УДК 004.415.52

# Основы предикатного программирования

**В.И. Шелехов**

Институт Систем Информатики СО РАН, г. Новосибирск,
e-mail: vshel@iis.nsk.su

## Введение

## 1. Корректность программ-функций

Предикатное программирование ограничено классом *программ-функций*. Это простейший класс в системе *классификации программ*. Классификация ориентирована на разработку адекватной технологии программирования для каждого класса программ. Например, принципиально различаются классы программ для задач вычислительной математики и систем реального времени. Соответственно, для этих классов следует разрабатывать разные методы и инструменты программирования.

Обычно рассматриваются классификации по назначению программ. Здесь же определяется классификация программ по их внутренней организации. Генеральная классификация определяет два класса программ: *программы-функции* и *реактивные системы*. Данные два класса составляют более 90% всех программ. Классификация реактивных систем, соответствующих автоматному программированию, рассматривается в работе [1]. Класс реактивных систем сложнее класса программ-функций в частности потому, что программа реактивной системы строится из частей, соответствующих программам-функциям.

Имеются другие, более сложные классы, например, языковые процессоры и операционные системы; они находятся на метауровне по отношению к первым двум классам. Языковые процессоры – это интерпретаторы программ, трансляторы, оптимизаторы, трансформаторы и т.д. Принципиально, что транслятор с одного языка на другой нельзя определить в виде программы-функции. Приведенная классификация не покрывает всего спектра программ и различных особенностей, например такой, как тесная интеграция программы с данными.

Программа принадлежит *классу программ-функций*, если она не взаимодействует с внешним окружением программы; точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод собран в конце программы. Если подобная перестройка программы принципиально невозможна, ее следует определять в виде реактивной системы. Программа-функция должна всегда нормально завершаться с получением результата, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов. Класс программ-функций, по меньшей мере, содержит программы для задач дискретной и вычислительной математики.

Программу-функцию U с набором аргументов x и набором результатов y будем записывать в виде U(x: y). *Спецификацией* программы-функции являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Спецификацию программы будем записывать в

виде $[P(x), Q(x, y)]$. Программу со спецификацией обычно принято записывать в виде тройки Хоара: $P(x)\ \{U(x\colon y)\}\ Q(x, y)$.

Известные понятия тотальности и однозначности функции $f\colon x \to y$, где $x$ и $y$ – непересекающиеся наборы переменных, естественным образом переносятся на предикат $H(x, y)$, рассматриваемый как функция из $x$ в $y$. Предикат $H(x, y)$ является *однозначным* в области $X$ для набора переменных $x$, если истинно:

$$\forall x{\in}X\ \forall y_1,y_2.\ H(x, y_1)\ \&\ H(x, y_2) \Rightarrow y_1 = y_2\ .$$

Предикат $H(x, y)$ является *тотальным* в области $X$ для набора переменных $x$, если:

$$\forall x{\in}X\ \exists y.\ H(x, y)\quad .$$

Далее потребуется следующая лемма.

**Лемма 1**. Допустим, предикат $D(x, y)$ является однозначным в области $X$, а предикат $Z(x, y)$ – тотальный в области $X$. Пусть истинна формула $\forall x{\in}X.\ Z(x, y) \Rightarrow D(x, y)$. Тогда истинна следующая формула: $\forall x{\in}X.\ D(x, y) \Rightarrow Z(x, y)$. Как следствие, предикаты $D$ и $Z$ оказываются тождественными в области $X$.

*Однозначность и тотальность спецификации* $[P(x), Q(x, y)]$ определяются, соответственно, формулами:

$$P(x)\ \&\ Q(x, y_1)\ \&\ Q(x, y_2) \Rightarrow y_1 = y_2.$$
$$P(x) \Rightarrow \exists y.\ Q(x, y);$$

Программа должна соответствовать спецификации. Это требование формулируется в виде условия *частичной корректности*: если перед исполнением программы $U(x\colon y)$ истинно предусловие $P(x)$, то в случае завершения программы должно быть истинно постусловие $Q(x, y)$ в момент ее завершения. Обязательным является также *условие завершения программы*: если перед исполнением программы $U(x\colon y)$ истинно предусловие $P(x)$, то программа обязана завершаться. Объединение условий частичной корректности и завершения программы определяет условие *тотальной корректности*.

Адекватная формализация условий корректности программы возможна лишь при использовании формальной операционной семантики языка программирования. *Операционную семантику* программы $U(x\colon y)$ определим в виде предиката:

$\mathcal{R}(U)(x, y) \cong$ для значения набора $x$ исполнение программы $U$ всегда завершается и существует исполнение программы, при котором результатом вычисления является значение набора $y$.

Данное определение исключает ситуацию, когда для некоторого значения набора $x$ существуют два разных исполнения программы, одно из которых завершается, а другое – не завершается. В этом случае $\mathcal{R}(U)(x, y)$ будет ложным для любых $y$.

*Однозначность и тотальность программы* $U(x\colon y)$ для некоторого значения $x$ определяются, соответственно, формулами:

$$\mathcal{R}(U)(x\colon y_1)\ \&\ \mathcal{R}(U)(x\colon y_2) \Rightarrow y_1 = y_2;$$
$$\exists y.\ \mathcal{R}(U)(x, y)\ .$$

Отметим, что тотальность программы для некоторого значения $x$ есть в точности условие завершения программы для этого значения $x$. Программа называется *однозначной*, если она однозначна для всех значений аргументов. Далее ограничимся рассмотрением лишь однозначных программ.

