Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

МЕТОДЫ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Сборник научных работ под редакцией В. И. Шелехова

Новосибирск 2003

Сборник включает работы по предикатному программированию. Представлен обзор концепции предикатного программирования и элементов языка предикатного программирования Р. Технология предикатного программирования демонстрируется на примерах конкретных задач. Описывается алгоритм склеивания переменных, используемый в системе трансформации предикатной программы в эффективную императивную программу.

Сборник представляет интерес для научных сотрудников, программистов и студентов, специализирующихся в области информатики.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

THE PREDICATE PROGRAMMING METHODS

Edited by V. I. Shelekhov

Novosibirsk 2003

This volume includes papers on predicate programming. The predicate programming concept and some part of the predicate programming language P is introduced. The predicate programming technology is demonstrated for some concrete problems. An algorithm of variable replacement used in transformation of a predicate program to effective imperative one is described.

The volume may be of interest for researchers in computer science, programmers and students.

ПРЕДИСЛОВИЕ

Настоящий сборник составлен из работ, относящихся к проблематике предикатного программирования. Большая часть работ иллюстрирует методы построения предикатных программ для конкретных задач, а также технику трансформации предикатной программы в эффективную императивную программу.

Предикатное программирование определяет программирование исключительно в рамках логически правильно построенных языковых конструкций. В этом плане предикатное программирование сходно с функциональным. Однако по стилю предикатное программирование принципиально отличается от функционального, а по возможностям изображения алгоритмов существенно шире его. Концепция и язык предикатного программирования изложены в препринтах [1].

Большая часть работ сборника демонстрирует технологию предикатного программирования на примере конкретных задач, а также технику трансформации предикатной программы в эффективную императивную программу. Для каждой задачи представлены два варианта предикатной программы. Второй вариант получается как развитие первого с более сложной спецификацией и программой и более эффективной императивной программой.

Спецификации предикатов, входящих в предикатную программу для рассматриваемых задач, даются в виде набора математических условий. Эти спецификации нетрудно оформить на любом формальном языке. Разработка методов автоматической верификации предикатной программы, т.е. автоматического доказательства того, что определение любого предиката соответствует его спецификации, является дальнейшей задачей в развитии предикатного программирования.

Начальная работа В.И. Шелехова “Предикатное программирование: основы, язык, технология” в данном сборнике является обзорной. В ней определяется ключевое понятие логически правильно построенных языковых конструкций, используемых для написания любой предикатной программы. Описываются элементы языка предикатного программирования Р, доста-

[1] Шелехов В.И. Введение в предикатное программирование. — Новосибирск, 2002. — 82с. — (Препр. / ИСИ СО РАН; № 100).

Шелехов В.И. Язык предикатного программирования Р. — Новосибирск, 2002. — 40с. — (Препр. / ИСИ СО РАН; № 101`)

точные для понимания предикатных программ в следующих работах данного сборника.

Работа В.И. Шелехова, Н.С. Карнаухова “Демонстрация технологии предикатного программирования на задаче сортировки простыми вставками” показывает типовые элементы технологии предикатного программирования.

Работа В.И. Шелехова, А.А Алгазина “Опыт предикатного программирования задачи нахождения кратчайшего пути между двумя городами” демонстрирует операции с разными структурами данных: одномерными и двухмерными массивами, последовательностями и множествами при построении и трансформации предикатной программы. Используется типовая техника работы с гиперфункциями. Показаны особенности функционального стиля записи языковых конструкций, результатами которых являются тройки или четверки значений.

Работа В.И. Шелехова “Трансформация предикатной программы сортировки слиянием в эффективную параллельную программу” описывает технику предикатного программирования, используемую для разработки параллельного алгоритма сортировки. Применяется нетривиальный способ склеивания массивов, являющихся параметрами предиката. Обычно рекурсивное определение предиката конструируется через рекурсивный вызов на ветви условного оператора или оператора расщепления. Здесь же рекурсивное определение реализуется через вызов, являющийся составной частью параллельного оператора. Новая форма рекурсии определяет также новую форму параллелизма, требующую специфической реализации в императивной программе, гарантирующей отсутствие конфликтов по разделяемым переменным.

Работа Э.Ю. Петрова “Склеивание переменных в предикатной программе” описывает алгоритм автоматического склеивания переменных в рамках одного определения предиката при условии, что склеивания входных и результирующих параметров определения известны. Склеивание переменных является первым этапом в системе трансформаций предикатной программы. Алгоритм склеивания будет использован в системе предикатного программирования, реализующей трансформацию предикатной программы в императивное расширение языка Р с последующей конвертацией на один из императивных языков.

В. И. Шелехов

В. И. Шелехов

ПРЕДИКАТНОЕ ПРОГРАММИРОВАНИЕ: ОСНОВЫ, ЯЗЫК, ТЕХНОЛОГИЯ

В данной работе дается обзор концепции, языка и технологии предикатного программирования, изложенных в препринтах [1, 2].

Любой язык программирования определяет набор языковых конструкций и правила их исполнения. Будем рассматривать произвольную исполняемую языковую конструкцию как **композицию** вида:

<имя композиции>(<спецификация1>,…,<спецификацияN>) ,

выражающую спецификацию конструкции через спецификации ее подконструкций. Подобное рассмотрение характерно при определении денотационной семантики программ [3]. Для типового набора конструкций:

begin <оператор1>; <оператор2>; …; <операторN> **end**
if <условие> **then** <оператор1> **else** <оператор2> **end**
while <условие> **do** <оператор> **end**

дадим соответственно представления их спецификаций в виде следующего набора композиций:

Block**(**<спецификация1>, <спецификация2>,…, <спецификацияN>)
If**(**<условие>, <спецификация1>, <спецификация2>)
While**(**<условие>, <спецификация>)

Язык программирования определяет логическое исчисление, в рамках которого реально работает программист. Конструирование программы (или любой ее части) реализуется исходя из некоторой априорной спецификации программы (или части). Типичной задачей, реализуемой при программировании, отладке и модификации программы, является проверка для некоторой конструкции правильности тождества вида:

<спецификация> ≡ <имя композиции>(<спецификация1>,…,<спецификацияN>)

Безусловно, программирование является логической деятельностью. Однако полноценная математическая работа с императивными программами невозможна; проведение математических доказательств для программ весьма проблематично. На практике применяется отладка и тестирование, что, как известно, не гарантирует отсутствия ошибок в программах.

Композиция является **логической**, если для произвольных подконструкций она представима в виде формулы исчисления предикатов второго порядка. Композиции Block и If являются логическими. Соответствующими формулами для них являются:

<спецификация1> & <спецификация2> & … & <спецификацияN>

(<условие> ⇒ <спецификация1>) & (¬<условие> ⇒ <спецификация2>)

Композиция While не является логической.

Конструкция считается **логически правильной**, если соответствующая ей композиция является логической. Язык программирования называется **предикатным** (а программа — **предикатной**), если все его исполняемые конструкции являются логически правильными.

В соответствии с определением, предикатная программа является правильным логическим (математическим) объектом. Следовательно, с предикатной программой можно работать в традиционном математическом стиле — в частности, проводить математическое доказательство ее правильности, в том числе посредством автоматизированной верификации. Это важнейшее свойство является ключевым в решении проблемы надежности программ.

Множество всевозможных логически правильных конструкций, допустимых в предикатных программах, определяется множеством вычислимых логических композиций, порождаемым **исчислением вычислимых предикатов**. Базисом исчисления являются композиции следующего вида: суперпозиция (композиция Block), альтерация (композиция If), параллельная композиция, применение предиката, порождение предиката, конструктор массива (цикл **forAll** FORTRAN 90) и расщепление. Возможно, этот базис не является полным.

Перечисленные базисные вычислимые логические композиции являются основой для построения языка предикатного программирования P (**P**redicate programming language). По набору языковых конструкций язык P значительно шире известных языков функционального программирования.

Предикатная программа есть набор рекурсивных **определений предикатов**, среди которых могут находиться описания типов и глобальных переменных. Определение предиката имеет следующий вид:

<имя предиката>(<описания исходных параметров>**:** <описания результатов>)
{ <оператор> }

Определение предиката состоит из **заголовка**, декларирующего имя предиката и параметры, и **тела предиката**. Исполнение определения предиката заключается в исполнении <оператора>, вычисляющего значения результирующих параметров, описанных после разделителя "**:**". Описание исходного или результирующего параметра предиката имеет следующий вид:

<тип><пробел><имя параметра>

Операторами являются: предикат равенства, вызов предиката, блок, параллельный оператор, условный оператор и другие. Предикатом равенства является конструкция: <переменная> = <выражение>. Вычислимая логическая композиция: <оператор1> & <оператор2> является блоком {<оператор1>; <оператор2>}, если результирующие переменные <оператора1> используются в <операторе2>. Если же результаты этих операторов в них самих не используются, то композиция является параллельным оператором и записывается в виде: <оператор1>|| <оператор2>. Исполнение каждого из двух операторов реализуется параллельно.

