Смешанные вычисления в Arageli
ЧЕРНОВОЙ ВАРИАНТ

С. С. Лялин

31 августа 2006 г.

1 Введение

В любой сколько-нибудь значительной программе, написанной с помощью Arageli, используется не один тип данных для представление одной и той же математической структуры. Например, встроенные числовые типы C++ служат примером многообразия типов с которыми приходится работать одновременно: несколько видов целых чисел и несколько видов чисел с плавающей запятой.

Использование нескольких типов в одном случае необходимо из-за экономии памяти, в другом случае из-за необходимости совместимости с чужими программами, и скорости вычислений — в третьем. Словом, есть причины не ограничиваться одним типом, моделирующем одну сущность, а использовать несколько доступных представлений.

Наличие нескольких объектов разных типов подразумевает, что они могут участвовать в арифметических операциях совместно. Например, что нам мешает сложить переменную типа **int** с переменной типа **double**, явно не преобразовывая тип одной из них в тип другой? Ведь и то и другого — числа. Результатом должен быть объект такого типа, который сможет его корректно представить. В случае нашего примера (см. стандарт C++) тип результата будет **double**.

Было бы логично, если в выражениях позволительно было комбинировать объекты разных, но в некотором смысле совместимых типов в качестве операндов одной бинарной операции. Таким способом построенные выражения в языках программирования называются смешанными выражениями, а вычисления, производимые с их помощью, — смешанными вычислениями (mixed computations).

Тому как эта идея распространяется на параметризованные типы библиотеки Arageli и посвещён настоящий документ.

2 Обзор возможностей

Наряду с симметричными операциями, в которых типы аргументов моделируют одну и ту же математическую структуру, например, сложение чисел, векторов и матриц, здесь будут рассматриваться операции с несимметричными типами операндов, например, умножение вектора на число, матрицы на скаляр. Причём, с позиции языка реализации типы не обладают каким-либо родством друг с другом и предлагается механизм, который определяет совместимые типы и смысл операций, которые к ним применимы.

С точки зрения манипулирования агрегатами (т. е., обектами, состоящими из большого числа других объектов; контейнерами) идея допустить смешанные вычисления в Arageli полезна с точки зрения эффективности производимых операций. Ведь она позволяет проводить операции не преобразуя один операнд к типу другого, что для агрегатов было бы напрасной и притом существенной тратой машинного времени и памяти.

Разрешено, например, складывать *vector*<**int**> и *vector*<**double**>, получая в итоге *vector*<**double**>. При этом не надо преобразовывать все операнды к *vector*<**double**> или к *vector*<**int**>, вместо этого нужно просто воспользоваться **operator+** для сложения:

```
vector<int> a = "(1, 2, 3)";
vector<double> b = "(0.1, 0.2, 0.3)";
cout << a + b; // выведет (1.1, 2.2, 3.3)
```

Реализованные схемы смешанных вычислений осуществлят эту операцию без отдельной фазы преобразования типов векторов так же, как это делается с отдельными переменными простых типов **int** и **double**.

Более того, в случае сложного шаблонного определения, механизм смешанных вычислений “заглядывает” на всю глубину типов операндов, “просматривая” тип полностью, рекурсивно. Так определяется истинная сущность каждого из типов операндов и их взаимное отношение в математическом понимании. Это нужно для правильного определения того, что нужно сделать с двумя операндами данных типов в данном выражении.

Рассмотрим следующий пример. Пусть дана матрица с полиномиальными элементами с целыми коэффициентами. В Arageli её можно смоделировать, например, следующими типами:

- *matrix*<*sparse_polynom*<**int**> >
- *matrix*<*sparse_polynom*<*big_int*> >

- *matrix*<*polynom*<*big_int*> >

Выбор зависит от решаемой задачи. Может так оказаться, что нам нужно будет домножить всю матрицу или какую-либо её строку или столбец на переменную, скажем, типа *polynom*<**int**>, так как он достаточно компактен для полиномов с небольшими степенями и малыми коэффициентами.

Вообще говоря, если рассматривать один из типов матриц выше, то ни один из них не пригоден к умножению с *polynom*<**int**> при наивном подходе в реализации операции умножения. Но, с точки зрения математики в такой операции нет ничего предосудительного, так как все матрицы представленные выше — это матрицы с полиномиальными элементами с целыми коэффициентами, а объект, на который домножают, — это полином с целыми коэффициентами.

