Алгебры как альтернатива численному параллелизму
Algebras as an alternative to numerical parallelism

Н. Н. Непейвода (Nikolai N. Nepejvoda)

В работе рассматриваются возможности, предоставляемые переходом от количественных моделей к качественным алгебраическим моделям, и трудности, возникающие на данном пути. Обращается внимание на особенности обратимого программирования и его тесную связь с алгебраическим.

Some new possibilities opened by qualitative algebraic computational models are considered here. Arising obstacles and hard points are also displayed. Aspects of reversive computing are stressed in this extent.

Алгебры программ, качественное моделирование, обратимые вычисления
Algebraic programming, qualitative models, reversive computing

Обозначения

Поскольку терминология алгебр и функционального программирования не известна в широких кругах специалистов, работающих с суперкомпьютерами, поясним используемые обозначения. Функция вместе с ее аргументами записывается как единый список, в котором последний член интерпретируется как функция. А предшествующие — как ее аргументы: (a b c f). Если T — некоторое выражение, возможно, содержащее переменные x,y,z, то λx,y,z.T — функция, перерабатывающая значения этих переменных x0, y0, z0 в значение выражения T[x0,y0,z0]:

$$(x0\ y0\ z0\ \lambda x,y,z.T)=T[x0,y0,z0].$$

Поскольку функции также могут быть объектами, необходима строгая типизация данных. То, что имя (константа либо переменная) n имеет тип τ, обозначается n: τ. Типы данных отнюдь не всегда можно считать множествами; слово «множество» резервируется для тех случаев, когда вычислительный смысл сомнителен или безразличен. Важнейшим частным случаем типа данных является тип функций от k аргументов: (τ1,... ,τk → τ). Здесь τi — типы аргументов, τ — тип результата.

Предпосылки

В последние десятилетия возникло два направления развития вычислительных применений алгебры: в чистой математической теории систем и в полуэкспериментальной информатике. Сейчас они начали сходиться и взаимодействовать.

В качественной теории систем, описываемых как дифференциальными уравнениями (обычными и в частных производных), так и разностными схемами, стали широко использоваться алгебраические методы, базирующиеся на алгебраической топологии и комплексах групп.

Если взять идеальную динамическую систему без диссипации Σ, то ее с точки зрения функционального подхода можно рассматривать как функционал, перерабатывающий элемент фазового пространства (начальное значение) в функцию, дающую положение системы в любой заданный момент времени. Ее уравнения обратимы согласно законам динамики и она порождает группу сдвигов, элементами

которой являются функции $\lambda \boldsymbol{x}$. (t ($\boldsymbol{x}$ Σ)) сдвига текущего положения системы на t. Далее, многообразие фазовых траекторий любой системы описывается методами алгебраической топологии как комплекс, у которого есть группы гомологий [1] (группы n-мерных циклов из клеток многообразия). Далее, фазовое пространство любой системы относительно некоторой совокупности преобразований координат (все равно, линейных или нет) образует группу преобразований. Далее, если некоторое многообразие обладает симметриями, то они составляют группу. Все эти группы могут использоваться для определения того, в каком качественном состоянии находится система, именно поэтому исследование качественных свойств сложных систем в современной теории обычно сводится к исследованию порождаемых ими комплексов групп [2,3].

Например, в простейшей диссипативной системе, в которой ускорение трения задается кусочно-постоянной формулой

$$d^2\mathbf{x}/dt^2 = \text{if } |d\mathbf{x}/dt|>0 \text{ then } -a((d\mathbf{x}/dt)/|d\mathbf{x}/dt|) \text{ else } 0 \text{ fi} \quad (1)$$

где a>0 — постоянная, возникает группа гомотопий фазового пространства (t,**x**, d**x**/dt), соответствующая его сжатию до **x**.

Даже в том случае, если мы не сводим задачу к некоторым явным либо скрытым симметриям, мы получаем полугруппу. Этой структурой, как известно, описывается любое пространство функций.