Среди операторов блока могут находиться описания локальных переменных:

<тип> <пробел> <список имен переменных>

Возможно также описание переменной с инициализацией:

<тип> <пробел> <имя переменной> = <выражение>

Оператор может вырабатывать значение и входить в состав выражений.

Пример 1. Определение предиката sign для знака вещественного числа x:

```
sign(real x: int s) {
    if x>0 then s = 1 elsif x = 0 then s = 0 else s = -1 end
}
```

Приведенная форма тела предиката sign является **предикатной** (или операторной). Возможна **функциональная** форма определения:

```
sign(real x: int) {
    if x>0 then 1 elsif x = 0 then 0 else -1 end
}
```

Вызов предиката sign(r: a) эквивалентен предикату равенства a = sign(r), где sign(r) является вызовом функции.

В императивном программировании регулярно возникают ситуации, когда алгоритм невозможно адекватно выразить в существующих формах

языка программирования, хотя содержательно алгоритм формулируется в естественной и правильной логике. В подобных ситуациях приходится либо оформлять часть программы в виде процедуры, используя оператор **return** в теле процедуры, либо искусственно вводить логические переменные для реализации управления, либо использовать другие трюки.

Невыразимость логически правильных фрагментов алгоритма при построении предикатной программы оказалась еще более острой проблемой. Анализ структуры таких алгоритмических фрагментов позволил обнаружить логическую вычислимую композицию нового вида: **оператор расщепления** на базе гиперфункции.

Предикат вида S(**x:** y), где x и y — списки переменных, определяет **функцию**. Допустим, наряду с предикатом S(**x:** y) имеется предикат Q(**x:** z). Списки переменных z и y не содержат общих переменных. Области определения функций S и Q, соответственно S_x и Q_x, не пересекаются. **Гиперфункция** H(x: y | z) с двумя **ветвями** есть предикат, определяемый формулой:

$$(x \in S_x \Rightarrow S(x: y)) \ \& \ (x \in Q_x \Rightarrow Q(x: z))$$

Гиперфункция H(x: y | z) определена на $Sx \cup Qx$ и совпадает с функцией S на Sx и с функцией Q на Qx. Исполнение гиперфункции завершается на одной из двух ветвей. При этом вычисляются значения результирующих переменных завершившейся ветви; переменные другой ветви не вычисляются. Для указания того, какой ветвью завершилось исполнение тела предиката-гиперфункции, в теле используется **указатель завершения**: #1 или #2.

Гиперфункция совмещает в себе преобразователь и распознаватель. Ветвь гиперфункции может быть **пустой**, т.е. не содержать результирующих переменных. По пустой ветви гиперфункция является только распознавателем.

Оператор расщепления состоит из заголовка и альтернатив расщепления:

```
split    <вызов предиката-гиперфункции>
do   <оператор – первая альтернатива расщепления>
do   <оператор – вторая альтернатива расщепления>
end
```

Если вызов предиката-гиперфункции завершается первой ветвью, далее исполняется первая альтернатива расщепления, иначе — вторая.

Определение гиперфункции и оператора расщепления обобщается на произвольное число ветвей.

Пример 2. Допустим, для последовательности s целых чисел требуется извлечь второй элемент и присвоить переменной e. Данная операция представляется в виде гиперфункции elemTwo(s: e |). Вторая (пустая) ветвь гиперфункции реализуется в случае, если s состоит менее чем из двух элементов. Для работы с последовательностями используется стандартный предикат Comp(s: | d, r). Первая ветвь Comp реализуется для пустой последовательности s, а во второй ветви: d — начальный элемент, r — последовательность, полученная из s удалением начального элемента. Определение гиперфункции elemTwo может быть реализовано следующим образом:

```
elemTwo(seq int s: int e | ) {
    int e1; seq int s1, s2;
    split Comp(s: | e1, s1)
    do   #2
    do   split Comp(s1: | e, s2)
         do    #2
         do    #1
         end
    end
}
```

Альтернатива оператора расщепления является пустой, если в ней нет других действий, кроме возможного указателя завершения. Действует следующее правило: в пустой альтернативе указатель завершения переносится в конец соответствующей ветви вызова гиперфункции. Определение гиперфункции elemTwo переписывается следующим образом:

```
elemTwo(seq int s: int e | ) {
    split Comp(s: #2 | int  e1, seq int s1)
    do
    do    Comp(s1: #2 | e, seq int s2 #1)
    end
}
```

В данном определении elemTwo кроме переноса указателей завершения реализован перенос описаний локальных переменных e1, s1 и s2 в позиции их **определения**, т.е. места в программе, где им присваиваются значения.

Оператор расщепления является вырожденным, если содержит только одну непустую альтернативу. В этом случае оператор расщепления заменяется блоком или параллельным оператором. Последнее определение elemTwo должно быть заменено следующим:

```
elemTwo(seq int s: int e | ) {
    Comp(s: #2 | _, seq int s1);
    Comp(s1: #2 | e, _ #1)
}
```

Дополнительно в этом определении проведена замена результирующих вхождений локалов e1 и s2 стандартным именем "_", означающем, что соответствующий результат игнорируется при исполнении.

Система типов языка P включает примитивные типы (**nat, int real, bool, char, complex**), подмножество типа (в частности, диапазон целых чисел), предикатный тип и структурные типы (массив, кортеж, объединение, последовательность, множество). Тип может быть **параметризован**, т.е. зависеть от значений переменной. Например, тип 1..n+1, определяющий диапазон целых чисел от 1 до n+1, параметризован переменной n. Если нам требуется ввести имя Diapason для данного типа, используется следующее описание:

```
type Diapason(int n) = 1..n+1
```

Структурный тип определяется **конструктором** объекта структурного типа по значениям компонентов объекта и **оператором разложения** объекта структурного типа для доступа к его компонентам.

Конструктор <структурный тип>(<список выражений>) определяет объект структурного типа по значениям компонентов, перечисленных <списком выражений>. Например, конструктор **seq int**(a, b, c) определяет последовательность целых чисел, где любой из аргументов a, b, c может быть последовательностью или целым. Допускается иная форма записи: a + b + c. Конструктор **seq int**( ) определяет пустую последовательность. Аналогичные правила действуют для конструктора объекта типа множества.

Предикат Comp(s: | d, r), приведенный в примере 2, является оператором разложения последовательности s. Для него используется следующая запись: s -> (| d, r). Дадим итоговую версию программы примера 2:

```
elemTwo(seq int s: int e | ) {
    s -> (#2 | _, seq int s1);
    s1 -> (#2 | e, _ #1)
}
```

Массивом является объект типа **array** I **of** T, где T — тип элементов, а I — конечный тип индексов. Конструктором массива A является оператор:

forAll k **in** I **do** A[k] = <выражение> **end**

Вычисление тела конструктора для разных индексов k реализуется независимо, возможно, параллельно. Телом конструктора может быть также вызов предиката вида S(...**:** A[k]). Конструктор массива может быть записан в другой форме:

A = (<выражение> **where** k **in** I)

Конструкция k **in** I называется итератором. Если тип индексов I является списком диапазонов, то итератор записывается в виде: k = I.

Конструктор должен определять массив для всех его элементов. Нельзя в качестве типа индексов использовать более широкий тип — тип индексов должен быть точно задан. По этой причине, тип индексов (и тип массива) обычно параметризован, т.е. зависит от значений переменных.

Следующие операции являются производными формами конструктора массива: поэлементное задание массива в виде (<список выражений>), объединение A + B массивов A и B с непересекающимися типами индексов и вырезка A[I] массива A по подмножеству I типа индексов. Объединение двух массивов (B_1 **where** k **in** I_1) + (B_2 **where** k **in** I_2) может быть записано как конструктор следующего вида: (B_1 **where** k **in** I_1, B_2 **where** k **in** I_2). Данная форма конструктора обобщается для объединения трех и большего числа массивов.

