Математика и её приложения: ЖИМО. 2011. Вып. 1 (8). С. 79 – 84.

УДК 004.424

А. М. Пеленицын[1]

Об использовании одного приёма метапрограммирования

Ключевые слова: C++, обобщённое программирование, метапрограммирование.

В задаче реализации полиномов многих переменных применяется метод рекурсивного инстанцирования шаблонов в языке C++ . Изучаются вопросы обработки структур данных подобного рода.

Keywords: C++, generic programming, metaprogramming.

We show an application of template recursive instantiation approach to the implementation of multivariate polynomials. We discuss challenges in processing of data structures of this kind.

1. Введение

В работе [1], гл. 21, изложен пример работы с рекурсивно определяемыми шаблонными типами в языке C++. Поясним это понятие на примере: для шаблона класса `Bar<T>` можно рассмотреть возможность использования "рекурсивных" инстанций этого шаблона, таких как: `Bar< Bar< Bar<int> > >`. Для такого использования шаблонов нужно определить, какие преимущества при проектировании принесёт данный подход и, если преимущества окажутся значительными, предоставить специальные средства, учитывающие рекурсивное использование шаблона класса. В [1] такой анализ и пример реализации представлен для моделирования типа `Tuple`-кортежа фиксированной длины значений возможно различных типов; было показано, что такой тип можно получить, используя рекурсивное инстанцирование типа, представляющего пару значений произвольных типов. При этом если указанный приём автоматизирован должным образом и приводит (как в данном случае) к формированию новой абстракции, моделирующей определённое понятие предметной области, то можно говорить о применении метапрограммирования: создании программного кода, генерирующего другой программный код, который являются частью решения поставленной задачи.

В настоящей работе рассматривается пример использования рекурсивного инстанцирования шаблонов C++ для моделирования полиномов многих переменных. Полный исходный код изложенного доступен в сети Интернет [2]. Библиотека, созданная на основе изложенных идей, была при-

[1]Южный федеральный университет; E-mail: `apel@sfedu.ru`.

менена в реализации BMS-алгоритма из алгебраической теории помехоустойчивого кодирования [3].

2. Шаблон класса `Polynomial<T>`

Рассмотрим шаблон класса `Polynomial<T>`, который представляет тип полинома, типовый параметр `T` обозначает тип коэффициентов соответствующего множества полиномов. Главная идея реализации полиномов многих переменных состоит в том, что, к примеру, полином от двух переменных с коэффициентами типа `T` неотличим по алгебраическим свойствам от полинома от одной переменной с коэффициентами – полиномами от одной переменной и коэффициентами типа `T` (см., напр., [4], гл. IV, § 1, п. 5); таким объектам соответствует тип `Polynomial< Polynomial<T> >`.

Для удобства использования полиномов многих переменных необходимо автоматизировать порождение инстанций рекурсивных типов, это может быть достигнуто применением известной метапрограммной техникой (см. [1], гл. 17). Определим шаблон класса `MVPolyType<n, T>`, где первый параметр служит для указания количества переменных, а второй задает тип коэффициентов многочленов n переменных так, чтобы следующие две строки кода определяли переменные одинакового типа, представляющего полином от трех переменных с целочисленными коэффициентами.

```
Polynomial< Polynomial< Polynomial<int> > > p;
MVPolyType<3, int>::type q;
```

Стандартный рекурсивный алгоритм метапрограммирования реализуется в данном случае следующим образом.

```
template<int VarCnt, typename Coef>
struct MVPolyType {
    typedef Polynomial< // "recursive call" to MVPolyType:
        typename MVPolyType<VarCnt - 1, Coef>::type >
        type;
};
template<typename Coef>
struct MVPolyType<1, Coef> {
    typedef Polynomial<Coef> type;
};
```

Как видно, рекурсия проводится по количеству переменных полинома, для полинома одной переменной определена специализация шаблона, которая отвечает за остановку рекурсии. Рекурсия по типу, таким образом, не сильно отличается от простейших алгоритмов, использующих рекурсию по данным. На этом примере также видно, как с помощью использования механизма шаблонов на этапе компиляции можно генерировать новые программные сущности – типы многочленов от разного числа переменных

с разными типами коэффициентов, что является одним из выражений метапрограммирования.

Аналогичным образом внутри типа полинома `Polynomial<T>` задается константа `VAR_CNT`, определяющая количество переменных полинома в зависимости от типа `T`, и синоним типа `CoefT` для обозначения фактического типа коэффициентов полинома в случае нескольких переменных. Например,

```
Polynomial<Polynomial<int>>::VAR_CNT
```

равно 2 и

```
Polynomial<Polynomial<int>>::CoefT
```

обозначает `int`.