Группа или полугруппа автоматически порождает сильный вариант функционального программирования. Любой элемент *a* может рассматриваться также как функция λx. ($x \circ a$). Соответственно, он может трактоваться как оператор над функциями, поскольку аргументы могут тоже пониматься как функции, затем как оператор над операторами и так далее.

Взяв пространство преобразований либо фазовое пространство и факторизовав получившуюся группу или полугруппу по некоторому отношению, отождествляющему состояния либо преобразования, разница между которыми с качественной точки зрения несущественна, получаем качественное описание системы. Например, диссипативная система (1) может быть в простейшем случае описана полугруппой, получаемой отождествлением всех точек фазового пространства, в которых система останавливается более чем за k секунд, за время от k до k–1 секунд и так далее до 0. Каждый класс эквивалентности обозначим количеством секунд до остановки системы. Произведение n○m — перейти к состояниям, получающимся через m секунд. Получается нильпотентная полугруппа с элементами {k,k–1, ... ,1,0} и правилом умножения

$$n \circ m = \text{if } n>m \text{ then } n-m \text{ else } 0 \text{ fi}$$

Если нас интересуют также области, в которых заканчивается процесс, то полугруппу легко модифицировать и на этот случай с помощью конструкции подпрямого произведения.

Поскольку переход от численного представления к алгебраическому, как было показано ранее, автоматически подключает понятия высших порядков и объекты высших типов, мы оказываемся в области действия теоремы В. П. Оревкова [4] о том, что определения и вычисления могут быть сжаты в башню экспонент раз. Это колоссально превосходит эффекты от распараллеливания грубой силой, но требует перехода на совершенно другой уровень описания.

Алгебры программ

Однако прямая ссылка на традиционные математические структуры подводит и

в данном случае, как и во многих других приложениях. А именно, если рассматривать физическую реализацию обратимых вычислений, то возникает операция обращения действия элемента (просто пропустить сигнал в обратном направлении). Но операция a^{-1} не может быть выражена как групповая функция вида $\lambda x.\ (x \circ \mu)$ ни для какого μ. Единственное исключение — когда обратные совпадают с самими элементами, то есть группа может быть представлена как совокупность невзаимодействующих элементов двоичной памяти. Рассматривая возникшую ситуацию подробнее, замечаем, что в принципе умножение на «обращение» неассоциативно. Возвращаясь к обычному программированию, видим, что такая же ситуация возникает в практических системах при наличии операции UNDO и в массе других практически важных случаев. Тем самым приходим к выводу: представление программ как функций адекватно лишь в случае, когда нет нетривиальных действий и все сводится к вычислению. В частности, действия кристаллического элемента могут быть представлены как группа, а способы соединения таких элементов в нетривиальную схему – уже не могут. Программы в традиционном языке могут представляться функциями, но если мы добавляем в язык методы преобразования программ — уже нет.

Тем самым мы приходим к общей концепции программных алгебр, независимых от конкретных операторов языка, названных в [5] GAPS. Имеется алгебра с двумя бинарными операциями: ассоциативной ○ (композиция, последовательное исполнение) и неассоциативной * (применение действия, преобразование). Они связаны между собой тождеством

((a*b)*c)= (a*(b○c)).

В такой алгебре ошибка представляется как 0, а ничего не делающее преобразование (тождественная программа) – как 1. Они обладают следующими свойствами:

(0*a)=(a*0)=0, (a*1)=(1○a)=a.

0 и 1 не обязаны присутствовать в программной алгебре.

Это элементарное представление дает описание традиционных языков программирования и функциональных языков [5]. ○ образует полугруппу, которая дает описание пространства вычисляемых функций; * расширяет ее действиями и преобразованиями программ.

Эта концепция дает возможность достаточно прямо переходить к алгебраическому программированию от алгебраического описания систем. Но языки алгебраического программирования и стиль работы с ними оказываются с самого начала принципиально отличными от традиционных.

Далее, концепция абстрактных алгебр программ дает возможность выделять классы программ, базируясь на физических и логических свойствах исходных элементов и получающихся программ, безотносительно к тому, какие именно конкретные конструкции используются в алгоритмическом языке. Тем самым мы избавляемся от привязки к стандартному битовому представлении памяти, программ и данных. Это становится особенно важно в связи с проблемой обратимости и реверсивных вычислений. Обзор мировых исследований по данной проблеме: [6].