Пример 3. Рассмотрим программу перестановки 5 и 6 элементов в массиве a из 10 элементов:

```
type  Ar10 = array 1..10 of  real;
Permutation(Ar10 a: Ar10 b) {
      b = (a[k] where k = {1..4, 7..10},
           a[6] where k =5,
           a[5] where k =6)
}
```

Часть конструктора массива “a[6] **where** k=5”, определяющая один элемент b[5], будем записывать в более компактной форме: “5: a[6]”. Далее, если в конструкторе имеется часть вида “a[k] **where** k **in** I” для некоторого подтипа I, то эту часть можно удалить и записать конструктор в виде “a(..., ...)”. С учетом этих правил тело предиката Permutation переписывается следующим образом:

b = a (5: a[6], 6: a[5])

Технология предикатного программирования предполагает написание программы на языке P, применение к ней набора **трансформаций** с полу-

чением эффективной программы на императивном расширении языка Р и конвертацию на любой из императивных языков: С, С++ и др. Эффективность конечной программы в значительной степени зависит от выбора эффективного алгоритма на языке Р. Универсальным методом предикатного программирования является **обобщение** исходной задачи для получения программы с хвостовой формой рекурсии.

Базовыми трансформациями являются:

- **склеивание переменных**, реализующее замену нескольких переменных одной;
- замена **хвостовой** рекурсии (tail-рекурсии в языке Лисп) циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов посредством низкоуровневых структур с использованием массивов и указателей.

Результатом применения трансформаций является императивная программа, не уступающая по эффективности написанной вручную.

Трансформации применяются к программе на императивном расширении языка Р, включающем: операторы присваивания, циклы вида **loop**, **while** и **for**, операторы перехода и групповые операторы присваивания следующего вида:

|<список переменных> | := | <список выражений> |

При исполнении этой конструкции вычисленные значения списка выражений одновременно присваиваются соответствующим переменным левой части.

Если тип исходного параметра x совместим с типом результирующего параметра y в определении предиката S(...x...**:** ...y...){...} и в реализации императивной программы предполагается склеивание переменных x и y, то всюду в определении вместо y используется имя x′. Кроме того, в определении вида S(...x*...**:**...){...} исходный параметр x декларируется вместе с результирующим параметром x′ того же типа.

Эффективная реализация оператора разложения s -> (| T d, **seq** T r) предполагает склеивание переменных s и r. Стандартным представлением объекта s типа последовательность является вырезка вида A[n**:** m] для некоторого массива A элементов типа T. Оператор расщепления вида

split s -> (| T d, **seq** T r) **do** B **do** C **end**

будет реализован для стандартного представления в виде следующего условного оператора:

if n>m **then** B **else** d := A[n]; n := n + 1; C **end**

Рассмотрим трансформацию предикатной программы elemTwo. Локальная переменная s1 склеивается с параметром s. Допустим, последовательность s представляется массивом A[1..m]. Тогда локальной переменной s1 соответствует вырезка A[2..m]. Заменяя операторы расщепления условными операторами, получим следующую императивную программу:

```
type  Ar(N) = array 1..N of  int;
elemTwo(Ar(m) A: int e | ) {
     if 1>m then #2
     elsif 2>m then #2
     else  e := A[2] #1
     end
}
```

Устранение избыточной ветви **if** 1>m **then** #2 реализуется в процессе классической оптимизации при трансляции.

Рассмотрим трансформацию предикатной программы Permutation(Ar10 a: Ar10 b) в предположении, что массивы a и b склеиваются, т.е. в качестве b используется переменная a′. Заголовок предиката принимает вид: Permutation(Ar10 a*:). Поскольку a и a′ склеиваются, конструктор a′ = a (5: a[6], 6: a[5]) преобразуется в следующий параллельный оператор: a′[5] := a[6] || a′[6] := a[5]. При замене a′ на a параллельный оператор превращается в групповой оператор присваивания:

$$| a[5], a[6] | := | a[6], a[5] |.$$

Раскрытие последнего оператора, т.е. его замена на обычные операторы присваивания, возможна лишь при введении дополнительной переменной. В итоге получим следующую программу:

```
type  Ar10 = array 1..10 of  real;
Permutation(Ar10 a*: ) {
       int t := a[5]; a[5] := a[6]; a[6] := t;
}
```

СПИСОК ЛИТЕРАТУРЫ

1. **Шелехов В.И.** Введение в предикатное программирование. — Новосибирск, 2002. — 82с. — (Препр. / ИСИ СО РАН; № 100).
2. **Шелехов В.И.** Язык предикатного программирования Р. — Новосибирск, 2002. — 40с. — (Препр. / ИСИ СО РАН; № 101).
3. **Stoy J.T.** Denotational semantics: The Scott-Strachy approach to programming theory. — MIT Press, 1977.

В.И.Шелехов, Н.С.Карнаухов

ДЕМОНСТРАЦИЯ ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ НА ЗАДАЧЕ СОРТИРОВКИ ПРОСТЫМИ ВСТАВКАМИ

В данной работе демонстрируются типовые элементы технологии предикатного программирования на примере задачи **сортировки простыми вставками** [1]. Показано построение двух вариантов предикатной программы и реализация эффективной императивной программы. Читателю предварительно следует ознакомиться с работой [2] настоящего сборника.

Допустим, имеется массив a[low..hgh] с элементами некоторого типа T, где low — нижняя, а hgh — верхняя граница массива. Полагаем, что для любой пары значений типа T определены операции сравнения <, <=, > и >=. Необходимо реализовать сортировку массива методом простых вставок [1]. В этом методе реализуется итерация по элементам. Очередной i-ый элемент перемещается в подходящее место среди ранее упорядоченных элементов a[low] <= a[low+1] <=...<= a[i-1]. Массив, полученный на i-ом шаге итерации, оказывается упорядоченным для элементов с индексами от low до i.

Тип массива a имеет следующее описание:

```
type Ar(int l, h, type T) = array l..h of T;
```

Поскольку low, high и T являются постоянными параметрами программы сортировки, они определяются следующими глобальными описаниями:

```
int low,high;
type T;
type ArT = Ar(low, high, T);
```

Последнее описание определяет обозначение ArT, в котором параметры отсутствуют, но подразумеваются. Обозначение ArT используется для сокращения записи.

Задача сортировки массива a представляется предикатом sort(ArT a: ArT a'), где a' есть упорядоченный массив, являющийся перестановкой массива a. Предполагается, что результат сортировки должен быть получен в том же массиве a, поэтому в реализации массивы a и a' должны быть склеены. Напомним, что вместо sort(ArT a: ArT a') можно писать sort(ArT a*:).

Определение предиката sort реализуется через обобщающий предикат sorti(a, i: a'), сортирующий массив a при условии, что его элементы a[low],...,a[i-1] уже отсортированы.

```
sort(ArT a*: ) // a' — перестановка a, a' — упорядочен
{       if low >= high then
                a' = a
        else
                sorti(a, low+1: a')
        end
}
```

В определении предиката sorti используется вызов предиката pop_into, вставляющий элемент a[i] внутрь отсортированной части a[low],...,a[i-1].

```
sorti(ArT a*, int i: )
        // a[low..i-1]  — упорядочен, a' — перестановка a, a' — упорядочен
{   pop_into(a, i, a[i]: ArT c);
    if i = high then
        a' = c
    else
        sorti(c, i+1: a')
    end
}
```

В определении предиката pop_into элемент e=a[j] вставляется внутрь отсортированной части a[low],...,a[j-1]. Массив a' является перестановкой a с отсортированными элементами a'[low],...,a'[j].

```
pop_into(ArT a*, int j, T e: )  // a[low..j-1], a'[low..j]  — упорядочены, e=a[j],
                                // a'[j+1..high] = a[j+1..high]
{   if j = low or a[j-1] <= e then
      a' = a
    else
      ArT c = a(j-1: e, j: a[j-1]);
      pop_into(c, j-1, e: a')
    end
}
```

Массив c строится из массива a с помощью конструктора: c совпадает с a за исключением элементов a[j] и a[j-1], которые меняются местами, так как e=a[j]. Поэтому c является перестановкой a.

Определения предикатов sort, sorti и pop_into, а также описания типов и глобальных переменных представляют полную предикатную программу

сортировки. В этой программе элемент e = a[j] последовательно обменивается с соседними элементами a[j-1], a[j-2] и т.д., пока не окажется отсортированным. Понятно, что это не самая эффективная программа, поскольку элемент e можно сразу перенести в нужное место, сдвигая на одну позицию вверх все промежуточные элементы. Построим новую версию программы, реализующую подобный перенос элемента e.

