С авторами можно связаться по адресу: mail@avrorasystems.com, "для Шалыто".

Статья опубликована в журнале "Программист", 2002. №8. С.82-90.

Ханойские башни и автоматы

Анатолий Шалыто, Никита Туккель, Никита Шамгунов

Рассматриваются алгоритмы и программы решения известной задачи о ханойских башнях. Показано, что применение автоматов позволяет формально переходить от рекурсивных алгоритмов к итеративным, либо строить их непосредственно. Установлено, что автоматы с двумя или тремя состояниями могут управлять 2^N состояниями объекта управления, в роли которого выступают диски и стержни.

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях [1–5], которая формулируется следующим образом. Имеются три стержня, на первом из которых размещено N дисков. Диск наименьшего диаметра находится сверху, а ниже — диски последовательно увеличивающегося диаметра. Задача состоит в определении последовательности перекладываний по одному диску со стержня на стержень, которые должны выполняться так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы, в конце концов, все диски оказались на другом стержне.

Покажем, что применение автоматов [6] позволяет формально переходить к итеративным алгоритмам решения этой задачи, либо строить такие алгоритмы непосредственно. В работах [7,8] приведены примеры преобразований рекурсивных программ в итеративные, однако подходы, обеспечивающие такие преобразования, не формализованы. Настоящая работа призвана устранить этот пробел.

В работе предлагаются три метода. Первый из них обеспечивает формальное построение автоматной программы (программы, построенной с использованием автоматов) на основе раскрытия рекурсии с применением стека. Второй метод также обеспечивает раскрытие рекурсии и состоит в построении автомата, осуществляющего обход дерева действий, выполняемых рекурсивной программой. Третий метод состоит в непосредственном управлении дисками и стержнями, и не использует таких абстракций как деревья и стеки. Отметим, что применение каждого из предлагаемых методов для рассматриваемой задачи порождает автоматы с двумя или тремя состояниями, управляющие "объектом управления" с 2^N состояниями, возникающими в процессе перекладывания дисков.

Классическое рекурсивное решение задачи

Известны рекурсивные алгоритмы для решения рассматриваемой задачи [1,5]. Приведем один из таких алгоритмов, построенный по методу "разделяй и властвуй". Задача о перекладывании N дисков с i-ого на j-тый стержень может быть декомпозирована следующим образом.

1. Переложить N–1 диск со стержня с номером i на стержень с номером 6–i–j.

2. Переложить диск со стержня с номером i на стержень с номером j.

3. Переложить N–1 диск со стержня с номером 6–i–j на стержень с номером j.

Таким образом, задача сводится к решению двух аналогичных задач размерности N–1 и одной задачи размерности один. При этом если для решения одной задачи размерности N–1 требуется F_{N-1} перекладываний, то их общее число определяется соотношением:

$$F_N = 2F_{N-1} + 1.$$

Из этого соотношения следует [2], что

$$F_N = 2^N - 1.$$

Указанный процесс декомпозиции можно изобразить с помощью дерева декомпозиции (рис. 1).

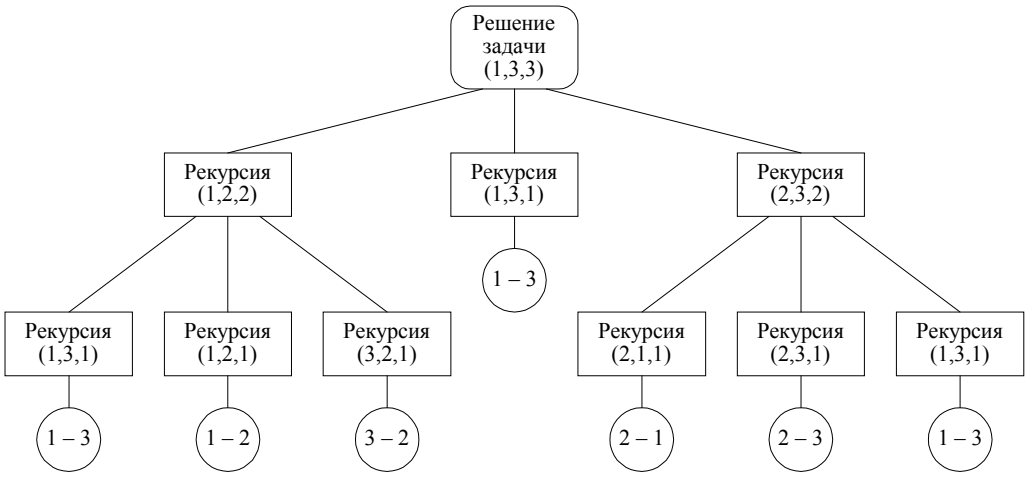

Рис. 1. Дерево декомпозиции для задачи о ханойских башнях при N = 3

В этом дереве вершины, обозначенные прямоугольниками, соответствуют подзадачам, решаемым при каждом вызове рекурсивной функции. Эти вершины помечаются номерами стержня, с которого перекладывается диск, стержня, на который перекладывается диск, и числом перекладываемых дисков. При этом для вершины с пометкой (i,j,k) левая вершина поддерева будет помечена (i,6–i–j,k–1), средняя — (i,j,1), а правая — (6–i–j,j,k–1).

