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.