Навстречу корректным программам

Цель данного документа – отметить, какие вспомогательные средства для нашего интеллекта мы имеем в своем распоряжении для разработки и понимания алгоритмов, продемонстрировать некоторые приемы программирования, которые мы можем попытаться применить к своим задачам без ущерба для понимания, и подчеркнуть потребность в том, чтобы наши программы (т.е. окончательные и промежуточные версии) как можно точнее отражали наше понимание задачи и алгоритма ее решения.

Среди вспомогательных средств для интеллекта я особо хотел бы упомянуть три:

1.Перечисление.

2.Математическая индукция.

3.Абстракция.

Я рассматриваю как применение Перечисления умственные усилия, необходимые для понимания либо последовательной программы, выполняющей фиксированную временную последовательность действий, либо условного оператора, либо оператора выбора (так называемая «конструкция case»). Я считаю, что одно из основных свойств человеческого разума – плохая способность к перечислению. В частности, это означает:

1a) что текст, описывающий фрагмент последовательной программы, должен выполнять небольшое количество действий (то есть соответствующие вычисления могут быть сгруппированы и восприняты как небольшое количество действий, последовательных во времени);

1b) что количество альтернативных ветвей в операторе выбора должно быть невелико.

Математическая индукция упомянута явно, поскольку это стандартный шаблон рассуждений для понимания рекурсивных процедур и (намного чаще встречаемых) циклов; я ограничусь циклами, образованными некоторыми разновидностями оператора повторения.

Абстракция рассматривается как главный умственный инструмент, нужный для применения математической индукции – то есть для формирования концепций, в терминах которых действие шага индукции может быть описано – и важный для уменьшения применения перечисления: в случае оператора выбора он должен предоставить концепции, в терминах которых действие может быть описано независимо от выбранного пути.

Памятуя о вышесказанном, я приступлю к следующей задаче. Заданы 32 позиции, расположенные по кругу; нужно составить программу, находящую все способы (если они есть), которыми эти позиции могут быть заполнены нулями и единицами (по одной цифре в каждой позиции) таким образом, что 32 пятерки, составленные из смежных позиций, представляют 32 различных цепочки из пяти двоичных цифр. Заполнения, которые отличаются друг от друга только поворотом, рассматриваются как эквивалентные, все решения должны начинаться с пяти нулей. (Последнее условие уменьшает количество «независимых решений» в 32 раза).

Лайтманс показал, что эта циклическая задача эквивалентна следующей линейной задаче. Задан линейный массив на 36 позиций; нужно составить программу, находящую все

способы (если они есть), которыми эти позиции могут быть заполнены нулями и единицами (по одной цифре в каждой позиции) таким образом, что 32 пятерки, составленные из смежных позиций, представляют 32 различных шаблона пяти двоичных цифр. Мы можем ограничиться последовательностями, начинающимися с пяти нулей, при этом 32 начальные цифры решения линейной задачи представляют решение циклической, и наоборот. [Доказательство Лайтманса выглядит следующим образом. Каждое решение линейной задачи начинается с 000001…, потому что последовательность 00000 может встретиться только единожды; кроме того, последовательность 10000 также может встретиться только единожды. Поскольку за этими последовательностями могут идти только 00000 или 00001 (уже имеющиеся в первых двух пятерках), то последовательность 10000 не может встретиться в середине линейной последовательности и поэтому может появиться только в конце. В результате каждое решение линейной задачи заканчивается четырьмя нулями, и таким образом кольцо замыкается!]. Мы должны решить линейную задачу, наложив дополнительное условие, что решения (если их несколько) должны выдаваться в алфавитном порядке.

Самое грубое описание программы – это единственная инструкция.

Версия 0:

```
«выполни всю работу».
```

Это описание совершенно общее, но также и совершенно бесполезное, поскольку не отражает ни нашего понимания задачи, ни структуру алгоритма. Чтобы двигаться дальше, мы должны глубже проанализировать задачу.

