СИСТЕМА АЛГЕБРАИЧЕСКОГО И ИНСЕРЦИОННОГО ПРОГРАММИРОВАНИЯ АПС

(руководство пользователя)

А.А. Летичевский, Ю.В. Капитонова, В.А. Волков, А. Чугаенко, В. Хоменко

Институт кибернетики имени В.М.Глушкова
Национальной академии наук Украины
Киев, Украина
E-mail: let@d105.icyb.kiev.ua

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгебраическое программирование – это программирование, основанное на переписывании. Алгебраическое программирование является расширением функционального программирования и применяется при решении задач компьютерной алгебры (таких как проблема слов в конечно определенных алгебрах, алгоритмы пополнения Кнута-Бендикса или Бухбергера), а также задач, связанных с операционной семантикой языков программирования (исполняемые алгебраические спецификации компонентов программного обеспечения, определение операционных семантик языков программирования, разработка интерпретаторов и прототипов компонентов программного обеспечения и др.).

В отличие от традиционного подхода, ориентированного на использование канонических систем правил переписывания с «очевидной» стратегией их применения, в АПС возможно сочетание любых систем правил переписывания и разнообразных стратегий переписывания. Такой подход значительно расширяет возможности техник переписывания, поскольку возрастает их гибкость и выразительность. АПС интегрирует четыре основные парадигмы программирования таким образом, что основная часть программы может быть написана в виде систем переписывания, императивное и функциональное программирование используются для определения стратегий, логическая парадигма реализуется на базе переписывания, использующего встроенную процедуру унификации.

Инсерционное программирование, развиваемое в настоящее время в Институте кибернетики им. В.М. Глушкова, продолжает традиции алгебраического подхода в прикладной теории алгоритмов, программировании и разработке кибернетических систем, основы которого были заложены В.М. Глушковым и его школой. В алгебре алгоритмов Глушкова программа рассматривается как алгебраически определенное преобразование множества состояний информационной среды (память, база данных, многоуровневые распределенные структуры данных и т.п.). В инсерционном программировании этот взгляд обобщается на объекты, обладающие поведением. Вместо пассивной среды, такой, например, как память, рассматривается активная среда, в которой преобразование информации системой взаимодействующих агентов определяет поведение этой среды, наблюдаемое пользователями информации, производимой внутри среды. Программа в инсерционном программировании рассматривается как агент, обладающий поведением. Его погружение в среду изменяет поведение этой среды. Переход от программ как преобразователей состояний к программам как преобразователям поведений можно сравнить с переходом от точечных к функциональным пространствам в математике.

Инсерционное программирование – это программирование на базе модели поведения агентов в средах [1]. В основе модели лежит понятие размеченной транзиционной системы, т.е. системы, определенной так же, как и автомат, множеством состояний и множеством переходов (пары состояний), размеченных действиями или событиями. Формально понятие транзиционной системы совпадает с понятием недетерминированного частично определенного автомата, однако, в отличие от теории автоматов отношение эквивалентности транзиционных систем более сильное.

Если для автоматов эквивалентность состояний определяется совпадением языков в алфавите входных сигналов, порождаемых этими состояниями, то эквивалентность состояний транзиционных систем определяется эквивалентностью деревьев поведений, размеченных действиями системы. Это отношение называется бисимуляционной эквивалентностью. Оно было введено в работах Р. Милнера и Д. Парка еще в 70-х годах. В настоящее время понятие транзиционной системы и бисимуляционной эквивалентности используется в качестве основного стандарта в поведенческой теории взаимодействующих процессов в сочетании с различными более слабыми, чем бисимуляция, отношениями эквивалентности на множестве состояний систем.

Содержательные мотивы инсерционного программирования строятся на представлениях о поведении систем (агентов), взаимодействующих со своим окружением (средой). Поэтому ближе всего к инсерционному программированию находятся современные декларативные языки моделирования типа UML, SDL, MSC, а также языки, основанные на алгебрах взаимодействующих процессов (CCS, CSP, и т.п.). Агентное программирование концентрируется в большей степени на проблемах интеллектуализации, однако его поведенческие аспекты естественным образом компенсируются инсерционным программированием.

Отличительной чертой инсерционного программирования является формализация понятия среды и функции погружения (insertion) агентов в среду. Функция погружения определяет композицию среды и агента, результатом которой является новая среда, готовая для погружения других агентов. Агенты и среды рассматриваются как объекты разных типов, обладающие поведением, представляемым с помощью транзиционных систем, состояния которых рассматриваются с точностью до бисимуляционной эквивалентности.

В качестве одного из основных примеров среды может служить компьютер, рассматриваемый как среда для программных агентов. Погружение программы в компьютер изменяет его поведение и превращает в иную среду. Первая программа (или система программ), погружаемая в компьютер, обычно представляет собой операционную систему (WINDOWS, UNIX и т.л.), которая расширяет возможности взаимодействия компьютера с программными агентами и с внешней, пользовательской, средой, поставляющей программные агенты для погружения в компьютер. Простая программа старого типа, которая получает данные, отрабатывает некоторый алгоритм и успешно завершает свою работу, изменяет поведение среды незначительно и лишь на короткое время. И совсем иное – пакеты прикладных программ, постоянно готовые к получению запросов (другой тип агентов, погружаемых в вычислительную среду) вместе с данными для решения задач из соответствующего класса. Погружение таких пакетов или интерактивных программ, взаимодействующих с внешней средой, существенно меняет поведение исходной среды.

Еще одним впечатляющим примером сложной среды является Интернет. Агенты в этой среде не только взаимодействуют друг с другом, но также обладают свойством мобильности и имеют возможность перемещаться в пространстве, создаваемом средой. В телекоммуникационной среде, поддерживающей мобильную связь, перемещение агентов (мобильных телефонов) осуществляется физически, а не виртуально, как в Интернете. Агенты и среды могут быть устроены иерархически. Любая среда с помещенными в нее агентами может быть закрыта для помещения в нее новых агентов извне, и, рассматриваемая как агент, может быть погружена в среду верхнего уровня.

Важной чертой инсерционного программирования является недетерминированность поведения агентов и сред, присущая реальным системам, которые моделируются инсерционными программами. Поэтому реализация систем инсерционного программирования, в общем случае, требует применения моделирующих программ (симуляторов) вместо интерпретаторов, а также постановки целей для получения конкретных результатов.

В структуре АПС можно выделить три способа структуризации используемых элементов:

– основные типы системных объектов;
– базовые вычислительные механизмы;
– допустимые языковые конструкции, объединенные в семейство языков АПЛАН.

Система алгебраического программирования организована таким образом, что пользователь имеет одновременный доступ ко всем уровням программирования, начиная с языка конкретной предметной области и заканчивая уровнем расширения языка си, которое называется L2C. Таким образом достигается эффективное распределение сложности разрабатываемой программной системы по различным уровням. Поддерживается

эволюционность процесса программирования, начиная с выполняемой алгебраической спецификации через оптимизирующие преобразования до эффективной программы на языке си. Самый верхний и специализированный уровень образуют конструкции, связанные с некоторой конкретной предметной областью. Они имеют локальный характер, но также активно используются в других предметных областях. Это связано с тем, что в базовом АПЛАНе нет типов данных. Строго говоря, существует единственный тип данных – алгебраический терм. В то же время имеются возможности для использования строгого типизирования, например, при программировании в объектно-ориентированном расширении АПЛАНа.

ПРАВИЛА ПЕРЕПИСЫВАНИЯ

Техника программирования в АПС базируется на переписывании термов. Рассмотрим простую задачу функционального программирования – вычисление чисел Фибоначчи. Хорошо известное рекурсивное определение n-го числа Фибоначчи задается следующей системой соотношений:

```
F(0) = 1 ,
F(1) = 1 ,
F(n) = F(n - 1) + F(n - 2)
```

Данную систему можно рассматривать как (рекурсивное) определение функции F или систему правил переписывания, которую можно использовать для вычисления F(n). Чтобы записать на языке АПЛАН программу, использующую эту систему, следует начать с предложения

```
INCLUDE <std.ap>,
```

если файл `std.ap` находится в том же каталоге, куда Вы собираетесь поместить Вашу программу, иначе следует указать путь к файлу `std.ap`. Это предложение включает некоторые стандартные определения, в частности, обеспечивает использование арифметических операций “+”, “-”, отношения “=”, символа “,”, а также некоторых других синтаксических понятий, определенных в модуле `std.ap`. Затем надо определить имя системы

```
NAME R;
```

и присвоить начальные значения

```
R:=rs(n)(
    F(0) = 1 ,
    F(1) = 1 ,
    F(n) = F(n - 1) + F(n - 2)
);
```

Первая строка присваивания указывает, что значением имени R является система правил переписывания (rs), а n – единственная переменная, используемая системой.

Общий вид определения синтаксиса систем правил переписывания следующий:

```
<rewriting system> ::= rs(<list of variables separated by ",">)
        (<list of rules separated by "," >)
<rule> ::= <simple rule> | <conditional rule>
<simple rule> ::= <algebraic expression> = <algebraic expression>
<conditional rule> ::= <condition> -> <simple rule>
<variable> ::= <identifier>
```

Систему R можно применить, например, к выражению T=F(10), и данное выражение будет преобразовано таким образом:

```
F(10)=F(10-1)+F(10-2)=F(9)+F(8).
```

В АПС данное преобразование выполняется за один шаг, так как арифметические операции являются интерпретируемыми и выполняются по мере возможности на каждом шаге переписывания. Следующий шаг переписывания может быть выполнен одним из двух способов в зависимости от того, к которому вхождению выражения F(n) применяется третье правило системы. Рассмотрим случай, когда выбирается первое вхождение, то есть F(9). Тогда имеем новое выражение:

```
T = (F(8)+F(7))+F(8)
```

Продолжая подобным образом, то есть выбирая всякий раз самое левое (самое внутреннее) вхождение, получим:

```
   T = ((F(7)+F(6))+F(7))+F(8) =
 = (((F(6)+F(5))+F(6))+F(7))+F(8) =
 ..................................
= (...((F(1)+F(0))+F(1))+....)+F(8)
```

Теперь применимы как третье, так и второе правила. Будем действовать по принципу, который используется во всех основных стратегиях переписывания в АПС. Он состоит в том, что правило, которое фактически применяется, – это первое из правил, применимых на текущем этапе вычислений. Таким образом, последующие два шага преобразований приведут к таким выражениям:

```
T = (...((1+F(0))+F(1))+....)+F(8) =
  = (...((1+1)+F(1))+...)+F(8)
```

В данной ситуации имеется еще одна возможность выбора: вычислить выражение 1+1 прежде, чем преобразовать F(1). Применение арифметических операций к постоянным величинам также можно рассматривать как переписывание с помощью неявно заданных правил переписывания (таблиц сложения и умножения). Поэтому в соответствии с левосторонней стратегией в первую очередь следует выполнить сложение. Следующий шаг переписывания:

```
T = (...(2+F(1))+...)+F(8) =
  = (...((2+1)+F(2))+...)+F(8) =
  = (...(3+F(2))+...)+F(8) = ...= 55+34 = 89
```

Для данной системы переписывания тот факт, что при переписывании выбирается лишь самое левое вхождение, не существен. Чтобы получить требуемый результат с помощью системы правил R, достаточно использовать любую стратегию переписывания (алгоритм переписывания), удовлетворяющую следующим условиям.

