И.Р.Агамирзян, А.С.Иванов

ПРОЦЕДУРНЫЕ МЕХАНИЗМЫ АБСТРАКЦИИ

I. Введение

Постоянно растущие круг и сложность задач, решаемых с помощью вычислительной техники, приводят к разработке достаточно больших программ. Затраты на эти разработки, включающие в себя затраты как финансовые, так и затраты человеческих ресурсов, оказываются слишком велики и часто не оправдываются получаемым в результате программным продуктом. Так, в одних случаях речь идет о программах одноразового использования, когда нужно, скажем, численно промоделировать какую-либо одну фиксированную ситуацию; в других случаях метод решения задачи безнадежно устаревает за время разработки программы, реализующей этот метод; иногда оказывается, что заказчик неправильно сформулировал требования на программу, или неправильно был понят разработчиком и т.п.

Этим вполне объясняется наметившейся в последнее время переход от разработки отдельных программ к разработке проблемно-ориентированных программных систем (пакетов прикладных программ), ориентированных на решение классов задач определенных предметных областей.

Проводя аналогию между программированием и математикой на "метауровне", при которой программе сопоставляется теорема, программной системе естественно сопоставить математическую теорию в том отношении, что подобно тому, как последняя формирует окружение, адекватное для формулирования и доказательства свойств определенного класса объектов, программная система формирует адекватное окружение для программирования процедур обработки объектов определенной предметной области.

В этом смысле программные системы являют собой качественно иное явление, нежели отдельные программы, и, естественно, требуют столь же качественно иных средств и методологии из разработки. Основа такой методологии была заложена в концепции модульного программирования. Модули представляют собой программные единицы, реализующие определенные программные абстракции, которые и формируют программное окружение, определяемое системой.

Понятие модуля в таком виде слишком общо и потребовало дальнейших уточнений в направлении выделения типов абстракций и структуры определяющих их модулей. Эти уточнения привели к выделению трех типов абстракции: операционной абстракции, абстракции данных

и абстракции управления (естественно сопоставляемых в рамках указаной аналогии теоремам, аксиоматически определяемым понятиям и правилам вывода соответственно).

Операциональные абстракции представляют собой хорошо известный тип, ибо уже первые языки программирования доставляли модульные средства их поддержки в форме подпрограмм и процедур.

Модульные средства поддержки абстракции данных были предложены в языках Клу [1] и Асбал [2] (в форме кластеров), в языке Альфард [3] и др. и имели своим источником концепцию класса языка Симула [4] , [5] .

Абстракции управления представляют собой значительно менее разработанный тип абстракции; предложенные здесь модульные средства в основном ориентированы на поддержку абстрактных циклов: итераторы в Клу и Асбале, генераторы, for- и first-suchthat-конструкции в Альфарде [6] .

На использование процедур как средства поддержки абстракции в программировании можно смотреть с двух точек зрения. Речь может идти о "моделировании" конструкций, специально ориентированных либо на поддержание абстракции данных (типа класса в Симуле), либо - абстракции управления (например, итераторов Клу). Во многом именно такой была исходная позиция авторов.

Однако постепенно эта позиция стала смещаться в сторону другой точки зрения, согласно которой упомянутые конструкции следует рассматривать как своего рода синтаксически оформленные образцы использования процедурного механизма абстракции, как, в действительности, более гибкого и фундаментального. На формирование этой точки зрения повлияли два обстоятельства.

Во-первых, при "моделировании" итераторов языка Клу с использованием техники proctype-параметризации было обнаружено, что последняя позволяет описывать более широкий класс абстракций управления, нежели тот, который ограничен концепцией итератора. В частности, на языке процедур естественно описываются такие классические управляющие структуры, как if-then-else, while-do, repeat-until и case-of; то же касается и менее традиционных структур управления, таких как, например, порождаемых концепцией охраняемых команд [7] .

Во-вторых, в результате рассмотрения механизма исключительных ситуаций, тесно связанного с механизмом абстрации данных, и структур управления типа exit и процедурного их "моделирования" авторы пришли к понятию того, что в этой работе названо абстракциями смешанного типа (в противоположность таким абстракциям "чистого" типа, как абстракция данных и абстракция управления).

Осмысленность этого рода абстракций проистекает из наблюдения, что с абстракциями данных всегда связывается определенная структура управления, и, наоборот, абстракции управления всегда неявно вводят определенный тип данных. Обобщение механизма исключительных ситуаций (в сторону усиления аспекта абстракции управления) с одной стороны, и конструкций типа **exit** (в сторону усиления аспекта абстракции данных) с другой стороны, формирует определенную базу для анализа соотношения структур данных и структур управления.

В отношении этой точки зрения, однако, следует сделать ряд замечаний. Авторы далеки от мысли, что механизм процедур в том чистом виде, как он рассматривается в данной работе, является полностью приемлемым механизмом введение абстракций. Нетрудно увидеть, что последовательное использование процедур в роли такого механизма приводит к программным текстам, сильно перегруженным вспомогательной символикой, относящейся к синтаксису изображения и вызова процедур, что проявляется, например, в неумеренном обилии скобок (-) и **begin-end**.

Важным недостатком процедур является наличие накладных расходов на организацию их вызова. Опять же последовательное использование процедурного механизма абстракции приводит к тому, что эти расходы начинают превышать собственно вычислительные расходы. Здесь, по-видимому, требуется своего рода балансирование между процедурным вызовом и макровызовом, хорошую основу для которого может составить концепция смешанных вычислений [8] .

И, наконец, требуется дальнейшее развитие и самого механизма процедур. Здесь следует отметить интересные находки в языке Ада [9] в отношении умолчания параметров вызова процедуры; механизм процедур в языке SL5 [I0] , позволяющий описывать абстракции управления, порождаемые сопрограммным режимом; механизм частичной параметризации [II] , позволяющий, например, в динамике формировать список фактических параметров вызова процедуры.