Реализованный механизм смешанных вычислений позволяет справиться с проблемой и в данной ситуации. Не говоря уже о том, что на работу не влияет второй параметр шаблона в этом примере, — включён подсчёт ссылок (по умолчанию **true**, т. к. явно не указан), но может быть и выключен в одном или обоих операндах. Имеется ввиду, что, например, *matrix*<*sparse_polynom*<**int**, **true**>, **true**> из примера совместим с *matrix*<*sparse_polynom*<**int**, **true**>, **false**>, с *matrix*<*sparse_polynom*<**int**, **false**>, **true**> и т. д.

Помимо осуществления бинарных операций, можно преобразовывать объект в "родственный" тип обычной операцией присваивания или копирования. Например, все три типа матриц, перечисленных выше, можно свободно преобразовывать друг в друга, т. к. они представляют собой одной и то же с математической точки зрения. Естественно, данное преобразование будет корректно, если соответсвующие значения представимы результирующим типом.

Заметим, что средства бибилиотека по части рассматриваемого вопроса являются чисто статическими. Вся логика, которую необходимо вычислить и о которой идёт речь в данной статье, вычисляется на этапе компиляции программы компилятором. Т. е. мы не рассматриваем какие-либо механизмы, основынные на RTTI[1], виртуальных функциях или на чём-либо подобном, сделанном искусственно, вне базовых возможностей языка.

Сложность состоит в том, что имея несколько параметризуемых типов-шаблонов и комбинируя их друг с дургом, мы не можем ограничить множество всех возможных типов, которые из них получаются. Поэтому

[1]RTTI — Run-Time Type Information, информация о типе на этапе исполнения программы, стандартный механизм языка C++.

правила смешанных вычислений не могут быть жёстко сформурированы для ограниченного числа типов (как это, например, сделано в стандарте C++), т. к. они были бы не полными в таком случае. Они должны быть математически осмысленно сформулированы как рекурсивно применяемые правила.

Более того, выше указанный аспект неограниченности множества типов усогубляется тем, что пользователь библиотеки может вносить в базовую систему типов Arageli свои типы. Возможность такой интеграции является одним из требований, предъявляемых к нашему механизму.

Итак, смешанные вычисления в Arageli призваны смягчить возможные проблемы с огромным типовым многообразием, которое она предоставляет, делая работу пользователя более лёгкой, а программы более гибкими. Система снимает ограничения по проведению операций над операндами, которые "слегка" отличаются друг от друга, что часто бывает в программах, к которым предъявлены жёсткие требования по производительности и/или экономии памяти, т. к. это вызывает использование нескольких типов данных для одной математической сущности. Механизм так же служит смягчению особенностей языка программирования при работе со сложными вложенными типами, не ограничивая мысль математика искуственными техническими ограничениями.

3 Категории типов

Для того, что бы реализовать понятие "родственных" типов вводится категория типа. В первую очередь нас будут интересовать сложные составные типы, состоящие из других типов. Каждый такой тип T может быть представлен как корневое дерево. Поддерево этого дерева соответсвует некоторому составному типу, а корню соответсвует сам тип T. Если узел имеет потомков, то считается, что параметризуемый тип (шаблон) представленный этим узлом параметризуется типами, представленными потомками. Листом является некоторый примитивный, неделимый тип, который не имеет составных частей-подтипов.

В терминах C++ отдельному внутреннему узлу соответствует некоторый шаблон (это *не тип* в терминах языка), например *vector*, *rational*, *sparse_polynom*. Поддереву — тип инстанцированный из этого шаблона (*vector*<*rational*<*big_int*> >), а листу — примитивный непараметризуемый тип, такой как *big_int*, **int**, **double**.

Каждому из типов ставится в соответствие категория, кратко описывающая интерфейс типа, основываясь на его математической сущности. Она определяет то, какую структуру (с точки зрения допустимых опе-

раций) он реализует с использованием зависимых типов (если они есть). Выделены, например, такие категории как число, вектор, матрица, полином. Имеет место классификация категорий типов.