1. На каждом шаге переписывания применяется одно из правил системы либо выполняется арифметическая операция.
2. Выбор правила осуществляется согласно последовательности, в которой правила записаны.
3. Переписывание продолжается до тех пор, пока это возможно, то есть пока существуют вхождения, к которым применимы правила системы, либо возможны выполнения арифметических операций над числами.

Стратегия, удовлетворяющая приведенным условиям, называется *финальной*. АПС позволяет использовать встроенные стратегии переписывания, а также писать собственные стратегии, которые наиболее подходят для решаемой задачи. Вызов встроенной стратегии является вызовом внутренней процедуры. Как правило, он имеет вид *s(T,R)*, где *s* – имя стратегии, *T* – имя алгебраической структуры данных, к которой применяется система, а *R* – имя системы правил переписывания.

Имеются две встроенные финальные стратегии переписывания: applytb (пошаговое применение сверху вниз) и applybt (пошаговое применение снизу вверх). Каждую из них можно использовать для вычисления n-го числа Фибоначчи. Следует лишь задать определение имени *T*, присвоить этому имени начальное значение, а также сформировать задание, соответственно которому будет вызвана стратегия и результат будет выведен на экран. Соответствующий текст имеет следующий вид:

```
INCLUDE <std.ap>
NAME T;
T:=F(10);
task:=applytb(T,R),prn(T);
```

Здесь “task” – стандартное имя системы. Оно определено в файле `std.ap`. Оператор prn(T) выводит значение T на экран. Для выполнения задания в среде UNIX нужно воспользоваться командным файлом dotask.sh следующего вида:

```
LD_LIBRARY_PATH=$HOME/aps-2004/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH

if [ ! -f can_tbl.tbl ]; then
 ln -s $HOME/aps-2004/bin/can_tbl.tbl can_tbl.tbl
fi

if [ ! -f proc_tbl.tbl ]; then
 ln -s $HOME/aps-2004/bin/proc_tbl.tbl proc_tbl.tbl
fi

if [ ! -f aps ]; then
 ln -s $HOME/aps-2004/bin/aps aps
fi

$HOME/aps-2004/bin/aps -i $1

true
```

который при необходимости требуется отредактировать, указав фактический путь к каталогу `aps-2004`, содержащему системные модули.
В результате мы получим десятое число Фибоначчи, то есть 89.

Обсудим теперь работу стратегий applytb и applybt. Сперва рассмотрим, как алгебраические выражения представляются в виде помеченных деревьев. Вершины такого дерева соответствуют подвыражениям данного выражения. Вершины, соответствующие элементарным подвыражениям (таким как число или символ), помечены самими этими подвыражениями. Каждая вершина, соответствующая неэлементарному подвыражению, помечена главной операцией этого подвыражения.

Вершина, соответствующая подвыражению $f(x_1,\ldots,x_n)$, где f – n-арная операция, соединяется дугами, пронумерованными числами от 1 до n, с вершинами, которые соответствуют подвыражениям $x_1,\ldots,x_n$. Дерево, соответствующее выражению $F(n\text{-}1)+F(n\text{-}2)$, можно представить одним из двух способов в зависимости от значения символа F. Он может быть определен в языке АПЛАН как одноместная операция с помощью оператора

```
MARK F(1);
```

В этом случае символ F является меткой двух вершин дерева. Если же символ F появляется без определения, то он рассматривается как атом, а главной операцией выражения $F(x)$ является операция аппликации, которая обозначается как обычная конкатенация двух выражений. Первым аргументом этой операции в выражении $F(x)$ является атом F, вторым – выражение x.

В основе обеих стратегий (applytb и applybt) лежит левосторонний обход дерева в глубину. При обходе каждая вершина посещается дважды: первый раз при движении сверху вниз, второй – при движении снизу вверх. Стратегия applytb применяет систему правил к данному выражению в соответствии с текущей вершиной, двигаясь к этой вершине сверху. Стратегия применяется к данной вершине столько раз, сколько возможно. Стратегия applybt работает аналогично, но движение к текущей вершине происходит снизу. Если в ходе полного обхода применилось хотя бы одно правило, то обход начинается снова, а стратегия работает до тех пор, пока не останется применимых правил. При движении снизу вверх в процессе работы стратегии выполняются все упрощения, которые возможны (такие, как вычисление арифметических выражений). Поэтому обе стратегии являются финальными, и каждая из них может использоваться для преобразования выражения F(n) при любом натуральном n. Систему R можно применять и к сложным выражениям, которые содержат вызовы функции F, например, к выражению F(F(n)). Однако в этом случае стратегии работают по-разному.

Стратегия applybt вычислит значение выражения, а стратегия applytb будет выполнять последовательность переписываний, которая не завершается:

```
F(F(10)) = F(F(10)-1)+F(F(10)-2) =
            (F((F(10)-1)-1)+F((F(10)-1)-2))+F(F(10)-2) = ...
```

Бесконечного переписывания можно избежать, если использовать правила перписывания с условиями (условные правила переписывания). Заметим, что третье правило системы R следует применять к выражению F(n) лишь в том случае, когда n есть неотрицательное целое число. Соответствующее условие записывается на языке АПЛАН следуюшим образом: `isint(n)&(n>0)`. Его можно добавить к третьему правилу. Новая система имеет вид:

```
R:=rs(n)(
                F(0) = 1 ,
                F(1) = 1 ,
     isint(n)&(n>0)->(
                F(n) = F(n - 1) + F(n - 2)
     )
);
```

Теперь, когда третье правило защищено от нежелательных применений, систему R можно применять, используя любую финальную стратегию. Если эта система применяется к произвольному выражению, содержащему вхождения F, то будут вычислены все подвыражения вида F(n), где n – неотрицательное целое число.

Рассмотрим правила вычисления чисел Фибоначчи с другой точки зрения. Нетрудно заметить, что число шагов, выполняемых любой финальной стратегией для вычисления F(n), ограничено снизу выражением, содержащим экспоненту. Действительно, после применения третьего правила вычисление F(n) сводится к вычислению F(n-1) и F(n-2). Эти вычисления будут выполняться независимо. Вычисление F(n-1) сведется к вычислению F(n-2) и F(n-3). Таким образом, выражение F(n-2) будет вычисляться дважды, выражение F(n-3) – трижды и т.д. Чтобы разобраться в том, как можно было бы улучшить процесс вычислений, обратимся снова к вычислению выражения F(10). Будем допускать теперь применение любых алгебраических упрощений, а не только выполнение арифметических операций с целыми числами. Имеем:

```
T = F(10) =
        (F(8)+F(7))+F(8) = 2*F(8)+F(7) = 2*(F(7)+F(6))+F(7) =
            3*F(7)+2*F(6) = 3*(F(6)+F(5))+2*F(6) = 5*F(6)+3*F(5) = ...
            ... = 89
```

Нетрудно найти общую форму записи примененных правил:

$$a*(x+y)+b*x = (a+b)*x+ a*y$$

Приведем также специальные случаи этого правила при a=b=1 и при b=1:

$$(x+y)+b*x = (1+b)*x+y,$$
$$a*(x+y)+x = (a+1)*x+a*y$$
$$(x+y)+x = 2*x+y$$

Если добавить эти правила к системе R, переписывание F(n) можно осуществить за число шагов, пропорциональное n, при условии применения подходящей стратегии. Стратегии applytb и applybt уже непригодны. Каждая из них будет применять третье правило перед тем, как применятся новые правила, что приведет к тем же затратам, что и в случае, когда новые правила не использовались. Требуемая стратегия должна работать таким образом, чтобы на каждом шаге переписывания правило системы переписывания применялось к самому левому внешнему вхождению выражения. И среди встроенных стратегий АПС такая стратегия имеется. Это стратегия lmt (самый левый-самый внешний, известная стратегия так называемого “ленивого” вычисления). Новая система правил такова:

```
    R1:=rs(n,a,b,x,y,z)(

        a * (x + y) + b * x = (a + b) * x + a * y,
```

```
        a * (x + y) +     x = (a + 1) * x + a * y,
            (x + y) + b * x = (1 + b) * x +     y,
            (x + y) +     x =       2 * x +     y,

        F(0) = 1 ,
        F(1) = 1 ,
        F(n) = F(n-1)+F(n-2)
    );
```

а новое задание следующее:

```
        task:=lmt(T,R1),prn(T);
```

Поскольку описание стратегии lmt и некоторых необходимых для ее работы процедур содержатся в системных файлах strat.ap и gen_obj.ap, следует включить в программу соответствующие предложения `INCLUDE < strat.ap>` и `INCLUDE <gen_obj.ap>`. Заметим, что несколько предложений включения можно заменить одним вида `INCLUDE <include_aplan.ap>`, содержащем предложения включения системных файлов.

Недочетом приведенной выше системы является большое число правил. Его можно устранить, если более внимательно проанализировать общее состояние вычислений. Такое состояние может быть описано выражением:

$$a*F(n)+b*F(n\text{-}1),$$

которое получается уже после четвертого применения системы и повторяется через каждые два шага. Обозначим функцию, определяемую данным выражением, через f(a,b,n). Нетрудно заметить, что при n>0

$$F(n) = 1*F(n)+0*F(n\text{-}1) = f(1,0,n),$$
$$f(a,b,n) = f(a+b,a,n\text{-}1)$$

Таким образом, вычисление F можно заменить вычислением f, полагая f(a,b,0)=a и применяя такую систему переписывания

```
R2:=rs(a,b,n)(
    f(a,b,0) = a,
    f(a,b,n) = f(a+b,a,n-1),
    F(n)     = f(  1,0,n  )
);
```

Для управления такой системой нужна более простая стратегия, которая применяет систему только к самому выражению и не рассматривает его подвыражения. Эта стратегия называется appls. Вычисления ускорятся, если символ f определить как трехместную операцию, а не атом или имя. Система R2, очевидно, соответствует хорошо известной процедурной программе вычисления F(n), но здесь она представлена в алгебраической форме и получена исключительно алгебраическими методами без использования процедурного мышления. Это основная особенность алгебраического программирования, которая позволяет приводить процедурные конструкции к простому виду, сократив их при этом до необходимого минимума, а также выражать наиболее существенные свойства алгоритмов в математическом (алгебраическом) виде.

ПРОЦЕДУРЫ

Императивные программы представляют собой последовательности операторов, разделенных символами “,” или “;”.

