повторная используемость и гибкость программного кода, представляющего эти понятия.

Поле — множество элементов, замкнутое по отношению к двум заданным в нем бинарным операциям: сложению ($+$) и умножению ($\cdot$). Для полей выполняются следующие аксиомы:

1. Для введённых операций выполняются правила ассоциативности $a + (b + c) = (a + b) + c$, $a(bc) = (ab)c$, коммутативности $a + b = b + a$, $ab = ba$ и дистрибутивности $(a + b)c = ac + bc$.

2. Поле всегда содержит мультипликативную единицу 1 и аддитивную единицу 0, такие, что $a+0 = a$ и $a \cdot 1 = a$ для любого элемента поля.

3. Для любого элемента a в поле существует обратный элемент по сложению $-a$ и обратный элемент по умножению a^{-1} (при $a \neq 0$), такие, что $a + (-a) = 0$, $a(a^{-1}) = 1$. Наличие обратных элементов позволяет наряду с операциями сложения и умножения выполнять также вычитание и деление: $a - b = a + (-b)$, $a/b = a(b^{-1})$.

Конечное поле или поле Галуа — поле, состоящее из конечного числа элементов. Конечное поле обычно обозначается $\mathbb{F}_q$, где q —

число элементов поля (мощность). С точностью до изоморфизма конечное поле полностью определяется его мощностью, которая всегда является степенью какого-либо простого числа. Специальным случаем являются простые поля – кольца вычетов $\mathbb{Z}_p$ по модулю простого числа p.

3. Реализация

Арифметика в конечных полях реализована в ряде C++-библиотек, например, библиотеке NTL [8], [9]. Метапрограммный подход позволяет использовать несколько различных полей, избегая дублирования кода, и обеспечивает лучший контроль ошибок, делая, в частности, использование нескольких полей более безопасным.

3.1. Арифметика в простых полях

Элемент x простого поля $\mathbb{F}_p$ представлен структурой FFE с параметрами шаблона `x` и `N`, определяющими значение элемента и мощность поля соответственно. Как было показано, метапрограммирование на шаблонах C++ чаще всего связано с вычислением типов; возвращение типа осуществляется typedef-объявлением и стандартной обёрткой `int_` для типа **`int`**. Для корректного объявления необходимых статических констант внутри структуры используется перечисление (**`enum`**) по причинам, указанным в 1.2.

Листинг 3.1. Структура `FFE`, реализующая элемент простого поля

```
template <int N, int x>
struct FFE {
    enum { value = x % N };
    typedef int_<value> type;
};
```

Арифметика в простых полях $\mathbb{Z}_p$ представляет собой целочисленное сложение и умножение по модулю p. Операции реализованы соответствующими метафункциями, для примера приведена функция `FFE_sum` (листинг 3.2).

Листинг 3.2. Метафункция `FFE_sum`, осуществляющая сложение двух элементов поля

```
template <int N, typename P1, typename P2>
struct FFE_sum {
};
template <int N, int x, int y>
struct FFE_sum<N, FFE<N, x>, FFE<N, y>> {
    typedef FFE<N, x + y> type;
};
```

При объявлении элемента поля его значение может быть задано любым числом из $\mathbb{Z}$, которое будет приведено по модулю внутри структуры, что обеспечит корректность вычислений. Следить за приведением значений внутри метафункций, реализующих сложение и умножение, не требуется, так как оно осуществляется внутри структуры `FFE`, а также на основании следующих свойств этих операций для $a, b \in \mathbb{Z}$:

$$(a + b) \mod n = ((a \mod n) + (b \mod n)) \mod n$$

$$(a * b) \mod n = ((a \mod n) * (b \mod n)) \mod n$$

При вызове функции `FFE_sum` вычисления производятся на этапе компиляции, что подтверждается ассемблерным листингом (листинг 3.4).