Для авторов является несколько удивительным, что процедурному механизму абстракции, несмотря на его прозрачность, уделено в литературе столь незаслуженно мало внимания (тем более в контексте интенсивного изучения механизмов абстракции в программировании). Мы можем сослаться лишь на одну работу [I2] , где упоминается, и то вскользь, возможность использования **proctype**-параметров для поддержки абстракции данных; большинство же учебников по программированию до сих пор объясняет использование **proctype**-параметризации на примерах типа процедуры вычисления интеграла. Авторы хотят надеяться, что данная работа сможет при-

влечь интерес к этому гибкому и мощному механизму формирования программных абстракций.

В качестве языка примеров в данной работе принят Алгол 68, расширенный концепцией модалов [I3] , и с учетом работы [I4] . Причиной такого выбора явилось отсутствие необходимых выразительных средств в распространенных языках программирования.

2. Абстракция данных

Традиционный программный текст является сложным переплетением вычислений, реализующих собственно алгоритм, и сопутствующих вычислений, отражающих структуру объектов, обрабатываемых программой. Рассмотрим, например, процедуру левостороннего обхода двоичного дерева, представленного двумерным массивом целых:

Фиг.I. Рекурсивный обход двоичного дерева, представленного массивом.

```
proc  traverse = (int node) void:
    if node ≠ 0
        then
            traverse (tree [ node,  1] );
            print (tree [node,  2 ] );
            traverse (tree [node, 3] );
    fi;
```

В этом примере вырезки в вызовах процедур и условие окончания рекурсивного спуска не относятся к существу алгоритма, загромождают его неалгоритмическими подробностями, задающими структуру представления двоичного дерева.

Эта процедура обладает, на взгляд авторов, двумя недостатками. Во-первых, она фиксирует конкретное представление двоичного дерева - в виде двумерного массива. Во-вторых, из текста этой процедуры, хотя и можно понять, что обрабатывается дерево, но характер его обработки остается неизвестным (например, невозможно определить направление обхода дерева, не зная подробностей его представления).

В качестве альтернативного примера можно привести ту же процедуру в виде, свободном от этих недостатков:

Фиг.2. Рекурсивный обход абстрактного двоичного дерева.

```
proc traverse = (tree node) void:
    if not empty (node)
        then
            traverse (left (node));
```

```
        print (item (node));
        traverse (right (node))
fi;
```

В этой процедуре структура объекта целиком определяется его типом. Под типом мы здесь понимаем совокупность описания вида (tree) в смысле Алгола 68 и процедур (empty, left,right), реализующих операции этого типа. Назначение этих операций - убрать отмеченные неалгоритмические детали из алгоритма и сосредоточить их в описании (инкапсулированного) типа.

Продолжая предлагавшуюся аналогию с математикой, можно отметить, что тип в этом смысле соответствует понятию в математической теории. Для того, чтобы выявить, какого рода механизм формирования понятий требуется в программировании, рассмотрим аналогичный механизм в математике. Любая область математики занимается изучением свойств определенного класса (абстрактных) объектов и позволяет при помощи корректно построенных рассуждений из одних свойств объектов выводить другие их свойства. Этим обусловлено аксиоматическое построение математических теорий: объекты теории (точнее, понятия о них) определяются как обладающие определенным набором изначальных свойств (аксиомы объекта). Классический пример доставляет понятие группы: это любой объект, являющийся множеством, для которого выполнены аксиомы группы. Заметим, что предикат "быть множеством" означает "обладать определенным набором свойств". В программировании мы имеем дело с обработкой объектов. Процедуры обработки строятся как определения последовательности выполняемых операций над объектами, т.е. последовательности выполнения некоторых других процедур обработки. Это обуславливает определение объекта набором связанных с ним операций. Классический пример: стек; стек определяется как объект, над которым определены такие исходные операции как "положить в стек" и "вынуть из стека".

ЗАМЕЧАНИЕ. Здесь мы рассматриваем стек лишь с точки зрения возможности его обработки. В действительности любая обработка, как выполнение определенной последовательности действий, не является самоцелью, а всегда обусловлена требованием получить объект, обладающий определенными свойствами. Однако в данной работе мы позволим себе абстрагироваться от проблематики корректности и синтеза программ, хотя и отдаем себе отчет в том, что в реальной дисциплине программирования такая абстракция недопустима.

Эти рассмотрения приводят нас к определению механизма абстракции данных, ключом к которому является представление о том, что данные в процедурах их обработки доступны только через ассо-

циируемые с ними операции.

Работы, выполняемые в направлении изучения и поиска средств поддержки механизма абстракции данных, привели к двум основным подходам, естественно связываемым с концепцией класса языка Симула и кластера языка Клу соответственно (и идентифицируемых в [I5] как **procedural data structures** и **user defined types**). Основное различие их может быть описано следующим образом.

Кластероподобный механизм основан на фиксировании общего для всех объектов данного типа внутреннего представления и ассоциировании типа с набором операций, которые считаются единственно имеющими доступ к этому представлению. Пример: комплексные числа, описываемые операциями сложения, умножения и т.п.

В случае симулаподобного механизма операции ассоциируются не с типом, а с объектом и характеризуют объекты данного типа эффектом применения к нему этих операций. Пример: стек, описываемый операциями **get** (вынуть из стека) и **put** (положить в стек).

Оба эти механизма являются взаимодополняющими. Симулаподобный механизм ориентирован на введение типов данных, объекты которых рассматриваются как имеющие (внешне видимую) структуру или обладающие определенным поведением. Кластероподобный механизм предназначен для описания типов данных алгебраического рода, объекты которых рассматриваются как атомарные. Важным свойством первого из них является возможность иметь для различных объектов одного и того же типа различные независимые представления, что затруднительно в случае кластероподобной абстракции. Однако симулаподобный механизм не позволяет адекватно описывать типы данных алгебраического рода (это потребовало бы, например, операцию сложения двух комплексных чисел вида **proc**(**compl**,**compl**)**compl** представлять как операцию прибавления числа вида **proc**(**compl**) **void**, ассоциируемую с каждым комплексным числом).

