Библиотека алгебраических вычислений Arageli. Принципы построения. ЧЕРНОВОЙ ВАРИАНТ

Н. Ю. Золотых, С. С. Лялин

3 августа 2006 г.

Аннотация

В статье рассматриваются принципы построения и дизайн библиотеки Arageli. Даётся краткий обзор её возможностей, приведены примеры испльзования. В процессе описания делается сравнительный анализ с другими известными библиотеками подобного класса.

1 Введение

Arageli — это библиотека для точных, т. е. символьных или алгебраических, вычислений. Она содержит определения базовых алгебраических структур, таких как целые числа неограниченной величины, рациональные числа, векторы, матрицы, полиномы и т. д. Реализована и постоянно расширяется алгоритмическая база для решения различных задач с использованием этих структур.

Несмотря на кажущуюся обыденность и стандартность перечисленных средств, библиотека вобрала в себя лучшие решения и разработки из уже существующих библиотек. С другой стороны, Arageli имеет некоторые особенности, которые определяются в большей степени историческими обстоятельствами и вкусами разработчиков.

Библиотека была создана Н. Ю. Золотых; в разное время в библиотеку сделали свой вклад Е. Агафонов, М. Алексеев, А. Бадер, С. Лялин и А. Сомсиков.

Библиотека начинала своё развитие ещё на языке Pascal. Но, в процессе разработки скоро стало ясно, что для комфортной работы с моделями

алгебраических объектов нужны параметризуемые типы. Они позволили бы создавать на этапе компиляции структуры, полностью моделирующие их алгебраические аналоги. Поэтому выбор языка в то время пал на C++, так как он был самым распространённым языком, имеющим необходимую выразительную мощь.

Сейчас Arageli — это библиотека, хотя и не слишком большая по объёму кода, но написанная с использованием всех возможностей C++. Мощь и выразительность языка позволила воплотить в жизнь некоторые смелые идеи и решения, которые отсутствуют в существующих библиотеках подобного класса.

Цель настоящей статьи — максимально кратко изложить архитектуру библиотеки с объяснением того, для чего был создан тот или иной рассматриваемый механизм, т. е. больший упор делается на технические аспекты, чем на состав и особенности реализованных алгоритмов и структур данных.

2 Алгоритмы и моделируемые объекты

Библиотека поддерживает моделирование следующих алгебраических систем (включая базовое множество применимых операций к каждой из них):

- целые числа неограниченной длины;
- рациональные числа;
- рациональные функции;
- полиномы от одной переменной (плотное и разреженное представления);
- векторы;
- матрицы;
- конечные поля;
- кольца вычетов (модулярная арифметика);
- *алгебраические числа*;
- *интервальная арифметика.*

Заметим, что реализация не всех из перечисленных структур полностью завершена. Некоторые операции на момент написания этой статьи были в процессе разработки. Часть структур выше, получается как комбинация других структур из этого же списка (например, конечные поля можно получить, комбинируя средства модулярной арифметики с полиномами от одной переменной и целыми числами, хотя есть отдельный шаблон класса, реализующий эту функциональность).

С точки зрения алгоритмического состава, библиотека включает следующие реализации:

- алгоритм Евклида для нахождения НОД и коэффициентов Безу;
- бинарный алгоритм Евклида;
- алгоритм Гаусса для нахождения ступенчатой формы матрицы (в том числе и целочисленный аналог);
- классический алгоритм нахождения нормальной диагональной формы Смита целочисленной и полиномиальной матриц;
- определение простоты числа методом Соловея-Штрассена, Миллера-Рабина и полиномиальным методом Агравала-Кайала-Саксены (Agrawal-Kayal-Saxena);
- разложение целого числа на простые сомножители ρ-методом Полларда и $p-1$ методом Полларда;
- решение задачи линейного программирования симплекс методом (прямой и двойственный, в строчечной и столбцовой форме), целочисленного линейного программирования алгоритмами Гомори;
- отделение вещественных корней рациональными границами алгоритмом Штурма;
- криптографический алгоритм RSA;
- алгоритм Моцкина-Бургера для нахождения остова многогранного конуса;
- LLL-алгоритм для разложения конуса на унимодулярные конусы.