Императивная программа может быть значением имени и может быть вызвана посредством системной команды:

Синтаксис оператора следующий:

```
<statement> ::= <basic statement>| <conditional statement>
    | <while statement> | <do statement> | <internal call>
    | <external call> | return | return(<algebraic expression>)
    | (<program>)
```

```
<basic statement> ::= <set statement> | <assignment statement>
<set statement> ::= <selector> --> <algebraic expression>
<assignment statement> ::= <name> := <algebraic expression>
<selector> ::= <name>
    | arg(<selector>,<sequence of expressions separated by ",">)
<conditional statement> ::= <condition> -> <statement>
    | <condition> -> <statement> else <statement>
<while statement> ::= while(<condition>, <statement>)
<do statement>  ::= do(<name>)
<internal call> ::= <internal procedure name>
    (<actual parameter list>)
<internal name> ::= <atom>
<external call> ::= <procedure definition or name of procedure
    definition>(<actual parameter list>)
<actual parameter> ::= <algebraic expression>
```

Синтаксис определения процедуры:

```
<procedure definition> ::= proc(<formal parameters list>)
    <local names> <statement>
<local names> ::= loc(<local names list>) | <empty>
<formal parameter> ::= <identifier>
```

Пример программы:

```
INCLUDE <std.ap>
NAME R;
R:=rs(a,b,n) (
        f(a,b,n) = a,
        f(a,b,n) = f(a+b,a,n-1),
        F(n) = f(1,0,n)
);
NAME(T);
T:=F(10);
task:=(
   prog_init();
   prn T;
   yes:=1;
   while(yes,
      garb();
      applr(T,R)
   );
   prn T
);
```

Данная программа имеет типичную структуру. Основная ее часть представляет собой систему правил переписывания R. Вызов внешней процедуры prog_init() обеспечивает выполнение некоторых первоначальных действий. Операторы garb() и applr(T,R) – это вызовы внутренних процедур. Они вызывают внутренние процедуры, написанные на языке си (который является языком нижнего уровня в АПС). “applr” – это одна из базовых стратегий переписывания. Она применяет систему R к корню терма T один раз. Если применение успешно, то имя yes получает значение 1, в противном случае – 0. Таким образом while-цикл равносилен appls, но оператор сборки мусора garb() выполняется после каждого применения R. Это существенно, так как сборка мусора указывается в АПС явно.

Следующий пример – определение процедуры для стратегии applytb.

```
NAMES applytb,appl_tb_rec;
applytb:=proc(t,R)loc(Y)(
    dowhile(Y,Y:=appl_tb_rec(t,R))Y
);


appl_tb_rec:=proc(t,R)loc(Y,s,i)(
    appls(t,R);
    Y:=yes;
    forall(s=arg(t,i),
        Y:=appl_tb_rec(s,R)|/Y
    );
    t:=can(t);
    return Y
);
```

Второе определение процедуры также используется как функция, поскольку возвращает значение Y; выражение “forall(s=arg(t,i),P)” обозначает цикл по всем непосредственным подтермам s терма t и может быть записано с использованием оператора while. Использование таких выражений для сокращения записи будет обсуждаться ниже.

ФУНКЦИИ

Имеется два способа введения функций: посредством систем правил переписывания и с помощью (внешних или внутренних) процедур, которые возвращают значение. Их можно вызывать явно или путем использования в алгебраических выражениях имен определений процедур систем правил переписывания либо имен внешних процедур.

Следующий пример – это программа, которая печатает таблицу чисел Фибоначчи (заданное число N строк), останавливаясь после вывода каждых десяти строк (внутренняя процедура wait_end()). Отметим, что внутренние процедуры могут использоваться как функции, которые возвращают значение 1.

```
INCLUDE <std.ap>
NAMES tab,turn_right,prtab;

tab:=rs(x,y,z,u,n,p)(
       ( (x,y,z), 0) = (F(0)=1),
       ( (x,y,z), 1) = (F(0)=1,F(1)=1),
       ( (x,y,z), 2) = turn_right(((z,y),x)),
       ( (F(n)=x,p=y,z), u) = tab((F(n+1)=x+y,F(n)=x,(z,p=y)), u-1)
);

turn_right:=rs(x,y,z)(
    ((x,y),z) = turn_right(x,y,z)
);

prtab:=rs(x,y,n,m)(
(n>0)&(n mod 10 == 0)->(
    (F(n)=m,y) = prn(F(n)=m)&wait_ent()&y
),
    (x,y) = prn(x)&y,
    (x=y) = prn(x=y)&wait_ent()
);
NAMES T,N;
task:=(
    N:=15;
    T:=tab((F(2)=2,F(1)=1,F(0)=1),N),
    appls(T,prtab);
    wait_ent()
);
```

СТРУКТУРА АПС

Структуры данных

Основным типом данных в системе является алгебра термов (представленных в виде деревьев) $T_\Omega(Z)$, порожденная множеством первичных объектов Z и операциями сигнатуры Ω. Данная алгебра рассматривается как абсолютно свободная Ω-алгебра. Ее расширение до алгебры бесконечных (но конечно представленных, или рациональных) деревьев обозначим $T^*_\Omega(Z)$. В качестве значений имен такие структуры могут иметь общие части и могут использоваться для представления произвольных помеченных графов (термов, представленнях в виде графов). Такая возможность реализована на императивном уровне и обычно не учитывается на уровне алгебраического программирования.

Системные объекты

Системные объекты бывают трех типов: алгебраические программы (ар-модули), алгебраические модули (а-модули) и интерпретаторы.

Алгебраические программы – это тексты на языке АПЛАН. Каждая программа содержит описание сигнатуры Ω и синтаксических правил построения алгебраических выражений (термов). Она определяет также множества имен X и атомов A. Эти объекты, а также числа и строки составляют множество первичных объектов Z. Упомянутые выше три множества задают тип (Ω,X,A) ар-модуля. Типы ар-модулей частично упорядочены отношением включения (символы сигнатуры Ω рассматриваются вместе с их определениями, которые, в частности, включают местность каждого символа). Если $(\Omega,X,A)\subset(\Omega',X',A')$, то будем говорить, что ар-модуль M типа (Ω',X',A') принадлежит классу $C(\Omega,X,A)$. Два класса назовем совместимыми, если они имеют общую нижнюю грань, которая является их общим подклассом. Параметры этого подкласса содержат параметры обоих совместимых классов. Алгебраическая программа определяет также и начальные значения имен, которыми являются объекты типа $T_\Omega(Z)$.

Алгебраические модули содержат внутренние представления структур данных, определенных в ар-модулях.Они создаются посредством системных команд, которые обращаются к ар-модулям как к генераторам новых объектов.Алгебраический модуль *M*, порожденный программой *P*, наследует ее тип и начальные значения имен. Понятие а-модуля является динамическим. Он имеет состояние, которое может со временем меняться. Изменение состояние а-модуля происходит в результате выполнения (посредством интерпретации) содержащихся в нем процедур. Упорядочение на множестве типов а-модулей, также как и понятие классов *CA(Ω,X,A)* а-модулей, определяется аналогично соответствующим понятиям для ар-модулей. Таким образом, ар-модули играют по отношению к а-модулям ту же роль, что и классы по отношению к объектам в объектно-ориентированном программировании.

Состояния

Состояние а-модуля типа (Ω,X,A) слагается из двух компонентов. Первый называется состоянием памяти и является отображением $\sigma: X\rightarrow T^*_\Omega(Z)$. Второй компонент – это отношение эквивалентности на множестве вершин Ω-деревьев, являющихся значениями имен. Две вершины могут быть отождествлены согласно данному отношению эквивалентности, если они помечены одинаковыми операциями и подтермы, порождаемые этими вершинами, изоморфны. Если какие-либо две заданные вершины отождествлены, то соответствующие дочерние вершины (аргументы операции, помечающей заданные вершины) также отождествляются.

Системные интерпретаторы

Системные интерпретаторы – это программы, предназначенные для интерпретации процедур, написанных на языке АПЛАН. Они разрабатываются на языке си на основе библиотек функций и структур данных для работы с внутренним представлением системных структур данных. Соответствующее расширение языка си называется L2C. Каждый интерпретатор связан с некоторым типом (Ω,X,A), определяющим классы CI(Ω,X,A), к которым принадлежит интерпретатор, так же,как и в случае модулей. Тип определяет то, какие алгебраические модули могут выполняться с помощью данного интерпретатора: это модули, принадлежащие классу, совместимому с классом CA(Ω,X,A.

Каждый интерпретатор определяет операционную семантику языка АПЛАН для данного класса а-модулей и обеспечивает эффективную реализацию процедур, функций и статегий переписывания для систем, находящихся в данном модуле.

АПЛАН
Порождение ар-модулей

Синтаксис, используемый для создания ар-модулей, – это синтаксис алгебраических структур данных. Специальные структуры данных, такие как правила переписывания, процедуры, объекты, подчиняются единым синтаксическим правилам. Алгебраическая программа – это последовательность предложений. Предложения бывают следующих типов:

- описания имен,
- описания отметок,
- присваивания начальных значений,
- включения,
- комментарии.

```
<name description> ::= NAMES <sequence  of  names separated by "," >;
     <name> ::= <identifier>

     <mark description> ::= MARK  <sequence  of  mark  descriptions
          elements separated by "," >;
     <mark description element> ::=<mark symbol>(<arity>)
          | <mark symbol>(2, <priority>, "<infix notation>")
     <mark symbol> ::= <identifier>
     <arity> ::= <positive integer> | UNDEF
     <priority> ::= <positive integer>
     <infix notation> ::= <sequence of signs>

     <initial assignment> ::= <name> := <algebraic expression>;
     <algebraic expression>::=<primary expression>|<prefix expression>
          |<application>|<infix expression>
     <primary expression>::=<integer or rational number> | <string>
          | <empty object> | <name> | <atom> | VAL <name>
          |(<algebraic expression>)
     <empty object>::= ()
     <application>::=<algebraic  expression> <algebraic  expression>
     <prefix expression>::=<mark   symbol>(<sequence   of   algebraic
          expressions separated by ",")
     <infix expression>::=<algebraic  expression><infix notation>
          <algebraic expression>
```

В выражении вида $\omega(x_1,\ldots,x_n)$, представленном в префиксной форме, где ω – это символ отметки, количество аргументов должно быть равным местности отметки, если местность – целое число, и может принимать произвольное значение, если местность имеет значение

UNDEF. Приоритет выражения вида xωy, представленного в инфиксной форме, определяется приоритетом ω. Выражение x должно быть либо первичным, либо аппликацией, либо инфиксным выражением, приоритет которого больше приоритета ω; если y – инфиксное выражение, то его приоритет должен быть не меньшим, чем приоритет ω.

```
<inclusion> ::=INCLUDE <file name inserted into "<>">
     | INCLUDE "<file name >"
```

Комментарии выделяются скобками /* */. Строки – это последовательности символов, заключенные в кавычки “ ”.

Предложение включения `INCLUDE` *x* означает, что на место данного предложения следует поместить текст модуля *x*.