Операционная семантика $\mathcal{R}(U)$ является эквивалентом программы $U$. Доказательство некоторого свойства программы $W(x, y)$ реализуется доказательством

истинности формулы: $\mathcal{R}(U)(x, y) \Rightarrow W(x, y)$. Кроме того, эту формулу достаточно доказать при истинном предусловии. С учетом этого, условие частичной корректности программы $U(x{:}\ y)$ записывается в виде формулы:

$$\forall x,y.\ P(x)\ \&\ \mathcal{R}(U)(x, y) \Rightarrow Q(x, y)\ .$$

Условие завершения программы $U(x{:}\ y)$ при истинном предусловии представляется формулой:

$$\forall x.\ P(x) \Rightarrow \exists y.\ \mathcal{R}(U)(x, y)\ .$$

Формула для условия тотальной корректности программы получается объединением двух формул:

$$\forall x.\ P(x) \Rightarrow [\forall y.\ \mathcal{R}(U)(x, y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ \mathcal{R}(U)(x, y)\ . \qquad (1)$$

**Лемма 2**. Если программа $U(x{:}\ y)$ тотально корректна относительно спецификации $[P(x), Q(x, y)]$, то спецификация тотальна.

Приведенные выше леммы и теорема доказаны в системе автоматического доказательства PVS: http://www.iis.nsk.su/persons/vshel/files/rules.zip.

# 2. Построение языка предикатного программирования P

*Предикатная программа* $H(x{:}\ y)$ с аргументами $x$ и результатами $y$ есть предикат в форме вычислимого оператора. Множество предикатных программ есть вычислимое подмножество языка исчисления предикатов. Минимальный полный *базис предикатных программ* определен в виде языка $P_0$ [1]. Для него построена формальная операционная семантика и доказано тождество $\mathcal{R}(H) = H$. Наша цель – расширить язык $P_0$ до удобного языка предикатного программирования P.

## 2.1. Язык $P_0$

Используется две эквивалентные формы языка $P_0$: *предикатная* на языке исчисления предикатов и *операторная* для проведения вычислений.

Произвольная предикатная программа определяется следующей конструкцией:

<имя предиката>(<аргументы>**:** <результаты>) { <оператор> }

Аргументы и результаты – разные непересекающиеся наборы имен переменных. Набор аргументов может быть пустым.

Простейшим оператором, используемым в составе других операторов, является *вызов предиката* $B(x{:}\ y)$, где B – имя предиката. Допускается также вызов предиката вида $A(x{:}\ y)$, где A – имя переменной *предикатного типа*. Значение переменной A есть имя предиката.

Предположим, что x, y и z обозначают разные непересекающиеся наборы переменных. Набор x может быть пустым, наборы y и z не пусты. В составе набора переменных x может использоваться логическая переменная e со значениями **true** и **false**. Пусть B и C – имена предикатов, A и D – имена переменных предикатного типа. Операторами являются: *оператор суперпозиции* $B(x{:}\ z);\ C(z{:}\ y)$, *параллельный оператор* $B(x{:}\ y)\ ||\ C(x{:}\ z)$, *условный оператор* **if (**e) $B(x{:}\ y)$ **else** $C(x{:}\ y)$, *вызов предиката* и *оператор каррирования*.

Ниже в таблице 1 представлен полный базис вычислимых предикатов и соответствующих им операторов. Левая колонка содержит различные определения предиката H, правая – соответствующие им предикатные программы.

| H(x: y) ≅ ∃z. B(x: z) & C(z: y) | H(x: y) { B(x: z); C(z: y) } |
|---|---|
| H(x: y, z) ≅ B(x: y) & C(x: z) | H(x: y, z) { B(x: y) \|\| C(x: z) } |
| H(x: y) ≅ (e ⇒ B(x: y)) & (¬e ⇒ C(x: y)) | H(x: y) { **if** (e) B(x: y) **else** C(x: y) } |
| H(x: y) ≅ B(x˜: y) | H(x: y) { B(x˜: y) } |
| H(A, x: y) ≅ A(x: y) | H(A, x: y) { A(x: y) } |
| H(x: D) ≅ ∀y,z. D(y: z) ≡ B(x, y: z) | H(x: D) { D(y: z) {B(x, y: z)} } |
| H(A, x: D) ≅ ∀y,z. D(y: z) ≡ A(x, y: z) | H(A, x: D) { D(y: z) {A(x, y: z)} } |

Таблица 1. Вычислимые предикаты и их программная форма

Набор x˜ составлен из набора переменных x с возможным добавлением имен предикатных программ.

Имеется два вида оператора каррирования: D(y: z) {B(x, y: z)} и D(y: z) {A(x, y: z)}. Результатом исполнения оператора каррирования является новый предикат D, получаемый фиксацией значения набора x.

*Полная программа* состоит из конечного набора предикатных программ. Исполнение полной программы начинается с некоторой предикатной программы, называемой *главной*. Любое имя предиката B, встречающееся в операторной части любой предикатной программы, должно быть определено одним из следующих способов:

- в составе полной программы имеется предикатная программа с именем B;
- предикат B является *базисным*;
- предикат B является *параметром* полной программы и определяется в другой полной программе.

*Базисный предикат* определяет одну из элементарных операций над числами или логическими значениями. Для базисного предиката нет предикатной программы – он считается предопределенным в языке $P_0$. Примеры базисных предикатов: +(a, b: c), <(a, b: e), =(a, b: e), где e – логическая переменная. Они соответствуют следующим отношениям: c = a + b, e = (a< b), e = (a = b). Базисный предикат =(a: b) соответствует отношению b = a. Базисный предикат является эквивалентом соответствующего *оператора присваивания*. Набор базисных предикатов языка $P_0$ может быть произвольным. Однако в данной работе мы ограничиваемся рассмотрением лишь однозначных базисных предикатов.

Определим формальную операционную семантику языка $P_0$. Для формальной операционной семантики произвольного предиката H(x: y) используется обозначение ℛ(H).

Пусть A – переменная предикатного типа. В операционной семантике переменную A будем представлять структурой (A.name, A.pref) из двух полей: name – имя предиката и pref – *префикс каррирования*, определяющий набор значений, фиксированных применением оператора каррирования. При подстановке имени программы C в качестве аргумента некоторого вызова предиката B(x˜: y) этот аргумент представляется структурным значением (C, ( )), где ( ) – обозначает пустой набор значений в качестве префикса каррирования.