Все же создается впечатление, что симулаподобный механизм можно рассматривать как основополагающий. Во-первых все объекты (любого типа) в конечном счете (на соответствующем уровне абстракции) рассматриваются как структурные. В противном случае мы не смогли бы просто реализовать кластерные операции. И порождать их как такие структурные объекты можно с использованием симулаподобного механизма. Достаточно прозрачно это иллюстрируется на примере матриц. Матрица, как структурный объект, описывается симулаподобной абстракцией, допускающей различные независимые для разных матриц представления. Операции же кластера, описывающего алгебру матриц, реализуются в терминах этой структуры:

```
begin
    matrix a = symmatrix (n),
           b = bandmatrix ( n, m1, m2 ),
           c = squarematrix ( n );
    . . .
    . . . matrix $ mult (a, b, c );
    . . .
end;
```

По-видимому, абстракция данных алгебраического типа всегда является двухуровневой; первый уровень формируется симулаподобным механизмом, второй уровень формируется кластероподобным механизмом, причем объекты первого уровня начинают здесь играть роль переменных.

Теперь заметим, что концепция кластера в языке Клу, с точки зрения поддержки абстракции данных не приносит ничего нового (по сравнению, скажем, с Алголом 68), кроме фиксации самой методологии работы с абстракцией данных и специальной дисциплины доступа к данным. Последнее, по мнению авторов, иногда является слишком жесткой дисциплиной, так как в реальном программировании типы данных очень редко существуют изолированно друг от друга. Зачастую удобнее описывать не отдельные типы, а предметные области (на алгебраическом языке, определять многоосновные алгебры). Например, при описании предметной области элементарной механики, естественно возникают три типа данных: расстояние, время, скорость. Однако очень трудно понять, как можно естественно соотнести с ними операцию, например, вычисления скорости по пройденному расстоянию и времени. Поэтому иногда удобнее считать, что весь набор операций некоторой предметной области равно относится ко всем типам данных этой предметной области, причем операция относится к кластеру некоторого типа, если она имеет доступ к его представлению. Такой подход применяется, например, в языке Декарт [I6] .

В данной работе внимание будет сосредоточено на рассмотрении способов поддержки симулаподобной абстракции данных при помощи механизма процедур.

Образец использования процедур в роли такого средства описан в [I7] и [I8] (правда, в несколько неявной терминологии; в [I8] модуль, определяющий абстракцию данных, вслед за [I9] назван "служба"). В этих же работах можно найти достаточно содержательные примеры. Поэтому дальнейшее изложение этого параграфа будет кратким и в основном сводится к иллюстрации этого образца на примере стека.

Стек, как абстракция данных, характеризуется двумя операциями: get (вынуть из стека) и put (положить в стек), имеющими соответственно виды **proc** () **stacktype** и **proc** (**stacktype**) **void**, где **stacktype** (вид элементов стека) является параметром этой абстракции.

Представим себе вначале, что абстракция "стек" не модуляризована, т.е. в нашем распоряжении нет подходящего модуля, вызов которого порождал бы (в каком бы то ни было смысле) требуемый для работы стековый объект. Тогда, если мы будем придерживаться методологии абстракции данных, соответствующий фрагмент нашей программы, возможно, примет следующий вид *):

```
...  begin  <внутреннее представление стека>
        proc get = ( ) int: ( ... );
        proc put = ( int x ) void : ( ... );
        begin
            <обработка стека в терминах  get  и  put>
        end
     end ...
```

Двинемся теперь в направлении модуляризации стека. С этой целью рассмотрим приведенный фрагмент не как статический, а как порождаемый динамически в результате вызова модуля, который мы назовем stack_module. Динамически такая картинка возникает при исполнении вызова ... stack_module (...); ... , если stack_module реализован в виде процедуры:

```
proc stack_module = ( ... ) void:
begin          <внутреннее представление стека>
   proc get = ( ) int: ( ... );
   proc put = ( int x ) void: ( ... );
   begin
              <обработка стека в терминах get и put>
   end
end;
```

Нам нужно еще абстрагироваться в этом модуле от присутствующего в нем блока конкретной обработки стека; опять же рассмотрим его как вложенный в тело процедуры динамически, для чего запроцедурим этот блок и поместим его в список параметров stack_mo-

*) здесь мы фиксируем тип элементов стека.

dule под именем scope (выбор этого имени станет ясным чуть ниже). При этом, поскольку в процедуре scope требуются операции get и put , мы соответствующим образом ее параметризуем, после чего stack_module примет вид:

```
proc stack_module = ( ..., proc ( stack ) void scope ) void:
begin <внутреннее представление стека>
   proc get = ( ) int: ( ... );
   proc put = ( int x ) void: ( ... );
   scope (  ( get, put ) )
end;
```

где

```
   mode stack = struct (proc() int get, proc (int) void put)
```

Исходный фрагмент нашей программы при этом будет динамической картинкой вызова:

```
... stack_module ( ..., proc ( stack s ) void:
       begin
           <обработка стека s>
       end ) ...
```

Рассмотренная схема доставляет общий образец использования техники proctype – параметризации для поддержки симулаподобной абстракции данных. Теперь мы дадим полное описание модуля, реализующего абстракцию "стек", выбрав достаточно простое его внутреннее представление: фиг.3.

