В. И. Шелехов, А. А. Алгазин

ОПЫТ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ ЗАДАЧИ НАХОЖДЕНИЯ КРАТЧАЙШЕГО ПУТИ МЕЖДУ ДВУМЯ ГОРОДАМИ

В данной работе демонстрируются разнообразные особенности технологии предикатного программирования. В предикатной программе применяются операции с разными структурами данных: массивами, множествами и последовательностями. Множества используются в качестве типа индексов одномерных и двухмерных массивов. Используются языковые конструкции в функциональном стиле, результатами которых являются тройки или четверки значений. Демонстрируется типовая техника работы с гиперфункциями. Описано построение двух вариантов предикатной программы. Показана техника кодирования множеств и последовательностей с помощью массивов при реализации эффективной императивной программы. Читателю предварительно следует ознакомиться с работой [1] настоящего сборника.

Допустим, имеется множество городов с произвольной сетью дорог между ними. Требуется найти кратчайший маршрут между двумя городами. Эта задача известна как задача нахождения минимального пути в графе.

Будем считать, что города пронумерованы от 1 до n. Каждый город идентифицируется его номером. Пусть a и b — города, между которыми требуется найти кратчайший маршрут. Обозначим через reached[i] множество городов, соединенных с городом i непосредственно. Обозначим через len[i,j] расстояние между городами i и j, непосредственно соединенных дорогой, т.е. если $i \in$ reached[j]. Отношение непосредственного соединения двух городов i и j является симметричным, причем len[i,j] = len[j,i]. Использование матрицы len подразумевает не более одного непосредственного соединения для любой пары городов.

Приведенные выше определения отображаются следующими описаниями на языке P:

```
type setn(nat m) = set 1..m;
nat n;
array 1..n of setn(n) reached;
array { nat i, j: i in reached[j] } of real len;
type Setn = setn(n);
```

Множество { **nat** i,j**:** i **in** reached[j] }, используемое в качестве типа индексов массива len, определяет множество кортежей (i, j), для которых i ∈ reached[j]. Во многих предлагаемых алгоритмах для исходной задачи массив len определяется для всех пар городов, полагая len[i,j]=L при отсутствии непосредственного соединения между городами i и j, где L — достаточно большое число, аналог машинной бесконечности. Подобное решение относится к стадии реализации предикатной программы. Выбирая конкретный способ реализации переменной reached в императивной программе, мы, в принципе, могли бы закодировать ложное значение условия i **in** reached[j] как len[i,j] = L.

Переменные n, reached и len являются глобальными по отношению к программе. Последнее описание определяет обозначение Setn, в котором параметр n отсутствует, но подразумевается. Это обозначение вводится для сокращения записи.

Исходную задачу представим предикатом

Route(**nat** a, b**: real** distance, **seq nat** route |) ,

где distance — длина кратчайшего маршрута, route — последовательность городов, образующих кратчайший маршрут. Вторая альтернатива гиперфункции Route реализуется при отсутствии маршрута между a и b.

Нахождение кратчайшего пути реализуется движением по множеству путей от a до b. При прохождении города k определяется dist[k] — минимальное расстояние от a до k по разным пройденным путям и соответствующий кратчайший маршрут от a до k. Для каждого проходимого города k достаточно хранить prev[k] — предыдущий город по кратчайшему маршруту; по массиву prev легко построить кратчайший маршрут.

Наиболее простым решением считается поиск в глубину (см. [2] и перевод [3]) движением от города a по всем путям с последовательным уточнением расстояния dist[k] для каждого города k. Повторное движение от города k реализуется в случае обнаружения нового, более короткого, пути до города k. Очевидно, что при исчерпании всей сети дорог dist[k] будет определять минимальное расстояние по всему множеству путей от a до k.