Все перечисленные алгоритмы реализованы как шаблоны.

3 Компонентность и многоуровневость

Как любая сложная система, Arageli имеет компонентную и многоуровневую архитектуру. С идейной точки зрения библиотека является комбинацией нескольких в некотором смысле ортогональных механизмов, подсистем и идей. Среди них можно отметить следующие:

- базовые алгебраические структуры;
- набор алгебраических алгоритмов;
- параметризуемость;
- дополнительная информация о типе;
- смешанные вычисления;
- подсистема контроля исключительных ситуаций;
- подсистема ввода/вывода;
- контролёры алгоритмов;
- внутреннее документирование в формате doxygen.

Все перечисленные черты не являются узлами некоторой иерархии, в этом и состоит их ортогональность. Наоборот, каждая из них представляет собой некоторую иерархию конструктивных понятий, выраженных тем или иным способом на языке реализации. Всё это делает Arageli более прозрачной и управляемой с точки зрения разработчика. Тем не менее, это не означает, что все эти системы совершенно независимы.

Например, класс *big_int*, моделирующий целое число, в реализации своих операций, таких как сложение, умножение и деление, опирается на набор функций, которые работают с последовательностями цифр, представляющими собой запись больших целых чисел в позиционной системе счисления. Т. е. сам *big_int* представляет только удобный интерфейс, а алгоритмы, реализующие необходимые операции, выделены в независимый компонент. Можно безболезненно заменить один компонент другим, реализующим тот же интерфейс. Пример такой же иерархии можно увидеть в библиотеке LiDIA [2].

Это закладывает огромный потенциал развития и гибкости библиотеки, как программного продукта.

4 Параметризуемые структуры и алгоритмы

Как уже было отмечено, почти все значимые алгебраические структуры моделируются в библиотеке параметризуемыми классами, т. е. являются шаблонными классами. То же касается и функций, которые реализуют алгоритмы с использованием этих структур.

Каждая такой класс или функция параметризуется необходимым набором типов, что позволяет рекурсивно конструировать произвольные типы, корректные с точки зрения математика. Далее мы кратко опишем некоторые из основных моделей, обращая внимание в первую очередь на набор и смысл параметров.

big_int — единственный класс, который не является параметризуемым; моделирует целое число с произвольным числом бит. Размер числа может изменяться на протяжении жизни объекта, и ограничен только размером доступной памяти.

rational<T, $Reducer$> — рациональная дробь. Может быть и рациональным числом и рациональной функцией, в зависимости от природы T, который является типом числителя и знаменателя. Таким образом, можно смоделировать кольцо частных над ассоциативным кольцом T. Дробь хранится в сокращённом виде, определяемом вторым параметром $Reducer$, который может оптимизировать число сокращений дроби при последовательности некоторых операций.

vector<T, $Refcounter$> — вектор с элементами типа T. Для подобных агрегирующих структур с произвольным числом элементов последним аргументом шаблона всегда указывается признак $Refcounter$, который контролирует включение счётчика ссылок для объектов конструируемого типа.

sparse_polynom<T, D, $Refcounter$> — полином от одной переменной с коэффициентами типа T, представимый в разреженном виде, т. е. в виде, который не предполагает хранения нулевых мономов. D — тип, используемый для хранения степени каждого монома.

Естественным образом конструируются конкретные типы (некоторые параметры шаблонов имеют значения по умолчанию, поэтому они не все указаны в этих примерах):

- *rational*<*big_int*> — рациональное число;
- *rational*<*sparse_polynom*<*rational*<*big_int*> > > — рациональная функция;
- *sparse_polynom*<*matrix*<*rational*<*big_int*> > > — полином с матричными коэффициентами.

- *sparse_polynom<finite_field<> >* — полином над $\mathrm{GF}(p^n)$ с задаваемыми p и n при создании конкретного объекта.

Предоставленная пользователю возможность конструировать типы самому, а не пользоваться зафиксированным набором структур, отличает Arageli, например, от такой библиотеки как NTL [3], в которой выделено несколько полезных, хорошо известных структур (например, целые числа, полиномы с целыми коэффициентами, конечные поля) и более ничего создать не удастся.