Фиг.3. Процедурная реализация модуля, реализующего абстрактный ограниченный стек

```
mode stack ( stacktype ) = struct
                              ( proc ( ) stacktype get,
                                proc (stacktype) void put );

proc bounded_stack = ( mode stacktype,
                       int maxsize,
     proc (stack(stacktype)) void scope ) void:

begin
    [0:maxsize-1]  stacktype s;
   int ptr := 0;
   proc get = ( ) stacktype:
```

```
    ( if ptr = 0
        then
           out ( "stack underflowed" );
           stop
    fi;
    s [ ptr -:= 1 ] );
proc put = ( stacktype x ) void:
  ( if ptr = maxsize
      then
         out ( "stack overflowed");
        stop
    fi ;
    s [ ptr ] := x;
    ptr +:= 1 );
scope ( ( get, put ) )
end;
```

Фиг.4. Использование абстракции ограниченного стека.

```
bounded_stack ( int, n,
    proc ( stack ( int ) s ) void:
       begin
          ...
          put of s ( 5 );
          ...
          k := get of s ();
          ...
       end );
```

На Фиг.4 иллюстрируется использование модуля bounded_stack. Следует обратить внимание на форму вызова модуля как своего рода "описания" стека s , которое по своей структуре очень сходно с описаниями в Алголе 68. Действительно, по аналогии со следующим фрагментом (описание тождества):

```
begin ref [ ] int a = loc [ 1:n ] int; ... end
```

мы можем переписать фрагмент на Фиг.4 в виде *);

```
begin stack ( int ) s = bounded_stack ( int, n );
   ... put of s ( 5 ); ... k := get of s (); ...
end;
```

*)- такая запись, конечно, недопустима в строгом языке и приведена здесь лишь с целью иллюстрации.

и рассматривать эту запись как своего рода "синтаксический сахар" записи на Фиг.4.

Теперь достаточно прозрачно происхождение идентификатора scope для формального proctype-параметра bounded_stack: блок процедуры, являющийся соответствующим фактическим параметром, становится областью действия стекового объекта (с указанным именем), порождаемого в результате вызова bounded_stack . В дальнейшем для такого proctype -параметра (как формального, так и фактического) мы будем использовать термин "скоуп".

Мы рассмотрели одну из возможных реализаций абстрактного стека. Можно указать и другие реализации, основанные, скажем, на представлении стека связным списком (на модуль, основанный на таком представлении стека, мы будем в дальнейшем ссылаться как на модуль с именем unbounded_stack). При этом видно, что поскольку выбор внутреннего представления объектов никоим образом не отражается на внешне видимом (абстрактном) их представлении при описанной процедурной реализации модулей, последняя оказывается весьма адекватно поддерживающей свойство множественной реализации симулаподобной абстракции данных. Другие же известные нам методы поддержки этого механизма абстракции требуют для этой цели введения специальных средств (например, упрятываемых и экспортируемых полей структур).

Заканчивая рассмотрение абстракции данных, хочется сделать небольшое замечание. В языках программирования существуют два различных правила связывания использующего и определяющего вхождений имени: (более традиционное) правило статической идентификации, по которому использующее вхождение имени связывается с определяющим в наименьшем статически охватывающем это использующее вхождение блоке, и (менее традиционное) правило динамической идентификации, по которому использующее вхождение имени связывается с определяющим в наименьшем динамически охватывающем это использующее вхождение блоке. Различие этих двух правил существенно для глобалов процедур, т.е. имен, которые используются в теле процедуры, но определяются вне его. В случае статической идентификации глобалы идентифицируются по контексту описания процедуры, а в случае динамической идентификации - по контексту ее вызова. Хотя динамическая идентификация и является весьма интересной и важной альтернативой (см., например, [20]), следует отметить, что именно статическая идентификация позволяет поддержать на языке процедур механизм абстракции данных.

3. Абстракция управления

В используемой аналогии с математикой структурам управления были сопоставлены правила вывода. Подобно тому, как последние представляют собой средство получения из одних свойств объектов других их свойств, структуры управления - это средство построения из одних процедур обработки объектов других (более сложных) процедур. Если в математике мы говорим о правилах (способах) рассуждения о свойствах объектов, то в программировании - о правилах (способах) построения процедур их обработки.

Абстракция управления составляет еще один, к сожалению, наименее разработанный в настоящее время класс программных абстракций. Интересно отметить, что в отличие от концепции абстракции данных, появившейся лишь в конце 60-х - начале 70-х годов и быстро завоевавшей признание, концепция абстракции управления начала складываться сравнительно давно (начало 60-х годов), но не получила достаточного развития. Для авторов явились своего рода открытием обнаруженные в [21] ссылки на концепцию сканеров, предложенную Вейзенбаумом в языке Слип [22] (1963 г.), и генераторов, предложенную Ньюэллом в IPL-V [23] (1964 г.). Похожие средства можно найти и в языке Лисп (функция MAPCAR).

По-видимому, основной причиной забвения этих работ явилось несколько преждевременное их появление, когда еще не успел сформироваться интерес к методологическим вопросам разработки больших программ и программных систем. Кроме того, необходимость в средствах поддержки абстракции управления возникает лишь в связи с обработкой достаточно сложных структур данных (кстати, упомянутые работы относятся к обработке списков, порождающих как раз такие структуры) и при наличии средств поддержки абстракции данных. Именно поэтому первые (весьма ограниченные) средства поддержки абстракции управления появляются несколько позже в Паскале в форме перечисляющего цикла (развитие этой идеи можно найти в работе [24]) и в языке Клу в форме итераторов, естественно привязанных к концепции кластера. В целом, однако, до сих пор механизм абстракции управления остается недостаточно понятым и, в основном, сводится к средствам поддержки абстрактных циклов в форме либо сканеров, либо генераторов.

Другой причиной явилось то, что эти работы оказались захлестнутыми мощной волной движения "структурного программирования", на первых порах воспринятого и развиваемого как "программирование без goto", в рамках которого осуществлялся поиск некоторой канонической системы управляющих структур. Здесь в первую очередь

следует упомянуть теорему Якопини [25] о полноте системы из операций композиции и if- и while -структур управления, сформулированную как теоретическое обоснование структурного программирования (и осознанную впоследствии как достаточно бессодержательную в таковом качестве; см., например, [26]). Несмотря на то, что в рамках этого направления были проведены очень важные методологические исследования по анализу структуры алгоритмического мышления, сложилась, к сожалению, бытующая и до сих пор точка зрения, что набор управляющих структур должен быть фиксирован как атрибут языка программирования (точнее, эта точка зрения была здесь закреплена, т.к. сложилась она уже в первых языках программирования). С другой стороны, в избыточности системы структур управления нет ничего плохого - такую картину мы видим практически в любом языке программирования.