Перекладывания дисков выполняются в вершинах, обозначенных кружками. Для каждой из этих вершин сохраняются первые две позиции пометки соответствующего прямоугольника — (i,j). Количество таких вершин равно F_N.

Приведем программу, написанную (как и все последующие) на языке Си и реализующую рассматриваемый рекурсивный алгоритм (листинг 1). Ее поведение эквивалентно обходу дерева декомпозиции слева направо, причем в вершинах обозначенных кружками производятся перекладывания.

ЛИСТИНГ 1. Рекурсивная программа

```
#include <stdio.h>

void hanoy( int i, int j, int k, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "   " ) ;
  printf( "hanoy(%d,%d,%d)\n", i, j, k ) ;

  if( k == 1 )
    move( i, j, d ) ;
  else
  {
    hanoy( i, 6-i-j, k-1, d+1 ) ;
    hanoy( i, j, 1, d+1 ) ;
    hanoy( 6-i-j, j, k-1, d+1 ) ;
  }
}

void move( int i, int j, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "   " ) ;
  printf( "move(%d,%d)\n", i, j ) ;
}
```

```
void main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input, 0 ) ;
}
```

В эту программу введено протоколирование рекурсивных вызовов и четвертый параметр рекурсивной функции, используемый для определения глубины рекурсии. Ниже приводится протокол работы программы (листинг 2), эквивалентный дереву декомпозиции (рис. 1).

ЛИСТИНГ 2. Протокол работы рекурсивной программы

```
Ханой с 3 дисками:
hanoy(1,3,3)
   hanoy(1,2,2)
      hanoy(1,3,1)
      move(1,3)
      hanoy(1,2,1)
      move(1,2)
      hanoy(3,2,1)
      move(3,2)
   hanoy(1,3,1)
   move(1,3)
   hanoy(2,3,2)
      hanoy(2,1,1)
      move(2,1)
      hanoy(2,3,1)
      move(2,3)
      hanoy(1,3,1)
      move(1,3)
```

Первый метод: моделирование рекурсии автоматной программой

Идея метода состоит в раскрытии рекурсии за счет моделирования работы рекурсивной программы. При этом для хранения локальных переменных стек используется явно. Явное выделение стека по сравнению со "скрытым" его применением в рекурсии, позволяет программно задавать его размер, следить за его содержимым и добавлять отладочный код в функции, реализующие операции над стеком.

Перейдем к изложению метода.

1. Каждый рекурсивный вызов в программе выделяется как отдельный оператор. По преобразованной рекурсивной программе строится ее схема (рис. 2), в которой применяются символьные обозначения условий переходов (x), действий (z) и рекурсивных вызовов (R). Схема строится таким образом, чтобы каждому рекурсивному вызову соответствовала отдельная операторная вершина. Отметим, что здесь и далее под термином "схема программы" понимается схема ее рекурсивной функции.

2. Для определения состояний эквивалентного построенной схеме автомата Мили в нее вводятся пометки, по аналогии с тем, как это выполнялось в работе [9] при преобразовании итеративных программ в автоматные. При этом начальная и конечная вершины помечаются номером 0. Точка, следующая за начальной вершиной, помечается номером 1, а точка, предшествующая конечной вершине — номером 2. Остальным состояниям автомата соответствуют точки, следующие за операторными вершинами. В рассматриваемом случае точка с номером 2 совпадает с точками, следующими за операторными вершинами R3 и z0.

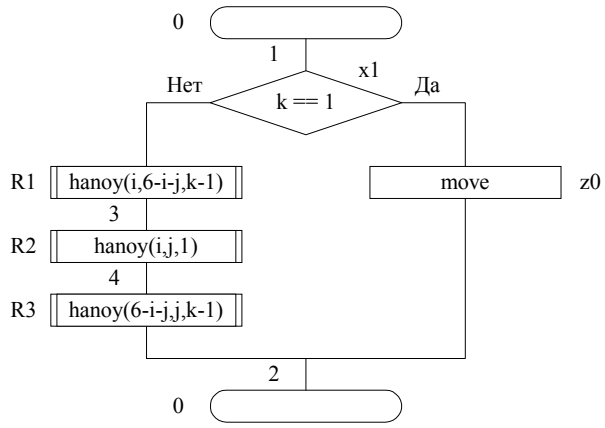

Рис. 2. Схема рекурсивной функции

3. В соответствии с пометками на рис. 2 строится граф переходов автомата Мили (рис. 3).