Само собой разумеется, что для заданной последовательности из 36 двоичных цифр может быть вычислена булевская функция, устанавливающая, является ли эта последовательность решением, и что мы можем написать алгоритм ее вычисления. В принципе мы могли бы написать программу, генерирующую все 36-цифровые последовательности с пятью ведущими нулями в алфавитном порядке и подвергающую все эти последовательности такой проверке, отбирая таким образом те, которые выдержали тест. Этот метод дает весьма нереальную программу, и мы не должны ему следовать; заметим только, что генерация пробных последовательностей в алфавитном порядке обеспечит, что решения, если таковые будут найдены, будут упорядочены по алфавиту.

Примечание. Наша окончательная программа может рассматриваться как производная от программы, набросок которой приведен выше, путем добавления нескольких мелких, но важных деталей. Сейчас я не склонен подчеркивать эту преемственность, так как она кажется тесно связанной с этой специфической задачей.

В нашем следующем приближении мы снова будем генерировать наши решения путем (генерации и) сканирования больших наборов последовательностей, из которых будут выбираться все решения по подходящему критерию.

Определим «длину последовательности» как количество пятерок, которое она содержит (то есть длина = число цифр – 4). Будем называть последовательность «приемлемой», если никакие две различные пятерки в ней не представляют одинаковых цепочек цифр. С такими определениями решения – это подмножество множества приемлемых последовательностей, а именно те, у которых длина = 32.

Мы не знаем, существуют ли решения вообще, но зато знаем, что множество приемлемых последовательностей не пусто (например, «00000»); у нас нет готового критерия для распознавания «последнего решения », когда мы его достигнем в нашем наборе приемлемых последовательностей; впрочем, мы можем пометить «виртуальное последнее

решение» (а именно «00001»): когда оно достигнуто, мы знаем, что все приемлемые последовательности с пятью ведущими нулями просмотрены и что больше решений найдено не будет. Подытожим, что мы знаем о множестве приемлемых последовательностей:

1.Оно непустое и конечное.

2.Мы знаем первый член («00000»).

3.Мы знаем виртуальный последний член («00001»)

4.Мы можем преобразовать приемлемую последовательность в следующую приемлемую последовательность

5.Решения – это все приемлемые последовательности (кроме виртуальной), удовлетворяющие условию «длина последовательности равна 32».

Переход от рассмотрения только набора решений к рассмотрению набора приемлемых последовательностей явился следующим шагом в нашем анализе, который является первым улучшением: программой, в которой инструкции оперируют объектом (все еще достаточно абстрактным), называемым «последовательностью».

Версия 1:

```
Установить последовательность на первую приемлемую;
repeat
  if длина последовательности = 32 do
    begin
      принять последовательность как решение;
      печатать решение
    end;
  преобразовать последовательность в следующую приемлемую
until последовательность – это (первый или) виртуальный последний член
```

Для пояснения этой программы уместно было бы сделать пару замечаний.

Замечание 1. Последняя проверка должна отличать последний член от всех, кроме первого, так как первый не проверяется. По существу, не должно быть возражений, если первый член будет равен последнему виртуальному (например: пустая последовательность).

Замечание 2. Оператор «принять последовательность как решение» может изрядно озадачить читателя. Он вполне может оказаться пустым. Он включен в тест затем, чтобы подчеркнуть, что за «следующей точкой с запятой» последовательность рассматривается как представляющая решение. (Можно представить его себе как вытаскивание первых 32 цифр).