Безусловно, что теории логического вывода фиксируют определенные системы правил вывода (также устанавливая факт их полноты). Но цель этих теорий состоит не в представлении математике рабочего инструмента формализации рассуждений, а в исследовании математического вывода и математических теорий как математических объектов, т.е. в разработке системы понятий и способов рассуждений, позволяющих наиболее адекватно формулировать (выражать) и доказывать их свойства (аналогичная ситуация с теорией алгоритмов). Рассуждения же рабочего математика (в том числе и логика), в том виде, как они оформляются в математических текстах, остаются основанными на "проблемно-ориентированных" правилах вывода (способах рассуждения) и используют логические шаги, хотя и достаточно очевидные, но являющиеся зачастую сложной композицией формальных правил вывода. Относительно этих рассуждений существует лишь определенная уверенность в потенциально возможной их формализации. Причина же, по которой эта потенциальная формализуемость остается нереализованной, в основном, одна: такая формализация (скажем, на языке натуральных выводов) привела бы к значительной утере прозрачности хода рассуждения в доказательствах. Для математических текстов очень важна именно их читабельность, т.е. то, чего крайне не хватает программным текстам.

При традиционном способе записи алгоритма, с использованием дисциплины структурного программирования, мы все равно вынуждены обходиться лишь очень слабыми структурами управления из "классического" набора. Это аналогично принуждению рабочего математика, доказывающего некоторую (например, геометрическую) теорему, не только записывать это доказательство в терминах системы правил

вывода какой-то теории (например, исчисления натуральных выводов), но и мыслить в тех же терминах. Программистской реакцией на такие жесткие рамки оказалось распространенное сейчас (особенно в системах управления пакетами прикладных программ) стремление к "непроцедурности", реализуемое синтезом программ на модели предметной области. Продолжая аналогию с математикой, это во многом аналогично автоматическому доказательству теорем по их формулировке (при условии существования соответствующей теории).

Основу абстракции управления составляет управление последовательностью исполнения абстрактных действий, т.е. безотносительно их конкретного процедурного содержания. При реализации модуля, определяющего абстракцию управления, (мы будем называть их контроллерами *)) мы абстрагируемся от конкретных действий, выполняемых управляемыми процедурами; при использовании контроллера в связи с конкретными процедурами мы абстрагируемся от его реализации (т.е. используем его как абстракцию).

Отсюда очевидна роль, которую играет proctype- параметризация как средство поддержки абстракции управления. На фиг.5 приведены контроллеры, условно реализующие "классические" управляющие структуры.

Фиг.5. Контроллеры классических структур управления

```
proc seq = ( proc () void stmt1, stmt2 ) void :
      -- контроллер последовательного выполнения
begin stmt1 (); stmt2 () end;

proc if = ( bool cond,
            proc () void -- then
                         stmt1,
                         -- else
                         stmt2 ) void :


      -- контроллер условного выполнения
begin
   if cond
      then stmt1 () -- then
      else stmt2 () -- else
```

*) - этот термин был предложен в свое время одним из авторов данной работы И.Р.Агамирзяном.

```
    fi
end;
proc while = ( proc () bool cont_cond,
               proc () void loop_body ) void :

            -- контроллер цикла
begin
   loop :
      if not cont_cond ( )
         then goto eof_loop
      fi;
      loop_body ( );
   goto loop;
   eof_loop : skip
end;
```

Несколько замечаний по поводу этих примеров. Реализация контроллеров seq и if условна в том смысле, что основана на использовании тех же управляющих структур, которые они сами и определяют. Для нас, однако, здесь существенна не их конкретная реализация, а то, что в принципе они могут быть представлены в виде процедур.

Следует обратить внимание на то, что while -контроллер управляет выполнением не одной процедуры loop_body (тело цикла), а двумя, включая и процедуру cont_cond (условие выполнения цикла), так что while -цикл имеет как бы два "тела", одно из которых должно вырабатывать булевское значение. Кстати, в практике программирования нередко можно встретить такой образец использования while -конструкции: **while ... do skip od**. В этой связи уместно рассмотреть еще и **repeat-until** -абстракцию с контроллером на фиг.6а. Здесь мы имеем также два "тела цикла",

Фиг.6а. Контроллеры **repeat-until** -структуры.

```
(a). proc repeat = ( proc () void loop_body,
                     proc () bool eof_cond ) void :

   begin
      loop :
         loop_body ( );
         if not eof_cond ( )
            then goto loop
         fi
   end;
```

(b). **proc** repeat = (**proc** () **bool** eof_cond) **void** :

```
begin
   loop :
     if not eof_cond ( )
        then goto loop
     fi
end;
```

которые в данном случае можно слить в один: см. фиг.6b . Теперь видно, что repeat – абстракция является частным случаем while--абстракции, поддерживающим образец **while not** (...) **do skip od**.

При реализации более сложных структур управления возникает необходимость в связи по данным между контроллером и выполняемыми им телами. Классическим примером такой структуры управления является for -цикл. Реализовать управляющую переменную for - цикла (в более сложных структурах управляющих переменных может быть и несколько - польза такой возможности доказана циклом по итератору в языке КЛУ) можно двумя путями - либо как параметр контроллера вида **ref indextype** (при этом контроллер превращается в процедуру с побочным эффектом), либо как параметр тела цикла вида **indextype**, причем в таком случае управляющая переменная не видна снаружи контроллера, что полезно с методологической точки зрения (можно вспомнить, что циклическое предложение в Алголе 68 локально описывает свою переменную цикла). Очевидно, что последовательное применение такого метода приведет к чисто функциональному стилю программирования. С другой стороны, этот пример показывает, что проводимое нами разграничение абстракций данных и управления не абсолютно, и позволяет нам перейти к другому, более широкому классу абстракций, обобщающему рассматривавшиеся ранее случаи.