Во-первых, вместо обмена элементов a[j-1] и e=a[j] в конструкторе ArT c = a(j-1: e, j: a[j-1]) следует просто перемещать элемент a[j-1] с помощью ArT c = a(j: a[j-1]). Новый массив c не является перестановкой для a. Перестановкой, соответствующей перемещению a[j-1], является (c[low],...,c[j-2], e, c[j],...,c[high]). Во-вторых, в качестве исходного упорядочиваемого массива следует рассматривать: d = (a[low],...,a[j-1], e,a[j+1],..., a[high]), поскольку условие e=a[j] уже не выполняется. Будем использовать новый предикат shift(a, j, e: a'), который перемещает элемент e массива a в соответствующее место, так что a'[low..j] становится упорядоченным.

```
pop_into(ArT a*, int j, T e: )  // a[low..j-1], a'[low..j]  — упорядочены, e=a[j],
                                // a'[j+1..high] = a[j+1..high], j > low
{     if a[j-1] <= e then
        a' = a
      else
        shift(a, j, e: a')
      end
}
```

Предикат shift вставляет элемент e внутрь отсортированной части a[low],...,a[j-1]. Массив a' является перестановкой массива (a[low],...,a[j-1], e, a[j+1],...,a[high]). Элементы a'[low],...,a'[j] являются упорядоченными.

```
shift(ArT a*, int j, T e: ) // a[low..j-1], a'[low..j]  — упорядочены, a[j-1] > e,
                            // a'[j+1..high] = a[j+1..high], j > low
                            // a' — перестановка (a[low..j-1], e, a[j+1..high])
  {   ArT c = a(j: a[j-1]);
      if j-1 = low or a[j-2] <= e then
          a' = c(j-1: e)
      else
          shift(c, j-1, e: a')
      end
  }
```

Доказательство того, что предикатная программа является правильной, т.е. что определение каждого предиката удовлетворяет его спецификации, не вызывает принципиальных затруднений. В случае записи спецификаций на формальном языке предикатная программа в принципе может быть автоматически верифицирована.

Для предикатной программы, составленной из определений предикатов sort, sorti, pop_into и shift, построим соответствующую императивную программу.

Сначала склеим переменные a, a' и c во всех определениях и проведем очевидные упрощения. При этом конструктор ArT c = a(**j:** a[j-1]) заменяется оператором присваивания. a[j] := a[j-1]. Далее, заменим хвостовую рекурсию циклом. Получим:

```
sort(ArT a*: )
{     if  low < high then
          sorti(a, low+1: a)
      end
}
sorti(ArT a*, int i: )
{     loop
          pop_into(a, i, a[i]: a);
          if i = high then
              exit
          else
              i:= i+1
          end
end
}
pop_into(ArT a*, int j, T e: )
{     if a[j-1] > e then
        shift(a, j, e: a)
      end
}
```

```
shift(ArT a*, int j, T e: )
    loop
        a[j] := a[j-1];
        if j-1 = low or a[j-2] <= e then
          a[j-1] := e; exit
        else
          j := j-1
        end
    end
}
```

В полученной программе проведем подстановку определения предиката shift в тело pop_into. Полученное определение pop_into подставляется в тело sorti, а затем sorti — в тело sort. В результате подстановки параметры i, j и e становятся локалами.

```
sort(ArT a*: )
{     if low < high then
          int i := low+1;
          loop
              int j := i; T e := a[i];
              if a[j-1] > e then
                  loop
                    a[j] := a[j-1];
                    if j-1 = low or a[j-2] <= e then
                         a[j-1] := e; exit
                    else j := j-1
                    end
                  end
              end
          end ;
          if i = high then  exit
          else  i:= i+1
          end
      end
}
```

Заменяя объемлющий цикл **loop** на цикл **for**, получаем итоговую императивную программу:

```
sort(ArT a*: )
{     if low < high then
          for int i := low+1.. high do
               int j := i; T e := a[i];
               if a[j-1] > e then
                    loop
                         a[j] := a[j-1];
                         if j-1 = low or a[j-2] <= e then
                              a[j-1] := e; exit
                         else j := j-1
                         end
                    end
               end
          end
     end
}
```

Последнее преобразование не меняет процесса исполнения программы. Оно относится к **оформлению**, улучшающему структуру программы.

СПИСОК ЛИТЕРАТУРЫ

1. **Кнут Д.** Искусство программирования для ЭВМ. Т.3: Сортировка и поиск. Пер. с англ. — М.: Мир, 1978. — 843с.
2. **Шелехов В.И.** Предикатное программирование: основы, язык, технология // Наст. сб. — С. 7–15.

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.

В.И.Шелехов

ТРАНСФОРМАЦИЯ ПРЕДИКАТНОЙ ПРОГРАММЫ СОРТИРОВКИ СЛИЯНИЕМ В ЭФФЕКТИВНУЮ ПАРАЛЛЕЛЬНУЮ ПРОГРАММУ

В данной работе описывается техника предикатного программирования, используемая для разработки параллельного алгоритма сортировки. Применяется нетривиальный способ склеивания массивов, являющихся параметрами предиката. Обычно рекурсивное определение предиката конструируется через рекурсивный вызов на ветви условного оператора или оператора расщепления. Здесь же рекурсивное определение реализуется через вызов, являющийся составной частью параллельного оператора. Новая форма рекурсии определяет также новую форму параллелизма, требующую введения в язык P потоков (**threads**) и боксов (**boxes**), в которых эти потоки локализуются. Читателю предварительно следует ознакомиться с работой [2] настоящего сборника.

Допустим, имеется вещественный массив элементов a[m..n], где $m \le n$. Необходимо реализовать сортировку массива методом слияния [1]. В этом алгоритме массив a делится на две части. В каждой части сортировка проводится независимо. Два отсортированных массива объединяются (сливаются) в один отсортированный массив просмотром по обоим массивам и сравнением текущих элементов. Для каждой из двух частей массива сортировка реализуется по той же схеме: массив делится на два и т.д.

Тип массива a имеет следующее описание:

```
type Ar(int x, y) = array x..y of real;
```

Для глобальных переменных m и n будем использовать описание:

```
int m, n; // m <= n
```

Спецификация программы сортировки слиянием представляется предикатом sort(Ar(n,m) a*:). Предполагается, что результат сортировки должен быть в том же массиве a, поэтому в реализации массивы a и a' должны быть склеены.

Непосредственная реализация сортировки слиянием может быть дана следующим определением:

```
sort(Ar(m,n) a* : ) // m <= n, a' — упорядочена, a' есть перестановка a
{     nat d = (n-m+2)/2;
      if d = 1 then a' = a
      else   sort(a[m..m+d-1] : Ar(m, m+d-1) b);
             sort(a[m+d..n]: Ar(m+d..n) c);
             merge(b, c: a')
      end
}
```

В теле определения предикат merge(b, c: a') реализует отсортированное объединение двух отсортированных массивов b и c.

Определение sort является рекурсивным, причем рекурсия является бинарной. Чтобы преобразовать бинарную рекурсию к хвостовому виду, необходимо сначала преобразовать ее к унарной рекурсии. Обычно применяемый метод — это обобщение исходной задачи для сортировки набора массивов, являющихся подмассивами одного массива. Для этой цели будем рассматривать массив a с разбиением на куски (подмассивы) из d элементов: a[m+d*i..m+d*i+d-1], где i = 0,1,2,...

Новый алгоритм сортировки определяется следующим образом. Сначала мы сортируем массив кусками из двух элементов, затем — из четырех, далее — из восьми. Последовательно удваивая куски, мы в конце концов накроем массив a одним отсортированным куском, и таким образом решим исходную задачу.