При обработке описания имен новые имена, содержащиеся в описании, добавляются к множеству имен. Описания отметок расширяют сигнатуру Ω. Отметки могут использоваться не только как алгебраические операции, но и в качестве: функциональных и предикатных символов, имен типов, конструкторов структур данных и т.п. По этой причине вместо операции или функции используется просто отметка терма. Если при описании отметки использована инфиксная запись, то отметка может использоваться для представления выражений в инфиксной форме. В этом случае приоритет позволяет обойтись без скобок. Если значение местности есть UNDEF, отметка может использоваться с произвольной положительной местностью (такую отметку можно считать связанной с бесконечным семейством операций). Имеются две предопределенные отметки, описывать которые необязательно. Это бинарная аппликация в инфиксной форме записи без обозначения знака операции и отметка ARRAY(UNDEF), используемая для построения массивов. В системе аппликация всегда имеет наивысший приоритет. Атомы – это идентификаторы, встречающиеся в программе и не описанные как имена, отметки или знаки операций в инфиксной форме записи.

Внутренним представлением алгебраических выражений являются Ω-деревья, способ построения которых очевиден. В результате обработки описания имени каждое имя получает начальное значение – пустой объект (это единственный объект, существующий изначально, то есть имеющийся до присваивания начальных значений). При присваивании начального значения вида *x:=y* значением *x* становится терм, представленный алгебраическим выражением *y*. При создании такого объекта значения вместо имен не подставляются, кроме случая, когда имя *z* следует после символа VAL. В этом случае произойдет обращение к *z*, а не к `VAL z`. Использование этого средства дает возможность отождествлять вершины внутреннего представления деревьев. Если VAL z=arg(y,p), то после такого присваивания появится эквивалентность arg(x,p)=arg(z,p)(ε).

При переопределении имени или отметки предыдущее определение удаляется. То же происходит и с начальными значениями. Таким образом, невозможно создание объектов, содержащих контуры. Действительно, после присваивания начального значения *x:=... VAL x...* все вхождения *VAL x* должны быть заменены пустым объектом, даже если x было уже присвоено значение.

Императивные программы

Синтаксис программ и внешних процедур описан выше.

В целях более точного описания процесса вычислений, а также семантики базовых предложений и передачи параметров, в настоящем разделе вводится формальная модель представления состояний модуля.

Представление состояний. Рассмотрим состояние (σ,ε) модуля типа (Ω,X,A). Согласно данному выше определению, σ есть отображение X в $T^*_\Omega(Z)$, а ε – отношение эквивалентности на множестве вершин рациональных деревьев из $T^*_\Omega(Z)$. Каждая вершина единственным образом представляется в виде пары (x,p), называемой X-вхождением, где $x \in X$, $p=(i_1,\ldots,i_m)$ – последовательность положительных целых чисел, представляющих путь в дереве (или вхождение в терм) σ(x). Существуют очевидные ограничения на составляющие этого пути,

налагаемые степенями промежуточных вершин. В языке АПЛАН эта вершина является корнем подтерма, представленного выражением вида $arg(t,(i_1,\ldots,i_m))$, где arg – это стандартная бинарная операция из Ω, $t=\sigma(x)$.

Пусть $U=\{u_1,\ldots,u_m\}$– множество символов (алфавит) равномощное множеству классов эквивалентности отношения ε. Символы U будут обозначать соответствующие классы. Выражение вида $(x,p)\in u$ обозначает, таким образом, что (x,p) принадлежит классу, который соответствует u. В то же время будем рассматривать символы U как вершины графа, который представляет структуры данных, содержащиеся в данном модуле в текущем состоянии.

Пусть $(x,p)\in u$, $\sigma(x)=t$, $arg(t,p)=\omega(t_1,\ldots,t_n)$. Тогда если $(x,(p,i))\in v_i$, $i=1,\ldots,n$, то будем писать $u\to\omega(v_1,\ldots,v_n)$ и назовем данное выражение декомпозицией вершины u. Если $arg(t,p)\in Z$, то декомпозицией вершины u является $u\to arg(t,p)$. Декомпозиция класса u не зависит от представления (x,p) этого класса и единственным образом определяется классом. Если $(x,())\in u$, то $x\to u$ назовем декомпозицией имени x. Множество декомпозиций вершин и имен называется вершинным представлением состояния (σ,ε). Нетрудно показать, что вершинное представление единственным образом определяет соответствующее состояние и для каждого состояния его вершинное представление определено с точностью до переименования вершин.

Целесообразно расширить понятие представления, считая, что правые части декомпозиции являются произвольными конечными термами над $T_\Omega(Z\cup U)$. Используя такое расширение, можно удалить некоторые лишние вершины. Вершину u назовем лишней, если она встречается в правых частях декомпозиции не более одного раза. Лишняя вершина u может быть удалена путем замены ее единственного вхождения правой частью ее декомпозиции. Представление, не содержащее лишних вершин, называется минимальным в отличие от представления, определенного выше, которое назовем максимальным.

Вычисление значений. Для алгебраического выражения t можно вычислить два вида значений. Значение первого вида обозначается val(t) и принадлежит множеству $T_\Omega(Z\cup U)$. Оно выражается с помощью минимального представления текущего состояния. Значение второго вида обозначается Val(t) и принадлежит множеству $T^*_\Omega(Z)$. Связь двух видов значений задается формулой:

$$\mathbf{Val}(t) = (\mathbf{val}(t))\tau^\infty$$

где τ^∞ есть предел τ^k, а τ – это подстановка правых частей вершины и декомпозиций имен вместо вершин и имен в правых частях всех декомпозиций вершин и имен. Значение второго вида не зависит от эквивалентности ε и используется в “инвариантных” рассуждениях об алгебраических программах. Первый вид используется для точного определения операционной семантики процедурных средств. Функция val(t) осуществляет подстановку значений имен и приводит выражение к основной канонической форме, используя интерпретаторы операций (функций) φ_ω. Формальное определение содержит следующие правила:

$$\begin{gathered}
\mathbf{isname}(x), x\to t\in r \Rightarrow \mathbf{val}(x)=t;\\
\mathbf{isfun}(f)\Rightarrow \mathbf{val}(f(x))=\varphi_{\mathbf{appl}}(\mathbf{nd}(f),\mathbf{val}(x));\\
\mathbf{val}(`(t))=t;\\
\mathbf{val}(\omega(t_1,\ldots,t_n))=\varphi_\omega(\mathbf{val}(t_1),\ldots,\mathbf{val}(t_n));\\
\mathbf{isname}(x), x\to t\in r\Rightarrow \mathbf{nd}(x)=\mathbf{nd}(t);\\
\mathbf{nd}(x)=x
\end{gathered}$$

В формулировке правил фигурирует текущее состояние, которое представлено в виде декомпозиции вершины (r). Каждое правило можно использовать лишь в том случае, когда неприменимо предыдущее. Выражение isfun(f) истинно, если F=nd(f) есть определение процедуры или системы правил переписывания. Если F – это определение процедуры, то эта процедура выполняется, при этом x служит списком фактических параметров. Значением

является то значение, которое возвращает процедура. Если F – это система правил переписывания, то она применяется к x со стратегией applr.

При выполнении процедур могут возникать побочные эффекты, в результате чего возможно изменение текущего состояния модуля. По этой причине порядок вычисления значений аргументов операции ω является существенным.

Базовые предложения. Как случае установки, так и в случае присваивания вычисляется значение $s \in T_\Omega(Z \cup U)$ правой части. Рассмотрим предложение установки. Если левая часть – это имя x, то его декомпозиция заменяется на $x \to s$. Рассмотрим предложение $arg(x,p) \to t$. Значением p должна быть последовательность $(i_1,\ldots,i_n)$ целых положительных чисел. Если $(x,i_1,\ldots,i_{n-1}) \in u$ и $u \to q \in r$, то вместо i_n-го аргумента q подставляется s. Разумеется, местность q не должна быть меньше i_n.

Присваивание x:=t функционирует иначе. Сперва, если s – вершина, вместо s берется правая часть декомпозиции данной вершины. Если правая часть декомпозиции x не является вершиной, то действие присваивания равнозначно действию предложения установки. В противном случае, если $x \to u$, $u \to q \in r$, декомпозиция $u \to q$ заменяется на $u \to s$.

Внешние вызовы. Формальные параметры и локальные имена временно добавляются к модулю в качестве имен. Формальным параметрам присваиваются значения фактических параметров, а затем выполняется тело процедуры, после чего формальные параметры и локальные имена удаляются из модуля. Предложение возврата порождает значение, которое используется, когда процедура встречается в алгебраическом выражении.

Внутренние вызовы. Обращение к процедурам реализовано на уровне языка си. Количество формальных параметров и то, как их передавать (вычислять значения или нет), определяется согласно спецификациям внутренней процедуры.

Выполнение программ. Выполнение программ осуществляется обычным образом. Предложения выполняются одно за другим. Условные предложения и циклы имеют обычный смысл. Одной интересной особенностью является использование расширения императивного языка АПЛАН. Если интерпретатор программы встречает неизвестное базовое предложение, он делает попытку преобразовать его с помощью системы переписывания compile, которая является стандартным средством АПС. Пользователь может изменить или расширить систему compile. Типичный состав системы такой:

```
NAMES _p,_sp,_ptr;
_p :=nil;
_sp:=nil;
MARK case(2);
```

```
compile:= rs(x,y,z,u,i)(
           define x = `(defcall `(x)),
     (arg(arg(x,y),z)-->u) =
                compile(arg(x,conc(y,z))-->u),
     (arg(x,y)--> z  ) = `(set(x,y,z)),
     (    x(i)--> z  ) = `(set(x,i,z)),
     (   (x.y)--> z  ) = `(set_comp(x,y,z)),
     (      x --> z.u) =   set_obj_comp `(x-->z.u),
     (      x --> z  ) = `(setname(x,z)),

       dowhile(x,y) = (x, while(y,x)),
       dowhile x y  = (x, while(y,x)),
       for(x,y,z,u) = (x, while(y,(u,z))),
       loop(x)      = while(1,x),

       forall(x=arg(y,i),z) =
                for(i-->1,i<= `(ART(y)),i:=i+1,
                     x-->arg(y,i); z
```

```
          ),
    forallw(x=arg(y,i),u,z) =
          for(i-->1,(i<= `(ART(y))) & u,i:=i+1,
               x-->arg(y,i); z

          ),
    forall(x in list y, z) =
         (_sp-->(_p,_sp);
          _p-->y;
          while( `(is_type( _p,((),()) ) ),
               x -->arg(_p,1);
               _p-->arg(_p,2);
               z
          );
          x-->_p;
          _p -->arg(_sp,1);
          _sp-->arg(_sp,2);
          z
         ),
    forall(x in set y, z) =
         (_sp-->(_p,_sp);
          _p-->y;
          while(`(is_type( _p,((),()) ) ),
               x -->arg(_p,1);
               _p-->arg(_p,2);
               z
          );
          _p -->arg(_sp,1);
          _sp-->arg(_sp,2)
         ),
    forallw(x in list y, u, z) =
         (_sp-->(_p,_sp);
          _p-->y;
          while(`(is_type( _p,((),()) )&u ),
               x -->arg(_p,1);
               _p-->arg(_p,2);
               z
          );
          x-->_p;
          _p -->arg(_sp,1);
          _sp-->arg(_sp,2);
          u->z
         ),
    forallw(x in set y, u, z) =
         (_sp-->(_p,_sp);
          _p-->y;
          while(`(is_type( _p,((),()) )&u ),
               x -->arg(_p,1);
               _p-->arg(_p,2);
               z
          );
          _p -->arg(_sp,1);
          _sp-->arg(_sp,2)
         ),
    case(x,y->z;u)= starg(x,2,y)->z else case(x,u),
    case(x,y->z  )= starg(x,2,y)->z,
    case(x,   z  )=               z,

    branch(y->z;u)= y->z else branch(u),
    branch(y->z  )= y->z,
    branch(   z  )=    z
);
```

СТРАТЕГИИ

Базовые стратегии

Основой стратегий переписывания в АПС служат две основные внутренние процедуры applr и appls. Предложение applr(t,R) осуществляет попытку применить одно из правил системы R к терму t. Если применимых правил нет, то имя `yes` принимает значение 0. В противном случае применяется первое применимое правило, а имя `yes` принимает значение 1. Простые правила применяются обычным образом: сначала левая часть унифицируется с t путем сопоставления по образцу, в случае успеха производится замена переменных левой части и t заменяется правой частью. Перед заменой редекса правая часть приводится к основной канонической форме с помощью правил подобно тому, как вычисляются значения, но без подстановки значений вместо имен.