Для произвольной предикатной программы H(x: y) { <оператор> } реализуется тождество ℛ(H)(x, y) ≡ ||<оператор>||, где ||<оператор>|| обозначает операционную семантику <оператора>. Определим операционную семантику различных видов операторов из Таблицы 1.

|| B(x: z); C(z: y) || ≅ ∃z. || B(x: z) || & || C(z: y) ||
|| B(x: y) || C(x: z) || ≅ || B(x: y) || & || C(x: z) ||
|| **if** (e) B(x: y) **else** C(x: y) || ≅ (e ⇒ || B(x: y) ||) & (¬e ⇒ || C(x: y) ||)
|| B(x˜: y) || ≅ ℛ(B)(||x˜||, y)
|| A(x: y) || ≅ || A.name(A.pref, x, y) ||
|| D(y: z) {B(x, y: z)} || ≅ D = (B, (x))
|| D(y: z) {A(x, y: z)} || ≅ D = (A.name, (A.pref, x)) .

Здесь (A.pref, x) обозначает новый префикс каррирования, полученный добавлением фиксированных значений набора x к префиксу A.pref; ||x˜|| обозначает замену любого вхождения в x˜ имени предиката C на (C, ( )).

**Теорема 3**. Для произвольной предикатной программы H(x: y) на языке $P_0$ истинно тождество ℛ(H) = H. Предполагается, что данное тождество истинно для всех базисных предикатов. Доказательство см. в работе [1].

Здесь ℛ(H) определяется для программы предиката H, а вхождение H справа понимается в смысле предиката, определенного в левой колонке Таблицы 1.

Имеются другие ограничения. Не допускаются сложные формы рекурсии, опосредованные через подстановку предиката C аргументом вызова B(x˜: y) в случае, когда C зависит от B. Для условного оператора **if** (e) B(x: y) **else** C(x: y) вхождение переменной e не может быть источником рекурсии. Операционная семантика определена лишь для однозначных предикатных программ.

### 2.2. Язык P1: блочная структура программы

Допустим, в полной программе на языке $P_0$ имеется предикатная программа B(x: y) { K(x: y) } и вызов предиката B(t: z) внутри программы некоторого предиката H. Здесь K(x: y) обозначает один из операторов Таблицы 1. Будем считать, что наборы переменных x, y, t и z не пересекаются. *Подстановкой предикатной программы* на место вызова B(t: z) является конструкция { K(t: z) }, называемая *блоком*. Оператор K(t: z) получается из K(x: y) заменой всех вхождений переменных наборов x и y на соответствующие переменные наборов t и z. Здесь и далее мы полагаем, что для любой предикатной программы ее параметры имеют уникальные имена, не встречающиеся в других предикатных программах. Кроме того, если K(x: y) – оператор суперпозиции, то при построении блока { K(t: z) } возможно проводится переименование локальных переменных, обеспечивающее их уникальность. С учетом данных соглашений замена x и y на t и z в конструкции K(x: y) не приведет к коллизиям. Операционная семантика блока определяется тождеством:

||{ K(t: z) }|| ≅ ||K(t: z)|| .

Замена вызова B(t: z) блоком { K(t: z) } в программе предиката H сопровождается аналогичной заменой терма B(t: z) в предикате H из левой части Таблицы 1. Для модифицированного предиката H˜ реализуется тождество ℛ(H˜) = H˜ .

Язык программы, получающийся многократным произвольным применением подстановок предикатных программ на место вызовов, обозначим через $P_1$. В предикатной программе на языке $P_1$ в позиции вызова предиката для операторов Таблицы 1 может находиться либо вызов, либо блок, причем блок может содержать блоки внутри себя, т.е.

иметь иерархическую структуру. Для произвольной предикатной программы H на языке $P_1$ реализуется тождество $\mathcal{R}(H) = H$.

### 2.3. Язык $P_2$: оператор суперпозиции и параллельный оператор общего вида

Операторы { B(...); C(...) }; E(...) и B(...); { C(...); E(...) } являются эквивалентными, поскольку их операционная семантика тождественна. Аналогичным образом устанавливается ассоциативность параллельной композиции, т. е. эквивалентность { B(...) || C(...) } || E(...) и B(...) || { C(...) || E(...) }. Свойство ассоциативности позволяет опускать внутренние фигурные скобки.

Рассмотрим оператор суперпозиции общего вида:

$$B_1(x{:}\ z_1);\ B_2(z_1{:}\ z_2)\ ;...;\ B_j(z_{j-1}{:}\ z_j)\ ;...;\ B_n(z_{n-1}{:}\ y)\ .$$

Здесь x, $z_1$, $z_2$, …, $z_{n-1}$, y – различные непересекающиеся наборы переменных; $n > 1$; $B_1$, $B_2$, …, $B_n$ обозначают вызовы предикатов или блоки языка $P_1$. Нетрудно доказать следующую формулу для операционной семантики оператора суперпозиции общего вида:

$$\|B_1(x{:}\ z_1);\ B_2(z_1{:}\ z_2)\ ;...;\ B_j(z_{j-1}{:}\ z_j)\ ;...;\ B_n(z_{n-1}{:}\ y)\| \equiv$$
$$\exists z_1,z_2,...,z_{n-1}.\ \|B_1(x{:}\ z_1)\| \,\&\, \|B_2(z_1{:}\ z_2)\| \,\&...\&\, \|B_j(z_{j-1}{:}\ z_j)\| \,\&...\&\, \|B_n(z_{n-1}{:}\ y)\|\ .$$

Рассмотрим параллельный оператор общего вида:

$$B_1(x{:}\ y_1)\ ||\ B_2(x{:}\ y_2)\ ||...||\ B_j(x{:}\ y_j)\ ||...||\ B_n(x{:}\ y_n)\ .$$

Здесь x, $y_1$, $y_2$, …, $y_n$ – различные непересекающиеся наборы переменных; набор y объединяет в себе наборы $y_1$, $y_2$, …, $y_n$; $B_1$, $B_2$, …, $B_n$ – предикаты и блоки языка $P_1$. Операционная семантика параллельного оператора общего вида определяется формулой:

$$\|B_1(x{:}\ y_1)\ ||\ B_2(x{:}\ y_2)\ ||...||\ B_j(x{:}\ y_j)\ ||...||\ B_n(x{:}\ y_n)\ \| \equiv$$
$$\|B_1(x{:}\ y_1)\| \,\&\, \|B_2(x{:}\ y_2)\| \,\&\, ...\,\&\, \|B_j(x{:}\ y_j)\| \,\&\, ...\,\&\, \|B_n(x{:}\ y_n)\|\ .$$