Параметризованность основных структур и функций даёт так же способ конструировать типы с дополнительными возможностями. Например, можно создать тип, подсчитывающий битовую сложность операций, которые производят с его объектами. С таким типом можно обращаться так же как со стандартными структурами, так как любая функция в Arageli параметризуется по своим аргументам и может принимать любой тип, реализующий нужный интерфейс в независимости от того, как конкретно этот интерфейс реализуется. Возможно использование этого типа в качестве составной части другого типа и т. д..

Типы не связаны какой-либо иерархией классов, как это имеет место, например, для чисел в CLN [4] или для всех объектов в GiNaC [5]. Это отчасти следствие статического подхода, и отличает Arageli от этих библиотек, в которых придерживаются динамического подхода.

Преимущество статической параметризации перед динамической заключается не только в быстродействии и отсутствии накладных расходов на этапе исполнения программы, но и в своевременной (на этапе компиляции) диагностики ошибок, связанных с неправильным использованием системы типов. Надо признать, что при наличии такого огромного потенциала по созданию новых типов, который даёт Arageli, часты ошибки связанные с несовместимостью типов объектов при вызове функций. И как можно ранняя диагностика очень важна.

Среди недостатков статического подхода параметризации следует отметить большой объём бинарного кода, который получается из нескольких определений после компиляции: объём бинарного кода пропорционален числу фактических типов, которые построены из шаблонов. Напротив, при динамическом подходе такой проблемы нет: объём кода пропорционален числу реализованных строительных блоков (что соответствует шаблонам) и не зависит от комбинаций их использования. Но, стоит отметить, что в рамках реализованного статического подхода возможно реализовать и динамический путём определения специального класса-обёртки с жирным интерфейсом [?].

С другой стороны, статический подход имеет недостаток в том, что

для обеспечения гибкости и полной совместимости типов друг с другом (см. следующие параграфы), требуется сложная, нетривиальная и аккуратная реализация не только самих классов, но и функций комбинирующих несколько различных типов. Словом, такой подход затрудняет работу разработчикам библиотеки, впрочем, облегчает её использование.

Реализованы средства, предоставляющие информацию о данном типе (шаблонная структура *type_tairs*) или о взаимоотношении двух данных типов (шаблонная структура *type_pair_traits*), путём абстрагирования от некоторых деталей. Так, вводится такое понятие как категория типа, которое позволяет обращать внимание только на математическую сущность модели, опуская ненужные детали реализации. На языке каждая категория формализуется в виде класса. Все алгоритмы написаны в таком обобщённом стиле, используя информацию не о конкретных типах, а о тех категориях, к которым они относятся.

Это позволяет писать гибкие алгоритмы и параметризуемые классы, которые не зависят от конкретных шаблонов или типов параметров, а только от их категорий. Полезность этого нельзя переоценить и в случае расширения библиотеки новыми классами и алгоритмами, и в случае использования других, не входящих в Arageli, классов.

5 Смешанные вычисления

В некоторых программах, написанных с помощью Arageli, используется не один тип данных для представления 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 для моделирования одного и того же алгебраического объекта необходимо в одном случае из-за экономии памяти, в другом — из-за требования совместимости с чужими программами, в третьем — из-за скорости вычислений. Словом, есть причины не ограничиваться одним типом, моделирующим 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 другое — числа. Результатом должен быть объект такого типа, который сможет его корректно представить. В случае нашего примера тип результата

будет **double** [1].

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).

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. Поэтому предлагается механизм, который определяет совместимые типы и уточняет смысл операций, которые к ним применимы.

С точки зрения манипулирования агрегатами (т. е. объекты, состоящие из большого числа других объектов, — контейнеры) идея допустить смешанные вычисления в Arageli полезна с точки зрения эффективности производимых операций. Ведь становится возможным проводить операции, не приводя 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. Реализованные схемы смешанных вычислений применяют эту операцию без преобразования типов так же, как это делается с переменными простых типов **int** и **double**.

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 посредством вызова **operator*** — это покомпонентное умножение, а умножение скаляра на вектор с использованием всё того же оператора — это умножение его на каждый из элементов.