Построим предикатную программу алгоритма поиска в глубину. Определение предиката Route реализуется через обобщающий предикат RouteDeep, вычисляющий массивы dist' и prev' движением от города k, используя значения массивов dist и prev, вычисленные к моменту достижения города k. Массивы dist и prev имеют, соответственно, следующие типы:

```
type arSreal (Setn s)= array s of real;
type arSnat (Setn s)= array s of 1..n;
```

Типом индексов является множество s, являющееся параметром типа. Это множество должно быть параметром предиката RouteDeep и быть также объектом вычисления. Обозначим через ts множество городов, пройденных перед приходом в город k, – это множество является типом индексов для массива prev. Массив dist, кроме того, определен для города a (dist[a] = 0), т.е. его тип индексов есть ts+a. Предикат должен вычислить множество ts', являющееся типом индексов для dist' и prev'. Итак, заголовок RouteDeep имеет следующий вид:

```
RouteDeep(nat a, b, k, Setn ts*, arSreal(ts+a) dist, arSnat(ts) prev :
                                  arSreal (ts'+a) dist', arSnat (ts') prev')
```

Напомним, что литера ' в имени результирующего параметра dist' означает, что в реализации императивной программы переменная dist' будет склеена с переменной dist. Ограничитель "*" после ts определяет также описание результирующего параметра ts'.

Дадим определение предиката Route:

```
Route(nat a, b: real distance, seq nat route | )
   // route, distance — кратчайший маршрут и расстояние от a до b
{     RouteDeep(a, b, a, Setn(), (a: 0), arSnat(Setn()) () :
                    Setn ts, arSreal(ts+a) dist, arSnat(ts) prev );
      if b in ts then
            distance = dist[b] ||
            BuildRoute(a, b, ts, prev, seq nat(b) : route)
            #1
      else
            #2
      end
}
```

В теле определения Setn() является конструктором пустого множества, (a: 0) — есть конструктор массива расстояний с единственным элементом 0 для индекса a, arSnat(Setn())() — конструктор пустого массива предыдущих городов для пустого типа индексов. Вызов BuildRoute строит по массиву prev кратчайший маршрут route с конца, начиная с последовательности из единственного города b (конструктор **seq nat**(b)).

Предикат BuildRoute строит последовательность городов route' в обратном порядке от города c до города a, используя массив prev. Последовательность route есть ранее построенная последовательность от b до c.

```
BuildRoute(nat a,c, Setn ts, arSnat (ts) prev, seq nat route*: )
                // route – маршрут от c до b, route' – маршрут от a до b
{     if c = a then
          route' = route
      else
          nat pre = prev[c];
          BuildRoute (a, pre, ts, prev, seq nat(pre, route) : route')
      end
}
```

Дадим определение предиката RouteDeep, его спецификация приведена выше.

```
RouteDeep(nat a, b, k, Setn ts*, arSreal(ts+a) dist, arSnat(ts) prev :
                          arSreal (ts'+a) dist', arSnat (ts') prev')
    // dist и prev – расстояния и предыдущие города при достижении города k
    // dist' и prev' – расстояния и предыдущие города при достижении города b
{     if k = b then
              ts' = ts || dist' = dist || prev' = prev
      else
              RouteDeepSet(a, b, k, reached[k], ts, dist, prev : ts', dist', prev')
      end
}
```

Предикат RouteDeepSet вычисляет массивы dist' и prev' движением от города k, используя значения массивов dist и prev, вычисленные ранее. Движение от города k реализуется через непосредственно соединенные с k города, принадлежащие множеству k_next. В вызове RouteDeepSet внутри определения RouteDeep множество k_next = reached[k] — есть множество всех достижимых из k.

В общем случае параметр k_next определяет некоторое подмножество городов, непосредственно достижимых из города k. Предполагается, что пути, идущие через город k и проходящие через города подмножества reached[k] \ k_next, уже просмотрены, причем dist и prev определены по всем этим просмотренным путям.