Наряду с B(x: z); C(z: y) возможна суперпозиция двух операторов вида B(x: z); C(x, z: y). Наиболее общей формой суперпозиции двух операторов является композиция вида:

$$B(x{:}\ z,\ t);\ C(x,\ z{:}\ y)\ .$$

Здесь x, y, z, t – различные непересекающиеся наборы переменных, причем наборы x и t могут быть пустыми; B и C обозначают предикаты или блоки языка $P_1$. Определим предикатные программы:

$$B1(x{:}\ x1,\ z,\ t1)\ \{B(x{:}\ z,\ t1)\ ||\ x1 = x\};$$
$$C1(x1,\ z,\ t1{:}\ y,\ t)\ \{C(x1,\ z{:}\ y)\ ||\ t = t1\}\ .$$

Определим композицию общего вида B(x: z, t); C(x, z: y) как эквивалент суперпозиции стандартного вида:

$$B1(x{:}\ x1,\ z,\ t1);\ C1(x1,\ z,\ t1{:}\ t,\ y)\ .$$

Нетрудно вывести следующую формулу для операционной семантики:

$$\|B(x{:}\ z,\ t);\ C(x,\ z{:}\ y)\| \equiv \exists z.\ \|B(x{:}\ z,\ t)\| \,\&\, \|C(x,\ z{:}\ y)\|\ .$$

Для комбинации оператора суперпозиции с параллельным оператором реализуются аналоги свойства дистрибутивности. Если в операторе E(x: y); {B(z: t) || C(u: v)} набор y пересекается с набором z и не пересекается с набором u, то оператор эквивалентен {E(x: y); B(z: t)} || C(u: v). Аналогичным образом оператор {E(x: y) || B(z: t)};C(u: v) эквивалентен E(x: y) || {B(z: t); C(u: v)}, если наборы y и u не пересекаются.

### 2.4. Язык $P_3$: выражения

Допускается использование *функциональной* формы операторов предикатной программы. Оператор вызова предиката B(x**:** y) эквивалентен оператору присваивания y = B(x), где B(x) интерпретируется как соответствующий *вызов функции*. В случае, когда y – набор более чем из одной переменной, будем также использовать форму записи |y| = B(x). Оператор суперпозиции B(x**:** z); C(x, z**:** y) может быть записан в виде y = C(x, B(x)). Фрагмент программы { B(x: y) || C(z: t) }; E(y, t: u) может быть записан в виде оператора u = E(B(x), C(z)). При этом вычисление значений аргументов вызова реализуется параллельно.

Для базисных предикатов языка $P_0$, соответствующих стандартных арифметических операциям, вместо функциональной формы z = A(t) будем использовать традиционную инфиксную и постфиксную нотацию. Например, базисные предикаты: +(x, y**:** z), –(x, y**:** z), –(x**:** y), <(x, y**:** b) – записываются в языке $P_3$ привычным образом: z = x + y, z = x – y, y = –x , b = x < y.

В языке $P_3$ вводятся изображения констант. Так, вместо базисных предикатов ConstZero( **:** x) и ConstOne( **:** x) используются операторы присваивания x = 0 и x = 1. Значение x константы целого типа по ее изображению s определяется вызовом предиката valInt(s **:** x), входящего в библиотеку языка $P_3$. Например, оператор x = 2089 интерпретируется как вызов valInt("2089" **:** x)[1]. Аналогичным образом определяются изображения констант других типов.

Оператор суперпозиции z = a ∗ b; y = z + c может быть записан в более компактном виде: y = (a ∗ b) + c. В общем случае, при подстановке a ∗ b вместо z действует правило: подставляемая операция a ∗ b обрамляется круглыми скобками. Таким образом, приходим к известному понятию *выражения*. Использование правил приоритетов операций, позволяющих опускать круглые скобки, определяет привычную форму записи выражений; например, y = a ∗ b + c.

Переменные, изображения констант, вызовы функций и их представление в виде операций являются частными случаями понятия выражения. В языке $P_3$ аргумент вызова в общем случае является выражением.

Композиция E(x**:** e); **if** (e) B(x**:** y) **else** C(x**:** y) эквивалентна оператору **if** (E(x)) A(x**:** y) **else** B(x**:** y). Таким образом, в позиции условия в условном операторе может находиться произвольное выражение логического типа.

### 2.5. Язык предикатного программирования P

Язык P строится на базе языка $P_3$. Детальное и полное описание языка P представлено в препринте [10]. Для возможности использования в практическом программировании язык должен обладать рядом важнейших свойств. Ниже рассматриваются аспекты, определяющие язык в целом.

**Структура полной программы**. *Модуль* есть единица компиляции, содержащая набор предикатных программ, определяющий полную программу или ее часть. Модуль также содержит описания глобальных переменных, типов и классов. *Глобальная переменная* встречается в одной или нескольких предикатных программах модуля и при

[1] Разумеется, предварительно следует определить произвольную константу строкового типа

этом не упоминается в списках аргументов и результатов. *Класс* определяет набор переменных – *полей класса* и *методов*, представленных предикатными программами в теле класса. Использование классов упрощает программу локализацией (упрятыванием) части связей между объектами программы внутри классов [11].

**Предикатная программа** в общем случае представляется следующей конструкцией:

<имя предиката>(<аргументы>**:** <результаты>)
**pre** <предусловие> **post** <постусловие> **measure** <мера>
{ <оператор> };

Обязательными являются имя программы и списки аргументов и результатов. Остальные подконструкции могут отсутствовать. Предусловие и постусловие являются формулами на языке исчисления предикатов; язык формул определен как часть языка P. *Мера* определяет натуральную функцию на аргументах рекурсивных программ для дедуктивной верификации и программного синтеза (см. разд. 3.1). *Тело программы* есть подконструкция { <оператор> }. Программа без тела программы есть *спецификация программы*.