Для более точного описания рассмотрим предложение applr(x,y) и положим t=val(x), R=val(y). Пусть z – произвольное имя с декомпозицией z→t, $x_1,\ldots,x_n$ – переменные системы R, l=r – правило, левая часть которого l унифицируется с t, а $u_1,\ldots,u_n$ – z-вхождения, соответствующие значениям переменных $x_1,\ldots,x_n$ (вершинам некоторого представления текущего состояния). Если правило не является леволинейным, то есть l имеет более одного вхождения какой-либо из переменных, то рассматривается первое вхождение этой переменной. Подстановка правой части, таким образом, равнозначна присваиванию z:=CAN(r), где функция CAN определена следующими правилами:

$$\mathbf{CAN}(x_i) = u_i;$$
$$\mathbf{isfun}(f) \Rightarrow \mathbf{CAN}(f(x)) = \varphi_{\mathbf{appl}}(\mathbf{nd}(f), x);$$
$$\mathbf{CAN}(\text{`}(s)) = s[x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n];$$
$$x \in Z \Rightarrow \mathbf{CAN}(x) = x;$$
$$\mathbf{CAN}(\omega(t_1, \ldots, t_n)) = \varphi_\omega(\mathbf{CAN}(t_1), \ldots, \mathbf{CAN}(t_n));$$

Условные правила применяются к термам следующим образом. Сначала выполняется унификация. В случае успеха условие приводится к основной канонической форме путем вычисления функции CAN. Если результат равен единице, применение правила продолжается обычным образом. В противном случае применение прекращается.

Предложение appls(t,R) вызывает applr(t,R), пока yes=1. Заметим, что если имя системы переписывания встречается в правой части системы, то система применяется рекурсивно. По этой причине стратегия, реализуемая процедурой applr, называется рекурсивной стратегией переписывания. Стратегия, реализуемая appls, называется итеративной.

Канонические формы

При алгебраических вычислениях принято рассматривать алгебраические выражения с точностью до конгруэнции, согласованной с тождествами алгебры, определяющей предметную область. В АПС эта идея реализована с помощью функции CAN. Эта функция определяет эквивалентность t=t′(CAN)⇔CAN(t)=CAN(t′), которая должна быть конгруэнцией: $t_1=t'_1(\mathrm{CAN}),\ldots,t_n=t'_n(\mathrm{CAN})\Rightarrow\omega(t_1,\ldots,t_n)=\omega(t'_1,\ldots,t'_n)(\mathrm{CAN})$. Это равносильно существованию функции `can`, такой что

$$\mathbf{CAN}(\omega(t_1, \ldots, t_n)) = \mathbf{can}(\omega(\mathbf{CAN}(t_1), \ldots, \mathbf{CAN}(t_n))).$$

Функция CAN, определенная выше, не задает конгруэнцию, так как этому мешают операции аппликации и ссылки. Но если последние игнорируются, то тогда CAN задает конгруэнцию. Действительно, функция `can` определяется с помощью интерпретаторов операций:

$$\mathbf{can}(\omega(t_1,\ldots,t_n)) = \varphi_\omega(\omega(t_1,\ldots,t_n))$$

Другим важным свойством CAN является то, что она должна определять каноническую форму данной конгруэнции: t=CAN(t)(CAN). Это свойство равносильно идемпотентности CAN: CAN(CAN(t))=CAN(t). Если CAN обладает свойством идемпотентности, она называется корректной.

Назовем терм t нормализованным относительно can, если can(s)=s для всякого подтерма s терма t. Корректность CAN обеспечивается следующим условием: если $t_1,\ldots,t_n$ нормализованы относительно can, то can($\omega(t_1,\ldots,t_n)$) также нормализован.

Это условие сводит проверку корректности CAN к анализу тех свойств интерпретаторов операций, которые связаны с нормализацией.

В качестве простого примера, реализованного в большинстве системных интерпретаторов АПС, можно привести интерпретаторы операций, которые вычисляют выражения, содержащие постоянные величины и арифметические и булевы операции, такие как x+0=x или x∨1=1.

Определения операций, которые встречаются наиболее часто, помещены в файл std.ap. Ниже приводится содержимое этого файла с комментариями по поводу толкования и использования операций (отметок).

```
MARKS
     /* Arithmetical and algebraic operations and functions */

     POW( 2, 60, "^"),   /* power x ^ y                          */
     MOD( 2, 59,"mod"),  /* residual x mod y                     */
     Mult( 2, 58, "*"),  /* multiplication x * y                 */
     DIV( 2, 57, "/"),   /* division x / y                       */
     MLT( 2, 56, "$"),   /* uninterpreted                        */
     SUB( 2, 55, "-"),   /* subtraction x - y                    */
     ADD( 2, 54, "+"),   /* addition  x + y                      */

     /* Predicates */

     LE ( 2, 40, "<="),  /* less or equal x <= y                 */
     LS ( 2, 40, "<"),   /* less x < y                           */
     ME ( 2, 40, ">="),  /* more or equal x >= y                 */
     MR ( 2, 40, ">"),   /* more x > y                           */
     EQ  ( 2, 11, "=="), /* equality of numbers x == y           */
     EQU ( 2, 11, "=" ), /* uninterpreted                        */

     /* Logical connections */

     ~ (  1, 30),        /* logical negation  ~(x)               */
     AND( 2, 29, "&"),   /* logical and x & y                    */
     OR ( 2, 28, "|/"),  /* logical or x |/ y                    */
     IFF( 2, 27, "<=>"), /* logical equivalence x <=> y          */

     /* L2B operations */

     SET  ( 2, 20, "-->" ),    /* set statement x-->y                */
     ASS  ( 2, 20, ":="  ),    /* assignement statement x:=y         */
     ELSE ( 2, 19, "else"),    /* used in conditional statement:
                                          x -> y else z           */
     IF   ( 2, 18, "->"  ),    /* implication x -> y and separator
                                   in conditional statements      */
     do   (1),                 /* do statement: do(p)                */
     while(2),                 /* while statement: while(x,y)        */



     /* Separators */
```

```
    comma( 2,  7, ","),
    LL ( 2,  5, ";"),

    /* Special functions */

    `(1),               /* `(x) = x                          */
    arg (2),            /* arg(f(x1,...,xn)) = xi            */
    ART (1),            /* ART(x) = arity of x               */
    CAN (1),            /* Basic canonical form              */
    IFTH(3),            /* IFTH(x,y,z) = if x then y else z  */
    vl  (1),            /* vl(x) = copy of value of name x          */
    VL  (1),            /* VL(x) = substitution of values to term x  */
    intr(2),
    copy(1),            /* copy(x) is copy of x if x has no loops  */
    mrg (1);



NAMES indprn,indprog,indpl,
      indts,indtm,task,nil,
      extcan;

NAMES in,out,stack,proc_atom,rs_atom;


in   :=0;
out  :=0;
stack:=0;
nil:=nil;
extcan:=nil;
```

АК-операции

В АПС можно вводить два вида ассоциативных и коммутативных операций (ак-операций): арифметические и булевоподобные ак-операции.

Арифметическая ак-операция ω вводится вместе с операцией итерации φ и двумя константами (введение которых не является обязательным): единичным элементом *e* и нулевым элементом *a*. Кроме ассоциативности и коммутативности выполняются следующие тождества:

$$(x\varphi y)\omega(x\varphi z) = x\varphi(y+z);$$
$$x\omega e = x;$$
$$x\omega a = a;$$
$$x\varphi 0 = e;$$
$$x\varphi 1 = x;$$
$$e\varphi x = e;$$
$$a\varphi x = a$$

В качестве булевоподобных операций используется унарная операция отрицания ν и, помимо единичного и нулевого элементов, вводится элемент o. Для булевоподобных операций выполняются тождества:

$$x\omega x = x;$$
$$x\omega e = x;$$
$$x\omega a = a;$$
$$\nu(\nu(x)) = x;$$
$$x\omega\nu(x) = o$$

Для хранения информации об ак-операциях используется структура данных, которая является значением стандартного имени ac_list и представляет собой массив ак-описаний. Каждое описание – это пятерка. Описание арифметических ак-операций имеет вид (()ω(),()φ(),*e*,*a*,nil), для булевоподобных – (()ω(),ν(),*e*,*a*,o). Если какая-либо из трех констант не используется, в соответствующем месте следует записать символ nil. В качестве примера рассмотрим описание:

```
ac_list:= ARRAY(
           (()+ (), ()$ (), 0,     nil, nil),
           (()* (), ()^ (), 1,       0, nil),

           (()& (), ~(()),  1,       0,   0),
           (()|/(), ~(()),  0,       1,   1),

           (()><(), ()>^(), Delta, nil, nil),
           (()||(), ()|^(), Delta,   0, nil)
);
```

Первые две операции арифметические. Они интерпретируются обычным образом. Следующие две являются булевыми и интерпретируются также обычным образом. Последние две операции – это соединение и параллельная композиция, применяемые при реализации алгебры параллельных процессов.

Поддержку ак-операций обеспечивают функция mrg и две внутренние процедуры merge и ord. Они используются для приведения выражений, содержащих ак-операции, к ак-канонической форме, которая обеспечивает упорядочение и приведение подобных членов для арифметических операций, а также упрощения для обоих типов ак-операций. Функция mrg и процедура merge осуществляют приведение к канонической форме выражений вида xωy, где x и y уже находятся в канонической форме.