В теле RouteDeepSet используется вызов гиперфункции RouteStep, вычисляющей эффект продвижения от города k к городу town. Вторая альтернатива гиперфункции реализуется при следующем условии: город town достигается не в первый раз и длина нового пути до town не меньше ранее вычисленного. При данном условии дальнейшее продвижение через город town не имеет смысла, поскольку не изменит значений dist и prev.

```
RouteDeepSet(nat a, b, k, Setn k_next, Setn ts*,
                arSreal(ts+a) dist, arSnat(ts) prev :
                arSreal (ts'+a) dist', arSnat (ts') prev')
   // dist и prev — расстояния и предыдущие города при достижении
   // города k, а также просмотром сети дорог от городов множества
   // reached[k] \ k_next
{  split k_next -> ( | nat town, Setn k_rest)
   do  ts'=ts || dist'=dist || prev'=prev
   do
      split RouteStep(a, k, town, ts , dist, prev:
                        Setn ts0 , arSreal(ts0+a) dist0, arSnat(ts) prev0 #1 | #2)
      do    RouteDeep(a, b, town, ts0, dist0, prev0 : ts1, dist1, prev1);
            RouteDeepSet(a, b, k, k_rest, ts1, dist1, prev1 :
                              ts', arSreal (ts'+a) dist', arSnat (ts') prev')
      do   ts'=ts || dist'=dist || prev'=prev
      end
   end
}
```

Если множество городов k_next пусто, то первая альтернатива объемлющего оператора расщепления **split** очевидным образом определяет результат. В противном случае множество k_next разлагается на город town и множество остальных городов k_rest, причем реализуется вторая альтернатива **split**. Здесь в заголовке вложенного оператора **split** вызов предиката RouteStep вычисляет величины ts0, dist0 и prev0, определяющие эффект продвижения от города k к городу town. Для этих значений в первой альтернативе **split** к городу town применяется предикат RouteDeep с получением ts1, dist1 и prev1. Последние значения определяют результат просмотра всех путей, ведущих из города town. Наконец, остается применить RouteDeepSet к множеству k_rest для получения требуемых значений ts', dist' и prev' по всем путям от a до b.

```
RouteStep(nat a, k, town, Setn ts* , arSreal(ts+a) dist, prev:
                                          arSreal(ts'+a) dist', arSnat(ts') prev' | )
   //  ts', dist' и prev' получаются из ts, dist и prev продвижением от города k к town
{       if !(town in ts) then
                ts' = ts+town ||
                arSreal (ts+town+a) dist' = dist + (town: dist[k]+len[k,town]) ||
                arSnat (ts+town) prev' = prev+(town: k)
                #1
        elsif dist[town] > dist[k]+len[k,town] then
                ts' = ts ||
```

```
                arSreal (ts+a) dist' = dist(town: dist[k]+len[k,town]) ||
                arSnat (ts) prev' = prev(town: k)
                #1
        else
                #2
        end
}
```

Конструктор (town: k) определяет массив с единственным элементом k, индекс которого есть town. Конструктор prev+(town: k) определяет массив, получающийся добавлением к массиву prev одного нового элемента k с индексом town. Конструктор prev(town: k) определяет массив, получающийся из массива prev заменой в нем элемента с индексом town на k.

Определение RouteStept можно переписать в функциональном стиле:

```
RouteStep(nat a, k, town, Setn ts* , arSreal(ts+a) dist, prev:
                                        arSreal(ts'+a) dist', arSnat(ts') prev' | )
    //  ts', dist' и prev' получаются из ts, dist и prev продвижением от города k к town
{     if !(town in ts) then
                ts+town, dist + (town: dist[k]+len[k,town]), prev+(town: k)
                #1
        elsif dist[town] > dist[k]+len[k,town] then
                ts, dist(town: dist[k]+len[k,town]), prev(town: k)
                #1
        else
                #2
        end
}
```

Приведенные выше определения составляют предикатную программу поиска в глубину. Рекурсия в определении RouteDeepSet является хвостовой. При построении императивной программы можно заменить рекурсию циклом и подставить определение RouteDeepSet на место вызова в теле RouteDeep. Однако рекурсия в полученном определении RouteDeep уже не будет хвостовой.

Алгоритм поиска в глубину является достаточно медленным с оценкой $O(n^3)$, где n — число городов, хотя приведенная версия этого алгоритма имеет оценку $O(m \times n^3)$, где m — среднее число дорог из одного города. Имеются улучшения алгоритма поиска в глубину. Один из них базируется на использовании некоторой известной априорной длины до города b,

большей, чем dist[b] [2]. Это позволяет ограничить поиск путями, длина которых меньше априорной.