**Типы**. Всякая переменная характеризуется *типом* – множеством допустимых значений. Описание переменной определяется в стиле языков C и C++ следующей конструкцией: <тип> <имя переменной>. В языке P принята строгая статическая типизация переменных. Это не исключает возможности построения бестипового языка предикатного программирования, в котором при описании переменной тип не указывается; при трансляции тип определяется по вхождениям переменной в программе.

Типы **bool**, **int**, **real** и **char** являются *примитивными*. Они используются при построении других типов. Значением типа **struct**($T_1$ $f_1$, ..., $T_n$ $f_n$) является *структура* из n значений, именуемых *полями* $f_1, f_2, ..., f_n$ и имеющих типы $T_1, T_2, ..., T_n$, соответственно. Тип **set**(T) определяет *множество подмножеств* конечного типа T. Значением типа **array**($T_e$, $T_i$) является *массив* с *элементами массива* типа $T_e$ и *индексами* конечного типа $T_i$. Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами.

Значением *алгебраического типа* **union**($K_1$, …, $K_n$) является один из *конструкторов* $K_1$, $K_2$, …, $K_n$. Конструктор определяется *именем конструктора* и набором полей как у структуры; набор полей может быть пустым. Типичными объектами алгебраических типов являются списки и деревья. *Список* определяет последовательность элементов некоторого типа T. Описание типа списка следующее:

**type** list (**type** T) = **union**( nil, cons(T car, list(T) cdr) );

Тип list имеет два конструктора: nil, обозначающий пустой список, и cons, определяющий список, первый элемент которого представлен полем car, а остальная часть списка («хвост» – все элементы, кроме первого) определяется полем cdr. Тип list считается определенным в языке P и не требует описания в программе. В языке P также определен *строковый* тип **string**, определяемый описанием:

**type string** = list(**char**) .

Тип *перечисления* **enum**($K_1$, …, $K_n$) есть нерекурсивный алгебраический тип, все конструкторы которого не имеют полей.

Пусть E(x) – логическое выражение. Тип **subtype**(T x**:** E(x)) определяет *подтип* типа T при истинном предикате E(x), т.е. множество {x∈T | E(x)}. Определенный в языке P тип целых чисел **nat** представляется описанием:

**type nat** = **subtype**(**int** x: x ≥ 0) .

Допускаются подтипы, параметризуемые переменными. Примером является тип *диапазона* целых чисел:

**type** Diap(**nat** n) = **subtype**(**int** x: x ≥ 1 & x ≤ n) .

В языке P для изображения типа диапазона используется конструкция 1..n.

Описания типов переменных являются частью спецификации программы. Описание переменной T x есть утверждение x∈T, которое становится частью предусловия, если x – аргумент предикатной программы, или частью постусловия, если x – результат программы. При этом утверждение x∈T обычно не пишется в составе предусловия или постусловия, хотя предполагается.

**Операция with для предикатного типа**. Для предиката B(x**:** y) определим программу:

with(B, x1, y1**:** C) { C(x**:** y) {**if** (x = x1) y = y1 **else** B(x**:** y)} } .

Вызов программы with(B, x1, y1**:** C) будем записывать в виде C = B **with** (x1**:** y1). Конструкция B **with** (x1**:** y1, x2**:** y2) определяется как (B **with** (x1**:** y1)) **with** (x2**:** y2).

**Конструкции императивного программирования**. Если результат программы требуется сохранить в той же переменной, что и аргумент b, то для результата используется имя b′; при трансляции b′ будет заменено на b. Однако удобнее модификацию переменных записывать явно в стиле императивного программирования. Например, оператор b = b + 1 трактуется как b′ = b + 1. Оператор присваивания вида a[i] = x является эквивалентом a′ = a **with** (i**:** x). Вследствие замены оператора вида b′ = b пустым оператором появляется укороченный условный оператор **if** (E(x)) A(x**:** y). Допускается вызов вида G(...**:** a[i]), трактуемый как G(...**:** x); a′ = a **with** (i**:** x). Для удобства работы с деревьями введены средства модификации поддеревьев [13].

В функциональном программировании внесение в программу императивных конструкций реализуется неявно через аппарат монад. Без монад функциональные программы потеряли бы свою компактность и привлекательность. В предикатном программировании императивные конструкции определены явно. Программа с императивными конструкциями легко приводима к правильной предикатной программе.

### 2.6. Язык $P_{imp}$: императивное расширение языка P

Эффективность предикатных программ достигается применением при трансляции оптимизирующих трансформаций, преобразующих программу на императивное расширение языка P – язык $P_{imp}$. Далее программа конвертируется на язык C++.

Дополнительными конструкциями языка $P_{imp}$ являются: циклы, оператор перехода и указатели, используемые для кодирования объектов алгебраических типов – списков и деревьев. Имеется цикл общего вида **loop** {<оператор>}. Выход из цикла реализуется оператором **break**. Кроме того, допускается использование циклов **while** и **for** языка C++. Использование перечисленных конструкций в исходной предикатной программе недопустимо.

# 3. Метод предикатного программирования

Рассмотрим *задачу* построения предикатной программы, удовлетворяющей спецификации [P(x), Q(x, y)]. Набор переменных x определяет *исходные данные* задачи, причем предусловие P(x) ограничивает допустимый набор исходных данных. Набор y определяет *неизвестные* задачи. Постусловие Q(x, y) определяет *условие* задачи, связывающее исходные и неизвестные задачи. *Решением* задачи является предикатная программа H(x: y), тотально корректная относительно спецификации [P(x), Q(x, y)].