Для полиномиальных типов перегружены простейшие арифметические операции: сложение полиномов, умножение полинома на скаляр, умножение полинома на моном (в форме `operator<<`) и вычисление полинома в точке. Особенности реализации этих операций представляют особый интерес, и их рассмотрение вынесено в следующий раздел.

3. Обработка рекурсивно заданных типов

В данном разделе рассматриваются особенности обработки рекурсивно заданных типов на примере реализации многочленов многих переменных, представленной в предыдущем разделе. Вначале описан один типичный простой случай такой обработки, затем – более сложный.

Реализация операции умножения полинома на скаляр не требует особо учитывать возможность рекурсивного инстанцирования.

```
Polynomial operator*=(CoefT const & c ) {
    for (typename StorageT::iterator it = data.begin();
            it != data.end(); ++it) {
        (*it) *= c;
    }
    return *this;
}
```

Роль поля шаблона класса `data` понятна, если помнить что полином является особого рода “контейнером для `T`”. Тип поля `data` скрыт за синонимом типа StorageT, который означает один из стандартных контейнеров C++.

Под умножением на скаляр понимается вызов для полинома операции $*$ с объектом, тип которого совпадает с `CoefT`. Непосредственно умножение производится вызовом той же операции для каждого элемента контейнера `data`. Если мы имеем дело с полиномом одной переменной, то `data` должен содержать элементы типа `CoefT`, для которого, таким образом, должна быть определена операция $*$ (тип коэффициентов полинома должен

допускать умножение). В ином случае (`data` содержит элементы другой инстанции `Polynomial`), снова будет вызвана приведенная выше функция, однако изменится тип `Polynomial` – теперь это будет тип полинома от переменных, число которых меньше на единицу. Еще раз подчеркнем, что описанная логика полностью реализована в приведенном выше коде, не требуется, к примеру, использовать специализацию шаблонов, чтобы явно останавливать рекурсию.

Сложение двух полиномов так же, как умножение на скаляр, не требует дополнительных модификаций по сравнению с тем, что можно было бы закодировать, не рассчитывая на использование в контексте рекурсивного определения. Можно сделать вывод о том, что такого рода рекурсивный обход одного (умножение на скаляр) или нескольких (сложение, сравнение двух полиномов на равенство) объектов одного рекурсивно инстанцированного типа может быть довольно удобен. Однако, при работе с полиномами многих переменных возникает задача рекурсивного обхода объектов двух разных типов одновременно (при этом глубина рекурсии разных типов должна быть одинаковой). Примерами решения такой задачи в данном случае являются операции умножения полинома на моном и обращения к коэффициенту полинома от многих переменных по его мультииндексу (иначе: по мультистепени монома, при котором стоит запрашиваемый коэффициент). В обоих случаях требуется осуществлять одновременный обход объекта типа `Polynomial<...Polynomial<T>...>` и объекта типа `Point<N>`, где число `N` совпадает с глубиной вложенности типа полинома (а значит, в соответствии с нашими соглашениями, – с числом переменных полинома).

Рассмотрим реализацию операции обращения к коэффициенту полинома по его (мульти)индексу. В частном случае, при работе с полиномом от одной переменной, обращаться к коэффициенту хотелось бы по обычному целочисленному индексу; по этой причине стоит реализовать операцию `operator[]`, имеющую один параметр типа `int`. Для обращения к коэффициенту полинома от многих переменных предоставляется перегруженная версия `operator[](Point<VAR_CNT>)` (размерность точки должна совпадать с количеством переменных полинома). Обсудим вопрос о том, как должна продвигаться рекурсия.

Пусть задана точка `p` типа `Point<VAR_CNT>`. Для полинома от многих переменных нужно обратиться к `p[0]`-му элементу в контейнере `data` (это полином от переменных, количество которых меньше на одну) и снова вызвать для него `operator[](Point<VAR_CNT>)` с аргументом, представляющим как бы срез точки `p`: точку со всеми элементами `p`, кроме `p[0]`-го. Для этой цели был создан шаблон класса `Slice<Dim, Offset>`, который хранит ссылку на исходную точку размерности `Dim` и поддерживает операцию взятия индекса, причем при обращении к `i`-му элементу по индексу возвращается `Offset+i`-й элемент исходной точки. Используя этот шаблон в реализации `operator[](Point<VAR_CNT>)`, хотелось бы написать что-то наподобие `(data[p[0]])[make_slice(p)]`, где функция `make_slice` создает

срез, исключающий первый элемент точки. При этом в интерфейс полинома нужно добавить `operator[](Slice<Dim, Offset>)`, в котором будет написано примерно то же, что и в исходной версии этой операции, имеющей в качестве аргумента точку.