Альтернативным подходом в решении задачи нахождения кратчайшего пути является алгоритм **поиска в ширину**. Алгоритм определяет распространение **фронта** городов, находящихся на разных путях движения от a до b. Структура "фронт" есть некоторое подмножество городов. Первоначально фронт образуют города, непосредственно соединенные с городом a. Допустим, город k принадлежит фронту. Распространение фронта от города k реализуется следующим образом: город k удаляется из фронта, а любой город, непосредственно соединенный с k, добавляется к фронту, если он не был ранее пройден.

Наиболее известным является алгоритм Э. Дейкстры [4]. Основная идея алгоритма состоит в выборе самого ближайшего (к городу a) города k из фронта в качестве кандидата для очередного распространения фронта. Как следствие, любые другие пути к городу k, обнаруживаемые позднее через другие города фронта, будут длиннее, и значит, вычисленные значения dist[k] и prev[k] будут окончательными.

В алгоритме Э. Дейкстры нет структуры «фронт». Вместо нее используется множество пройденных городов — это города, которые когда-либо попадали во фронт. Поиск ближайшего города k с минимальным dist[k] ведется не по фронту, а по всем городам. При работе алгоритма город b рано или поздно попадает во фронт. Как только город b будет выбран для распространения фронта, мы будем иметь требуемый кратчайший путь от a до b, который необходимо выбрать из массива prev.

Наш алгоритм, представленный в настоящей работе, явно использует структуру «фронт», и поэтому быстрее алгоритма Э. Дейкстры и сложнее его. Ниже определяется предикатная программа, а также ее трансформация в эффективную императивную программу.

Определим prev[a]=a, для того чтобы массивы prev и dist имели один и тот же тип индексов ts. Множество городов, принадлежащих фронту, определяется переменной front. Множество городов, пройденных в алгоритме, за исключением городов фронта, обозначим через passed. Справедливо следующее равенство: ts=front+passed. Если при распространении фронта встречается город из passed, то новый путь до этого города не будет короче предыдущего, и поэтому города из passed игнорируются.

Определение основного предиката Route использует предикат

```
RouteFront(nat a, b, k, Setn passed, front, ts*,
                        arSreal(ts) dist, arSnat(ts) prev :
                        arSreal (ts') dist', arSnat (ts') prev' | ),
```

вычисляющий ts', dist' и prev' распространения фронта от города k, используя значения массивов dist и prev, соответствующие фронту front до его распространения от города k. Параметр passed определяет множество пройденных городов, за исключением городов фронта. Вторая альтернатива RouteFront реализуется при отсутствии маршрута между a и b.

Дадим определение предиката Route:

```
Route(nat a, b: real distance, seq nat route | )
   // route, distance — кратчайший маршрут и расстояние от a до b
{       RouteFront(a, b, a, Setn(a), Setn(), Setn(a), (a: 0), (a: a) :
                        Setn ts, arSreal(ts) dist, arSnat(ts) prev | #2 );
        dist[b],
        BuildRoute(a, b, ts, prev, seq nat(b))
        #1
}
```

Фрагмент в функциональном стиле "dist[b], BuildRoute(a, b, ts, prev, **seq nat**(b))" эквивалентен паре операторов:

```
distance=dist[b]; BuildRoute(a, b, ts, prev, seq nat(b): route)
```

Определение предиката BuildRoute приведено выше для первой версии Route поиска в глубину.

Дадим определение предиката RouteFront.

```
RouteFront(nat a, b, k, Setn passed, front, ts*,
                            arSreal(ts) dist, arSnat(ts) prev :
                            arSreal (ts') dist', arSnat (ts') prev' | )
// k — ближайший город из текущего фронта,
// front — текущий фронт, за исключением k, passed — пройденные, включая k,
// passed+front=ts, passed∩front=∅
// dist и prev — расстояния и предыдущие города при достижении города k
// dist' и prev' — расстояния и предыдущие города при достижении города b
{ if k = b then
          ts, dist, prev
          #1
  else
          SpreadFront(k, reached[k], passed, front, ts, dist, prev :
                        Setn front1,ts1, arSreal(ts1) dist1, arSnat(ts1) prev1);
          Nearest(front1, ts1, dist1 : nat town_nea  | #2 );
          RouteFront(a, b, town_nea, passed+town_nea, front1\town_nea,
                        ts1, dist1, prev1 : ts', dist', prev' #1 | #2 )
  end
}
```