4. Абстракции смешанного типа

Цель наших рассмотрений состоит в том, чтобы выявить механизм процедур как достаточно мощное средство поддержки программных абстракций. С этой целью мы анатомировали такие механизмы, как абстракция данных и абстракция управления и попытались вскрыть их процедурную семантику, т.е. рассмотреть их, в конечном счете, как определенные образцы использования процедур и техники proctype -параметризации. Теперь мы хотим обратить внимание на еще одно крайне важное, по нашему мнению, качество процедур в роли такого средства - обеспечение возможности вводить абстракции

"смешанного типа".

В реальном программировании мы имеем дело не только с "чистыми" абстракциями, т.е. понятиями, относящимися к одному определенному типу (классу) абстракций; достаточно обычными являются понятия, принадлежащие одновременно к нескольким типам абстракции.

В качестве простого примера можно взять (уже упоминавшуюся выше) абстракцию, близкую к for-конструкции Алгола 60 или DO-оператору Фортрана:

```
proc for = ( ref int k,
             int lwb, upb, step,
             proc () void loop_body ) void :

begin

   ...

end;
```

выполняющую не только функции управления исполнением тела цикла (loop body) , но и определяющую оператор, присваивающий переменной цикла (параметр k) некоторое значение.

Другие примеры доставляют так называемые накопители, рассмотренные в работе [I7], представляющие собой ассорти абстракции данных и операциональной абстракции.

Доказательством относительности деления на абстракцию управления и данных может являться следующий пример: предположим, что мы имеем модуль для реализации стека, параметризуемый по типу элементов. При использовании стека мы всегда записываем и читаем два целых числа. Можно параметризовать стек по целому и выполнять две операции записи и чтения, но можно параметризовать его по паре целых и выполнять одну операцию. В первом случае внешняя семантика стека задается управлением, во втором - структурой данных.

Одним из систематических источников абстракций смешанного типа является концепция исключительных ситуаций, тесно связанная с механизмом абстракции данных и обусловленная тем, что отдельные операции (абстрактного) типа могут быть определены не на всем пространстве состояний объектов этого типа.

Так, в примере с абстракцией "стек" операция get не определена на пустом стеке и применение ее к пустому стеку рассматривается как приводящее к возникновению исключительной ситуации (underflow); аналогично в случае bounded stack операция put не определена для уже полного стека (ситуация overflow).

С исключительной ситуацией всегда связана некоторая процеду-

ра ее"обработки", включающая, в частности, акт принятия решения о том, что делать дальше. В модуле bounded_stack на фиг.3 это решение "вшито" в форме аварийного останова программы. Можно, однако, представить себе и несколько иную реализацию, при которой возникновение underflow- ситуации будет трактоваться не как результат некорректного использования стека, а как сигнал об окончании его обработки (исчерпаны данные в стеке), требующий прекращения исполнения скоупа и нормального продолжения исполнения программы. В других случаях реакция на возникновение исключительной ситуации может приводить к продолжению исполнения скоупа, инициированию его заново и т.п. Другими словами, модуль абстракции данных может оказываться естественно вовлеченным в выполнение (внешне видимых) управляющих функций, т.е. может определять одновременно и некоторую абстракцию управления.

Хорошей практикой является предоставление пользователю абстракции данных возможности самому определять процедуру обработки (реакцию на возникновение) исключительной ситуации. Только что мы указали две возможные различные точки зрения на смысл underflow -ситуации. Кроме того, в программе обычно существует несколько объектов одного и того же типа, и в случае возникновения исключительной ситуации типа "ошибка" необходимо уметь идентифицировать (при ошиб_выдаче), при обработке какого именно объекта эта ситуация возникла. Далее, пользователем одной абстракции данных (например, массива) может являться модуль другой абстракции данных (например, стек) со всеми вытекающими отсюда последствиями (важно, например, чтобы сообщение о переполнении стека не оказалось "реализованным" в виде сообщения о выходе за пределы массива). В общем случае, дело здесь в том, что разработчик абстракции данных отделен от конкретных процедур обработки определяемых им абстрактных объектов и потому не может, вообще говоря, предусмотреть адекватной реакции на исключительные ситуации, возникающие именно в контексте этих конкретных процедур. Ясно, что внутри модуля, реализующего абстракцию, мы не знаем внешней семантики использования этой абстракции, а именно она и определяет реакцию на исключительные ситуации. Более того, одна и та же абстракция может использоваться в нескольких разных смыслах, например, массив может быть использован не только как представление стека, но и как представление вектора.

Механизм процедур позволяет достаточно естественно поддержать такую практику определяемых пользователем процедур обработки исключительных ситуаций, что иллюстрируется примером с bounded _stack на фиг.7. Описываемый здесь модуль, управляющий

Фиг.7. Модуль bounded stack с определяемой пользователем обработкой overflow- и underflow -ситуаций

```
proc  bounded_stack
        ( mode stacktype,
          int maxsize,
          proc ( stack ( stacktype )) void scope,
          proc () void overflow, underflow ) void;
begin
    [0:maxsize-1] stacktype s;
  int ptr := 0 ;

  proc get = () stacktype :
      (if ptr = 0
          then underflow ();
               goto exit
       fi;
       s [ ptr -:= 1] );
  proc put = (stacktype x ) void :
      (if ptr = maxsize
          then overflow ();
               goto exit
      fi;
      s [ ptr ] :=x;
      ptr +:= 1);
  scope ( ( get, put ));
  exit : skip
end;
```

выполнением трех процедур, определяет совершенно явную абстракцию управления.