### 3.1. Решение задачи

Правильная предикатная программа H(x: y) должна удовлетворять формуле тотальной корректности (1). Поскольку для предикатных программ реализуется тождество $\mathcal{R}(H) = H$, перепишем эту формулу в следующем виде:

$$H(x: y)\ \textbf{corr}\ [P(x), Q(x, y)] \cong P(x) \Rightarrow [\forall y.\ H(x: y) \Rightarrow Q(x, y)\ ] \ \&\ \exists y.\ H(x: y) \quad (3)$$

Здесь H(x: y) **corr** [P(x), Q(x, y)] используется как обозначение формулы тотальной корректности. Далее будем независимо использовать две части данной формулы:

$$P(x)\ \&\ H(x: y) \Rightarrow Q(x, y) \quad (4)$$

$$P(x) \Rightarrow \exists y.\ H(x: y)$$

Формула (4) определяет условие частичной корректности. Формула $P(x) \Rightarrow \exists y.\ H(x: y)$ есть условие завершения или условие тотальности программы H(x: y).

Предикат H(x: y), тотальный в области истинности предусловия P(x) и удовлетворяющий формуле (4), является решением задачи. Предикат H(x: y) является результатом *абдукции*, т.е. *обратного вывода* из постусловия Q(x, y): необходимо построить предикат H(x: y) таким образом, чтобы из него можно было вывести Q(x, y).

**Лемма 5**. Допустим, программа H(x: y) тотально корректна относительно однозначной спецификации [P(x), Q(x, y)]. Тогда истинна следующая формула:

$$P(x) \Rightarrow (H(x: y) \equiv Q(x, y))\ .$$

Поскольку в случае однозначной спецификации программа H(x: y) тождественна постусловию Q(x, y), программа может быть получена из спецификации прямым выводом в дополнении к возможности получить программу посредством абдукции.

### 3.2. Формы декомпозиции предикатных программ

Оператор суперпозиции, условный и параллельный операторы определяют базисные *формы алгоритмирования* при построении программы в процессе решения задачи.

**Независимые подзадачи**. Иногда исходная задача H(x: y, z) распадается на две независимые не связанные между собой подзадачи B(x: y) и C(x: z). В этом случае программа строится в виде параллельного оператора B(x: y) || C(x: z).

**Лемма 6**. Допустим, B(x: y) **corr** [P(x), Q(x, y)] и C(x: z) **corr** [P(x), R(x, z)]. Тогда {B(x: y) || C(x: z)} **corr** [P(x), Q(x, y) & R(x, z)].

В соответствии с данной леммой, если постусловие представимо в форме конъюнкции Q(x, y) & R(x, z) тотальных предикатов Q и R, то исходная задача H(x: y, z) сводится к двум независимым подзадачам.

**Разбор случаев**. Рассматривается дополнительное логическое условие – предикат E(x). Исходная задача H(x: y) делится на две подзадачи. Первая подзадача B(x: y) – это исходная задача H(x: y) с включением дополнительного условия E(x), т.е. задача, определяемая спецификацией [P(x) & E(x), Q(x, y)]. Вторая подзадача C(x: y) определяется включением отрицания условия E(x) к исходной задаче H(x: y), т.е. задача [P(x) & ¬E(x), Q(x, y)]. В данном случае программа строится в виде условного оператора **if (**E(x)) B(x: y) **else** C(x: y).

**Лемма 7**. Пусть B(x: y) **corr** [P(x) & E(x), Q(x, y)] и
C(x: z) **corr** [P(x) & ¬E(x), Q(x, y)].
Тогда {**if (**E(x)) B(x: y) **else** C(x: y)} **corr** [P(x), Q(x, y)].

**Сведение к другой задаче**. Определяется новый объект z, связанный с исходными данными задачи H(x: y). Построение объекта z определяется в виде задачи B(x: z). Далее решается задача C(x, z: y) нахождения искомого объекта y по объектам x и z. Таким образом, решение исходной задачи H(x: y) представляется как последовательное решение задач B(x: z) и C(x, z: y). В данном случае программа строится в виде оператора суперпозиции B(x: z); C(x, z: y).

Суперпозиция является наиболее сложной формой композиции программ. Есть правила вывода для разных видов суперпозиции [99]. Здесь рассмотрим лишь один частный случай – *сведение к более общей задаче*.

**Лемма 8**. Пусть истинны: $P(x) \Rightarrow \exists z.\ B(x: z)$, $P(x) \equiv P_C(x, B(x))$ и
C(x, z: y) **corr** [$P_C$(x, z), Q(x, y)].
Тогда C(x, B(x): y) **corr** [P(x), Q(x, y)].

Задача C(x, z: y) является более общей по отношению к исходной задаче H(x: y). Объект z определяется как z = B(x) для тотального предиката B(x: z). В соответствии с данной леммой условие $P(x) \equiv P_C(x, B(x))$ гарантирует корректность следующей программы:

H(x: y) **pre** P(x) **post** Q(x, y) { C(x, B(x): y) }

Таким образом, задача H(x: y) сводится к более общей задаче C(x, z: y). В предикатном программировании сведение к более общей задаче часто применяется для приведения рекурсии к хвостовому виду.

Приведенные выше формы декомпозиции программы являются базисными. На практике используются самые разнообразные формы операторов в языках $P_2$ и $P_3$, в частности, суперпозиция общего вида B(x: z, t); C(x, z: y). Особыми являются формы декомпозиции для гиперфункций [7, 3, 6].

Далее рассмотрим методы построения предикатных программ по формуле (4)) на конкретных примерах.

### 3.3. Пример. Вычисление наибольшего общего делителя

Рассмотрим программу GCD (great common divisor), вычисляющую наибольший общий делитель положительных значений a и b. Спецификацию программы на языке P можно представить следующим образом:

GCD(**nat** a, b**: nat** c) **pre** pGCD(a, b) **post** gcd(a, b, c);

Предусловие программы определяется предикатом:

$pGCD(a, b) \cong a > 0 \ \&\ b > 0$ .

Постусловие gcd(a, b, c) определяет свойство: значение c есть *наибольший общий делитель* значений a и b. Формальное определение представлено ниже в виде цепочки из трех определений.

$$divisor(a, x) \cong \exists\ \mathbf{int}\ z > 0.\ x * z = a$$
$$divisor2(a, b, c) \cong divisor(a, c)\ \&\ divisor(b, c)$$
$$gcd(a, b, c) \cong divisor2(a, b, c)\ \&\ \forall x.\ (divisor2(a, b, x) \Rightarrow x \le c)\ .$$

Для положительных значений a и x предикат divisor(a, x) определяет свойство «x является *делителем* a». Свойство «значение c есть *общий делитель* значений a и b» определяется предикатом divisor2(a, b, c).