В приведенном определении вызов SpreadFront реализует продвижение текущего фронта front от города k до непосредственно доступных из него городов, вычисляя расстояния для новых городов и уточняя его для старых. Вызов предиката Nearest для городов фронта front1 определяет ближайший город town_nea к городу a. Вторая ветвь Nearest реализуется для пустого front1 — в этом случае город b оказывается недостижим. Город town_nea назначается в качестве следующего города для продвижения фронта в рекурсивном вызове RouteFront.

Дадим определение предиката Nearest. Предикат Nearest определяет ближайший город town_nea к городу a среди городов фронта front1, используя массив расстояний dist1. Если front1 пуст, реализуется вторая альтернатива.

```
Nearest(Setn front1, ts1, arSreal(ts1) dist1 : nat town_nea | )
// town_nea — ближайший к городу a среди городов front1
{     front1 -> ( #2 | nat town, Setn front2);
      Nearest1(front2, ts1, dist1, town : town_nea);
      #1
}
```

Оператор разложения s -> (| **nat** e, Setn s1) реализует итерацию множества s: если s — пусто, то реализуется первая альтернатива, иначе в множестве s выбирается элемент e и определяется остаток множества s1 без элемента e, т.е. справедливо s = {e} + s1.

Предикат Nearest1 является обощением предиката Nearest и реализует определение ближайшего города town_nea'. Фронт front2 является остатком фронта front1 без его начальной части, по которой уже определен ближайший город town_nea.

```
Nearest1(Setn front2,ts1, arSreal(ts1) dist1, nat town_nea* : )
  // town_nea' — ближайший среди town_nea и городов множества front2
{  split front2 -> ( | nat town, Setn front3)
   do  town_nea
   do  Nearest1(front3, ts1, dist1,     if dist1[town] < dist1[town_nea] then
                                              town
                                        else
                                              town_nea
                                        end
               )
   end
}
```

Предикат SpreadFront реализует продвижение текущего фронта от города k на некоторое подмножество k_next городов, непосредственно соединенных с городом k. Переменная front определяет текущий фронт за исключением города k. Предикат также вычисляет расстояния для новых городов, формируя массивы dist′ и prev′. Для старых городов, принадлежащих фронту, корректируются расстояния, если они оказались короче предыдущих.

```
SpreadFront(nat k, Setn k_next, passed, front*, ts*,
                arSreal(ts) dist, arSnat(ts) prev :
                arSreal(ts') dist', arSnat(ts') prev');
// k — ближайший город из текущего фронта,
// k_next — города, на которые распространяется фронт,
// front — текущий фронт, за исключением k,
// passed — пройденные, включая k,
// passed+front=ts, passed∩front=∅,
// dist и prev — расстояния и предыдущие города
// до распространения фронта
// front', ts', dist' и prev' — после распространения фронта
// на множестве k_next
{   split k_next -> ( | nat town, Setn k_rest )
    do      front, ts, dist, prev
    do SpreadFront(k, k_rest, passed,
                        if town in passed then
                                front, ts, dist, prev
                        elsif !(town in front) then
                                front+town, ts+town,
                                dist + (town: dist[k]+len[k,town]),
                                prev + (town: k)
                        elsif dist[town] > dist[k]+len[k,town] then
                                front, ts, dist(town: dist[k]+len[k,town]),
                                prev(town: k)
                        else
                                front, ts, dist, prev
                        end
                    )
    end
}
```

В приведенном определении рекурсивный вызов SpreadFront является вызовом функции, вычисляющей четверку значений, которая присваивает-

ся результирующим параметрам front', ts', dist' и prev' предиката SpreadFront. Каждая ветвь условного оператора вычисляет четверку значений, подставляемых в качестве фактических параметров в рекурсивном вызове SpreadFront, которые соответствуют формальным параметрам front, ts, dist и prev.