С технической стороны, каждая категория представлена классом в пространстве имён *Arageli::type_category*, которое определено в файле *type_traits.hpp*. Отношение между двумя данными категориями выражено наследованием соответсвующих классов. Далее представлена небольшая часть этого пространства имён, для того что бы стало ясно, что из себя представляют эти определения:

```
namespace type_category
{
    class type {}; // Общий предок для всех категорий.

    class matrix : public type {};
    class vector : public type {};
    class dense_vector : public vector {};
    class polynom : public type {};
    class sparse_polynom : public polynom {};

    ...
}
```

Категория ставится в соответсвие данному типу с помощью специализации структуры *type_traits*. Для этого служат локальный тип *category_type* и статическое константное поле *category_value* данного типа.

В *type_traits* есть ещё два элемента, которые помогают механизму смешанных вычислений. Они позволяют определить является ли данный тип агрегатом, т. е. содержит элементы других типов, которые являются частью его математической сущности и основной тип этих элементов. Для этого служит константа *is_aggregate* и тип *element_type*. Например, для *vector* все перечисленные выше поля определены следующим образом:

```
template <typename T, bool REFCNT>
struct type_traits<vector<T, REFCNT> >
{
    ...

    typedef type_category::dense_vector category_type;
    static const category_type category_value;
    static const bool is_aggregate = true;
    typedef T element_type;
};
```

Ничто не мешает пользователю определять свои категории типов. Они могут быть подкатегориями уже существующих или какими-либо независимыми, совершенно новыми. Рекомендуется размещать определения классов категорий в пространстве имён *Arageli::type_category* и наследовать от *Arageli::type_category::type*.

Если в дереве, соответсвующему конкретному типу T, забыть о точной структуре каждого узла, оставив только его категорию, то мы получим дерево категорий для данного типа T. Идея реализации смешанных вычислений состоит в том, что бы при попытке осуществить бинарную, тернарную или более арную операцию нужно операться не на конкретные типы, а принимать во внимание только соответствующие деревья категорий.

4 Связь типов

Опираясь на информацию о категориях, устанавливается связь между двумя данными типами и их взаимное отношение друг к другу. Хотя основной информацией представляемой этим механизмом является определение возможности преобразования одного типа к другому (с возможной частичной потерей данных или без неё), главная цель — это разрешение вызова бинарной операции.

Как и информация о категории типа, сведения о паре типов формулируются на уровне шаблонов, т. е. на уровне одного отдельного узла дерева типа, но имеют зависимость по параметрам-типам. Это обеспечивает автоматический проход по обоим деревьям, соответствующим двум данным типам.

Контейнером для подобной информации является шаблонная структура *type_pair_traits*, общая форма которого определена в *type_pair_traits.hpp*. Это шаблон с двумя параметрами, соответвующими двум сравниваемым типам. В нём содержится несколько статических полей, которые сообщают о возможности конвертируемости одного типа в другой посредством

различных способов, с потерей или без потери информации:

```
template <typename T1, typename T2>
struct type_pair_traits_default
{
    static const bool is_specialized;
    static const bool is_convertible;
    static const bool is_safe_convertible;
    static const bool is_assignable;
    static const bool is_safe_assignable;
    static const bool is_initializable;
    static const bool is_safe_initializable;
};
```

Эти поля инициализируются значениями, которые зависят от структуры T_1 и T_2, опираясь на их категории и, если один или оба эти типа являются агрегатами, на категории составных типов и связи между ними. В этом и состоит рекурсивное применение правил, определяющих взаимное отношение двух данных типов.

5 Свойства функций и операторов

Каждая операция, например +, *, /, полиморфна по типам операндов. Предполагается, что существует некоторое множество действий, которые скрываются за одной и той же операцией. Например, умножение двух векторов — это покомпонентное умножение, а умножение скаляра на вектор — это умножение его на каждый из элементов.

Проблема заключается в том, что смотря только на корени деревьев типов участвующих в операции нельзя определенить то действие, которое подразумевал пользователь вызывая её. Т. е., в контекте нашего примера с умножением векторов, нельзя определить что умнажается: вектор на скаляр или вектор на вектор: *vector*<*vector*<**int**> > и *vector*<**int**> — два вектора, но при их умножении, второй должен выступать в роле скаляра[2]. Смотря же на корневой шаблон (*vector*) мы не можем их различить.