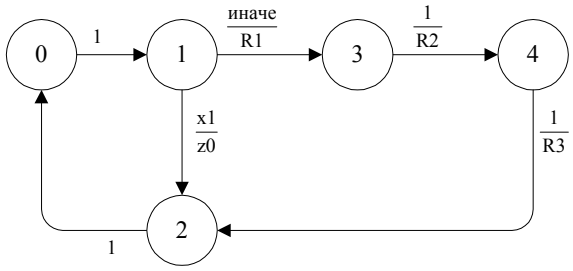

Рис. 3. Граф переходов, построенный по схеме рекурсивной функции

4. Выделяются действия, совершаемые над параметрами рекурсивной функции при ее вызове. Обозначим такие действия для рекурсивных вызовов R1, R2 и R3 как z1 (j = 6–i–j; k– –), z2 (k = 1) и z3 (i = 6–i–j; k– –) соответственно.

5. Составляется перечень параметров и других локальных переменных рекурсивной функции, определяющий структуру ячейки стека. Если рекурсивная функция содержит более одного оператора рекурсивного вызова, то в стеке также запоминается значение переменной состояния автомата. В данном примере элемент стека содержит значение переменной состояния автомата y и значения локальных переменных i, j и k.

6. Выполняется преобразование графа переходов для моделирования работы рекурсивной функции, состоящее из трех этапов (рис. 4).

6.1. Дуги, содержащие рекурсивные вызовы (R), направляются в вершину с номером 1. В качестве действий на этих дугах указываются: операция запоминания значений локальных переменных push(s), где s — номер вершины графа переходов, в которую была направлена рассматриваемая дуга до преобразования и соответствующее действие, выполняемое над параметрами рекурсивной функции.

6.2. Безусловный переход на дуге, направленной из вершины с номером 2 в вершину с номером 0, заменяется условием "стек пуст" с первым приоритетом.

6.3. К вершине с номером 2 добавляются дуги, направленные в вершины, в которые до преобразования графа входили дуги с рекурсией. На каждой из введенных дуг выполняется операция pop, извлекающая из стека верхний элемент. Условия переходов на этих дугах имеют вид stack[top].y == s, где stack[top] — верхний элемент стека, y — значение переменной состояния автомата, запомненное в верхнем элементе стека, а s — номер вершины, в которую направлена рассматриваемая дуга. Таким образом, в рассматриваемом автомате имеет место **зависимость от глубокой предыстории** — в отличие от классических автоматов, переходы из состояния с номером 2 зависят также и от ранее запомненного в стеке номера следующего состояния.

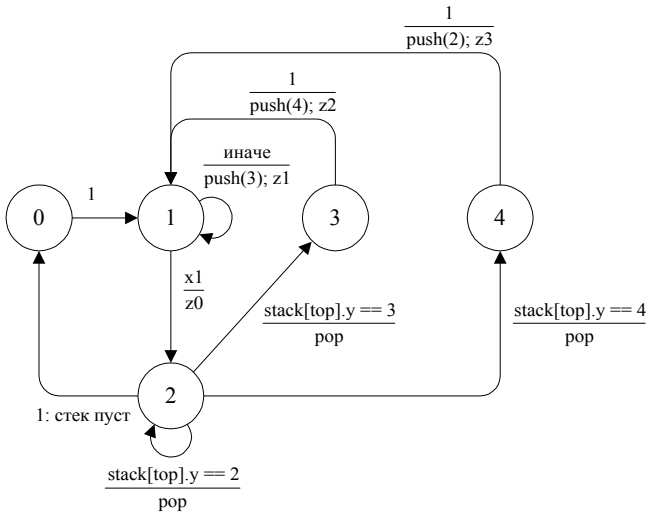

Рис. 4. Граф переходов, моделирующий работу рекурсивной функции

7. Граф переходов на рис. 4 может быть упрощен за счет исключения неустойчивых вершин (кроме вершины с номером 0). Такой граф переходов приведен на рис. 5.