Определение исходного предиката sort сводится к более общей задаче sortMulti:

```
sort(Ar(m,n) a* : ) // m <= n, a' — упорядочена, a' есть перестановка a
{     sortMulti(1, a: a')     }
```

Массив a в предикате sortMulti является отсортированным кусками длиной d: каждая из вырезок a[m+d*i.. m+d*(i+1)], (i=0,1,2,...) является упорядоченной. Это условие будем обозначать предикатом sortP(a, d, m, n). Очевидно, что куски длиной d=1 всегда являются отсортированными.

```
sortMulti(nat d, Ar(m,n) a* : )
        // m <= n, sortP(a, d, m, n), a' — упорядочена, a' есть перестановка a
{     if n-m+1 <= d then a
      else sortMulti(d*2, sortDouble(d, m, a))
      end
}
```

Определение sortMulti дано в функциональном стиле. В нем сортировка в целом рекурсивно определяется через сортировку вдвое большими кусками. Предикат sortDouble имеет на входе массив a[p..n], отсортированный кусками длиной d, и сортирует его кусками по 2*d.

```
sortDouble(nat d, int p, Ar(p,n) a* : )
      // sortP(a, d, p, n), sortP(a', 2*d, p, n)
{     if p+d-1 >= n then a
      else int q = p+2*d;
            if q-1 >=n then sortPiece(d, p, n, a)
            else a'[p..q-1] = sortPiece(d, p, q-1, a[p..q-1]) ||
                  a'[q..n] = sortDouble(d, q, a[q..n])
            end
      end
}
```

Предикат sortPiece сортирует один кусок a[p..r] длиной не более 2*d при условии, что две его половины отсортированы. Сначала в первой половине a[p..p+d-1] с помощью предиката findUnsorted находится наименьший индекс t, для которого a[t] > a[p+d]. Если условие ложно на всем куске a[p..p+d-1], то реализуется первая ветвь предиката findUnsorted. Кусок a[p..t-1] является отсортированной частью в массиве a[p..r].

После определения индекса t оставшийся кусок a[t..p+d-1] переписывается в массив b[t..p+d-1]. Далее куски b[t..p+d-1] и a[p+d..r] сливаются в отсортированном виде в массив a[t..r]. Это слияние реализуются предикатом merge.

```
sortPiece(nat d, int p, r, Ar(p,r) a* : )
      // sortP(a, d, p, r), a' — упорядочена, a' есть перестановка a
{     int s = p+d;
      real e = a[s];
      split findUnsorted(p, r, s, a, e: | int t)
      do  a
      do a'[p..t-1] = a[p..t-1] ||
         a'[t..r] = merge(t, t, t, s-1, s+1, r, (t: e), a[t..s-1], a[s+1..r])
      end
}
```

Предикат merge сливает в отсортированном виде массивы b[j..k] и c[l..r] в массив a′[t..r], причем начальная часть a′[t..i]=a[t..i] — есть отсортированная часть, полученная ранее путем слияния.

В реализации императивной программы массив a[t..i] склеивается с начальной частью массива a'[t..r], а массив c[l..r] — с конечной частью a'[t..r]. Массив b не склеивается с куском a[t..s-1], а получается в результате его копирования.

```
merge(int t, i, j, k, l, r, Ar(t,i) a, Ar(j,k) b, Ar(l,r) c : Ar(t,r) a')
      // a, b, c — отсортированы, a′[t..i]=a[t..i], a[i]<=b[i], a[i]<=c[l],
      // a′ - отсортирован  и является перестановкой a+b+c
{   if j > k then a' = a + c
    elsif l > r then a' = a + b
    else
        int i1 = i+1;
        if b[j] < c[l] then
                merge(t, i1, j+1, k, l, r, a + (i1: b[j]), b[j+1..k], c : a')
        else    merge(t, i1, j, k, l+1, r, a + (i1: c[l]), b, c[l+1..r] : a')
        end
    end
}
```

Оператор **int** i1 = i+1 не является необходимым. Однако в случае его отсутствия при замене в определении merge хвостовой рекурсии циклом, на каждой из ветвей условного оператора появится оператор i := i+1, который далее следовало бы выносить из обоих ветвей перед условным оператором. Понятно, что подобное вынесение проще сделать с помощью оператора **int** i1 = i+1 в рамках предикатной программы. Отметим, что в императивной программе переменные i и i1 будут склеены.

Предикат findUnsorted определяет минимальный индекс t, для которого a[t] > e. Если условие a[t] > e ложно на всем куске a[p..r], реализуется первая ветвь предиката findUnsorted

```
findUnsorted(int p, r, s, Ar(p,r) a, real e : | t)
      // t=min{x | a[x]>e}
{     if p > s then #1
      elsif a[p] > e then t = p #2
      else findUnsorted(p+1, r, s, a[p+1..r], e : #1 | t #2)
      end
}
```

Приведенные выше определения предикатов представляют полную предикатную программу. Преобразуем ее в эффективную императивную программу.

На первом этапе реализуется склеивание переменных и замена хвостовой рекурсии циклом в каждом из определений предикатов. Предварительно рассмотрим реализацию параллельного оператора в определении sortDouble:

```
a'[p..q-1] = sortPiece(d, p, q-1, a[p..q-1]) || a'[q..n] = sortDouble(d, q, a[q..n])
```

Вызов предиката sortPiece может быть запущен параллельным процессом (потоком), выполняемым одновременно со вторым оператором. Будем использовать конструкцию **thread** <блок>. При ее исполнении создается и запускается новый параллельный процесс, тело которого есть <блок>. Далее исполнение программы, породившей новый процесс, передается конструкции, следующей за **thread** <блок>. Будем также использовать конструкцию **box** <блок> — ее исполнение завершается при завершении исполнения всех параллельных процессов, запущенных при исполнении <блока>.

Таким образом, приведенный параллельный оператор реализуется в императивной программе следующим образом:

```
box { thread { sortPiece(d, p, q-1, a[p..q-1]: a'[p..q-1]) };
      sortDouble(d, q, a[q..n]: a'[q..n])
    }
```

Рекурсия в преобразованном определении предиката sortDouble не является хвостовой. Для приведения рекурсии к хвостовому виду вынесем конструкцию **box** из определения и применим ее для вызова sortDouble в определении sortMulti. Далее проведем склеивание переменных и замену хвостовой рекурсии циклов во всех определениях. Во всех определениях переменная a' склеивается с a. При раскрытии возникающих групповых операторов иногда требуется поменять порядок присваивания переменных. Предварительно функциональный стиль записи заменяется операторным.

```
sort(Ar(m,n) a* : )
{     sortMulti(1, a: a)
}
sortMulti(nat d, Ar(m,n) a* : )
{     loop
            if n-m+1 <= d then exit
            else    box { sortDouble(d, m, a: a) };
                    d := d*2
            end
      end
}
```

```
sortDouble(nat d, int p, Ar(p,n) a* : )
{   loop
        if p+d-1 >= n then exit
        else   int q = p+2*d;
               if q-1 >=n then sortPiece(d, p, n, a: a); exit
               else    thread {sortPiece(d, p, q-1, a[p..q-1]: a[p..q-1]) };
                       p:=q
               end
        end
   end
}
sortPiece(nat d, int p, r, Ar(p,r) a* : )
{       int s = p+d;
        real e = a[s];
        findUnsorted(p, r, s, a, e: #1 | int t);
        merge(t, t, t, s-1, s+1, r, (t: e), a[t..s-1], a[s+1..r]: a[t..r])
}
findUnsorted(int p, r, s, Ar(p,r) a, real e : | t)
{       loop   if p > s then #1
               elsif a[p] > e then t = p #2
               else p:=p+1
               end
        end
}
```

В определении merge начальный кусок a'[t..i] склеивается с a[t..i], а переменная — c[l..r] склеивается с конечным куском a'[l..r]. Массив b не склеивается со средним куском массива a, а копируется из него. Переменная i1склеивается с i.

```
merge(int t, i, j, k, l, r, Ar(t,i) a, Ar(j,k) b, Ar(l,r) a : Ar(t,r) a)
{     loop
              if j > k then a[t..r] = a[t..i] + a[l..r]; exit
              elsif l > r then a[t..r] = a[t..i] + b; exit
              else
                       i := i+1;
                       if b[j] < c[l] then     a[i] := b[j]; j:=j+1
                       else                    a[i] := c[l]; l:=l+1
                       end
              end
      end
}
```

В операторе a[t..r] = a[t..i] + a[l..r] выполняется равенство l = i+1, поскольку j = k+1 и l-i = k-j+2. Данный оператор является тождественной пересылкой и поэтому удаляется из программы.