Встроенные стратегии

Стратегии, описываемые в настоящем разделе, специфицированы в языке АПЛАН с помощью простых рекурсивных процедур и реализованы на языке си как внутренние процедуры. Все стратегии применимы только к термам, представленным в виде ациклических ориентированных графов.

```
ntb:=proc(t,R)loc(s,i)(                  /* top-bottom */
    appls(t,R);
    forall(s=arg(t,i),
        ntb(s,R)
    );
    t:=can(t)
);

nbt:=proc(t,R)loc(s,i)(                  /* bottom-up */
    forall(s=arg(t,i),
        nbt(s,R)
    );
    appls(t,R);
    t:=can(t)
);
```

Одна из стратегий осуществляет одномаршрутный обход снизу вверх, вторая – обход сверху вниз. Они соответствуют простейшим случаям вычислений по имени и по значению. Результатом применения ntb является терм в базовой канонической форме.

```
applytb:=proc(t,R)loc(Y)(
    dowhile(Y,Y:=appl_tb_rec(t,R))Y
);
```

```
appl_tb_rec:=proc(t,R)loc(Y,s,i)(
    appls(t,R);
    Y:=yes;
    forall(s=arg(t,i),
        Y:=appl_tb_rec(s,R)|/Y
    );
    t:=can(t);
    return Y
);
applybt:=proc(t,R)loc(Y)(
    dowhile(Y,Y:=appl_bt_rec(t,R))Y
);

appl_bt_rec:=proc(t,R)loc(Y,s,i)(
    Y:=0;
    forall(s=arg(t,i),
        Y:=appl_bt_rec(s,R)|/Y
    );
    appls(t,R);
    Y:=yes|/Y;
    t:=can(t);
    return Y
);
```

Обе стратегии являются финальными в том смысле, что терм-результат нормализован, если он определен (то есть если стратегия завершается).

```
ntr:=proc(t,R)loc(s,i)(
    yes:=1;
    while(yes,
        t:=can(t);
        appls(t,R);
        yes:=0;
        forallw(s=arg(t,i), ~(yes),
            ntr(s,R)
        )
    );
    t:=can(t);
    appls(t,R)
);
```

Эта стратегия в некоторых случаях оказывается эффективнее прочих.

```
lmt:=proc(t,R)loc(Yes)(
    dowhile(Yes:=lmt_rec(t,R))Yes
);

lmt_rec:=proc(t,R)loc(Yes,s,i)(
    t:=can(t);
    appls(t,R);
    yes->return(1);
    forall(s=arg(t,i),
        Yes:=lmt_rec(s,R);
        Yes->return(1)
    );
    t:=can(t);
    return(0)
);
```

Стратегия “самый левый-внешний” хорошо известна, она полна для лямбда исчисления.

```
ntb2:=proc(t,R)loc(s,i)(
          appls(t,R);
          (ART(t)>0)->ntb2 (arg(t,1),R);
          t:=can(t);
          (ART(t)>0)->ntb2 (t,R)
);
```

```
can_right:=proc(x,R)(
    appls(x,R);
    while(yes,
        (ART(x)>1)->can_right(arg(x,2),R);
        appls(x,R)
    )
);
```

Эти две стратегии используются для обработки списков, особенно для вычислений на ветвях дерева поиска.

```
NAME ntb0;
ntb0:=proc(t,R)loc(s,i)(
    applr(t,R);
    forall(s=arg(t,i),
        ntb0(s,R)
    )
);

ntb0s:=proc(t,R)loc(s,i)(
    appls(t,R);
    forall(s=arg(t,i),
        ntb0(s,R)
    )
);
```

Эти две стратегии используются в том случае, когда не требуется приведение к каноническому виду, например, при решении задач, связанных с обработкой языков.

```
can_ord:=proc(t,R1,R2)loc(s,i)(         /* top-bottom R1, bottom-up R2 */
    t:=can(t);
    appls(t,R1);
    forall(s=arg(t,i),
        can_ord(s,R1,R2)
    );
    can_up(t,R2)
);
can_up:=proc(t,R)loc(s,i)(
    appls(t,R);
    while(yes,
        forall(s=arg(t,i),
            can_up(s,R)
        );
        appls(t,R)
    );
    t:=can(t);
    merge(t)
);
```

Далее приводится одна из наиболее важных эффективных стратегий для преобразований с ак-операциями. Она использует две системы правил переписывания. Первая применяется, когда осуществляется обход сверху вниз, вторая – при движении снизу вверх. Одним из недостатков стратегии, проявляющимся в некоторых случаях, является то, что она не всегда финальна. Последующие стратегии финальны для более широкого класса термов по сравнению с `can_ord`. Они используют вспомогательную структуру данных (имя `rpar`, значением которого является правая часть правила, примененного стратегией `applr`).

```
NAME rpat;
can_ord2:=proc(t,R,S)loc(i,ar)(
    t:=can(t);
    appls(t,R);
```

```
    ar=ART(t);
    for (i:=1,i<=ar,i:=i+1,
        can_ord2(arg(t,i),R,S)
    );
    can_up2(t,S,0)
);
can_up2:=proc(t,S,pat)loc(s,i,ar,ar0)(
    dowhile(
        ~(equ(pat,0))->(
            ar0 = ART(pat);
            ar  = ART(t);
            (ar0==0)->return 0;
            for(i:=1, i<=ar, i:=i+1,
                (i<=ar0)->s-->i else s-->ar0;
                can_up2(arg(t,i),S,arg(pat,s))
            )
        );
        applr(t,S);
        yes->pat-->rpat
    )~(yes);
    t:=can(t);
    merge(t)
);

can_ord3:=proc(t,R,S)loc(i,ar)(
    t:=can(t);
    appls(t,R);
    ar=ART(t);
    for (i:=1,i<=ar,i:=i+1,
        can_ord3(arg(t,i),R,S)
    );
    can_up3(t,S,0)
);
can_up3:=proc(t,S,dep)loc(i,ar)(
    dowhile(
        (dep>1)->(
                        ar = cl_aryty(t);
                        for(i = 1; i <= ar; i++) {
                                can_up3(cl_arg(t,i), S, dep-1);
                        }
        )
        t:=can(t);
        merge(t);
        applr(t,S);
        yes->dep:=depth(rpat)
    )~(yes)
);
depth:=proc(t)loc(i,ar,m,n)(
    m=0;
    ar=ART(t);
    for(i:=1,i<=ar,i:=i+1
        n:=depth(arg(t,i))+1;
        (n>m)->m:=n
    );
    return(m)
);
can_atr:=proc(t,R1,R2,m)loc(s,i)(   /* top-bottom R1, bottom-up R2 */
    appls(t,R1);
    forall(s=arg(t,i),
        can_atr(s,R1,R2,m)
    );
    can_atr_up(t,R2,m,0)
);
can_atr_up:=proc(t,R,m,n)loc(s,i)(
```

```
    (n>0)->(
        forall(s=arg(t,i),
            can_atr_up(s,R,m,n-1)
        )
    );
    appls(t,R);
    (yes)->(
        forall(s=arg(t,i),
            can_atr_up(s,R,m,m)
        )
    )
);Strategy can_atr is similar to attribute grammars.
lisp:=proc(t,R)loc(ar,i)(
    dowhile(
        ar=ART(t);
        (ar>0)->for(i:=1,i<=ar,i:=i+1,
            lisp(arg(t,i),R);
            t:=can(t)
        );
        applr(t,R)
    )yes
);
```

Стратегия в духе ЛИСПа.

```
stb:=proc(t,R)loc(ar)(
    t:=can(t);
    applr(t,R);
    yes->(
        ar=ART(t);
        for (i:=1,i<=ar,i:=i+1,
            stb(arg(t,i),R)
        )
    )
);
```

Эта стратегия аналогична ntb, но вместо appls использует applr. Новые стратегии можно разрабатывать путем преобразования спецификаций встроенных стратегий и введением новых свойств. Кроме того, спецификации встроенных стратегий полезны при доказательстве свойств алгебраических программ (обычно путем структурной индукции).

АЛГЕБРА ЛОГИКИ

Далее приводится в полном объеме пример алгебраической программы, хранящейся в файле log.ap. Программа содержит некоторые средства обработки пропозициональных формул.

```
"INCLUDE <std.ap>
INCLUDE <ext.ap>"
MARK subs(2);
/*         Rules for eliminating <=>, ->, and de Morgan rules          */
NAME R;

R :=rs(x,y)(
        (x <=>  y) = ((x -> y) & (y -> x)),
        (x  ->  y) = (~(x) |/ y),
        ~( ~(x)  ) = x,
        ~(x |/  y) = (~(x) & ~(y)),
        ~(x  &  y) = (~(x) |/ ~(y)),
        ~(x <=> y) = (~(x -> y) |/ ~(y -> x)),
        ~(x  -> y) = (x & ~(y))
```

```
);

/*                          Rules for CNF
*/

NAMES R1,Q1;

R1 :=rs(x,y,z,u,v)(
            (x & y |/ z & u) & v =
                  (x |/ u) & (y |/ z) & (y |/ u) &(x |/ z) & v,
            (x & y |/ z) & u = ( y |/ z) & (x |/z) & u,
            (x |/ y & z) & u = ( x |/ z) & (x |/ y)  & u,
            x & y |/  z  & u = ((x |/ z) & (y |/ z)) & (x |/ u) &(y |/ u) ,
            x & y |/  z      = ( x |/ z) & (y |/ z),
            x |/ y &  z      = ( x |/ y) & (x |/ z)
     );

Q1 :=rs(x,y,z)(
        (x  & y) |/ z  = (x |/ z) & (y |/ z) ,
         x |/(y  & z) = (x |/ y) & (x |/ z),
        (x |/ y) |/ z  =  x |/ y  |/ z
     );

/*                          Rules for DNF                */
NAMES DR1,DQ1;

DR1 :=rs(x,y,z,u,v)(
           (x |/ y) & (z |/ u) |/ v =
                   x & u |/ y & z |/ y & u |/ x & z |/ v,
            (x |/ y) & z |/ u =  y & z |/ x &z |/ u,
            x & (y |/ z) |/ u =  x & z |/ x & y  |/ u,
            (x |/ y) & z |/ u = (x & z |/ y & z) |/ x & u |/ y & u ,
            (x |/ y) &  z     =  x & z |/ y & z,
            x & (y |/  z)     =  x & y |/ x & z
     );

DQ1 :=rs(x,y,z)(
        (x  |/ y)& z  = (x & z) |/ (y & z) ,
         x &(y  |/ z) = (x & y) |/ (x & z),
        (x & y)& z  =  x & y  & z
     );

NAME cnf;
cnf:=proc(x)(
       x-->copy(x),
       ntb(x,R),
       can_ord(x,R1,Q1),
       return(x)
);

NAME dnf;
dnf:=proc(x)(
       x-->copy(x),
       ntb(x,R),
       can_ord(x,DR1,DQ1),
       return(x)
);

NAME deM;
deM :=rs(x,y)(
        ~( ~(x)  ) = x,
        ~(x |/  y) = deM(~(x)) & deM( ~(y)),
        ~(x  &  y) = deM(~(x)) |/ deM(~(y))
);
```

```
/*           Rules for proving identities in logic          */
NAME I1;
I1:=rs(x,y,z)(

         1       ->      0  = 0,
         0       ->      x  = 1,
         x       ->      x  = 1,
         x       ->      1  = 1,
         x       -> y |/ z  = x & deM( ~(y) ) -> z,
         x       -> y &  z  =(x -> y) &( x -> z),
         x |/ y  ->      z  =(x -> z) &(  y -> z),
         x &  y  ->    ~(z) = subs(z=1,x)  -> deM( ~(subs(z=1,y))),
         x &  y  ->      z  = subs(z=0,x)  -> deM( ~(subs(z=0,y))),
         x       ->      y  = 0
);

/*
-------------------------------------------------------------------------
                        Strategy ntb2
NAME ntb2;

ntb2:=proc(t,R)loc(s,i)(
          appls(t,R);
          (ART(t)>0)->ntb2 (arg(t,1),R);
          t:=can(t);
          (ART(t)>0)->ntb2 (t,R)
);

-------------------------------------------------------------------------
*/

NAME is_id;

is_id:=proc(x)(
        x-->copy(x);
        ntb(x,R);
        x-->(1->x);
        ntb2(x,I1);
        return(x)
);

        /* printing list constructed by means of binary operation y */


NAME prn_list;
prn_list:=proc(x,y)loc(z,n)(
               line();
               z-->x;
               n:=0;
               while(equ(type(z),y),
                    n:=(n+1) mod 10;
                    print(arg(z,1));
                    line();
                    (n==0)->wait_ent();
                    z-->arg(z,2)
               );
              print(z);
              put("\nend of list")
          );
```