Проблема заключается в том, что, смотря только на корни деревьев типов участвующих в операции (т. е. на внешний шаблон), не всегда можно определить то действие, которое подразумевал пользователь, вызывая её. Т. е. в контексте нашего примера с умножением векторов, нельзя определить что умножается: скаляр на вектор или два вектора покомпонентно, в случае если типы аргументов такие: *vector*<*vector*<**int**> > и *vector*<**int**> — два вектора, но при их умножении, второй должен вы-

ступать в роле скаляра. Смотря же на корневой шаблон (*vector*), мы не можем их различить.

Механизм смешанных вычислений "заглядывает" на всю глубину типов операндов, "просматривая" тип полностью, рекурсивно. Так определяется истинная сущность каждого из типов и их взаимное отношение в математическом понимании. Это нужно для правильного определения того, что нужно сделать с двумя операндами данных типов при вызове данной операции в 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 умножить всю матрицу или какую-либо её строку или столбец на переменную, скажем, типа *polynom*<**int**>, так как она достаточно компактна для полиномов с небольшой степенью и малыми коэффициентами. Как это сделать максимально удобно с точки зрения пользователя?

Вообще говоря, ни один из рассматриваемых типов матриц 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 умножают, — это полином с целыми коэффициентами, т. е. перед нами обычное умножение матрицы на скаляр.

Реализованный механизм смешанных вычислений позволяет справиться с проблемой в данной ситуации и использовать обычный **operator***. Не говоря уже о том, что на работу не влияет второй параметр шаблона в этом примере, — включенный подсчёт ссылок (по умолчанию **true**, так как явно не указан), но может быть и выключенным в одном или обоих операндах. (Имеется в виду, 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.)

К тому же можно преобразовывать объект в "родственный" тип обычной операцией присваивания или конструктором копирования. Например, все три типа матриц, перечисленных выше, можно свободно преоб-

разовывать друг в друга, так как они представляют собой одно и то же с математической точки зрения.

Библиотека реализует шаблонную структуру *type_pair_traits*, которая принимает параметры двух типов и определяет связь между ними. Специализация *type_pair_traits* позволяет влиять на механизм смешанных вычислений.

Заметим, что все средства библиотека по части вопроса о типах в контексте смешанных вычислений являются чисто статическими. Вся логика, которую необходимо вычислить и о которой идёт речь в данном параграфе, вычисляется на этапе компиляции программы компилятором. Т. е. мы не рассматриваем какие-либо механизмы, основанные 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 другом, мы не можем ограничить множество всех возможных типов, которые из них получаются. Поэтому правила смешанных вычислений не могут быть жёстко сформулированы для ограниченного числа типов (как это, например, сделано в стандарте C++ для встроенных типов), так как 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 усугубляется ещё и тем, что пользователь библиотеки может вносить в базовую систему типов Arageli свои компоненты. 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, так как 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 искусственными техническими ограничениями.

У выбранного способа реализации смешанных вычислений в сочета-

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

нии с тотальной параметризуемостью есть и ряд недостатков.

Почти везде, например, 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* для числового аргумента должен быть явным, а для строки — неявным.

К тому же перегрузка функций с одинаковым числом аргументов должна уже проходить не по типу аргумента, а по его категории. Это ведёт к тому, что одна функция порождает ещё один "слой" перегруженных функций, которые имеют дополнительные параметры-категории.

Всё это затрудняет реализацию, но не преуменьшает полезность смешанных вычислений в Arageli. Во многих библиотеках сделана возможность смешанных вычислений (LiDIA [2], NTL [3]), но запрещена автоматическая конверсия одного типа в совместимый. Нигде этот механизм не представлен в таком общем виде, как в Arageli. Обычно в библиотеках ограничивалаются операциями с типами чисел, совместимостью со встроенными типами данных и типами связанными с тем же шаблоном заменой некоторых параметров.