Формула (4), используемая для построения программы GCD, представляется в следующем виде:

$$pGCD(a, b)\ \&\ GCD(a, b\mathbf{:}\ c) \Rightarrow gcd(a, b, c) \qquad (5)$$

При этом решение ищется среди тотальных программ в области истинности предусловия pGCD(a, b).

При истинном предусловии pGCD(a, b) для предиката gcd реализуются следующие известные математические свойства:

$$gcd(a, a, a)$$
$$gcd(a, b, c) \equiv gcd(a + b, b, c)$$
$$gcd(a, b, c) \equiv gcd(b, a, c)$$

Следствием является свойство:

$$a < b \Rightarrow gcd(a, b, c) \equiv gcd(a, b - a, c)$$

Решение реализуется разбором трех взаимоисключающих случаев: a = b, a < b и a > b.

Рассмотрим случай a = b. Очевидным решением является оператор c = a. Формула корректности (5) истинна ввиду свойства gcd(a, a, a). Таким образом, получаем первый *вариант решения* задачи GCD:

$$a = b \rightarrow c = a$$

Рассмотрим случай a < b. Истинно тождество: gcd(a, b, c) ≡ gcd(a, b - a, c). Это дает возможность свести исходную задачу GCD(a, b**:** c) к GCD(a, b - a**:** c). В итоге получаем второй вариант решения:

$$a < b \rightarrow GCD(a, b - a\mathbf{:}\ c)$$

Формула (5) для второго варианта переписывается следующим образом:

$$pGCD(a, b)\ \&\ a < b\ \&\ GCD(a, b - a\mathbf{:}\ c) \Rightarrow gcd(a, b, c)\ . \qquad (6)$$

Предикат GCD(a, b - a**:** c) «ближе» к решению по сравнению с GCD(a, b**:** c). Чтобы установить это, введем *меру* – натуральную оценочную функцию, определенную на аргументах: m(a, b) = a + b. Аргументы GCD(a, b - a**:** c) строго убывают по мере m, поскольку m(a, b - a) < m(a, b). Для доказательства формулы (6) будем использовать индукцию по мере m. В соответствии с индукционным предположением истинна формула

$$pGCD(a, b - a)\ \&\ GCD(a, b - a\mathbf{:}\ c) \Rightarrow gcd(a, b - a, c)$$

Поскольку посылки формулы истинны, то истинна формула gcd(a, b - a, c). Из ее истинности следует истинность gcd(a, b, c), а далее – истинность формулы (6).

Аналогичным образом определяется третий вариант решения:

$$a > b \rightarrow GCD(a - b, b\mathbf{:}\ c)$$

Объединение трех вариантов решения дает следующую программу:

```
GCD(nat a, b:  nat c)
pre pGCD(a, b) post gcd(a, b, c) measure a + b
{       if (a = b) c = a
        else   if (a < b) GCD(a, b – a: c)
               else GCD(a – b, b: c)
};
```

Для доказательства тотальности программы достаточно доказать тотальность двух рекурсивных вызовов программы. Для этой цели используется индукция по мере m.

### 3.4. Пример. Обмен элементов массива

Рассмотрим программу обмена элементов массива. Имеется массив чисел $a_1, a_2, \ldots, a_n$, причем $a_1=0$. Программа Swap обменивает нулевой элемент $a_1$ с любым другим ненулевым элементом, если такой существует. Данная программа является частью программы решения системы линейных уравнений по методу Гаусса-Жордана.

Спецификация программы представлена следующими определениями:

Swap(a: a′) **pre** pSwap(a) **post** qSwap(a, a′);

pSwap(a) ≅ n > 0 & a[1] = 0;

qSwap(a, a′) ≅

∃ j=1..n. (a[j] ≠ 0 & a′ = a **with** (1: a[j], j: 0)) ∨ a′ = a & ∀ j=1..n. a[j] = 0;

Здесь a – исходный массив, a′ – результирующий. Штрих в имени массива означает, что в реализации программы a и a′ – один и тот же массив, т.е. переменная a′ будет заменена на a при трансляции программы. Переменная n является константой для программы, причем n > 0. Отношение a′ = a **with** (1: a[j], j: 0) определяет, что массив a′ получается из a заменой первого элемента на a[j], а также заменой j–го элемента нулем. Постусловие qSwap(a, a′) постулирует, что либо массив a полностью нулевой и тогда a′ = a, либо имеется ненулевой элемент, который обменивается с первым элементом.

Отметим, что спецификация неоднозначна.

Данная программа считается тривиальной в императивном программировании и реализуется перебором в цикле элементов массива. В функциональном программировании программа не является простой, поскольку массивы – объекты логики второго порядка.

В предикатном программировании применяется метод обобщения исходной задачи. Вводится новый объект p – индекс очередного просматриваемого элемента массива a. Рассматривается более общая задача Swap1, в которой предполагается, что первые p элементов массива a – нулевые, и требуется обменять первый элемент с ненулевым элементом массива a. Задача Swap1 определяется следующей спецификацией:

Swap1(p, a: a′) **pre** pSwap1(p ,a) **post** qSwap(a, a′);

pSwap1(p, a) ≅ n > 0 & p > 0 & p ≤ n & ∀ j=1..p. a[j] = 0;

Здесь постусловие то же самое, что и для задачи Swap. Предусловие определяет, что первые p элементов массива a нулевые. Поскольку pSwap(a) ≡ pSwap1(1, a), то в соответствии с Леммой 8 задача Swap сводится к более общей задаче Swap1:

```
nat n;
type Arn = array(real, 1..n);
Swap(Arn a: Arn a′) pre pSwap(a) post qSwap(a, a′)
{ Swap1(1, a: a′) };
```

Формула (4), используемая далее для построения программы Swap1, конкретизируется в следующем виде:

$$pSwap1(p, a) \,\&\, Swap1(p, a{:}\ a') \Rightarrow qSwap(a, a') . \qquad (7)$$

Построение программы Swap1 реализуется разбором случаев $p = n$ и $p < n$, учитывая, что p меняется от 1 до n.