Определения предикатов Route, BuildRoute, RouteFront, Nearest, Nearest1 и SpreadFront вместе с описаниями типов и глобальных параметров составляют полную предикатную программу. Преобразуем ее в эффективную императивную программу.

На первом этапе реализуем склеивание переменных и замену хвостовой рекурсии циклом в каждом из определений предикатов. Предварительно функциональная форма записи преобразуется в предикатную. Приведем группы склеиваемых переменных по всем определениям предикатов. Переменные в каждой группе заменяются первой.

```
SpreadFront:
        k_next <- k_rest; front <- front'; ts <- ts';
        dist <- dist'; prev <- prev';
Nearest1:
        front2 <- front3; town_nea <- town_nea';
Nearest:
        front1 <- front2; town_nea <- town;
RouteFront:
        k <- town_nea; front <- front1;  ts <- ts1, ts';
        dist  <- dist1, dist';  prev <- prev1, prev';
BuildRoute:
        c <- pre; route<- route';
```

Проиллюстрируем первый этап только для определения SpreadFront. Склеивая указанные выше переменные и заменяя рекурсию циклом, получим:

```
SpreadFront(nat k, Setn k_next, passed, front*, ts*,
        arSreal(ts) dist, arSnat(ts) prev :
        arSreal(ts) dist, arSnat(ts) prev);
{  loop
        split k_next -> ( | nat town, k_next )
        do     exit
        do     | front, ts, dist, prev | :=
               if town in passed then
                   front, ts, dist, prev
```

```
                elsif !(town in front) then
                    front+town, ts+town,
                    dist + (town: dist[k]+len[k,town]),
                    prev + (town: k)
                elsif dist[town] > dist[k]+len[k,town] then
                    front, ts, dist(town: dist[k]+len[k,town]),
                    prev(town: k)
                else
                    front, ts, dist, prev
                end
        end
    end
}
```

Далее, вносим групповой оператор на каждую из ветвей условного предложения и раскрываем групповые операторы на каждой ветви:

```
SpreadFront(nat k, Setn k_next, passed, front*, ts*,
        arSreal(ts) dist, arSnat(ts) prev :
        arSreal(ts) dist, arSnat(ts) prev);
{   loop
        split k_next -> ( | nat town, k_next )
        do    exit
        do    if town in passed then

                elsif !(town in front) then
                                        front := front+town;
                                        ts := ts+town;
                                        dist := dist + (town: dist[k]+len[k,town]);
                                        prev := prev + (town: k)
                elsif dist[town] > dist[k]+len[k,town] then
                                        dist := dist(town: dist[k]+len[k,town]);
                                        prev := prev(town: k)
                end
        end
    end
}
```

На втором этапе трансформации предикатной программы подставим определения предикатов на место всех их вызовов. В итоге получим следующую программу:

```
type setn(nat m) = set 1..m;
nat n;
array 1..n of setn(n) reached;
array { nat i, j: i in reached[j] } of real len;
type Setn = setn(n);
type arSreal (Setn s)= array s of real;
type arSnat (Setn s)= array s of 1..n;
Route(nat a, b: real distance, seq nat route | )
{     nat k:=a; Setn passed:= Setn(a); Setn front:= Setn(); Setn ts:= Setn(a);
      arSreal(ts) dist := (a: 0); arSnat(ts) prev := (a: a);
      loop
          if k = b then
             exit
          else
            Setn k_next:= reached[k];
            loop
              split k_next -> ( | nat town, k_next )
              do     exit
              do     if town in passed then
                     elsif !(town in front) then
                             front := front+town;
                             ts := ts+town;
                             dist := dist + (town: dist[k]+len[k,town]);
                             prev := prev + (town: k)
                     elsif dist[town] > dist[k]+len[k,town] then
                             dist := dist(town: dist[k]+len[k,town]);
                             prev := prev(town: k)
                     end
              end
            end;
            Setn font1 := front;
            front1 -> ( #2 | k, front1);
            loop     split front1 -> ( | nat town, Setn front1)
                     do  exit
                     do  if dist[town] < dist[k] then k:=town end
                     end
            end;
            passed := passed+k; front := front\k
          end
      end;
```

```
    distance := dist[b];
    nat c:=b; route:= seq nat(b);
    loop    if c = a then          exit
            else        c := prev[c]; route :=seq nat(c, route))
            end
    end
    #1
}
```