Отметим одно немаловажное, по нашему мнению, обстоятельство. Одна и та же абстракция данных может допускать несколько различных ее реализаций, и вполне нормально, что с различными реализациями окажутся связаны различные наборы исключительных ситуаций. Например, с unbounded -реализацией стека связывается одна ситуация: underflow ; с bounded -реализацией – две ситуации: underflow и overflow . Таким образом, два модуля могут определять одну и ту же абстракцию данных, но разные абстракции управления (unbounded stack и bounded stack в действительности и являют собой различные абстракции, обладающие различным поведением; но в том виде, как эти абстракции видны изнутри скоупа, они неразличимы, т.е. в рамках скоупа их поведение полностью идентично).

В общем случае, две абстракции могут совпадать по одному типу, но различаться по другому.

Интересные абстракции смешанного типа можно подучить, двигаясь в направлении обобщений картины с исключительными ситуациями. Имея в виду эти обобщения, нам удобно ввести понятие исключительного состояния объекта: мы будем рассматривать пространство состояний (объектов данного типа) как объединяющее класс "нормальных" и класс "исключительных" состояний, операции данного типа - как всюду определенные, а исключительные ситуации - как возникающие в связи с переходом объекта в одно из исключительных состояний (именно, в то, которое сопоставлено данной исключительной ситуации).

Так, например, пространство состояний ограниченного стека будет включать в себя два исключительных состояния: **underflowed** и **overflowed** , а операции **get** и **put** будут считаться всюду определенными.

В таком контексте правильнее будет говорить далее не об обработке исключительных ситуаций, а об обработке объекта в его исключительном состоянии. Точнее, исключительные состояния объекта рассматриваются как исключительные по отношению к его скоупу,т.е. исключают возможность нормальной обработки его в таких состояниях процедурой скоупа, а требуют некоторой специальной обработки другими процедурами. Последняя может осуществляться в терминах специальных ("привилегированных") операций (скажем, позволяющих осуществить более глубокий доступ к внутреннему представлению объекта). Таким образом, модуль абстракции данных может иметь несколько скоупов, в каждом из которых определена своя "точка зрения" на (абстрактную) структуру определяемого модулем объекта.

Рассмотренная выше идеология (образец использования) исключительных ситуаций в значительной степени подразумевала необратимость перехода объекта в исключительное состояние и потому их возникновение приводило к прекращению исполнения скоупа, выполнению некоторых **post mortum** -действий и, возможно, выходу из модуля. Это было естественно, т.к. в процедуре обработки исключительной ситуации не предусматривалось доступа к самому объекту (поэтому мы и говорили об обработке ситуации, а не объекта) и, следовательно, возможности возврата его в нормальное состояние. Последнее, однако, оказывается в принципе возможным, если такой доступ обеспечен, и другая идеология исключительных состояний может состоять в том, что в таких состояниях просто требуется некоторая дополнительная, т.е. непредусмотренная в скоупе явно, но всегда

подразумеваемая, обработка объекта *).

Продемонстрируем это опять же на примере со стеком. Мы можем использовать ограниченный стек, если имеем хорошую верхнюю оценку максимальной высоты стека для реализуемого алгоритма. Допустим, однако, что эта оценка не абсолютна, но верна в большинстве случаев использования алгоритма. Тогда также весьма соблазнительно использовать ограниченный стек, т.к. получаемая при этом реализация алгоритма будет эффективной для большинства случаев. Но нам бы хотелось сохранить работоспособность программы и для оставшегося меньшинства. С этой целью мы можем следующим образом использовать исключительные состояния стека: при переходе в **overflowed** -состояние стек будет копироваться в некоторый буфер; при возникновении же **underflow** -ситуации буфер будет перегоняться обратно в стек. При этом, поскольку нам может не хватить одного буфера, следует организовать стек буферов, для чего можно использовать неограниченный стек.

Не выписывая подходящей для наших целей модификации модуля **bounded_stack** явно (она достаточно очевидна), отметим только, что в качестве операций, представляющих стек в его исключительном состоянии, мы выбрали операции доступа к представляющим его массиву и указателю:

```
mode stackrep ( stacktype ) =
     struct ( ref [ ] stacktype array,
              ref int            pointer )
```

Фиг.8. Обработка ограниченного стека с перекачкой в буфер и обратно

```
unbounded_stack ( ref [ ]int,
                  proc ( stack ( ref [ ]int ) bufs ) void :
begin
    bounded_stack ( int, n,
                    proc ( stack (int ) s ) void :
begin

          — основная обработка стека
          ...
```

*) – развиваемый здесь образец использования исключительных состояний очень близок к механизму "демонов" в базах данных систем искусственного интеллекта [27].

```
    end,
    proc ( stackrep ( int ) s ) void : -- on s overflow
    begin
       ref [ ]int buf := heap [0:n-1] int;
       for i from 0 to n-1
          do
             buf   [i]  := ( array of s ) [i]
          od;
       ( put of bufs ) ( buf );
       pointer of s := 0
    end,
    proc ( stackrep ( int ) s ) void :  -- on s underflow
    begin
       ref [ ] int buf := get of bufs ( ) ;
       for i from 0 to n-1
          do
             ( array of s ) [ i] := buf [i]
          od;
          pointer  of s := n
    end )
end,
proc ( ... ) void : -- on bufs underflow
begin
   ...
end );
```

ЗАМЕЧАНИЕ. Underflow и overflow -обработку стека в этом примере можно организовать и с использованием операций get и put. Мы использовали операции pointer и array потому, что они позволяют осуществить эту обработку несколько более эффективно. К тому же, мы хотим подчеркнуть, что специальная обработка может потребовать и специальных операций. Имеется, однако, один принципиальный вопрос: могут ли возникать исключительные ситуации в самих процедурах их обработки, когда последняя осуществляется в терминах тех же операций, что и в скоупе объекта? Да, могут. Это будет приводить к рекурсивному вызову этих процедур (интересная возможность, для которой авторы не смогли, к сожалению, найти хорошего образца использования).

Другой интересный пример абстракции смешанного типа доставляет структурный выход (exit).