Листинг 3.3. Пример сложения элементов простого поля

```
int main(){
    typedef FFE<7, 5> a;
    typedef FFE<7, 4> b;
    return FFE_sum<7, a, b>::type::value;
}
```

Листинг 3.4. Сложение элементов простого поля: ассемблерный листинг

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $2, %eax
        popq    %rbp
        ret
```

Как видно, при исполнении программы вместо вызова функции в ячейку памяти передается результат, возвращаемый метафункцией.

3.2. Проверка на простоту

Кольцо вычетов по модулю составного числа не является полем, поэтому для обеспечения корректности вычислений необходимо организовать проверку на простоту модуля поля, в котором создается элемент. Для осуществления проверки используется метод перебора делителей.

Так как возвращаемым значением метафункции является тип, реализация в соответствии со стандартом C++03 предполагает описание собственных типов `true_type` и `false_type` для значений $true$ и $false$.

Листинг 3.5. Описание типов для логических значений

```
struct false_type {
  typedef false_type type;
  enum { value = 0 };
};
```

Такого рода обёртки над статическими константами реализованы в библиотеке Boost [10], а позже в стандарте C++11 (<type_traits>).

Условный оператор также требует собственного описания в контексте типов (листинг 3.6).

Листинг 3.6. Описание типа для условного оператора

```
template<bool condition, class T, class U>
struct if_{
  typedef U type;
};
template <class T, class U>
struct if_<true, T, U>{
  typedef T type;
};
```

Поскольку метапрограммирование с использованием шаблонов предполагает парадигму функционального программирования, проверка на простоту реализуется рекурсивно с использованием вспомогательной функции `is_prime_impl`, которая осуществляет все вычисления. Основная функция `is_prime` вызывает вспомогательную, запуская рекурсию. Кроме того, для неё описываются особые случаи вызова для чисел 0 и 1, опущенные в листинге 3.7 для краткости. Полное описание приведено на стр. 17 (листинг 3.8).

Листинг 3.7. Проверка на простоту по стандарту C++03

```
template<size_t N, size_t c>
struct is_prime_impl{
  typedef typename if_<(c*c > N),
                       true_type,
                       typename if_<(N % c == 0), false_type,
                                    is_prime_impl<N, c+1>
                                   >::type
                      >::type type;
  enum { value = type::value };
};
template<size_t N>
struct is_prime{
  enum { value = is_prime_impl<N, 2>::type::value };
};
```

Стандарт C++11 позволяет сократить код алгоритма при использовании спецификатора `constexpr` за счёт отсутствия необходимости описывать собственные типы для условного оператора и логических переменных [11]. Как и в версии с применением шаблонов, алгоритм реализуется рекурсивно с использованием вспомогательной функции `is_prime_recursive` (листинг 3.9). Очевидно, такое описание алгоритма короче и понятнее, чем приведенное в листинге 3.8 (стр. 17).

Листинг 3.9. Проверка на простоту с использованием `constexpr`

```
constexpr bool is_prime_recursive(size_t number, size_t c){
  return (c*c > number) ? true :
           (number % c == 0) ? false :
              is_prime_recursive(number, c+1);
}

constexpr bool is_prime_func(size_t number){
  return (number <= 1) ? false : is_prime_recursive(number, 2);
}
```

Листинг 3.8. Проверка на простоту по стандарту C++03

```
struct false_type {
  typedef false_type type;
  enum { value = 0 };
};

struct true_type {
  typedef true_type type;
  enum { value = 1 };
};

template<bool condition, class T, class U>
struct if_ {
  typedef U type;
};

template <class T, class U>
struct if_<true, T, U> {
  typedef T type;
};

template<size_t N, size_t c>
struct is_prime_impl {
  typedef typename if_<(c*c > N),
                       true_type,
                       typename if_<(N % c == 0),
                                    false_type,
                                    is_prime_impl<N, c+1> >::type
                                        >::type type;
  enum { value = type::value };
};

template<size_t N>
struct is_prime {
  enum { value = is_prime_impl<N, 2>::type::value };
};

template <>
struct is_prime<0> {
  enum { value = 0 };
};