Обратимые вычисления как важнейший случай алгебраических

Как известно еще с 60-х годов, имеется физический нижний предел на выделение тепла для выработки одного бита информации: $E_{diss}= k_B*T*\ln 2$. k_B — постоянная Больцмана. Эта формула называется предел Ландауэра. Единственный

способ обойти его — сделать вычисления обратимыми. Помимо соображений тепловыделения, к обратимым вычислениям ведут также уменьшение активных элементов до молекулярного уровня, работа в сверхпроводящей среде, нанокристаллические базовые элементы (все это неизбежные компоненты новой элементной базы) и квантовые элементы, если когда-нибудь удастся что-нибудь на них реализовать. Без обратимости физический предел вычислителя с плавающей точкой – 10^{22} операций в секунду (без учета тепловых шумов и требований надежности) [8]. Таким образом, обратимость с неизбежностью возникает на всех главных путях развития вычислительной техники.

К. Беннет показал, что теоретически любую машину Тьюринга можно реализовать обратимо, но при этом экспоненциально увеличивается число операций и объем используемой памяти. Далее, он сделал важнейшее предупреждение о том, что нереверсивность хотя бы на одном уровне полностью разрушает положительные эффекты. Таким образом, необходимо поддерживать обратимость на всех трех уровнях: физическая модель вычислений, архитектура машины (команды тоже должны образовывать реверсивную алгебраическую структуру), языки высокого уровня и алгоритмы.

Если учитывать все эти ограничения, то реверсивные вычисления оказываются принципиально неполными по Тьюрингу. По сути дела, любой реверсивный процессор будет реализовывать вычисления на конечной алгебре, а именно, на GAPS без 0, с единицей и с дополнительной командой (константой) M (обращение действия):

((b*M)○b)=(b○(b*M))= 1.

Тем самым мы приходим к структуре, в которой базовая группа по операции ○ пополняется алгеброй обратимых соединений и преобразований по *.

Если учитывать, что техника группового описания систем уже глубоко отработана в теории, то остается дело за малым: перенести ее на практику. Но такой перенос является фундаментальной теоретической и одновременно практической работой. Например, непрерывные пространства для перехода к практическим алгоритмам пришлось приближать сеточными. И здесь с необходимостью возникает теория приближений бесконечных групп, возникающих в математике, конечными, которые нужны для практической реализации.

Ситуация с обратимыми языками программирования одновременно и лучше, и хуже. Сам язык описывается бесконечной алгеброй. Но любая конкретная программа всюду определена в силу обратимости и порождает для любых конкретных исходных типов конечную алгебру. Вопрос остается «всего лишь» в разработке эффективных методов оптимизации таких алгебр.

Даже логика обратимых программ принципиально отличается от известных логик традиционных алгоритмических программ. Опишем ее.

Моделью является группа. Каждой пропозициональной букве A сопоставляется подмножество этой группы ®A. Логические связки делятся на классические →&V¬ с традиционной булевой интерпретацией и конструктивные: конъюнкция ; импликация =>, отрицание ~ и константа E. Их семантика следующая.

®A;B={a○b | a®A& b®B}

®A=>B={c | Если a®A, то a○c®B}

®~A={a^{-1}| a®A}

®E={1}.

Эта логика не является конечнозначной и что конструктивные конъюнкция и следование выразимы друг через друга. Например, A=>B = ~¬(~¬B;A). (результаты

Непейвода А. Н.) Конструктивные формулы в этой логике дают построения программ над группой, а их преобразования — допустимые преобразования программ.