6 Контролёры алгоритмов

Типичный алгоритм, реализованный в библиотеке, оформлен в виде функции, которая принимает некоторые входные значения и передаёт результат своей работы через аргументы-ссылки или возвращаемое значение. Для некоторых алгоритмов требуется организовать более жёсткий контроль. Например, если функция может выполняться достаточно долгое время, то было бы неплохо иметь механизм, который позволит аварийно завершить работу и выйти из функции. Это полезно в интерактивных приложениях и параллельных программах. К тому же, в процессе работы процедуры может понадобиться сделать вывод промежуточных результатов или узнать о степени завершённости выполняемой работы. Это актуально, например, для учебных интерактивных сред или при отладке.

Такой механизм в библиотеке Arageli имеется. Это специальным образом оформленные объекты-контролёры, которые передаются в "сложную" функцию как дополнительный параметр. Требования к интерфейсу для типа этого объекта полностью определяются функцией; тип должен

поддерживать все методы, которые вызывает для него целевая функция. Вызов любой из них является сигналом от алгоритма контролёру.

Например, функция *rref* (приведение матрицы к ступенчатому виду) принимает последним аргументом такой объект. Она ожидает присутствия метода *preamble*, которому она передаёт в качестве параметра входную матрицу; в начале и конце каждой гауссовой итерации *rref* вызывает для этого объекта методы *before_iter* и *after_iter* соответственно, которым передаёт текущую расширенную матрицу, базис, значение детерминанта; и т. д. — есть ещё ряд методов, которые вызываются в различных ключевых местах алгоритма.

Что делает объект контролёра, когда вызываются эти его методы, зависит от создателя контролёра и может быть весьма разнообразным. Например, он может формировать вывод в виде исходного текста документа для LaTeX, который представляет собой подробное пошаговое решение задачи, генерируемое алгоритмом. Или на каком-то этапе будет принято решение досрочно остановить выполнение функции (без завершения вычисления всех выходных параметров), тогда контролёр в одном из своих методов может сгенерировать специальное исключение, которое укажет функции на необходимость аварийного выхода. Например, контролёр может управляться интерактивным интерфейсом пользователя в параллельном потоке.

Т. е. контролёр функции — это механизм передачи дополнительной информации в/из исполняемой функции, *в процессе её работы*. Тип контролёра всегда задаётся как параметр шаблона, и может быть сконструирован, например, таким образом, что ничего не будет делать по сигналам, поступающим из функции, и никак не будет влиять на исполнение. Для контролёра функции *rref* это означает, что все методы будут с пустым телом. *Нормальный* компилятор не оставит в получаемом бинарном коде и следа вызова таких методов. Таким образом, данный механизм не лишает возможности вызвать целевую функцию как раньше, без дополнительных затрат на контроль, если важно в первую очередь эффективное получение результата.

Функцию, сконструированную так, что она принимает контролёра, будем называть *контролируемой функцией*.

Для любой контролируемой функции имеется как минимум две её перегруженные версии: с явно указываемым объектом контролёра и без оного. Вторая версия просто вызывает первую с контролёром по умолчанию, который ничего не делает (idler-контролёр). Т. е. для контроли-

руемой шаблонной функции f у нас есть следующие определения:

```
template </* параметры шаблона без контроля */, typename Ctrler>
void f
(
    /* параметры функции без контроля */,
    Ctrler ctrler    // контролёр
)
{ /* код с вызовами методов контролёра */ }

template </* параметры шаблона без контроля */>
inline void f
(
    /* параметры функции без контроля */
)
{ f(/* аргументы функции без контроля */, ctrl::f_idler()); }
```

Класс *ctrl::f_idler* — это класс контролёра, который ничего не делает (в библиотеке все такие классы находятся в пространстве имён *Arageli::ctrl*).

Основным критерием, по которому определяется, делать ли некоторую функцию контролируемой или неконтролируемой, является объём промежуточных результатов и предполагаемая продолжительность работы функции. Например, сложение, умножение векторов и матриц — это неконтролируемая операция, а приведение матрицы к нормальной диагональной форме Смита — контролируемая. При принятии решения нужно учитывать множество факторов в каждом конкретном случае.

К примеру, для основной контролируемой функции *simplex_method :: primal_row_iters* (итерации прямого строчного симплекс-метода) контолёр может понадобиться для выхода при зацикливании алгоритма симплекс-метода. Данная функция позволяет установить любые правила выбора текущего элемента в симплекс таблице, и некоторые из них могут привести к зацикливанию. Сохраняя список баз, контролёр может прервать исполнение, когда появится база уже рассматривавшаяся в ходе решения.