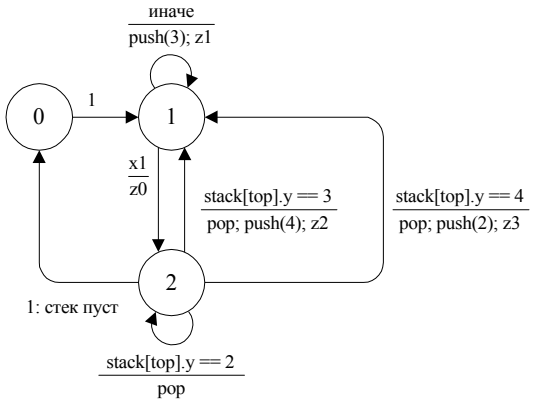

Рис. 5. Упрощенный граф переходов

8. Строится автоматная программа, содержащая функции для работы со стеком и цикл do-while, телом которого является оператор switch, формально и изоморфно построенный по графу переходов [9].

Программа, построенная по графу переходов (рис. 4) и содержащая функции для работы со стеком, приведена на листинге 3. В этой программе операторы, реализующие дуги, исходящие из вершины с номером 2 (за исключением дуги 2–0), закомментированы и заменены одним оператором pop(&y, &i, &j, &k). Такое упрощение программы, однако, приводит к тому, что по указанному оператору невозможно определить, в какое состояние будет выполнен переход.

ЛИСТИНГ 3. Автоматная программа, моделирующая рекурсию

```
#include <stdio.h>

// Элемент стека.
typedef struct
{
  int y, i, j, k ;
} stack_t ;

stack_t  stack[200] ;  // Стек.
int      top = -1 ;    // Индекс верхнего элемента в стеке.
```

```
push( int y, int i, int j, int k )
{
  top++ ;
  stack[top].y = y ;
  stack[top].i = i ; stack[top].j = j ; stack[top].k = k ;
  printf( "push {%d,%d,%d,%d}: ", y, i, j, k ) ; show_stack() ;
}

pop( int *y, int *i, int *j, int *k )
{
  printf( "pop  {%d,%d,%d,%d}: ", stack[top].y, stack[top].i,
                                  stack[top].j, stack[top].k ) ;
  if( y ) *y = stack[top].y ;
  *i = stack[top].i ; *j = stack[top].j ; *k = stack[top].k ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

// Вывод содержимого стека для протоколирования.
void show_stack()
{
  int i ;
  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d,%d} ", stack[i].y, stack[i].i, stack[i].j, stack[i].k ) ;
  printf( "\n" ) ;
}

void hanoy( int i, int j, int k )
{
  int y = 0 ;

  do
    switch( y )
    {
      case 0:
                                    y = 1 ;
      break ;

      case 1:
        if( k == 1 ) { move( i, j ) ;  y = 2 ; }
        else
        {
          push( 3, i, j, k ) ;
          j = 6-i-j ; k-- ;
        }
      break ;

      case 2:
        if( stack_empty() )         y = 0 ;
        else
          pop( &y, &i, &j, &k ) ;
        /* Эта операция pop() заменяет закомментированные операторы,
           но при этом теряется однозначное соответствие программы
           графу переходов. */
        /*
        if( stack[top].y == 4 )
        {
          pop( NULL, &i, &j, &k ) ;    y = 4 ;
        }
        else
        if( stack[top].y == 3 )
        {
          pop( NULL, &i, &j, &k ) ;    y = 3 ;
        }
        else
        if( stack[top].y == 2 )
        { pop( NULL, &i, &j, &k ) ; }
        */
      break ;
```

```
      case 3:
        push( 4, i, j, k ) ;
        k = 1 ;                          y = 1 ;
      break ;

      case 4:
        push( 2, i, j, k ) ;
        i = 6-i-j ; k-- ;                y = 1 ;
      break ;
    }
  while( y != 0 ) ;
}

void move( i, j )
{
  printf( "%d -> %d\n", i, j ) ;
}

int main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
  return 0 ;
}
```

Возможны и другие варианты реализации функций pop() и push(), например, с передачей в качестве параметра указателя на переменную типа stack_t.

Если функцию hanoy() в этой программе реализовать не по графу переходов (рис. 4), а по графу, полученному после его упрощения (рис. 5), то она сокращается (листинг 4).

ЛИСТИНГ 4. Автоматная программа с минимизированным числом состояний, моделирующая рекурсию

```
void hanoy( int i, int j, int k )
{
  int y = 0 ;

  do
    switch( y )
    {
      case 0:
                                         y = 1 ;
      break ;

      case 1:
        if( k == 1 ) { move( i, j ) ;  y = 2 ; }
        else
        {
          push( 3, i, j, k ) ;
          j = 6 - i - j ; k-- ;
        }
      break ;

      case 2:
        if( stack_empty() )              y = 0 ;
        else
        if( stack[top].y == 4 )
        {
          pop( NULL, &i, &j, &k ) ;
          push( 2, i, j, k ) ;
          i = 6 - i - j ; k-- ;          y = 1 ;
        }
        else
        if( stack[top].y == 3 )
        {
          pop( NULL, &i, &j, &k ) ;
          push( 4, i, j, k ) ;
          k = 1 ;                        y = 1 ;
        }
        else
```

```
        if( stack[top].y == 2 )
        { pop( NULL, &i, &j, &k ) ; }
      break ;
    }
  while( y != 0 ) ;
}
```

Отметим, что при такой реализации, в отличие от программы, приведенной в листинге 3, не удается выполнить ее дальнейшее упрощение за счет замены трех условий переходов одной функцией pop().