Подставим определения findUnsorted и merge на место вызовов в определение sortPiece. Подставим определение sortDouble в sortMulti, а затем sortMulti в sorti. Приведем текст итоговой императивной программы.

```
type Ar(int x, y) = array x..y of real;
int m, n; // m <= n
sort(Ar(m,n) a* : )
{     int d:=1;
      loop
            if n-m+1 <= d then exit end;
            box {     int p:=m;
                      loop
                            if p+d-1 >= n then exit end;
                            int q = p+2*d;
                            if q-1 >=n then sortPiece(d, p, n, a: a); exit end;
                            thread {sortPiece(d, p, q-1, a[p..q-1]: a[p..q-1])};
                            p:=q
                      end
                };
            d := d*2
      end
}
sortPiece(nat d, int p, r, Ar(p,r) a* : )
{     int s = p+d; real e = a[s];
      loop if p > s then return
            elsif a[p] > e then t = p; exit
            else p:=p+1
            end
      end;
      int i:=t; int j:=t; int k:=s-1; int l:=s+1;
      Ar(j,k) b := a[t..s-1]; a[t]:=e;
      loop
            if j > k then exit
            elsif l > r then a[i+1..r] = b[j..k]; exit
            else i := i+1;
                  if b[j] < c[l] then     a[i] := b[j]; j:=j+1
                  else                    a[i] := c[l]; l:=l+1
                  end
            end
      end
}
```

Очевидно, что массив b[m..n] можно сделать глобальным по отношению к определению sortPiece, используя в теле sortPiece его единственную копию. Кроме того, следует удалить переменные d и a из списка параметров определения sortPiece, трактуя их в контексте определения sort.

СПИСОК ЛИТЕРАТУРЫ

1. **Кнут Д.** Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. Пер. с англ. — М.: Мир, 1978. — 843с.
2. **Шелехов В.И.** Предикатное программирование: основы, язык, технология // Наст. сб. — С.7–15.

Э.Ю. Петров

СКЛЕИВАНИЕ ПЕРЕМЕННЫХ В ПРЕДИКАТНОЙ ПРОГРАММЕ

Склеивание переменных есть замена в программе нескольких переменных одной. Например, склеивание переменных x, y и z представляет систематическую замену всех вхождений имен y и z на имя x. Разумеется, программа после склеивания должна быть эквивалентна предыдущей.

Склеивание переменных рассматривалось ранее в рамках задачи экономии памяти в классических работах А.П. Ершова, С.С. Лаврова, В.В. Мартынюка. Склеивание переменных определялось как выбор переобозначения аргументов и результатов из заданного множества корректных переобозначений, которое позволяет в наибольшей степени уменьшить объем необходимой памяти [3]. В оптимизирующей трансляции императивных программ склеивание переменных обычно реализуется на основе анализа локальных свойств программы [4, 5].

Технология предикатного программирования предполагает написание программы на языке предикатного программирования Р [2], применение к ней набора оптимизирующих трансформаций с получением программы на императивном расширении языка Р и конвертацию на любой из императивных языков: С, С++ и др. Склеивание переменных реализуется на первом этапе трансформации предикатной программы в императивную.

Предикатная программа представляет собой набор рекурсивных определений предикатов. В определении предиката склеиванию подлежат совместимые по типу локальные и результирующие переменные с исходными переменными. Целью склеивания является оптимизация, прежде всего, по времени исполнения. Значительный эффект достигается при склеивании структурных переменных — массивов и последовательностей.

Задача склеивания переменных в данной постановке рассматривается впервые. Она вряд ли может стать актуальной для оптимизации функциональных программ**:** в определении функции нет результирующих переменных, и там можно было бы рассматривать лишь склеивание локальных и исходных переменных. Эта задача не возникает для императивных программ, поскольку практически все рассматриваемые здесь склеивания обычно реализуются программистом в императивной программе.

1. МОДЕЛЬ ПРОГРАММЫ

Задача склеивания переменных формулируется для модели программы, соответствующей языку предикатного программирования Р [2].

1.1. Определение предиката

Предикат есть условие, определяющее зависимость между значениями переменных. Пусть дан предикат A(x, z), где A — обозначает имя предиката, x — набор **входных** (используемых) переменных $x_1,x_2,...,x_n$ ($n \geq 0$), а z — набор **результирующих** (определяемых) переменных $z_1,z_2,...,z_m$ ($m>0$); наборы x и z не пересекаются. Будем использовать запись A(**x:** z), где "**:**" разделяет входные и результирующие переменные.

Определение предиката имеет следующую форму:

<имя предиката> **(**<входные параметры> **:** < результирующие параметры>**)**
{ <тело предиката> **}**

Вычисление предиката реализует исполнение тела предиката для получения значений результирующих параметров по значениям входных.

Описание каждого параметра предиката имеет вид: <тип> <имя параметра>. Тип может быть примитивным, подмножеством типа или структурным. Примитивные типы это **nat**, **int**, **real** или **bool**. Структурными типами являются: массив, кортеж, объединение, последовательность и множество.

Частным случаем подмножества типа является диапазон, задаваемый следующим образом:

<выражение>..<выражение>

Например, 1..n+1 обозначает диапазон натуральных чисел, где переменная n является параметром типа.

Описание типа "последовательность" имеет вид:

seq <тип элементов последовательности>

Описание типа "массив" имеет вид:

array < тип индекса> **of** < тип элементов массива>

Тип индекса обычно является диапазоном.

Предикат вида A(x**:** y) соответствует функции. Определим предикат более общего вида в форме **гиперфункции.** Допустим, наряду с предикатом A(x**:** z) имеется предикат B(x**:** y). Списки переменных z и y не содержат общих переменных. Области определения функций A и B, соответственно, A_x и

B_x, не пересекаются. Гиперфункция C(x**:** z | y) с двумя **ветвями** есть предикат, определяемый формулой:

$$(x \in A_x \Rightarrow A(x{:}\ z)) \ \&\ (x \in B_x \Rightarrow B(x{:}\ y))$$

Гиперфункция C(x**:** z | y) определена на $A_x \cup B_x$ и совпадает с функцией A на A_x и с функцией B на B_x. Вычисление гиперфункции завершается на одной из двух ветвей с получением значений результирующих переменных завершившейся ветви; переменные другой ветви не вычисляются. Гиперфункция C реализует ветвление — она совмещает в себе преобразователь и распознаватель. Ветвь гиперфункции может быть **пустой**, т.е. не содержать результирующих переменных. По пустой ветви гиперфункция является только распознавателем.

В общем случае, гиперфункция может содержать произвольное количество ветвей.

Исполнение гиперфункции завершается некоторой ветвью с вычислением значений набора результирующих параметров ветви. Для указания того, какой ветвью завершилось вычисление гиперфункции, используется завершитель ветви: **#**<номер ветви>.

1.2. Операторы

Первичными операторами являются: предикат равенства, вызов предиката и оператор разложения.

Предикат равенства имеет вид:

<переменная> **=** <выражение>

<переменная> это либо результирующий параметр определения предиката, либо локальная переменная, декларированная выше в программе описанием вида**:**

< тип> <имя переменной>**.**

Вызов предиката имеет следующий вид:

<имя предиката>([<аргументы>]**:**<результаты>)

Вычисление вызова предиката производится следующим образом. Сначала независимо (параллельно) вычисляются все выражения, находящиеся в позиции аргументов предиката. Полученные значения присваиваются во входные параметры определения вызываемого предиката. Далее выполняется тело предиката, в котором происходит вычисление результирующих параметров. Значения результирующих параметров присваиваются переменным, являющимися результирующими фактическими параметрами в вызове предиката.

На базе примитивных предикатов строятся следующие операторные композиции: суперпозиция (блок), условный оператор, оператор расщепления, параллельный оператор и конструктор массива.

Пусть имеются два оператора: A(x: y) и B(u: z), причем наборы x и z не содержат общих переменных. Оператор A(x: y); B(u: z) является **суперпозицией**, если наборы y и u содержат общие переменные. **Оператор** A(x: y) || B(u: z) является **параллельным оператором**, если в наборах u и y нет общих переменных.

Суперпозиция нескольких операторов образует **блок.** Перед любым оператором блока могут помещаться описания локальных переменных. Блок обычно обрамляется фигурными скобками.

Условный оператор имеет следующий вид:

if <выражение> **then** <сегмент>
else <сегмент>
end

Сегментом является блок и следующий за ним необязательный завершитель ветви. Фигурные скобки блока, составляющего сегмент, можно опускать. Если сегменты условного оператора заканчиваются одинаковыми завершителями, то каждый сегмент должен определять одинаковый набор результирующих переменных.

Оператор расщепления имеет следующий вид:

split <вызов предиката-гиперфункции>
do <первая альтернатива расщепления>
do <вторая альтернатива расщепления>
end

Каждая альтернатива расщепления есть <сегмент>. Исполнение оператора расщепления начинается с вызова предиката-гиперфункции. Если исполнение вызова гиперфункции завершается первой ветвью, то далее исполняется первая альтернатива расщепления, иначе — вторая. На первой ветви расщепления используются переменные, вырабатываемые первой ветвью вызова гиперфункции; аналогично — для второй ветви расщепления.

Определение оператора расщепления обобщается на произвольное число ветвей предиката-гиперфункции.