Справедливости ради нужно заметить, что на определённом уровне отладки библиотека отлавливает зацикливание стандартной отладочной проверкой. Но, данное поведение не перекрывает возможности контролёра и срабатывает лишь тогда, когда контролёр не прервал исполнение в нужный момент. Т. е. вызывающей программе даётся шанс узнать о зацикливании без активации механизма отладочных проверок библиотеки. Вызов симплекс-метода с таким правилом выбора ведущего элемента, который может привести к зацикливанию, очевидно, полезен в иллюстративных целях в образовательном процессе.

Опыт реализации контролёров показывает, что почти всегда код простых контролёров, которые выводят все промежуточные результаты в поток (с несложным оформлением), занимает больше места, чем сам алгоритм. Это вызвано с одной стороны громоздкостью шаблонных определений методов контролёра, а с другой — многословностью определений, делающих "красивый" вывод. Но, отсюда вовсе не следует сложность реализации этих методов.

Конечно, можно предложить альтернативу выбранному методу контроля алгоритмов. Если реализовать какой-нибудь алгоритм в виде, который уже предполагает некоторый вполне определённый и зашитый в коде вид контроля, то строчек кода будет заметно меньше, чем в нашем решении с разделением кода алгоритма и контролирующего кода. Не будет вызова функций контролера, всё будет в одном месте: и код алгоритма и код контроля. Т.е. вместо создания одной реализации алгоритма и n реализаций контролёров (по одному на каждый вид контроля), мы будем реализовывать n алгоритмов с зашитым контролем, но избавимся от общих громоздких определений отдельных контролёров.

У подобной схемы есть один большой недостаток, — для ликвидации которого и создан механизм контроля алгоритмов в библиотеке, — каждый раз придётся переписывать один и тот же алгоритм, но по-разному. Это повлечёт за собой большее количество ошибок, не говоря уже о том, что сопровождение такого кода очень сильно затруднится. К тому же, выделение основных ортогональных по отношению друг к другу концепций в независимые механизмы, представляется более элегантным.

7 Обработка ошибок

Система контроля и обработки исключительных ситуаций в библиотеке представлены двумя механизмами. Во-первых, это система классов исключений и описание ситуаций, когда они генерируются. Все функции и методы классов сконструированы таким образом, что они не теряют целостности, когда через них проходит исключение.

Следует отметить, что это в особенности важно для шаблонных функций, так как на самом деле разработчик в момент создания любой шаблонной функции не знает, какие исключения могут пройти через неё, так как их типы и ситуации, когда они возникают, зависят от типов-параметров.

Все исключения организованы в единую иерархию классов с одним общим предком, который задаёт минимальный интерфейс класса исключения. Такой дизайн является стандартным, и его можно увидеть, напри-

мер, в STL [1], GiNaC [5] и LiDIA [2].

Во-вторых, механизм обработки ошибок включает в себя отладочные проверки. Это система предикатов в различных частях функций и методов классов библиотеки вместе с кодом, который проверяет их значение. В корректно работающей программе каждый такой предикат должен быть истинен. Вычисление предиката и проверка вычисленного значения может быть включена или выключена с помощью специального конфигурационного макроса. Такой же принцип действия имеют проверки, построенные на основе макроса *assert* из стандартной библиотеки C++.

В отличие от стандартной библиотеки, Aregeli содержит несколько уровней отладочных проверок. Уровни можно включать последовательно, в зависимости от того, какую глубину проверки хочет осуществить пользователь. Дело в том, что некоторые проверки достаточно затратные по времени исполнения и памяти, и хотелось бы иметь возможность отключить их отдельно от проверок, которые относительно легки.

Опять же в отличие от стандартного *assert*, реализованная система проверок может работать в одном из двух режимов. Они различаются реакцией на ситуацию, когда вычисленный предикат имеет ложное значение. В первом режиме происходит аварийный останов, с выводом в стандартный поток сообщения об ошибке (так же делает *assert*). А во втором режиме проверочным кодом генерируется исключение с исчерпывающей информацией о не сработавшем предикате.