Выше был расмотрен метод, обеспечивающий построение автоматов Мили. Аналогичный метод может быть предложен и для построения автоматов Мура.

Второй метод: обход дерева действий

Дерево декомпозиции (рис. 1), вершинами которого являются рекурсивные вызовы, может быть преобразовано в дерево действий, выполняемых рассмотренной рекурсивной программой, путем исключения всех вершин кроме тех, в которых выполняется перекладывание (рис. 6).

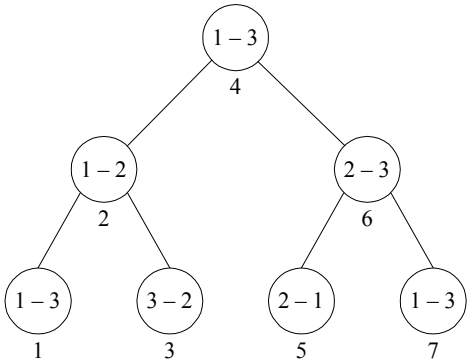

Рис. 6. Дерево действий при N = 3

Это дерево обладает следующим свойством: для вершины с пометкой (i,j) вершина левого поддерева имеет пометку (i,6–i–j), а вершина правого поддерева — (6–i–j,j).

Обход полученного дерева может быть выполнен как рекурсивно, так и итеративно. Построение итеративного алгоритма обхода может рассматриваться как способ раскрытия рекурсии. При этом необходимая последовательность перекладываний получается в результате обхода этого дерева слева направо. Такому обходу соответствуют числа, указанные рядом с каждой из вершин.

Построим итеративный алгоритм обхода этого дерева. Не храня его в памяти, будем осуществлять двоичный поиск каждой вершины, начиная с корневой, так как для нее известны номера стержней и способ определения их номеров для корневых вершин левого и правого поддеревьев. Например, для вершины с номером 5, на первом шаге алгоритма осуществляется переход от вершины 4 к ее правому поддереву — вершине 6, так как пять больше четырех. На втором шаге осуществляется переход от вершины 6 к ее левому поддереву — искомой вершине 5, так как пять меньше шести. Таким образом, несмотря на то, что обход дерева осуществляется слева направо, алгоритм работает сверху вниз.

Листинг 5 содержит программу, реализующую этот алгоритм.

ЛИСТИНГ 5. Итеративная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int j, int k )
{
  int max_nodes = (1 << k) - 1 ; // Всего вершин.
  int root = 1 << (k-1) ;        // Номер корневой вершины.
  int node ;                     // Номер искомой вершины в дереве.
```

```
  // Определить номера стержней для каждой вершины в дереве.
  for( node = 1 ; node <= max_nodes ; node++ )
  {
    int a = i, b = j ;
    // Начальная позиция поиска соответствует корневой вершине.
    int current = root ;
    // Изменение номера вершины при переходе к следующему поддереву.
    int ind = root / 2 ;

    // Двоичный поиск нужной вершины.
    while( node != current )
    {
      if( node < current )
      {
        // Искомая вершина в левом поддереве.
        b = 6-a-b ;
        current -= ind ; // Переход к левому поддереву.
      }
      else
      {
        // Искомая вершина в правом поддереве.
        a = 6-a-b ;
        current += ind ; // Переход к правому поддереву.
      }
      // Разница в номерах вершин при переходе к
      // следующему поддереву уменьшается в два раза.
      ind /= 2;
    }