Для присваивания значений переменным типа «массив» определен оператор — конструктор массива:

forAll <простая переменная> = <диапазон> **do** <оператор> **end**

Для каждого значения <простой переменной>, принадлежащего <диапазону>, независимо (параллельно) исполняется <оператор>, вычисляющий элемент массива. Например, в следующем фрагменте конструируется массив, элементы которого равны 1:

```
int n;
....
array 1..n of int arr;
forAll i = 1..n do arr[i] = 1 end;
```

Оператор разложения предназначен для определения компонент объекта структурного типа. Например, разложение последовательности s->(| e, r) представляет следующее утверждение: последовательность s либо пуста, и тогда реализуется первая ветвь гиперфункции, либо s состоит из элемента e и подпоследовательности r.

Пример 1.

```
Length(seq int s: int len) {
    Length2(s, 0: len)
}

Length2(seq int s, int acc : int len) {
    seq int r;
    int e;
    split s->( | e, r)
    do  len = acc;
    do  Length2 (r, acc+1: len);
    end;
}
```

В примере 1 предикат Length определяет длину последовательности s. Используется вспомогательный предикат Length2, определяемый условием:

Length2 (s, acc) = Length(s)+acc

1.3. Дополнительные определения

Для условного оператора и оператора расщепления будем использовать общее понятие — **оператор ветвления**, состоящий из **ветвей** и **заголовка** — части, предшествующей ветвлению.

Для некоторого оператора G, находящегося в теле некоторого определения предиката, введем обозначения, используемые для описания алгоритма склеивания.

Назовем **аргументами** G параметры предиката и локалы, использующиеся в G. Аргументы G обозначим **Args(G)**. Переменные из Args(G), которые не используются при продолжении исполнения после исполнения оператора G, обозначим **A(G)**. Склеивание переменной из A(G) с другими переменными является допустимой трансформаций, сохраняющей эквивалентность программы по результатам исполнения.

Множество переменных, определяемых (присваиваемых) в операторе G и либо использующихся при дальнейшем исполнении определения предиката после исполнения G, либо являющимися результирующими параметрами предиката, будем называть результатами оператора G и обозначим **R(G)**.

Переменные, определяемые в G и неиспользуемые далее при исполнении тела предиката после G, назовем **собственными локалами** G.

2. ПОСТАНОВКА ЗАДАЧИ

Склеивание переменных есть преобразование предикатной программы, при котором несколько переменных, совместимых по типу, заменяются одной. Например, при склеивании переменных c и d оператор c=d+1 будет преобразован в оператор присваивания c:=c+1, а оператор a = b при склеивании a и b превратится в тождественный предикат a = a, удаляемый из императивной программы. В отличие от задачи экономии памяти [3], склеиванию подлежат только те переменные, между которыми имеется информационная связь.

Типы считаются совместимыми, если они либо тождественны, либо являются конкретизациями одного общего типа с разными значениями параметров типа. Понятие совместимости типов используется здесь в более узком смысле, чем в описании языка Р [2].

Склеивание переменных реализуется на первом этапе в процессе получения эффективной императивной программы и предваряет все остальные трансформации: преобразование хвостовой рекурсии в цикл, подстановку тела предиката на место вызова и конкретизацию операций с переменными структурных типов.

В данной работе рассматривается алгоритм склеивания переменных в пределах одного определения предиката. Правилами языка Р предусматривается возможность склеивания аргументов и результатов для произвольного определяемого предиката. При наличии такого склеивания функциями алгоритма является проверка корректности такого склеивания. Полный ал-

горитм, реализующий также склеивание входных и результирующих параметров предикатов, предполагается разработать в дальнейшем.

В работе не рассматривается склеивание переменных структурных типов разной длины, за исключением склеивания, подразумеваемого в операторе разложения последовательности. Также не рассматривается склеивание переменных типа «массив» в операторе forAll. Например, в следующем фрагменте можно было бы заменить переменную b на a при условии, что в дальнейшем вычислении a не используется.

Пример 2.

```
int n;
array 1..n of int a;
array 1..n+1 of int b;
...
forAll i = 1..n+1 do
        if i <5 then b[i] = a[i] elsif i =5 then b[5] = 0 else b[i] = a[i-1] end;
end;
```

3. ОБЩИЙ АНАЛИЗ И ОПИСАНИЕ АЛГОРИТМА

Обозначим через **<x: y> команду склеивания**, определенную по отношению к некоторому оператору G, где x — аргумент из A(G), а y — результирующая переменная в G. Выполнение этой команды заменяет y на x в G, а также во всех операторах, исполняющихся после завершения исполнения G.

Определим алгоритм построения совокупности команд склеивания для некоторого определения предиката. Применение этой совокупности команд к определению предиката реализует полный набор склеиваний в определении предиката.

Для каждой результирующей переменной $y_k \in R(G)$ $(k = 1,...,m)$ строится **кандидатная пара** $<A_k(G): y_k>$, где: y_k информационно зависима от всех переменных из $A_k(G)$, причем y_k и переменные из $A_k(G)$ — совместимы по типу. $A_k(G)$ называется **множеством кандидатов** для склеивания с y_k.

Перечислим свойства склеивания переменных для произвольного оператора G.

(3.1) Количество команд склеивания, которые можно определить для аргументов и результатов G, не превышает $\min(m, |\cup A_k(G)|)$, где $k = 1,...,m$.

(3.2) Склеивание аргумента G с некоторой собственной локальной переменной может воспрепятствовать склеиванию аргумента с результатом,

если результат определяется до последнего использования локальной переменной.

3.1. Определение кандидатных пар в первичных операторах

Первичными являются предикат равенства, вызов предиката и оператор разложения. Рассмотрим на примерах правила построения кандидатных пар для этих операторов.

Кандидатной парой для предиката равенства a = b+c является <b,c: a>, что допускает возможность выполнения одной из команд склеивания <c: a> или <b: a>.

Пусть имеется определение предиката со следующим заголовком: Foo (**int** a,b: **int** a′, b′). Здесь, правилами языка определены следующие склеивания параметров Foo: <a: a′> и <b: b′>. На основе этой информации для оператора вызова Foo (p1+p2, p3+p4: r1, r2) строятся следующие кандидатные пары <p1, p2: r1> и <p3, p4: r2>.

Результатом разложения непустой последовательности (или множества) являются элемент и подпоследовательность (подмножество), которую можно склеить с исходной последовательностью (множеством).

Пример 3.

```
seq int a;
...
int c;
seq int b;
a→(|int c, seq int b)
```

В примере 3 можно склеить переменные a и b.

3.2. Склеивание в операторах ветвления.

Рассмотрим оператор ветвления G, имеющий форму простой функции, т.е. когда каждая ветвь оператора определяет одно и тоже множество результатов.

Пример 4.

```
if a < 5 then x = a else x = b end;
d = x*x;
```

В примере 4 при независимом склеивании аргументов и результатов на разных ветвях получим взаимоисключающие команды <a: x> и <b: x>. Во избежание подобных конфликтов построение кандидатных пар для резуль-

татов оператора ветвления G реализуется для него в целом на основе анализа информационных связей результатов и аргументов G. Во множество кандидатов, определенных по отношению к G для некоторой результирующей переменной x, войдут все аргументы G, от которых зависит x на ветвях G. Кандидатной парой для оператора ветвления G в примере 4 является <a,b: x>.

Склеивать собственные локалы ветвей между собой и с аргументами можно независимо на каждой ветви.

Рассмотрим построение кандидатных пар для оператора ветвления.

Пример 5.

```
if x < 5 then y = sqrt(x) || q = x*x  else  y = x*x ; q = y*x*b end;
z = q + y;
```

В примере 5 переменную y нельзя склеить с x по ветви **else**, поэтому x не войдет в кандидатную пару для y. В кандидатную пару для некоторой результирующей переменной оператора ветвления не будем включать аргументы, которые нельзя склеить с этой результирующей переменной, хотя бы на одной ветви.

Пример 6.

```
if a < 5 then c = b || e =b  else  c = a || e =b  end;
d = c*e;
```

Пример 6 показывает, что выбор различных вариантов склеивания аргументов и результатов может влиять на количество склеиваний. В примере возможны два варианта множеств кандидатных пар: {< b: c >} и {<a: c >, <b: e>}.