На третьем этапе трансформации предикатной программы объекты структурных типов кодируются через массивы. Объект s типа последовательности или множества кодируется тройкой: массивом s и целыми переменными smin и smax. Значением объекта является вырезка массива s[smin..smax]. В результате кодирования оператор расщепления вида:

```
split s -> ( | nat e, s ) do A do B end
```

с произвольными операторами A и B заменяется условным оператором:

```
if smin>smax then A else nat e:=s[smin]; smin:=smin+1; B end
```

Последовательность route представляется вырезкой route[routemin..n]. Оператор route:=**seq nat**(b) представляется парой операторов: routemin:=n; route[routemin]:=b.

Оператор route:=seq nat(c, route) будет представлен в виде:

```
routemin:=routemin-1; route[routemin]:=c
```

Произвольное множество в качестве типа индексов массива заменяется диапазоном 1..n. Массив len, определенный на множестве пар, представляется двумерным массивом типа **array** 1..n; 1..n **of real.**.

Специфику преобразование покажем для следующего фрагмента программы:

```
Setn font1 := front;
front1 -> ( #2 | k, front1);
loop        split front1 -> ( | nat town, Setn front1)
            do  exit
            do  if dist[town] < dist[k] then k:=town end
            end
end
```

Применяя приведенные выше правила, получим:

```
Setn font1 := front; nat front1max := frontmax;
nat front1min := 1;
if front1min > front1max then #2 end;
k := front[front1min]; front1min := front1min+1;
loop      if front1min > front1max then exit
          else nat town := front1[front1min]; front1min:= front1min+1;
               if dist[town] < dist[k] then k := town end
          end;
end
```

Очевидно, что необходимо специализированное правило преобразования оператора расщепления для константного значения front1min, которое породило бы:

```
if 1 > frontmax then #2 end; k := front[1]; nat  front1min :=2;
```

Во-вторых, нет необходимости в копировании Setn font1 := front, поскольку при итерации множества массив font1 не модифицируется. Учитывая это и заменяя цикл **loop** на **for**, получим:

```
if 1 > frontmax then #2 end;
k := front[1];
for nat frontmin := 2..frontmax do
          nat town := front[frontmin];
          if dist[town] < dist[k] then k := town end
end;
```

Реализация кодирования множеств и последовательностей, а также замена циклов **loop** на циклы **while** и **for** дает следующую программу:

```
type setn(nat m) = array 1..m of 1....m;
nat n;
type Setn = setn(n);
array 1..n of Setn reached;
array 1..n of nat reachedmax;
array 1..n, 1..n of real len;
type arSreal = array 1..n of real;
type arSnat = array 1..n of 1..n;
```

```
Route(nat a, b: real distance, seq nat route | )
{       nat k:=a;
        Setn passed; passed[1] := a; nat passedmax := 1;
        Setn front; nat frontmax := 0; Setn ts; ts[1]:=a; nat tsmax := 1;
        arSreal dist; dist[a] := 0; arSnat prev; prev[a] :=a;
        while  k != b do
              Setn k_next:= reached[k]; nat k_nextmax := reachedmax[k];
              for nat k_nextmin := 1.. knextmax do
                        nat town:=k_next[k_nextmin];
                        if town in passed then
                        elsif !(town in front) then
                                frontmax:=frontmax+1; front[frontmax] := town;
                                tsmax := tsmax+1; ts[tsmax] := town;
                                dist[town] := dist[k]+len[k,town];
                                prev[town] := k
                        elsif dist[town] > dist[k]+len[k,town] then
                                dist[town] := dist[k]+len[k,town];
                                prev[town] := k
                        end
              end;
              if 1 > frontmax then #2 end;
              k := front[1];
              for nat frontmin := 2..frontmax do
                  nat town := front[frontmin];
                  if dist[town] < dist[k] then k := town end
              end;
              passedmax := passedmax+1; passed[passedmax] := k;
              front := front\k
        end;
        distance := dist[b];
        nat c:=b; routemin := n; route[routemin] := b;
        while c != a do
                  c := prev[c]; routemin := routemin-1; route[routemin] := c;
        end
        #1
}
```