Файл ext.ap содержит определения отметок, а также имен ac_list и compile, которых нет в файле std.ap. Функция subs (интерпретируемая бинарная инфиксная операция) определяет

подстановки: subs((x=a,y=b,…),z) осуществляет в z подстановку a,b,… вместо первичных объектов x,y,…. Процедура prn_list используется для печати больших списков (например, формул, представленных в конъюнктивной или дизъюнктивной нормальной форме) в пошаговом режиме.

В данном файле определены три функции: cnf и dnf осуществляют приведение пропозициональных формул соответственно к конъюнктивной и дизъюнктивной нормальной форме, is_id(x) принимает значение 1, если x является тавтологией, и 0 в противном случае.

Стратегия can_ord используется для приведения логических формул. В обоих случаях используются две системы правил переписывания. На самом деле можно взять две одинаковые стратегии. Однако использование дополнительных правил ускоряет приведение.

ПОЛИНОМЫ

Следующий пример относится к области компьютерной алгебры. Рассмотрим ap-модуль `pl_ac.ap`, который задает алгоритм приведения полиномов, представленных в обычной форме, то есть построенных с помощью арифметических операций сложения и умножения (имеются и другие представления: рекурсивные представления, представления, в которых для мономов используется векторная и экспоненциальная форма, векторная форма для коэффициентов и т.п.).

```
INCLUDE <rat.ap>
/*
      Expanding polynomials represented in natural form
*/
NAMES rdn, canpl;
NAMES pw,pow,bn;
/*                    Specification of canpl            */

canpl:=proc(t)(can_ord(t,rdn,rdn));

rdn:=rs(q,x,y,z,u,k,n,a)(

     isnum(x) -> (x*y = y$x),
     isnum(y) -> (x*y = x$y),
     isnum(x) -> (x$y = y*x),

                 x - y = x+(-1)*y,

                 x $ 0 = 0,
                 x $ 1 = x,
           (x + y) $ z = x $ z + y $ z,
           (x $ y) $ z = x $ (y * z),

               (x+y)*z = x*z+y*z,
               x*(y+z) = x*y+x*z,

               (x$y)*(z$u) = (x*z)$(y*u),
               (x$y)* z    = (x*z)$y,
                x   *(y$z) = (x*y)$z,

               (x$y)^n = x^n$y^n,
               (x*y)^n = x^n*y^n,
               (x^y)^n = x^(y*n),
(isint(n)&(n>0))->       (       (x+y)^n = pw((x+y)^n) )

);


pw:=proc(t)(
    can_ord(t,pow,rdn);
```

```
    return(t)
);

pow:=rs(x,y,z,n,k,q,a)(
    n>1 ->   ((x+y)^n = bn(1,x,y,1,n,n) +   y^n),
    n>1 -> (q*(x+y)^n = bn(q,x,y,1,n,n) + q*y^n)
);

bn:=rs(q,x,y,k,n,a)(
    (q,x,y,n,n,a) = q*x^n,
    (q,x,y,k,n,a) = (x^k*q$a)*y^(n-k)+bn(q,x,y,k+1,n,(a*(n-k))/(k+1))
);

NAME T;

T:=((a+b)*(b+1/2)+(a-5*b)*(b+c))*(a+b+c)*(b+c-d);

task:=(
    canpl(T);
    prnpl(T)
);
```

Функция canpl реализована на самом деле как интерпретатор унарной операции, но ее первоначальная реализация – это простая алгебраическая программа. Чтобы использовать в качестве коэффициентов вещественные числа, применяется дополнительный механизм для полиморфных функций.

Для понимания этого механизма необходимо обратиться к более подробному рассмотрению интерпретаторов операций. Имеется три уровня интерпретаторов:

- внутренние интерпретаторы, реализованные в виде программ на языке си;
- интерпретаторы полиморфных операций, определенных таблицей `polist`, находящейся в распоряжении пользователя;

Вызов этих интерпретаторов осуществляется с помощью функции can, аргументом которой является вершина состояния модуля с декомпозицией $t \to \omega(t_1,\ldots,t_n)$. Операция ω определяет внутренний интерпретатор φ'_ω, написанный на языке си. Применение этого интерпретатора к терму t может быть успешным или нет. В случае успеха терм t заменяется на $\varphi'_\omega(t)$, иначе осуществляется вызов полиморфного интерпретатора. При этом используется таблица polist. В рассматриваемом здесь примере для обращения к таблице используется файл rat.ap.

```
"INCLUDE <std.ap>
INCLUDE <ext.ap>"

NAME  rational,
      polist,simp,israt,
      deration, prn_rat;

MARKS rat(2),
      quote(2,57,"//"),   /* integral part of a quotient */
      GCD(2);             /* greatest common devisor     */

rational:=1;

polist:=ARRAY(
      /* Polymorphic functions */
            (()+(),add),
            (()-(),sub),
            (()*(),mul),
            (()/(),div),
            (()^(),pwc),
            (()<(),lt),
            (()>(),gt),
```

```
            (()<=(),lte),
            (()>=(),gte),
      /* Polymorphic data constructors */
            (rat((),()),0)
);

extcan:=nil;
simp:=rs(x,y,z)(
     rat(x,y)     = simp(rat(x,y),GCD(x,y)),
     (z==y) -> ((rat(x,y),z) = x//z),
     (rat(x,y),z) = rat(x//z,y//z)
);

deration:=rs(x,y)(
      x $ y = y*x,               (add spaces to "$" !!!)
      rat(x,1) = x,
      rat(x,y) = x / y
);

prn_rat:=proc(T)loc(S)(
       T-->copy(T);
       rational->(
          S:=T;
          markcan(()/(),can0);
          nbt(S,deration);
          prn(S);
          markcan(()/(),div_rat)
       ) else prn(T)
);

israt:=rs(x,y)(
      rat(x,y)     = 1,
                 x = 0
);

/* ######################## Specifications ##############################

cor_sg:=rs(x,y)(
     (y<0) -> (rat(x,y) = rat((-1)*x,(-1)*y))
);

GCD:=rs(x,y)(
       (x,0) = x,
       (0,x) = x,
      (x >  y)->((x,y) = GCD(y,x mod y)),
      (x <= y)->((x,y) = GCD(x,y mod x))
);

Abs:=rs(x)(
       (x<0)->(x = (-1)*x)
);

add:=rs(x1,x2,y1,y2)(
     rat(x1,y1)+rat(x2,y2) = simp(rat(x1*y2+x2*y1,y1*y2)),
     isint(x1)->(x1+rat(x2,y2) = simp(rat(x1*y2+x2,y2))),
     isint(x2)->(rat(x1,y1)+x2 = simp(rat(x1+x2*y1,y1)))
);


subt:=rs(x1,x2,y1,y2)(
     rat(x1,y1)-rat(x2,y2)     = simp(rat(x1*y2-x2*y1,y1*y2)),
     isint(x1)->(x1-rat(x2,y2) = simp(rat(x1*y2-x2,y2))),
     isint(x2)->(rat(x1,y1)-x2 = simp(rat(x1-x2*y1,y1)))
);
```

```
mul:=rs(x1,x2,y1,y2)(
     rat(x1,y1)*rat(x2,y2)     = simp(rat(x1*x2,y1*y2)),
     isint(x1)->(x1*rat(x2,y2) = simp(rat(x1*x2,y2))),
     isint(x2)->(rat(x1,y1)*x2 = simp(rat(x1*x2,y1)))
);



div:=rs(x1,x2,y1,y2)(
     rat(x1,y1)/rat(x2,y2) = cor_sg(simp(rat(x1*y2,x2*y1))),
     isint(x1)->(x1/rat(x2,y2) = cor_sg(simp(rat(x1*y2,x2)))),
     isint(x2)->(rat(x1,y1)/x2 = cor_sg(simp(rat(x1,y1*x2)))),
     (isint(x1)&isint(x2)) ->
             (x1/x2 = cor_sg(simp(rat(x1,x2))))
);

pwc:=rs(x1,x2,y1)(
     (isint(x2)&(x2>0))->
          (rat(x1,y1)^x2 = simp(rat(x1^x2,y1^x2))),
     (isint(x2)&(x2<0))->
          (rat(x1,y1)^x2 = simp(rat(y1^((-1)*x2),x1^((-1)*x2))))

);

lt:=rs(x,y)(
     (x < y) = tst(subt(y-x))
);

gt:=rs(x,y)(
     (x > y) = tst(subt(x-y))
);


tst:=rs(x,y)(
       rat(x,y) = (x>0),
             x  = (x>0)
);

 ########################################################################## */
```

Таблица polist содержит строки двух типов: полиморфные операции с именами соответствующих интерпретаторов и полиморфные конструкторы данных. Вызов полиморфного интерпретатора происходит только в том случае, когда один из аргументов t образован с помощью полиморфной структуры данных. Именем полиморфного интерпретатора может быть атом, именующий внутреннюю процедуру, или имя функции, написанной пользователем на языке АПЛАН. Если полиморфный интерпретатор получает отрицательный результат, то к терму t в качестве интерпретатора третьего уровня применяется система extcan.

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

В данном разделе рассмотрим решатель задач общего назначения, развитый в духе CLP (constraint logic programming). Описываемый здесь решатель осуществляет поиск решения задачи на предметной области, определенной множеством аксиом, представленных в виде бесквантворных формул, переменные которых предполагаются связанными кванторами всеобщности. Каждая формула является либо элементарной, то есть атомом или отрицанием атома, либо имеет вид импликации $P \Rightarrow Q$, где P – произвольная формула, а Q – элементарная формула. Для решения задач решатель использует прологоподобную стратегию. Аксиомы,

содержащие импликацию, используются, как правила пролога или правила сведения задачи к подзадачам.