Фиг.9. Реализация оператора exit.

```
proc exit = ( proc ( proc () void ) void body ) void :
begin
   proc signal = () void : goto exit_lab ;
   body ( signal);
   exit_lab : skip
end;

exit ( proc ( proc () void signal ) void :
          begin
            ...
            signal ( );
            ...
          end );
```

Следует заметить, что в этом примере процедура exit неявно определяет некий объект с двумя состояниями: нормальным и исключительным. Этот объект имеет единственную операцию signal, осуществляющую принудительный перевод объекта в исключительное состояние.

Приведенные примеры и рассуждения, на взгляд авторов, дают хорошую основу для понимания связи абстракции управления и абстракции данных и доказывают относительность деления механизма абстракции на непересекающиеся классы.

В контексте данной работы возникает закономерный вопрос: как соотносится абстракция смешанного типа с математикой? Авторы склонны считать, что аналогом абстракции смешанного типа являются сложные правила рассуждений, применяемые рабочим математиком при доказательстве теорем и сочетающие правила вывода и аксиомы.

5. Заключение

В заключение следует сделать несколько замечаний по поводу синтаксического оформления процедурных средств абстракции. Как уже подчеркивалось во введении, авторы далеки от мысли, что механизм процедур в том виде, как он предлагается в этой работе,

приемлем для поддержки абстракций в реальном программировании. Поэтому желательна разработка новых синтаксических форм вызова и описания процедуры. В настоящее время существуют некоторые работы, в которых предлагаются различные способы представления вызовов процедур. Среди них следует упомянуть трактовку присваивания как вызова [I,2], распределенный вызов (в приложении к макрогенераторам) [28], ключевые параметры [9].

Можно заметить, что запись некоторых абстракций в нетрадиционной форме существенно проясняет их смысл. Например, при использовании процедур с ключевыми параметрами становится ясно, что размеченные структуры управления (например, **case** в языке Паскаль [29]) эквивалентны контроллерам с ключевыми параметрами.

Авторы пользуются случаем выразить благодарность за проявленный к данной работе интерес С.С.Лаврову, И.В.Романовскому, Н.Н.Непейводе, Т.И.Петрушиной, Ф.А.Новикову.

Литература

1. Liskov B., Atkinson R., Bloom T., Moss E., Schaffert C., Scheifler B., Snyder A. CLU reference manual. - MIT Lab.Comput.Sci. Techn.Rept., 1979, N 225, 166 pp.
2. Moss J.E.B. Abstract data types in stack based languages. - MIT Lab.Comput.Sci.Techn.Rept., 1978, N 190, 154 pp.
3. Wulf W.A. Languages and structured programs. - Current Trends in Programming Methodology, Prentice-Hall,Englewood Cliffs. 1977, p.33-60.
4. Дал У.-И., Мюрхауг Б., Нюгорд К. Симула 67. Универсальный язык программирования. М., I969.
5. Дал У.-И., Хоор К. Иерархические структуры программ. - В кн.: Дал У., Дейкстра Э., Хоор К. Структурное программирование. М., I975, с.I98-245.
6. Shaw M., Wulf W.A., London R.L. Abstraction and verification in Alphard: defining and specifying iteration and generators. - Comm.ACM, 1977, v.20, N 8, p.553-564.
7. Дейкстра Э. Дисциплина программирования. М., I978.
8. Ершов А.П. О сущности трансляции. - Программирование, I977, № 5, с.2I-39.

9. Preliminary Ada reference manul. ACM, 1979.
10. Hanson D.R., Griswold R.E. The SL5 procedure mechanism. - Comm.ACM, 1978, v.21, N 5, p.392-400.
11. Lindsey C.H. Specification of partial parametrization proposal. - Algol Bull., 1976, N 39, p.6-9.
12. Boom H. Separate compilation, definition modules and block structured languages. - Amsterdam Math. Centre preprint IW 77/77.
13. Lindsey C.H. Modals. - Algol Bull., 1973, N 37, p.26-29.
14. Boom H. Extended type checking. - Amsterdam Math.Centre preprint, IW 60/76.
15. Reynolds J.C. User defined types and procedural data structures as complementary approaches to data abstraction. - Conf.New Directions in Algorithm. Languages, IFIP WG 2.1, Munich, 1975.
16. Бабаев И.О., Новиков Ф.А., Петрушина Т.И. Язык Декарт - входной язык системы СПОРА. - Прикладная информатика, 1981, вып.1, с.35-73.
17. Иванов А.С. Использование механизма параметров процедурного типа в модульной организации пакетов. - Зап.науч. семин.ЛОМИ, 1979, т.90, с.24-38.
18. Иванов А.С. Об организации служб. - Зап.науч.семин. ЛОМИ, 1980, т.102, с.19-26.
19. Воденин Д.Р., Романовский И.В. Программирование в терминах служб. - Кибернетика, 1979, № 5, с.70-75.
20. Цейтин Г.С. Нематематическое мышление в программировании. - В сб.: "Перспективы системного и теоретического программирования", Новосибирск, 1979, с.128-132.
21. Пратт Т. Языки программирования: разработка и реализация. М., 1979.
22. Weizenbaum J. Symmetric list processor. - Comm. ACM, 1963, v.6, N 9, p.524-544.
23. Newell A. (ed.) Information Processing Language Manual. Prentice-Hall, Englewood Cliffs, N.J., 1964.
24. Abramson H. Ordered types and generalised for statement. - SIGPLAN Notices, 1977, v. 12, N 12, p.55-59.

25. B ö h m C., J a c o p i n i G. Flow diagrams,Turing machines, and languages with only two formation rules. - Comm.ACM, 1966, v.9, N 5, p.366-371.
26. K n u t h D.E. Structured programmingwith go to statement. - Computer Surveys, 1974, v.6, N 4, p.261-302.
27. У и н с т о н П. Искусственный интеллект. М., 1980.
28. Б р а у н П. Макропроцессы и мобильность программного обеспечения. М., 1977.
29. A d d y m a n A.M. et al. A draft description of Pascal. Software - Pract. & Exper., 1979, v.9, N 5.