    // Номера стержней для рассматриваемой вершины определены.
    printf( "Вершина %d. %d -> %d\n", node, a, b ) ;
  }
}

void main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}
```

Построим схему этой программы (рис. 7).

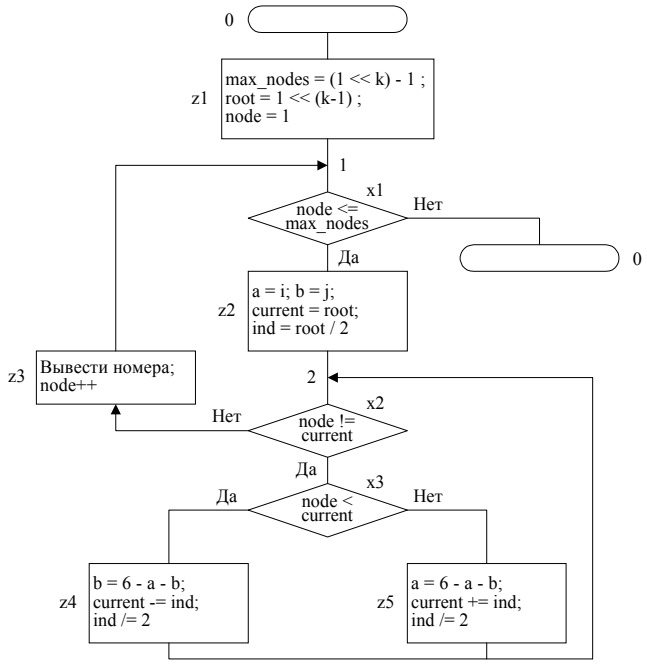

Рис. 7. Схема программы обхода дерева действий

В соответствии с методикой, изложенной в работе [9], построим по этой схеме граф переходов автомата Мили. При этом состояниям автомата Мили на схеме программы соответствуют точки, следующие за операторными вершинами. Нулевому состоянию соответствуют вершины схемы, обозначающие начало и конец программы. Построенный граф переходов приведен на рис. 8.

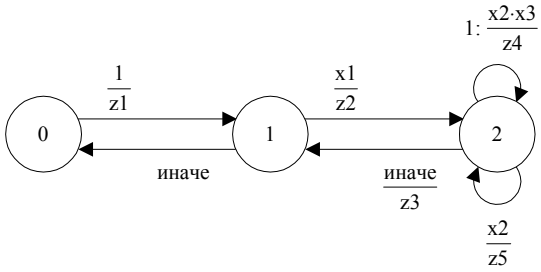

Рис. 8. Граф переходов автомата, реализующего обход дерева действий

Программа, реализующая этот автомат, приведена на листинге 6.

ЛИСТИНГ 6. Автоматная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int j, int k )
{
  int y = 0 ;
  int max_nodes, root, node, a, b, current, ind ;

  do
    switch( y )
    {
      case 0:
        max_nodes = (1 << k) - 1 ;
        root = 1 << (k-1) ;
        node = 1 ;                    y = 1 ;
      break ;

      case 1:
        if( node <= max_nodes )
        {
          a = i ; b = j ;
          current = root ;
          ind = root / 2 ;            y = 2 ;
        }
        else                          y = 0 ;
      break ;

      case 2:
        if( node != current && node < current )
        {
          b = 6-a-b ;
          current -= ind ;
          ind /= 2 ;
        }
        else
        if( node != current )
        {
          a = 6-a-b ;
          current += ind ;
          ind /= 2 ;
        }
        else
        {
          printf( "Вершина %2d. %d -> %d\n", node, a, b ) ;
          node++ ;                    y = 1 ;
        }
      break ;
    }
  while( y != 0 ) ;
}

void main()
{
  int input = 4 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}
```

Отметим, что при использовании изложенного метода номер перекладываемого диска явно не указывается. Эти номера могут быть определены с помощью подхода, изложенного в работе [4], который состоит в следующем: строится таблица, строки которой содержат двоичное представление номера шага. При этом номер разряда двоичного числа, в котором размещена "младшая единица" (при условии, что счет разрядов начинается с единицы), является номером перекладываемого на данном шаге диска.

Ниже приведена такая таблица для N = 3.

Таблица. Определение номера диска.

Номер шага	Номер разряда			Номер диска
	3	2	1	
1	0	0	1	1
2	0	1	0	2
3	0	1	1	1
4	1	0	0	3
5	1	0	1	1
6	1	1	0	2
7	1	1	1	1

Как отмечено в работе [3], последовательность номеров перекладываемых дисков является палиндромом.

Отметим, что аналитически номер перекладываемого диска может быть определен как единица плюс количество делений номера шага на два без остатка. При этом для нечетных номеров шагов количество делений на два без остатка равно нулю, и, поэтому, номер диска равен единице. Таким образом, на каждом нечетном шаге всегда перекладывается диск наименьшего диаметра.

Обобщая изложенное выше, построим дерево решения задачи (рис. 9), которое содержит исчерпывающую информацию для перекладывания дисков. При этом для каждой вершины снизу указан номер шага, а внутри — номера диска и стержней, участвующих в перекладывании.