Сигнатура предикатов может содержать предикат равенства. Аксиомы определяют некоторую теорию равенства (эквациональную теорию). Она состоит из всех равенств, которые являются следствиями аксиом. Решатель может использовать специальные алгоритмы решения уравнений в заданной теории и, таким образом, не работать непосредственно с аксиомами теории. То же касается и некоторых других предикатов.

Будем обозначать абсолютно свободную алгебру термов с заданной сигнатурой, порожденную множеством констант A и множеством переменных Z T(A,Z) (инициальная алгебра с множеством $A \cup Z$ операций арности 0). Предполагается, что переменные, встречающиеся в аксиомах, принадлежат множеству W={W(1),W(2),…}.

Элементарной задачей назовем пару (P,X), где P – произвольная предикатная формула без кванторов, свободные переменные которой принадлежат множеству V={V(1),V(2),…}, X есть V-контекст со значениями в T(A,V), то есть подстановка вида $\{V(1)\leftarrow t_1,\ldots,V(n)\leftarrow t_n\}$, $t_1,\ldots,t_n \in T(A,V)$. Решение элементарной задачи (P,X) – это новый V-контекст $Y=\{V(1)\leftarrow s_1, \ldots,V(n)\leftarrow s_m\}$ такой, что $m \geq n$, s_i является примером t_i, i=1,…,n, а PY является следствием аксиом. Точный смысл понятия “следствие” определяется операционной (исчислением) или денотационной (теоретико-множественной) семантикой языка.

В ходе решения задачи образуется множество подзадач. По этой причине можно говорить о комплексных задачах, соответствующих множествам элементарных задач и их решений. Синтаксически понятие задачи определяется следующим образом:

1) элементарная задача есть задача;
2) контекст есть (решенная) задача, или решение;
3) `fail` есть (неразрешимая) задача;
4) если P и Q – задачи, то P|Q есть задача;
5) если P – формула, Q – задача, то (P,Q) есть задача.

Говоря неформально, решением задачи P|Q есть решение задачи P или решение задачи Q; решением задачи (P,Q) есть решение (P,X), где X – это одно из решений Q. Таким образом, понятие (неразрешимой) задачи соответствует И/ИЛИ-дереву поиска. Если предполагается решать задачу P с помощью специальных алгоритмов (решения уравнений или обработки других предикатов), то с задачей следует связать имя (z) соответствующей алгебры. Задача z(P) называется специализированной.

Операционная семантика языка определяется с помощью частичной функции solve, отображающей задачи в задачи. Основное свойство данной функции задается равенством solve(P)=X, где X – одно из решений задачи P (если такое существует) или solve(P)=X|Q, где X – одно из решений, и возможно получение других решений в результате применения solve к Q. Лучше всего, когда такой процесс исчерпывает все решения. Функция solve осуществляет применение к задаче системы `solve_rs` с итеративной стратегией appls. Таким образом, правила переписывания этой системы можно рассматривать как правила вывода соответствующего исчисления. Решаемая задача должна быть специализированной. Если алгоритмов для этой задачи не имеется, ее следует специализировать с помощью имени свободной алгебры `fr`. Далее следуют определения на языке АПЛАН.

```
solve:=proc(p)(
    appls(p,solve_rs);
    return(p)
);
solve_rs:=rs(P,Q,R,X,Y,a,b,x,y,z)(

                  fail|Q =  Q,                          /*  1 */
                 (P|Q)|R = P|Q|R,                       /*  2 */
    is_not_sol(x)->(x|y = solve(x)|y),                  /*  3 */

      z(    P,    fail) = fail,                         /*  4 */
```

```
      z(~(~(P))      ,X) = z(P,X),                          /*  5 */
      z(~(  P &  Q ),X) = z(~(P) |/ ~(Q),X) ,               /*  6 */
      z(~(  P |/ Q ),X) = z(~(P) &  ~(Q),X),                /*  7 */

      z(    P &  Q  ,X) = z(Q,solve(z(P,X))) ,              /*  8 */
      z(    P |/ Q  ,X) = z(P,X) | z(Q,X) ,                 /*  9 */

            z(P,  Q|R ) = z(P,Q)| z(P,R),                   /* 10 */
            z(P,y(Q,X)) = z(P,solve y(Q,X)),                /* 11 */

      z(P,(a,Y)|-(Q,X)) = z(P,solve((a,Y)|-(Q,X)) ),        /* 12 */

      z ( (P = Q),   X) = (vl(z).solve_eq)((P = Q),X),      /* 13 */
      z (  P,        X) = (vl(z).solve_pr)(P,X),            /* 14 */

      ((a, b),Y)|-(P,X) = (a,Y)|-(P,X) | (b,Y)|-(P,X),      /* 15 */
      ( R=>Q, Y)|-(P,X) = try (R,ART(X),unf(P=Q,X,Y)),      /* 16 */
      ( a,    Y)|-(P,X) = unf(P=a,X,Y)                      /* 17 */


);

is_not_sol:=proc(x)(return(~(mark(x)==mark_ar)));


try:=rs(R,n,X)(
    (R,n,fail) = fail,
    (R,n,X  )  = fr(rename(R,n,X),X)
);
```

Первые 11 правил предназначены для сведения задачи к элементарным задачам. Знак `|/` обозначает дизъюнкцию. Правила 13 и 14 решают элементарные задачи путем обращения к соответствующей алгебре. Имя алгебры в специализированной задаче есть имя структуры данных, называемой `valuation` (означиванием). Компоненты этой структуры именованы атомами, обращение к ним производится с помощью операции z.x, которая обозначает тело компонента с именем x означивания z. Означивание алгебры должно содержать по меньшей мере два имени (`solve_eq` и `solve_pr`) алгоритмов решения уравнений и обработки предикатов. Далее следует описание свободной алгебры:

```
fr:=(
    solve_pr: rs(P,z,x,X)(
        (~(P(x)),X)  = (vl(P).neg,make_nil(vl(P).head))|-(~(P(x)),X),
        (  P(x), X)  = (vl(P).pos,make_nil(vl(P).head))|-(  P(x), X)
    );
    solve_eq: rs(P,Q,X)(
        ((P = Q),X) = unify((P = Q),X)
    )
);
```

Функция `solve_eq` для этой алгебры осуществляет вызов процедуры унификации, которая реализована средствами нижнего уровня. Она возвращает новый контекст, который представляет собой наиболее общий унификатор P и Q или символ `fail`, если унификация термов невозможна. При обработке предикатов (элементарных формул) функция `solve_pr` обращается к определению предиката. Неинтерпретированный предикатный символ есть имя означивания, в котором все аксиомы, относящиеся к данному предикату, разделены на две группы: положительную и отрицательную. Приведем пример определения предиката:

```
P1:=ax(x,y,z)(
    pos: (
        P1(A,B),
        P1(x,A+x),
        P1(x,y) |/ ~(P2(y,z) => P1(x+z,y)
    ),
    neg:(
```

```
        ~(P1(C,D)),
          P3(y,y)=> ~(P1(x,y,z))
    )
);
```

Здесь приведено внешнее представление. Заголовок определения содержит список переменных, который следует преобразовать к виду W(1),W(2),… Заголовок используется также для создания пустого контекста (функция `make_nil`). Функция `solve_pr` возвращает задачу другого типа – задачу вывода, которая имеет вид A||B, где A – список аксиом, B – элементарная задача. Для работы с задачами такого рода служат правила 15-17. Функция `un` представляет собой модификацию унификации для двух контекстов, решение (новый контекст) может содержать переменные двух типов: переменные аксиом и переменные задачи. Функция `rename` служит для переименования переменных аксиом, если такие имеются, заменяя их новыми переменными задачи.

Контексты представлены в виде одномерных массивов. С помощью условия `is_not_sol(x)` (не является решением) в правиле 3 производится проверка, является ли `x` массивом. Из этого правила видно, что для поиска решения используется процедура поиска в глубину. Для получения стратегии поиска в ширину или стратегии параллельного поиска следует изменить данное правило следующим образом: `is_not_sol(x)->(x|y = solve(y)|x)`, функцию, которая осуществляет лишь некоторые шаги решения задачи, следует вызывать не из `solve`, а из `solve_rs`.

Другой пример алгебры – это алгебра `lin_alg`, которая содержит алгоритм решения системы линейных уравнений над полем:

```
lin_alg:=(
    solve_pr:rs(P,X)((P,X) = (vl(fr).solve_pr)(P,X));
    solve_eq: proc(p)(
        canpl(p);
        yes:=1;
        appls(p,solve_lin_rs);
        ntb(p,del_mlt);
        return(p)
    )
);
```

Для решения предиката данная алгебра обращается к свободному случаю. Процедура `canpl` изменена таким образом, что символы, отличные от неизвестных `V(i)`, рассматриваются как константы и считаются коэффициентами. В `solve_eq` используются следующие системы переписывания:

```
solve_lin_rs:=rs(A,B,E,X,i)(

    (V(i)$A+B = 0,V(i)=nil,X) = starg(X,i,canplf((-1)*(1/A)*B)),
    (V(i)  +B = 0,V(i)=nil,X) = starg(X,i,canplf((-1)*      B)),
    (V(i)$A   = 0,V(i)=nil,X) = starg(X,i,0),
    (V(i)     = 0,V(i)=nil,X) = starg(X,i,0),

    (E,           V(i)=A,  X) = (canplf(sub(V(i)=A,E)),X),

    (0=0,X) = X,
    (A=0,X) = make_sys(A=0,unknown(A),X),
    (A=B,X) = (canplf(A+(-1)*B=0),X)
);

make_sys:=rs(E,i,X)(
    (E,nil,X) = fail,
    (E,  i,X) = (E,V(i)=arg(X,i),X)
);

unknown:=rs(A,B,i)(
    A+B = unknown(A),
    A$B = unknown(A),
    V(i)= i,
```

```
    A    = nil
);
```

Функция `canplf` – это модификация функции `canpl`. Функция `starg(X,i,z)` обновляет массив X, приписывая i-му аргументу значение z.

Семантика решателя может быть определена в терминах трехзначной логики Клини на основе работы [2], как это было сделано в [3].

СПИСОК ЛИТЕРАТУРЫ

1. A.Letichevsky, D.Gilbert. A Model for Interaction of Agents and Environments // In D.Bert, C.Choppy, P.Moses (Eds), Resent Trends in Algebraic Development Techniques, LNCS 1827, pp.311-328, 1999.
2. M.Fitting. A kripke-kleene semantics for logic programs. *J.Logic Programming*, (4):295-312, 1985.
3. A.A.Letichevsky J.V.Kapitonova. On constructive mathematical descriptions of subject domains. *Kibernetica*, (4):17-35, 1988.