Чтобы показать некоторые особенности алгебраического программирования, рассмотрим язык программирования Botik, предложенный в [7]. Общая структура программы в этом языке следующая.

```
DEFINITIONS
*описания получаемых из внешнего источника групп
INPUT
* определения исходных значений, что существенно необратимо
END INPUT
MAIN
* главная секция, полностью обратимые действия
END MAIN
OUTPUT
* выводимые результаты, существенно необратимо
```

Общая структура блока TYPE следующая:

```
DEFINITIONS
{name}={Imported ;{name} Imported ~{name}}*
```

Здесь **;** — групповая операция умножения, ~ — операция взятия обратного. Если тип стандартный (например, $<Z_2,+>$) то записывается короче:

```
{name}={name of the standard group}
```

Определяется конструктор типов. {name}={name1};{name2} дает прямое произведение групп {name1} и {name2}. Таким образом, элемент массива может быть представлен как G ; Z_n.

Блок ввода производит операции задания начальных значений, которые принципиально не могут быть реверсивными.

```
INPUT
  {{group type} {name} {=name}|{=value}|}*
END INPUT
```

Если переменной не присваивается значения, то она задается динамически, в момент исполнения программы, с помощью чтения из внешней среды. Этот блок единственный, где допускается дублирование значений. Но никакие вычисления, даже статические, недопустимы.

Рассмотрим теперь раздел MAIN.

```
{segment}={{name}{;{function}}+}|~ {{name}{;{function}}+}
{function}=~{function}|{name}|{segment}|
        IF {name} THEN {function} ELSE {function} FI|
        BY {constant} DO {segment} OD
MAIN
{segment}*
END MAIN
```

Первые переменные сегментов называются базовыми и не могут использоваться в последующем функциональном блоке. В базовых переменных накапливается результирующее значение. Переменная в условии может

использоваться в сегментах, но не может там изменяться.
BY a DO {section} OD

Это оператор цикла. Переменная a называется итератором. Цикл BY вычисляет a, a^2, ... a^n, ... пока не окажется $a^k=1$ (само значение a остается неизменным). Из-за конечности типов данных элемент a имеет конечный порядок, и поэтому цикл всегда заканчивается. Группа, соответствующая циклу — прямое произведение типа a и типа {section}.

Условное предложение с двоичным условием a
IF a THEN b ELSE c
может быть интерпретировано как группа со следующей операцией умножения:
$(a,b1,c1)\circ(0,b2,c2)=(a,b1\circ b2,c1\circ c2)$
$(a,b1,c1)\circ(1,b2,c2)=(a+1,b1\circ c2,c1\circ b2)$

Эта операция может быть получена как частный случай полупрямого произведения групп. b1 может интерпретироваться как текущее состояние, а c1 — как альтернатива, которую приходится запоминать ради обратимости. Таким образом, при прямой интерпретации каждое исполнение условного оператора удваивает число элементов группы, соответствующей программе.

Приведем пример программы, вычисляющей степень элемента, являющуюся числом Фибоначчи.

```
* k является номером числа Фибоначчи
INPUT Z2 flag=E, Zk n=1, G a, G b=E, G c=E
END INPUT
MAIN
        BY n DO
                IF flag THEN {c;a} ELSE {c;b} FI
                IF flag THEN {b;a} ELSE {a;b} FI
                {flag;1}
        OD
END MAIN
OUTPUT c
```

В исходном виде язык Botik работает с абстрактными группами, но в каждом конкретном случае, когда типы данных уточнены, программы можно и должно оптимизировать. Рассмотрим пример абстрактной программы и ее специализации для случая группы Z_2.

Абстрактная программа

```
INPUT Z2 flag=E, Zn a=1, G b, G c=E
END INPUT
MAIN
        {c;a}
        IF c=E THEN {flag;1} ELSE {flag;E} FI
        BY a DO
                IF flag=E THEN {c;a} ELSE {c;E} FI
                IF c=E THEN {flag;1} ELSE {flag;E} FI
        OD
END MAIN
OUTPUT c,flag
```

Ее специализация.

```
TYPE G=Z2
INPUT Z2 flag=E, Z2 a=1, G b, G c=b
END INPUT
MAIN
    IF a=E THEN
        IF b=E THEN {c;~b} {flag;1} ELSE {c;E} {flag;E}
        FI
    ELSE {c;~b} {flag;1} FI
END MAIN
OUTPUT c,flag
```

Но необходимо заметить, что операции преобразования программ существеннейшим образом нереверсивные, и их необратимость тем сильнее, чем выше эффективность. Таким образом, подготовка реверсивных программ к исполнению должна производиться традиционным процессором.