template <>
struct is_prime<1> {
  enum { value = 0 };
};
```

Если передаваемый при использовании `constexpr` параметр является интегральной константой, результат может быть вычислен на стадии компиляции, если же параметр вычисляется во время исполнения, результат также будет вычислен во время выполнения программы без вывода сообщения об ошибке. При этом если возвращаемое `constexpr`-функцией значение нигде не используется таким образом, что его необходимо вычислить во время компиляции, то оно не будет вычислено. Таким образом, одна и та же функция может быть использована в двух контекстах, что, однако, усложняет контроль за тем, на каком этапе выполняется вычисление. Например, использование результата проверки на простоту в качестве программного утверждения в функции `static_assert` гарантирует его вычисление во время компиляции (листинг 3.10).

Листинг 3.10. Пример проверки на простоту на разных этапах

```
int main(){
   // Computed at compile-time
   static_assert(is_prime_func(7), "...");
   int i = 11;
   int j = is_prime_func(i)); // Computed at run-time
}
```

3.3. Многочлены

Многочлены над полем $\mathbb{F}_p$ естественным образом представимы в виде массивов элементов типа `FFE`. Однако их использование требует явного указания их значений, так как значение элемента входит в число параметров шаблона для `FFE`. Ввиду этого для реализации арифметики многочленов создан класс `FFE_x` с единственным шаблонным параметром `N`, представляющим мощность поля, и полем `value`, обозначающим значение элемента. Арифметика для `FFE_x` реализована перегрузкой операторов `+=`, `*=` (внутри класса) и `+`, `*` (вне класса). Так-

же определён оператор потокового вывода <<. Для предотвращения возможности создания поля, мощность которого является составным числом, в конструкторе класса используется `static_assert` с проверкой параметра `N` на простоту (листинг 3.11).

Листинг 3.11. Класс `FFE_x`

```
template<long N>
class FFE_x {
private:
    int value;

public:
    FFE_x(int value) {
        static_assert(is_prime_func(N), "Field order must be a
           prime number");
        FFE_x<N>::value = value % N;
    }
};
```

Стандарт C++11 позволяет создавать шаблонные классы и функции, параметризованные переменным числом параметров (variadic templates) [12], что использовано при реализации структуры `Poly`, определяющей многочлен над простым полем как массив из переменного числа элементов `FFE_x` (листинг 3.12).

Листинг 3.12. Структура `Poly`, реализующая многочлен над простым полем

```
template <int N, int ... Nums>
struct Poly {
    static FFE_x<N> elements[sizeof ...(Nums)];
    enum { elements_count = sizeof ...(Nums) };
};

template<int N, int... Nums>
FFE_x<N> Poly<N, Nums ...>::elements[] = { Nums ... };
```

Такого рода параметры шаблонов называются «пакетами параметров» («parameters packs») и не могут быть использованы везде, где используются обычные параметры. Их использование допустимо в следующих контекстах:

- В перечислении базовых классов шаблона (base-specifier-list);
- В списке инициализации членов данных в конструкторе (mem-initializer-list);
- В списках инициализации (initializer-list);
- В списках параметров других шаблонов (template-argument-list);
- В спецификации исключений (dynamic-exception-specification);
- В списке атрибутов (attribute-list);
- В списке захвата лямбда-функции (capture-list).

Использование пакета параметров называется «раскрытием пакета» (pack expansion). При раскрытии элементы пакета интерпретируются в соответствии с контекстом его использования.

Сложение многочленов осуществляется согласно функциональной парадигме программирования: с использованием частичной специализации шаблона пакет параметров разделяется на «голову» и «хвост» (предполагается, что коэффициенты при степенях хранятся в порядке их возрастания). Вычисление выполняется рекурсивно: результат сложения коэффициентов при младших степенях объединяется с результатом вызова функции для оставшейся части параметров при помощи метафункции `concat` (листинг 3.13). Таким образом, задача сводится к сложению многочлена нулевой степени с другим многочленом, возможно, также нулевой степени; эти случаи описаны частичной специализацией метафункции `sum` и опущены в листинге для краткости.

Метафункция `concat` возвращает конкатенацию двух многочленов и не подразумевает самостоятельное использование в библиотеке. Подобная операция не несёт алгебраического смысла и используется как вспомогательная.

Листинг 3.13. Сложение многочленов