Критерий «приемлемости» имеет важное свойство:

6.никакое расширение неприемлемой последовательности не является приемлемым

и это его важное свойство будет использовано при уточнении оператора «преобразовать последовательность в следующую приемлемую». Прямое следствие свойства

6: проверка приемлемости должна применяться только лишь к так называемым «потенциально приемлемым последовательностям», которые можно определить как приемлемую последовательность, к которой добавлена одна цифра. Это приводит к следующему улучшению программы:

Преобразовать последовательность в следующую приемлемую:

```
преобразовать приемлемую-последовательность в следующую потенциально-
приемлемую-последовательность ;
    while потенциально-приемлемая-последовательность не является приемлемой do
    преобразовать потенциально-приемлемую-последовательность в следующую
потенциально-приемлемую-последовательность ;
    принять последовательность как приемлемую
```

Когда мы рассматриваем «преобразовать последовательность в следующую приемлемую» как доступный примитив, значение «последовательности» всегда приемлемо; только лишь при рассмотрении его внутреннего строения – в данном случае возле точки с запятой в предыдущем фрагменте – текущее значение «последовательности» - это потенциально-приемлемая-последовательность.

Ввиду требования алфавитной упорядоченности «преобразовать потенциально-приемлемую-последовательность в следующую потенциально-приемлемую-последовательность» образует новую последовательность, которая равна старой, дополненной нулем, тогда как «преобразовать потенциально-приемлемую-последовательность в следующую потенциально-приемлемую-последовательность» образует новую последовательность, равную старой, за исключением последнего нуля, дополненной единицей. Итак, очередное усовершенствование программы выглядит так:

преобразовать приемлемую-последовательность в следующую потенциально-приемлемую-последовательность:

```
    while последовательность заканчивается единицей do
      удалить последнюю цифру из последовательности;
      заменить последний нуль единицей
```

В приведенных выше пошаговых усовершенствованиях программы мы сосредоточили внимание на действиях программы с последовательностью. Теперь пора более явно представить решения, как в действительности должен быть представлен абстрактный объект под названием «последовательность». Введем integerk и положим k = длина последовательности. Затем введем arrayd[-3:33] для представления цифр, при этом d[-3] d[-2] ……. d[k] представляют последовательность. (1)

Замечание 1: с d[-3] по d[0] должны быть равны нулю.

Замечание 2: Максимальная длина приемлемой-последовательности = 32; так как алгоритм оперирует с потенциально-приемлемыми-последовательностями и потенциально-приемлемая-последовательность – это приемлемая-последовательность, дополненная одной цифрой, то максимальная длина последовательности – 33.

Представленные выше соглашения служат базой для более детального определения свойства «приемлемости». Мы должны характеризовать цифровые цепочки как представленные пятерками, содержащимися в текущем значении последовательности. Я предлагаю характеризовать такие цепочки цифр целым числом, которое получается при интерпретации такой пятерки цифр как двоичного числа. Другими словами, мы определяем функцию H(i) для 1 <= i <= k:

H(i) = d[i-4]*16 + d[i-3]*8 + d[i-2]*4 + d[i-1]*2 + d[i] (2)

Свойство «приемлемости» выглядит так:

для 1<=i,j<=k, i<>j означает H(i)<>H(j) (3)

В любой момент функция H(i) определена на текущем значении последовательности для1<=i<=k. Вместо того, чтобы вычислять значения этой функции каждый раз, когда они нам понадобятся (например, в проверке приемлемости), мы можем протабулировать их значения в h[1:33].

По нашему соглашению для всех значений последовательности

h[i] = H(i) для 1<=i<=k. (4)

Это соглашение подразумевает, что изменение значения последовательности в общем случае включает изменение массива h, чтобы следовать отношению (4).

Проверка приемлемости теперь – по аналогии с (3) – такова:

для 1<=i,j<=k из i<>j следует h[i]<>h[j].

Теперь мы воспользуемся фактом, что тесту на приемлемость подвергаются только потенциально-приемлемые последовательности, т.е. приемлемые последовательности, дополненные одной цифрой. Для потенциально-приемлемой последовательности мы можем заключить, что она является приемлемой тогда и только тогда, когда

h[i]<>h[k] для 1<=i<k (5)

т.е. когда последняя пятерка представляет цепочку, отличную от всех других. Это снова предполагает сканирование, но мы можем повторить трюк и протабулировать, появлялась ли уже некоторая цепочка цифр. Наиболее элегантный способ – это ввести booleanin[0:31], в котором для 0<=m<=31 in[m] означает:

для приемлемой-последовательности: m присутствует среди h[1]…h[k] для потенциально-приемлемой последовательности: m присутствует среди h[1]…h[k-1] (6)

Примечание. Если последовательность приемлема, каждая цепочка может присутствовать только один раз, поэтому для того, чтобы отметить ее появление, достаточно булевой переменной. Для потенциально-приемлемой последовательности h[k] может быть равным h[i] для i<k; таким образом, соглашение (6) различно для приемлемой и для потенциально-приемлемой последовательностей.