Реализации рекурсивного обхода указанным выше образом мешают ограничения языка C++. Для остановки рекурсии в шаблонных типах обычно используется специализация шаблона (как в примере с `MVPolyType`). В данном случае нужно было бы создать специализацию `operator[](Slice<Dim, Dim-1>)`. Однако в языке C++ имеется ограничение (см. [1], п. 12.3.3), которое, в частности, запрещает специализировать шаблонные члены шаблонов класса, в то время как именно шаблоном, зависящим от целочисленных параметров `Dim` и `Offset`, является `operator[](Slice<Dim, Offset>)` – член шаблона класса `Polynomial<T>`.

Чтобы обойти указанное ограничение, необходимо ввести дополнительный уровень косвенности: вместо `(data[p[0]])[make_slice(p)]` использовать вызов отдельной свободной функции `apply_subscript(data[p[0]], make_slice(p))`, которая либо продвигает рекурсию, вызывая `operator[](Slice<Dim, Offset>)` с переданным ей срезом, либо (в своей специализации) останавливает рекурсию, вызывая `operator[](int)` с нулевым элементом полученного среза (этот элемент стоит в последней координате исходной точки).

Удобно считать, что при обращении по индексу, выходящему за пределы текущего контейнера, нужно вернуть “нулевое значение”. Такое значение получается с помощью шаблона `CoefficientTraits`, представляющего собой пример шаблона класса характеристик или, в другом переводе, свойств типов (type traits) (см. [1], гл. 15; [7], п. 2.10).

Итоговый код, реализующий операцию обращения по индексу, выглядит следующим образом.

```
template<typename T, typename S, typename Pt>
T applySubscript(S const & el, Pt const & pt) {
    return el[pt]; // call for operator[](Slice<...>)
}
template<typename T, typename S, int Dim> // recursion stop
T applySubscript(S const & el, Slice<Dim, Dim-1> const & pt {
    return el[pt[0]]; // call for operator[](int)
}
template<typename T>
typename Polynomial<T>::CoefT
Polynomial<T>::operator[](Point<VAR_CNT> const & pt) const {
    if (pt[0] < 0 || data.size() <= pt[0])
        return CoefficientTraits<CoefT>::addId();
    else
        return
            applySubscript<Polynomial<T>::CoefT>(data[pt[0]],
                make_slice(pt));    }
```

Первая из приведенных версий `applySubscript`, продвигая рекурсию, вынуждена работать со все новыми инстанциями шаблона `Slice`, но поскольку конкретные значения шаблонных параметров `Slice` не важны для нее, то здесь можно говорить о применении подхода обобщенного программирования: алгоритм сделан независимым от представления данных настолько, насколько это возможно.

Здесь не приводится код `operator[](Slice<...>)`, так как он полностью повторяет `operator[](Point<VAR_CNT>)`, а `operator[](int)` отличается только тем, что вместо вызова `apply_subscript` используется `data[pt]` (`pt` – полученное целое число).

Программный код, осуществляющий обход рекурсивной структуры типа полинома одновременно с итерацией по типу `Point<N>`, является довольно громоздким. Учитывая то, что реализация еще нескольких операций (например, вывода полинома в поток и вычисления полинома для некоторого значения переменных) выполняется аналогично, возникает вопрос, можно ли создать общую обёртку для реализации подобных операций. На данном этапе такую обёртку создать не удалось.

Список литературы

1. *Вандевурд Д., Джосаттис Н.* Шаблоны C++: справочник разработчика. – М.: Вильямс, 2003. – 544 с.
2. *Pelenitsyn A.* Multivariate polynomials for C++. – URL: http://code.google.com/p/cpp-mv-poly/
3. *Пеленицын А. М.* О реализации n-мерного BMS-алгоритма средствами обобщенного программирования // Труды научной школы И. Б. Симоненко. – Ростов н/Д: Изд-во ЮФУ, 2010. – С. 197–203.
4. *Бурбаки Н.* Алгебра (Многочлены и поля. Упорядоченные группы). – М.: Наука, 1965. – 300 с.
5. International Standards Organization: Programming Languages – C++. International Standard ISO/IEC 14882:2011.
6. Programming Language C++ Draft February 2011 (n3242). – URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
7. *Александреску А.* Современное проектирование на C++. – М.: Вильямс, 2002. – 336 с.

Поступила в редакцию 26.11.2011.