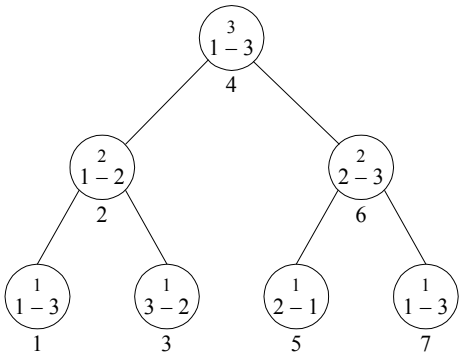

Рис. 9. Дерево решения задачи при N = 3

Из рассмотрения рис. 9 следует также, что номера перекладываемых дисков во всех вершинах одного уровня равны и совпадают с номером этого уровня.

Третий метод: непосредственное перекладывание дисков

Если алгоритм решения рассматриваемой задачи задавать непосредственно в терминах объекта управления (дисков и стержней), как это предлагается П. Бьюнеманом и Л. Леви [10], то его удается описать в виде графа переходов автомата с двумя состояниями (рис. 10).

Этот автомат по очереди выполняет всего два действия: перекладывает наименьший диск циклически по часовой стрелке (z1) и перекладывает единственно возможный диск, кроме наименьшего (z2). После выполнения действия z1 автомат проверяет условие завершения алгоритма (x1).

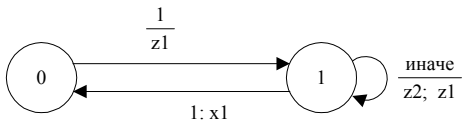

Рис. 10. Граф переходов при непосредственном перекладывании дисков

Текст программы, реализующей описываемый автомат, приведен в листинге 7. В этой программе функции z1, z2 и x1 могут быть реализованы по-разному. Например, определение номеров перекладываемого диска и участвующих в перекладывании стержней может выполняться с помощью перебора (как это имеет место ниже), либо аналитически, например, как это предложено в работе [11].

Особенность рассматриваемой программы состоит в том, что несмотря на простоту автомата, она содержит достаточно много строк, так как выполняемые в ней действия z1 и z2 являются сложными. При этом программа непосредственно управляет дисками, и поэтому необходимо запоминать их расположение и реализовать соответствующие вспомогательные функции.

Кроме того, в используемом алгоритме направление перекладывания первого диска (функция z1) зависит от четности числа дисков и номера стержня, на который их требуется переложить. При перекладывании дисков с первого на третий стержень, первый диск следует перекладывать в порядке номеров стержней 1–2–3 при четном количестве дисков, и в порядке 1–3–2 при их нечетном количестве.

ЛИСТИНГ 7. Автоматная программа, реализующая непосредственное перекладывание дисков

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int   y = 0 ;
int   N ;             // Количество дисков.
int   dest ;          // На какой стержень перекладываем.
int   step = 0 ;      // Номер текущего шага.
int   max_steps = 0 ; // Необходимое количество шагов.
int   first_on = 1 ;  // На каком стержне находится первый диск.

// Состояние объекта управления - "содержимое" стержней.
char  s[4][100] = { "", "1", "", "" } ;

void main()
{
  int input = 14 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}

void hanoy( int from, int to, int disk_num )
{
  int i ;

  N = disk_num ;
  max_steps = (1 << N) - 1 ;
  dest = to ;

  // Заполнить первый стержень дисками.
  for( i = 2 ; i <= N ; i++ )
  {
    sprintf( s[0], "-%d", i ) ;
    strcat( s[1], s[0] ) ;
  }

  do
    switch( y )
    {
      case 0:
        z1() ;              y = 1 ;
      break ;
```

```
      case 1:
        if( x1() )          y = 0 ;
        else
        { z2() ; z1() ; }
      break ;
    }
  while( y != 0 ) ;

  // Вывести результат.
  printf( "\nРезультат:\n" ) ;
  for( i = 1 ; i <= 3 ; i++ )
    printf( "%d: %s\n", i, s[i] ) ;
}

// Проверить, завершено ли перекладывание.
int x1() { return step >= max_steps ; }

// Переложить первый диск по часовой стрелке.
void z1()
{
  int from = first_on ;
  int i ;

  // Определить номер следующего стержня.
  if(    (dest == 2 && N%2 != 0)
      || (dest == 3 && N%2 == 0))
    first_on = (from + 1)%3 ; // Порядок стержней 1-2-3.
  else
    first_on = (from + 2)%3 ; // Порядок стержней 1-3-2.
  if( first_on == 0 ) first_on = 3 ;
  move( 1, from, first_on ) ;
}

// Переложить единственный возможный диск, кроме наименьшего.
void z2()
{
  int  i, j ;
  int  disk_from, disk_to ;