Рассмотрим следующий фрагмент:

Пример 7.

```
if a < 5 then
    c = b ||
    if a>2 then x=sqr(a) else x=a end          //(1)
else
    c = a || x =b
end;
d = c*x;
```

В примере 6 бессмысленно строить кандидатные пары для переменной x внутри вложенного в ветвь **then** оператора ветвления (1), так как на уровне оператора (1) нет полной информации о зависимости x от аргументов. По-

этому если результирующие переменные G являются также результирующими для некоторого объемлющего оператора ветвления, то для таких переменных нельзя строить кандидатные пары — такое построение будет проходить на уровне объемлющего оператора ветвления.

Для операторов ветвления, имеющих форму гиперфункции, склеивание переменных будем проводить на каждой ветви гиперфункции независимо, так как каждая ветвь определяет независимый набор результирующих переменных.

3.3. Склеивание переменных в параллельном операторе

Пример 8.

G(a: y)||H(a,b: x)

Переменная a используется в обеих альтернативах параллельного оператора. Чтобы применить команду склеивания <a: y>, необходимо перед параллельным оператором запомнить значение переменной a, поскольку оно используется для вычисления H(a,b: x). Однако при замене параллельного исполнения двух компонент параллельного оператора на последовательное: H(a,b: x); G(a: y) с перестановкой операторов, команду <a: y> можно применить без предварительного сохранения значения a. Рассмотрим преобразование, заменяющее параллельный оператор на блок с последовательным исполнением, в котором для проведения склеиваний мы жертвуем параллелизмом.

Рассмотрим параллельный оператор G вида G_1 ||...|| G_m. Для каждой компоненты G построим кандидатные пары $<A_k(G_i), y_k>$, где $A_k(G_i)$ — переменные из $A(G_i)$, используемые при вычислении k-го результата y_k в G_i (i=1..m). Входящие в A(G) аргументы назовем **уникальными,** если они используются только в одной альтернативе, а аргументы, использующиеся в более чем одной альтернативе, назовем **общими**.

Определим следующие критерии эффективности склеивания переменных. Если результирующие переменные одного типа, то будем добиваться максимально возможного количества пар кандидатов. Если результирующие переменные разного типа, причем, размер занимаемой ими памяти определен однозначно, то будем склеивать переменные так, чтобы размер занимаемой памяти склеенных переменных был максимален. В общем случае, будем использовать в качестве критерия систему приоритетов, когда в первую очередь склеиваются переменные динамического размера, затем последовательности и множества, и в последнюю — те переменные, для

которых размер занимаемой памяти определен однозначно, в порядке убывания их размеров.

Очевидно, что склеивание некоторого уникального аргумента из $A_k(G_i)$ с y_k не препятствует параллельному исполнению других компонент G, поэтому будем рассматривать компоненты, в которых имеются кандидатные пары с общими аргументами. Склеивание общего для нескольких компонент аргумента с одним из результатов не позволяет сохранить параллелизм между этими компонентами. Поэтому заменим параллельное исполнение последовательным, причем будем искать расстановку операторов с максимальной эффективностью склеивания.

Сгруппируем компоненты G, в кандидатных парах которых есть общие аргументы. При этом оператор G_i будет принадлежать некоторой группе Г, если существует хотя бы один оператор из Г, с которым G_i имеет общие переменные-кандидаты. Компоненты, принадлежащие разным группам, можно исполнять параллельно, так как по построению у них нет общих аргументов.

Найдем для каждой группы порядок исполнения компонент, допускающий склеивание переменных с максимальной эффективностью. В большинстве случаев каждая группа будет содержать небольшое количество компонент, поэтому расстановку можно проводить полным перебором всех возможных вариантов. Менее затратные способы склеивания переменных в параллельных операторах с общими аргументами, в том числе учитывающих условие (3.1), будут рассмотрены в дальнейшем.

Пример 9.

G(a,b: y_2) || H(a: y_1) || F(a,b,c: y_3) || E(a,b,c,d: y_4)

Для параллельного оператора из примера 9, лучшим порядком следования операторов будет:

E(a,b,c,d: y_4); F(a,b,c: y_3); G(a,b: y_2); H(a: y_1)

при котором можно провести 4 склеивания: <a: y_1>, <b: y_2>, <c: y_3>, <d: y_4>.

Рассмотрим ситуацию, когда в параллельном операторе определяются результаты некоторого объемлющего оператора ветвления.

Пример 10.

if a<b **then**
G(a,b: y_2) || H(a: y_1) || F(a,b,c: y_3) || E(a,b,c,d: y_4)
else
G(b: y_2) || H(a,b: y_1) ||F(a,b,c: y_3) || E(a,b,c,d: y_4)
end

В примере 10 лучшие варианты склеивания на разных ветвях конфликтуют между собой:

	Лучший порядок следования	Лучшее склеивание на ветви
Ветвь **then**	H(a: y_1); G(a,b: y_2); F(a,b,c: y_3); E(a,b,c,d: y_4)	<a: y_1>, <b: y_2>, <c: y_3>, <d: y_4>
Ветвь **else**	G(a: y_2);H(a,b: y_1); F(a,b,c: y_3); E(a,b,c,d: y_4)	<a: y_2>, <b: y_1>, <c: y_3>, <d: y_4>

В таких случаях необходимо отказаться от построения команд склеивания, конфликтующих со склеиванием на другой ветви.

3.4. Применение команд склеивания

Для первичных операторов G, где построены кандидатные пары <A_k(G), y_k>, определим команду <x: y_k> выбором любого x из A_k(G). Для оператора ветвления выберем такие команды, которые обеспечивают максимальное количество склеенных переменных при сохранении максимального количества подконструкций с параллельным исполнением.

Построение полученных команд реализуется в процессе рекурсивного обхода рассматриваемого определения предиката. В каждой конструкции будем обходить все подконструкции в глубину до достижения всех первичных операторов. В случае блока, в том числе полученного преобразованием параллельного оператора, обход начнем с последнего оператора. Для оператора ветвления установим следующий порядок обхода**:** сначала обходятся все ветви, затем заголовок оператора. В процессе обхода при достижения первичного оператора проведем замены в определении предиката в соответствии с командой склеивания.

Результатом применения к определению предиката произвольной команды склеивания, будет эквивалентная программа, так как по построению и в случае предложенного порядка обхода определения предиката переменные из A_k(G) не встречаются далее по исполнению, с другой стороны, y_k не может быть переопределено. Таким образом, до оператора G определение

предиката останется неизменным, а начиная с оператора G и далее по исполнению произойдет замена названия переменной, что не изменит процесса и результатов вычислений.

3.5. Замечания

Общий алгоритм склеивания является линейным по отношению к числу переменных в определении предиката. Применение полного перебора для определения порядка исполнения компонент параллельного оператора с числом компонент n определяет оценку сложности этой части алгоритма как O(n!) для худшего случая, когда число общих аргументов больше n.

Для указанных пользователем склеиваний входных параметров предикатов с результирующими будем проверять наличие информационной связи между склеиваемыми переменными. При отсутствии такой связи склеивание квалифицируется как некорректное.

ЗАКЛЮЧЕНИЕ

В данной работе описан линейный алгоритм склеивания переменных, учитывающий ограничения и особенности операторов ветвления и параллельных операторов предикатной программы.

В дальнейшем предполагается решить следующие задачи:

- склеивание переменных структурных типов разной длины, в том числе массивов;
- нахождение менее затратного способа реорганизации параллельных операторов;
- определение взаимосвязей между склеиванием переменных и последующими трансформациями предикатной программы.

СПИСОК ЛИТЕРАТУРЫ

1. **Шелехов В.И.** Введение в предикатное программирование — Новосибирск, 2002. — 82 с. — (Препр. / ИСИ СО РАН; №100).
2. **Шелехов В.И.** Язык предикатного программирования. — Новосибирск, 2002. — 40 с. — (Препр. / ИСИ СО РАН; №101).
3. **Ершов А.П.** Введение в теоретическое программирование — М.: Наука, 1977. — 288 с.

4. **Поттосин И.В.** Направленные преобразования линейного участка // Языки и системы программирования. — Новосибирск, 1981. — С.14–28.
5. **Bacon D., Graham S., Sharp O.** Compiler transformations for high-performance computing // ACM Computing surveys. — 1994. — Vol. 26, No. 4 — P. 345–420.

СОДЕРЖАНИЕ

МЕТОДЫ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Сборник научных работ под редакцией В.И.Шелехова

Рукопись поступила в редакцию 10.12.03
Редактор З. В. Скок

Подписано в печать 29.12.03
Формат бумаги 60 × 84 1/16 Объем 3.6 уч.-изд.л., 3.9 п.л.
Тираж 70 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 34-22-02