В приведенной программе остались нераскрытыми конструкции: town i**n** passed, town **in** front и front:=front\k. Определим операцию **in** для представления множества s в виде вырезки s[min..max] следующим образом:

```
IN(T el, array min..max of T s: bool)
{       if min > max then false
        elseif s[min] = el then true
        else IN(el, s[min+1, max])
        end
}
```

После замены рекурсии в теле IN циклом и подстановки на место операций **in** получим программу, в которой формируемое логическое значение является явно лишним. Этого можно избежать, если определить операцию **in** в форме гиперфункции IN(T el, **array** min..max **of** T s**:** |) с заменой оператора **if** town **in** passed **then** A **else** B **end** на оператор расщепления **split** IN(town, passed**:** |) **do** A **do** B **end.**

Итак, заменяя рекурсию циклом в определении

```
IN(T el, array min..max of T s: | )
{       if min > max then #2
        elseif s[min] = el then #1
        else IN(el, s[min+1,max]: #1 | #2)
        end
}
```

получим требуемую реализацию операции **in**:

```
IN(T el, array min..max of T s: | )
{       loop    if min > max then #2
                elseif s[min] = el then #1
                else min:=min+1
                end
        end
}
```

Покажем результат подстановки реализации **in** для внутреннего цикла **for**, полученный заменой пары вложенных операторов **split** на фрагмент обычной императивной программы.

```
for nat k_nextmin := 1.. knextmax do
        nat town:=k_next[k_nextmin];
        for nat passedmin:=1.. passedmax do
            if passed[passedmin] = town then goto #MP1 end
        end;
        for nat frontmin:=1.. frontmax do
            if front[frontmin] = town then goto #MF1 end
```

```
            end;
            frontmax:=frontmax+1; front[frontmax] := town;
            tsmax := tsmax+1; ts[tsmax] := town;
            dist[town] := dist[k]+len[k,town]; prev[town] := k;
            goto #MP1;
    #MF1: if dist[town] > dist[k]+len[k,town] then
                    dist[town] := dist[k]+len[k,town]; prev[town] := k
            end;
    #MP1:
end;
```

Метка #MF1 соответствует первой ветви внутреннего **split**, а #MP1 — первой ветви внешнего. Использование меток и переходов оказывается неизбежным при раскрытии операторов расщепления. Этот факт свидетельствует, что расщепление предоставляет существенно большие возможности изображения алгоритмов и ему нет аналогов в существующих языках императивного программирования.

Приведенная программа нахождения кратчайшего пути методом распространения фронта, безусловно, быстрее программы поиска в глубину. Однако она имеет ту же самую оценку: $O(m \times n^3)$, где m — среднее число дорог из одного города. Понятно, что предложенная реализация императивной программы не самая эффективная. Например, для множеств front и passed следовало бы использовать другую форму кодирования: в виде логических массивов (битовых шкал). Кроме того, множество ts оказалось ненужным в императивной программе, и его вычисление можно было бы удалить из программы. Реализация этих изменений могла бы улучшить оценку программы до $O(m \times n^2)$.

СПИСОК ЛИТЕРАТУРЫ

1. **Шелехов В.И.** Предикатное программирование: основы, язык, технология // Наст. сб. — С. 7–15.
2. **Stout B.** Smart Moves: Intelligent Path finding // Game Developer. — 1996. — P. 28–35.
3. **Программирование** магических игр. Алгоритмы поиска путей. Поиск путей на графе. Перевод с англ. — http://pmg-ru.narod.ru/russian/stout.htm
4. **Dijkstra E.W.** A note on two problems in connection with graphs // Numerische Mathematik. — 1959.