  // Определить перекладываемый диск.
  for( i = 1 ; i <= 3 ; i++ )
  {
    disk_from = disk_on(i) ;
    if( disk_from > 1 )
      // Определить на какой стержень перекладывать.
      for( j = 1 ; j <= 3 ; j++ )
      {
        disk_to = disk_on(j) ;
        if( disk_to == 0 || disk_from < disk_to )
        {
          move( disk_from, i, j ) ;
          return ;
        }
      }
  }
}

// Вернуть номер диска на указанном стержне.
int disk_on( int s_num ) { return atoi( s[s_num] ) ; }

// Переложить заданный диск.
int move( int disk, int from, int to )
{
  char  *str_pos = strchr( s[from], '-' ) ;

  if( str_pos == NULL )
    s[from][0] = 0 ;
  else
    strcpy( s[from], str_pos+1 ) ;

  if( s[to][0] == 0 )
    sprintf( s[to], "%d", disk ) ;
  else
  {
    strcpy( s[0], s[to] ) ;
    sprintf( s[to], "%d-%s", disk, s[0] ) ;
  }
```

```
  step++ ;
  printf( "Шаг %d. Диск %d: %d -> %d\n", step, disk, from, to ) ;

  return 0 ;
}
```

Заключение

Обычно под состоянием программы понимается значение ячеек ее памяти [12]. Однако, такое определение не конструктивно, так как практически в любой программе, реализующей вычислительный алгоритм, число указанных значений чрезвычайно велико. Для решения этой проблемы обратим внимание на то, что в машине Тьюринга небольшое количество состояний в автомате может управлять произвольным количеством состояний на ленте [6]. Такая же ситуация имеет место и в рассмотренных примерах, в которых автомат, содержащий два или три состояния, управляет 2^N состояниями "объекта управления", где N — число дисков.

Отметим, что приведенные выше графы переходов по структуре различны, но порождают одинаковые последовательности перекладываний.

Несмотря на то, что длина автоматных программ превышает размер традиционных, такие программы обладают тем преимуществом, что по их тексту может быть формально построен граф переходов, который является наиболее компактной и удобной для визуализации и документирования графической формой представления программ и их алгоритмов.

В заключение отметим, что настоящая работа во многом посвящена вопросу устранения рекурсии, как и работа [13]. Однако она отличается от последней тем, что посвящена такой разновидности итеративных программ, как автоматные программы, для построения которых первый из предложенных методов является формализованным и применим для устранения не только "концевой рекурсии".

Настоящая работа была выполнена на кафедре "Компьютерные технологии" Санкт-Петербургского государственного института точной механики и оптики (технического университета). Авторы благодарны студенту этой кафедры Крылову Р.А., который в своей курсовой работе начал рассмотрение второго из изложенных методов. Работа выполнена при поддержке Российского фонда фундаментальных исследований по гранту №02-07-90114 "Разработка технологии автоматного программирования".

Литература

1. Седжвик Р. Фундаментальные алгоритмы на C++. Киев: ДиаСофт, 2001.
2. Грэхем Р., Кнут Д., Поташник О. Конкретная математика. М.: Мир, 1998.
3. Романовский И.В., Столяр С.Е. Стек и его использование. http://ips.ifmo.ru.
4. Гарднер М. Математические головоломки и развлечения. М.: Мир, 1971.
5. Бобак И. Алгоритмы: "возврат назад" и "разделяй и влавствуй" //Программист. 2002. №3.
6. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному //Мир ПК. 2002. №2.
7. Стивенс Р. Delphi. Готовые алгоритмы. М.: ДМК, 2001.
8. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. М.: Вильямс, 2000.
9. Шалыто А.А., Туккель Н.И. Реализация вычислительных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2001. №6.
10. Анисимов А.В. Информатика. Творчество. Рекурсия. Киев: Наукова думка, 1988.
11. Быстрицкий В.Д. Ханойские башни. http://alglib.chat.ru/paper/hanoy.html
12. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, СПб.: Невский диалект, 1998.
13. Боровский А. Как избежать рекурсии // Программист. 2002. №6.

ОБ АВТОРАХ

Шалыто Анатолий Абрамович — ученый секретарь Федерального научно-производственного центра (ФНПЦ) — федерального государственного унитарного предприятия (ФГУП) "НПО "Аврора"", профессор кафедры "Компьютерные технологии" СПбГИТМО (ТУ). С ним можно связаться по адресу: mail@avrorasystems.com ("для Шалыто").

Туккель Никита Иосифович — инженер-программист ФНПЦ — ФГУП "НПО "Аврора"". С ним можно связаться по адресу: cynical@mail.ru.

Шамгунов Никита Назимович — трехкратный чемпион Урала по программированию, двухкратный финалист чемпионатов мира по программированию ACM, аспирант СПбГИТМО (ТУ).