[2]Возможно и другое трактование смысла этой операции, а именно, как умножение двух векторов покомпонентно. Тогда при умножении элементов возникает комбинация vector<int> и int. Но, в некотором роде эта интерпретация более сложна, так как её можно описать схемой vector<vector<T_1> > * vector<T_1>, тогда как интрепретацию, выбранную нами, — схемой vector<T_2> * T_2, где T_1 и T_2 — родственные типы, имеющие один и тот же математический смысл.

Именно для разрешения этой ситуации служит *type_pair_traits*. Каждая операция, используя особым образом информацию об аргументах, определяет какое действие нужно произвести. В том числе определяется и тип возвращаемого значения.

Для того, что бы такими операциями можно было пользоваться в контексте, который не знает выбранного действия, а следовательно и не знает типа возвращаемого значения, нужно как-то предоставить информацию об операциях с конкретными типами аргументов. Это нужно, например, что бы хранить объект вычисленный данной операцией. Причём эта информация будет зависить от действия, которое по данным аргументам производится, т. е. в том числе, будет содержать точный тип возвращаемого значения.

Для представления подобной информации в библиотеке есть семейство шаблонных структур *unary_function_traits* и *binary_function_traits*, по одной для каждой перегруженной или шаблонной функции (в зависимости от арности специализирована одна из структур). Общие версии этих шаблонов определены следующим образом:

```
template <typename TAG, typename ARG>
struct unary_function_traits
{
    static const bool is_specialized = false;

    typedef void result_type;
    typedef ARG argument_type;

    typedef TAG tag;

    static const bool alternates_argument = false;
    static const bool has_side_effect = false;
};


template <typename TAG, typename ARG1, typename ARG2>
struct binary_function_traits
{
    static const bool is_specialized = false;

    typedef void result_type;
    typedef ARG1 first_argument_type;
    typedef ARG2 second_argument_type;

    typedef TAG tag;
```

```
    static const bool alternates_first_argument = false;
    static const bool alternates_second_argument = false;
    static const bool has_side_effect = false;
};
```

Параметр *TAG* указывает на функцию, для которой специализируютсе структура. Для каждого семейства перегруженных (в том числе и шаблонных) функций в Arageli, поставлен в соответствии идентификатор из пространства имён *function_tag*. Его имя совпадает с именем соответсвующей функции или естественно соответствует имени оператора (например *function_tag::plus* для **operator+**).

Для каждого из тегов и типов, которые могут обозреть разработчики Arageli данные структуры определены таким образом, что позволяют сделать следующее. Например, даны объекты a и b типов A и B соответственно. Далее, если мы запишем выражение $a+b$, то, очевидно, должен будет сделан вызов какой-либо из перегруженных функций **operator+**. Так вот, специализация

binary_function_traits<*function_tag::plus*, A, B>

позволяет получить следующую информацию:

- точные типы аргументов в реализации этой функции (*first_argument_type*, *second_argument_type*);
- точный тип возвращаемого значения (*result_type*);
- изменяются ли первый или второй аргументы (*alternates_first_argument*, *alternates_second_argument*);
- сохраняется ли какая-либо информация при вызове оператора, которая может повлиять на объекты доступные пользователю после выхода из **operator+** кроме возвращаемого значения и аргументов (*has_side_effect*).

Например, для **operator+** от **int** и **const float&** в соответсвии со стандартом C++ должено быть такое определение[3]:

```
template <>
struct binary_function_traits<function_tag::plus, int, const float&>
{
    static const bool is_specialized = true;
```

[3]Естественно, определение этой специализации может быть выглядеть по другому, но значения полей такие же.

```
    typedef double result_type;
    typedef double first_argument_type;
    typedef double second_argument_type;

    typedef function_tag::plus tag;

    static const bool alternates_first_argument = false;
    static const bool alternates_second_argument = false;
    static const bool has_side_effect = false;
};
```

6 Разрешение вызовов операторов

Недостатки выбранного способа реализации смешанных вычислений:

- В ряде случаев мы лишаемся возможности создавать отдельно явные и неявные конструкторы для разных типов аргументов. Это происходит из-за того, что мы вынуждены объявлять только один конструктор с шаблонным параметром, и второй такой конструктор создать не возможно. Для ряда категорий типа аргумента хотелось бы определить неявный конструктор, а для других — явный, например, одноаргументные конструкторы *vector* и *matrix*.