Далее, реверсивные программы на Botik по определению примитивно-рекурсивны. Botik совпадает по выразительной силе с элементарной конечной арифметикой. Главный процессор выполняет подготовительную работу и окончательное преобразование полученных результатов к понимаемому человеком виду. И то, и другое требует анализа громадного числа условий и по сути своей не примитивно-рекурсивно. И то, и другое существенно необратимо. А алгебраический процессор выполняет вычислительную работу. Мы с необходимостью приходим к возрождению идей гибридных машин.

Общий случай

Теперь вернемся к общему случаю алгебраического моделирования. Эффект Оревкова возможного суперэкспоненциального сокращения выполнен для любой системы алгебраического моделирования. Рассмотрим пример, восходящий к исходной идее Оревкова.

Если у нас есть лишь исходная операция прибавления единицы, то вычисление экспоненты требует экспоненциального числа шагов. Если же записать ее неявное определение с помощью равенств

$\varphi(x,0)=x+1$

$\varphi(x,y+1)=\varphi(\varphi(x,y),y)$

то экспонента вычисляется в линейное число шагов.

Если теперь задать определение функции второго порядка

$\Psi(\varphi,x,0)=\varphi\ (0,x+1)$

$\Psi(\varphi,x,y+1)=\Psi(\varphi,\Psi(\varphi,x,y),y)$

то за линейное число шагов вычисляется уже суперэкспонента и так далее.

Рассмотрения в [5] показывают, что алгебры гибко специализируются под различные классы программ, определенные ортогонально тому, как привыкли делать в информатике (например, полуобратимые программы для работы с экономической информацией и ответственными базами данных; автоматы высших типов …) Алгебраические программы с самого начала построены так, что их нужно не распараллеливать, а укладывать на конкретную структуру вычислительной системы. Весь возможный параллелизм сразу задан в алгебраическом описании.

Но программирование для алгебраических моделей в любом варианте

становится совершенно нетрадиционным. Системы должны быть специализированы, поскольку универсальность сразу убивает все преимущества. В терминах битов, чисел и последовательного исполнения в этой области работать не получится.

Приношу благодарность Непейвода А. Н. за предложения по улучшению языка Botik и за помощь в редактировании статьи.

Литература.

1. В. В. Прасолов. Элементы теории гомологий. М.: МЦНМО, 2006.
2. А. Т. Фоменко. Дифференциальная геометрия и топология. Дополнительные главы. Ижевск, 1999.
3. В. И. Арнольд. Геометрические методы в теории обыкновенных дифференциальных уравнений. Ижевск, 2000.
4. Н. Н. Непейвода. Прикладная логика. Новосибирск, 2000.
5. Н. Н. Непейвода. От численного моделирования к алгебраическому. Параллельные вычисления и задачи управления PACO'2012. Труды шестой международной конференции. Том 1. Москва. 2012, стр. 93–104.
6. А. Н. Непейвода. Реверсивные вычисления: обзор мирового опыта. Параллельные вычисления и задачи управления PACO'2012. Труды шестой международной конференции. Том 2. Москва. 2012, стр. 129–142.
7. N. N. Nepejvoda. Reversivity, reversibility and retractability. Third Int. Workshop on Metacomputation. Pereslavl, 2012, pp 203—227.
8. E. DeBenedictis, Will Moore\rq s Law be Sufficient, // Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, 2004.

Непейвода Николай Николаевич, г.н.с. ИПС им. А.К. Айламазяна РАН, закончил Московской Государственный Университет в 1970 г., д.ф.-м.н., профессор, около 200 научных работ, в том числе 5 монографий, логика, информатика. email nepejvodann@gmail.com +79109635479
Nikolai N. Nepejvoda, main research fellow IPS A.K. Ailamazian RAS, Moscow state University in 1970, dr.sc., professor, approx. 200 works incl 5 books, logics, computer science, e-mail nepejvodann@gmail.com