Первый режим работы полезен на этапе ручной отладки приложения, написанного с использованием библиотеки, а второй режим — в готовом продукте, который, впрочем, может содержать ошибки (например, в бета-версии). Во втором случае мы хотим оставить часть мягких отладочных проверок, которые не будут повергать неподготовленного пользователя в шок аварийным завершением программы, но попробуют дать какое-нибудь осмысленное решение возникшей проблемы.

8 Ввод и вывод

Объекты классов, моделирующих математические сущности можно представлять как строку символов, а так же осуществлять обратное преобразование: по последовательности символов восстанавливать состояние объекта нужного класса. Эти два процесса называются выводом и вводом соответственно.

Система ввода/вывода отвечает следующим требованиям. Во-первых, может существовать несколько способов представления объектов в виде

строки, но обязательно существует как минимум один из них, который пригоден как для операции ввода, так и для операции вывода, т. е. позволяет осуществить обратимый ввод/вывод.

Существует три основных формата ввода/вывода. Следует сразу отметить, что так как вывод существенно проще реализовать чем ввод, то нет ничего странного, что из трёх форматов, только один имеет функции по вводу (он же является обратимым), а два других работают только при выводе. Короче говоря, существует только один способ для ввода, и три способа для вывода.

Первый способ ввода/вывода — обратимый; в нём все объекты записываются в виде списков. Для группировки используются скобки, а для отделения элементов при перечислении — запятая.

Второй формат служит только для вывода — формат с выравниванием при выводе на моноширинную консоль. Особенно удобен при выводе структурированной табличной информации, например, матриц и векторов.

И наконец, третий формат вывода — это вывод на языке LaTeX. Этот способ нельзя переоценить при подготовке качественных документов с включением информации, автоматически генерируемой программой, которая использует Arageli. Такой способ вывода обычно встречается только в больших математических пакетах, подобных MATLAB и Maple.

К системе ввода/вывода так же можно отнести и преобразование объекта любого из основных классов Arageli из строки. Это позволяет вводить своего рода литералы произвольных типов, записывая их в виде строки прямо в исходном коде программы, что очень удобно при написании простых расчётных программ или в иллюстративных целях.

9 Интеграция типов пользователя

Множество шаблонов Arageli не является замкнутым. Пользователи могут создавать свои шаблоны, как классов, т. е. увеличивать число моделей алгебраических структур, так и функций, т. е. расширять алгоритмическую базу библиотеки с уже имеющимся набором структур. Правильно сконструированные компоненты пользователя не будут уступать по возможностям и универсальности компонентам, уже реализованным в библиотеки. Это залог расширяемости и гибкости библиотеки.

Правила построения и рекомендации даны в соответствующих документах, подробно описывающих архитектуру библиотеки. Вкратце опишем некоторые общие стилевые требования к классам и алгоритмам Arageli.

Во-первых, каждый новый тип T (впрочем, так же как и любой стандартный тип Arageli) должен иметь корректно определённую специализацию структуры *type_traits*, которая предоставляет некоторую дополнительную информацию о нём. Во-вторых, тип должен корректно соотноситься с другими типами, информацию о чём даёт соответственно реализованная специализация *type_pair_trais*. В-третьих, в библиотеке есть набор вспомогательных функций, которые возвращают элементы с часто используемыми значениями, например, нулевой элемент, единичный элемент и т. п., их следует реализовать, если они имеют смысл для вновь конструируемого типа T.

При выполнении всех выше перечисленных условий, объекты типа T могут быть пригодны для передачи функциям Arageli, если, конечно, они к тому же реализуют необходимый интерфейс, требуемый этими функциями. Никаких связей родства по наследованию, или какой-либо дополнительной регистрации типов на этапе исполнения не требуется (в отличие, например, от GiNaC [5]). Хотя возможно, что в будущем развитии библиотеки это свойство появится.