Теперь дадим окончательную версию программы. То, что было именами примитивов, теперь становится метками (либо операторов, либо выражений)

```
begin
   integer k;
   integer array d[-3:33];
   integer array h[1:33];
   boolean array in[0:31]; // установить последовательность на первую
приемлемую
begin
  d[-3] := d[-2] := d[-1] := d[0] :=d [1] := 0;
  k := 1;
  h[1]: = 0;
  in[0] := true;
  begin
    integer m;
    m := 1;
```

```
          repeat
             in[m] := false;
             m := m + 1
          until m = 32
      end
end;
   repeat
      if (k = 32) then // длина последовательности равна 32
        begin // принять последовательность как решение:
        new line carriage return; //
        print solution:
          begin
            integer m;
            m := 0;
            repeat
               print(d[m - 3]);
               m := m + 1
            until m = 32
          end
     end;
// преобразовать последовательность в следующую приемлемую:
begin // преобразовать последовательность в следующую потенциально-приемлемую:
   begin
     k := k + 1;
     d[k] := 0;
     h[k]:= 2 * h[k - 1] - 32 * d[k - 4]
   end;
   while (in[h[k]]) do
     // потенциально-приемлемая последовательность не является приемлемой:
     // преобразовать потенциально-приемлемую последовательность в следующую
приемлемую:
begin
    while (d[k] = 1) do
     // последовательность заканчивается единицей // удалить последнюю цифру из
последовательности:
      begin
        k := k - 1;
        in[h[k]] := false
      end;
// заменить последний нуль единицей:
    begin
      d[k] : = 1;
      h[k] := h[k] + 1
    end
end;
```

```
// принять последовательность как приемлемую:
  in[h[k]] := true
end
until (k = 1) // последовательность - виртуальный последовательный член end
```

Пояснения:

1.«принять последовательность как решение» могло бы быть пустым оператором, но вместо этого задает переход на новую строку;

2.«принять последовательность как приемлемую» получает содержание в соответствии с соглашением (6), где делается различие между приемлемой и потенциально-приемлемой последовательностями.

Заключение.

Мы продемонстрировали последовательность версий программы от начальной постановки задачи до окончательной программы. В окончательном варианте программы вручную было произведено слияние последовательных версий, при этом более абстрактные версии постепенно выродились в комментарии.

Для больших программ сам по себе подобный процесс слияния становится серьезной задачей обработки данных, и я ожидаю появления интерактивных технологий составления программ, в которых этот процесс будет упрощен посредством привлечения помощи компьютера.

Более того: в настоящий момент более абстрактные версии исполняют роль только пояснительных комментариев, включенных для облегчения понимания программы человеком. Источник этого – наше желание иметь программы, сформулированные на постоянном семантическом уровне, в данном случае – на уровне языка программирования. На сегодняшний день более абстрактные версии совершенно бесполезны для выполнения их на машине. Я ожидаю, что в будущем более абстрактные версии станут неотъемлемой частью программы.

Наконец, мы рассмотрели только одну линию программ от постановки задачи до рабочей программы, написанной на желаемом семантическом уровне. Я ожидаю, что в будущем эта единственная линия будет расширена до более или менее структурированного в виде дерева класса программ, в котором найдется также место и альтернативам, таким образом связывая вместе составление программ и их модификацию.

Edsger W. Dijkstra

Department of Mathematics Technological University

The Netherlands

(Эта статья была написана в связи с докладом, прочитанным в университете

Гренобля в декабре 1967 г.).