```
template<int N, typename P1, typename P2>
struct sum {
};

template<int N, int C1, int C2, int... Nums1, int... Nums2>
struct sum<N, Poly<N, C1, Nums1 ...>, Poly<N, C2, Nums2 ...>> {
typedef typename concat<N,
                        Poly<N, C1+C2>,
                        typename sum<N,
                                     Poly<N, Nums1 ...>,
                                     Poly<N, Nums2 ...>>::type
                        >::type type;
};
```

Умножение многочленов реализовано рекурсивно схожим со сложением образом. Частичной специализацией шаблона описаны случаи, когда одним из множителей является многочлен нулевой степени, к которым сводится умножение двух произвольных многочленов. Результатом метафункции является сумма произведения одного из многочленов на коэффициент при нулевой степени («голову» пакета параметров) второго и результата выполнения операции для оставшихся элементов пакета («хвоста») с умножением на моном первой степени (листинг 3.14). То есть, алгоритм реализует «раскрытие скобок» при умножении многочленов.

Умножение на моном представляет собой сдвиг степеней многочлена и осуществляется конкатенацией с многочленом, имеющем нулевые коэффициенты. При этом происходит добавление нулевых

коэффициентов в начало пакета параметров, что и реализует необходимый сдвиг коэффициентов при степенях.

Листинг 3.14. Умножение многочленов

```
template <int N, typename P1, typename P2>
struct mult {
};

template <int N, int C1, int... Nums1, int C2, int... Nums2>
struct mult<N, Poly<N, C1, Nums1...>, Poly<N, C2, Nums2...>> {
typedef typename sum <N,
                      Poly<N, C1, Nums1...>,
                      typename concat<N,
                                      Poly<N, 0>,
                                      typename mult<N, Poly<N, C1
                                         , Nums1...>, Poly<N,
                                         Nums2...>>::type
                                      >::type
                      >::type type;
};
```

Заключение

Метапрограммирование с использованием шаблонов C++ несёт в себе большой потенциал для написания гибких быстродействующих программ, а изученный в ходе выполнения работы стандарт C++11 ещё больше расширяет его возможности.

В работе были рассмотрены принципы шаблонного метапрограммирования на языке C++ и разработана метапрограммная реализация библиотеки арифметики в конечных полях, включающая арифметику в простых полях и многочленов над ними.

Полный исходный код библиотеки доступен по адресу [13].

Список литературы

1. *Бартлетт Д.* Искусство метапрограммирования, Часть 1: Введение в метапрограммирование. — URL: http://www.ibm.com/developerworks/ru/library/l-metaprog1.
2. *Unruh E.* Prime number computation: тех. отч. / ANSI X3J16-94-0075/ISO WG21-462. — 1994.
3. *Veldhuizen T.* Using C++ Template Metaprograms // C++ Report. — 1995. — № 4. — С. 3643.
4. *Vandervoode D.*, *Josuttis N.* C++ Templates: The Complete Guide. — Boston, MA : Addison-Wesley, 2002.
5. *Veldhuizen T. L.* C++ Templates are Turing Complete: тех. отч. — 2003.
6. *Abrahams D.*, *Gurtovoy A.* C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. — Addison Wesley Professional, 2004.
7. *Лидл Р.*, *Нидеррайтер Г.* Конечные поля. — М. : Мир, 1988.
8. *Shoup V.* NTL: A library for doing number theory. — 1996-2008. — URL: http://www.shoup.net/ntl.
9. *Shoup V.* A Computational Introduction to Number Theory and Algebra. — Cambridge University Press, 2008.
10. BOOST C++ Library. — 1998-2008. — URL: http://www.boost.org.
11. Constexpr specifier. — URL: http://en.cppreference.com/w/cpp/language/constexpr (дата обр. 02.10.2014).
12. C++11 – New features – Variadic templates. — URL: http://www.cplusplus.com/articles/EhvU7k9E/ (дата обр. 28.11.2014).
13. URL: https://github.com/Metida/Finite_Fields.