Как видно, условия для интеграции нового типа не слишком обременительны. Задача облегчается ещё и тем, что все три группы компонент, перечисленных выше, которые нужно реализовать для интеграции типа, имеют достаточно универсальные реализации по умолчанию. Может случиться так, что при интеграции некоторых типов не нужно будет беспокоиться о реализации этих элементов, потому что глобальная реализация, даваемая Arageli, полностью соответствует желаниям пользователя.

К алгоритмам, которые проектируются из расчёта хорошей сочетаемости с уже имеющимися функциями и классами, применяются ещё меньшие требования, чем к типам. В основном это набор рекомендаций, которые помогают написать функции “в стиле Arageli”, но ни в коем случае не ограничивают творческого размаха мысли математика, конструирующего их.

10 Совместимость с различными платформами и окружением

Ещё в начале проектирования библиотеки было поставлено требование платформенной независимости. В первую очередь, это касается качества исходного кода библиотеки: он может быть откомпилирован любым компилятором C++ в достаточной мере поддерживающим стандарт [1]. В коде Arageli нет ни одной вставки на ассемблере, какой бы то не было

машины.

Степень приближенности кода библиотеки к стандарту определяется набором компиляторов, на котором она тестируется. Базовым инструментом при разработке библиотеки является компилятор Visual C++ .NET 2003 (т. е. версия 7.1). В процессе тестирования проверяется совместимость с компиляторами Intel C++ Compiler 8.0 и 9.0 (как под операционной системой Microsoft Windows, так и под Linux), Gnu Compiler Collection (GCC) версии 3.3.3 и более поздними под операционной системой Linux, а так же Microsoft Visual C++ .NET 2005.

Для тестирования совместимости и работы библиотеки создана и постоянно пополняется база тестов.

Кроме совместимости с компилятором так же есть требование по совместимости с требуемыми библиотеками и программами для сборки и использования библиотеки. Хотя для сборки самой библиотеки нужен только компилятор C++, для создания документации требуется дополнительное программное обеспечение: lgrind, один из дистрибутивов LaTeX, например MiKTeX [9], и doxygen [8]. К тому же для сборки тестовой системы необходима библиотека boost [7].

11 Заключение

Несмотря на заложенный потенциал, библиотека остаётся пока ещё на этапе развития. Не все концепции чётко выделены и очерчены, некоторые из уже существующих требуют доработки или коррекции. Тем не менее, библиотека интенсивно используется уже по крайней мере 5 лет.

В данный момент делается основной упор на расширение множества поддерживаемых алгебраических структур и на расширение алгоритмической базы. Существующий набор кажется совершенно не достаточным для большинства задач, которые встречаются на практике и выходят из круга элементарных.

К тому же, существуют планы по реализации возможности интеграции пакетов, которые являются лидерами по производительности в тех или иных областях. В первую очередь это касается пакетов линейной алгебры (векторно-матричные вычисления) и быстрой работы с большими целыми и рациональными числами. Например, многие пакеты (NTL [3], LiDIA [2]) подключают GMP [6] для быстрой работы с числами произвольной точности.

Предполагается реализовать более функциональные средства измерения характеристик выполняемых алгоритмов, а именно точный подсчёт количества операций с различными объектами, анализ работы с памятью

с точки зрения локальности производимых операций. Это нужно, например, для проверки теоретических гипотез или для исследования практических аспектов улучшения эффективности реализуемых программ.

Существует необходимость в расширении системы ввода/вывода. Например, не реализован удобный ввод/вывод во внутреннем бинарном формате. Это нужно, в частности, для быстрого сохранения и загрузки состояния объектов или передачи по сети в приложениях на однородных кластерах.

Отдельной ветвью развития библиотеки является введение формальных символьных вычислений с машиной логического вывода.

Список литературы

[1] Programming languages — C++. International Standard ISO/IEC 14882:1998(E).

[2] LiDIA home page http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

[3] NTL home page http://www.shoup.net/ntl/index.html.

[4] CLN home page http://www.ginac.de/CLN/.

[5] GiNaC home page http://www.ginac.de/.

[6] GMP home page http://www.swox.com/gmp/.

[7] Boost home page http://www.boost.org.

[8] Doxygen home page http://www.stack.nl/~dimitri/doxygen/.

[9] MiKTeX project page http://www.MiKTeX.org/.