Пусть $p = n$. В этом случае массив a полностью нулевой, и единственно возможным решением, при котором истинно qSwap(a, a′), является оператор присваивания $a' = a$. Таким образом, получаем вариант решения:

$$p = n \rightarrow a' = a .$$

Рассмотрим случай $p < n$. Далее проводится разбор случаев $a[p+1] = 0$ и $a[p+1] \neq 0$.

Пусть $a[p+1] \neq 0$. Элемент $a[p+1]$ можно обменять с $a[1]$. Получаем другой вариант решения:

$$p < n \,\&\, a[p+1] \neq 0 \rightarrow a' = a \textbf{ with } (1{:}\ a[p+1], p+1{:}\ 0) .$$

Рассмотрим другой случай: $p < n \,\&\, a[p+1] = 0$. Здесь становится истинным pSwap1(p+1, a), что позволяет свести исходную задачу к Swap1(p+1, a: a′), т.е. получаем третий вариант решения:

$$p < n \,\&\, a[p+1] = 0 \rightarrow Swap1(p+1, a{:}\ a') .$$

Формула (7) для данного случая переписывается следующим образом:

$$pSwap1(p, a) \,\&\, p < n \,\&\, a[p+1] = 0 \,\&\, Swap1(p+1, a{:}\ a') \Rightarrow qSwap(a, a') . \qquad (8)$$

**Доказательство** формулы (8) проводится по индукции от $p = n$ до $p = 1$. Предположим, что формула (7) доказана для $p+1$ и докажем ее для p. Пусть истинны посылки формулы (8). Докажем истинность qSwap(a, a′). Используем индукционное предположение:

$$pSwap1(p+1, a) \,\&\, Swap1(p+1, a{:}\ a') \Rightarrow qSwap(a, a')$$

Из истинности pSwap1(p, a) и $a[p+1] = 0$ получаем истинность pSwap1(p+1, a). Поскольку в индукционном предположении истинны посылки, то истинно и заключение qSwap(a, a′). □

Объединение трех вариантов решения дает программу:

```
Swap1(nat p, Arn a: Arn a′)
pre pSwap1(p ,a) post qSwap(a, a′) measure n - p
{       if (p = n) a′ = a
        else  if (a[p+1] = 0) Swap1(p+1, a: a′)
              else a′ = a with (1: a[p+1], p+1: 0)
};
```

Тотальность программы Swap1 достаточно подтвердить для рекурсивного вызова Swap1(p+1, a: a′). Для этой цели используется функция меры $m(p) = n - p$. Для рекурсивного вызова реализуется $m(p+1) < m(p)$, что обеспечивает доказательство тотальности Swap1.

Отметим тривиальность программы Swap1: операторы программы фактически извлекаются из формулы для постусловия qSwap применением элементарных рассуждений.

### 3.5. Стили предикатного программирования

Метод предикатного программирования, иллюстрированный на примерах программ в разд. 3.3-3.4, является универсально применимым для всех программ из класса программ-функций. Построение программы реализуется обратным логическим выводом (абдукцией) на базе формулы (4) с использованием математических свойств, определяемых спецификацией программы. Общие методы построения программы на базе формулы (4) для разных видов операторов базируются на Леммах 6-8. В дополнении к этому применяется универсальный метод доказательства по индукции; использование меры позволяет покрыть почти все используемые при доказательствах схемы индукции.

Отметим, что Леммы 6-8 являются аналогами некоторых правил вывода при доказательстве тотальной корректности предикатной программы [!!!]. В этом наборе пока нет методов построения программы в форме оператора суперпозиции.

Описываемый в данной работе метод предикатного программирования определяет возможность реализация новой версии экспериментальной системы предикатного программирования, в которой обычный стиль предикатного программирования интегрирован с *программным синтезом* на базе формулы (4) небольших фрагментов создаваемой предикатной программы и дедуктивной верификацией для модифицируемых фрагментов предикатной программы.

В обычном стиле предикатного программирования, представленном в публикациях [3–6, 12–16], программист самостоятельно конструирует программу на основе свойств, вытекающих из спецификации, без проведения формального доказательства ее корректности. Спецификация программы не всегда определена формально. Математические рассуждения при построении программы, как правило, неполны и реализуются на содержательном интуитивном уровне. Сами математические доказательства в большинстве случаев не проводятся – достаточно уверенности, что такие доказательства в принципе реализуемы. При этом степень уверенности может быть разной. Перечисленные особенности характерны и для императивного программирования. Тем не менее, по сравнению с аналогичной императивной программой предикатная программа «ближе» к спецификации и свойствам, на основе которых она построена. Процесс построения предикатной программы лучше контролируется, и она более понятна.

Программа, полученная по формуле (4), является корректной. Разумеется, при условии, что истинность формулы для конкретной программы является достоверной. Ошибка в доказательстве может стать причиной ошибки в программе. В этом смысле традиционный стиль математического доказательства, в частности, представленный в разд. 3.3 и 3.4, не является надежным. Надежность доказательства гарантируется лишь при его проведении в системе автоматического (компьютерного) доказательства, применяемого в рамках системы дедуктивной верификации или программного синтеза.

В текущей версии экспериментальной системе предикатного программирования реализована подсистема дедуктивной верификации. Построена универсальная система правил доказательства тотальной корректности предикатных программ. Разработан генератор формул корректности программы с выходом на систему интерактивного

доказательства PVS [17] и SMT-решатель CVC3. Данный метод опробован для дедуктивной верификации более чем 50 небольших программ [3–6].

# 4. Оптимизирующие трансформации предикатных программ

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций, переводящих программу на императивное расширение языка P:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков и деревьев) с использованием массивов и указателей.

После трансформаций программа конвертируется на язык C++. В дополнение к базисным трансформациям используются трансформации: упрощения и оформления программы, а также простые эквивалентные преобразования.

Перечисленные трансформации реализуют *оптимизацию среднего уровня* с переводом предикатной программы в эффективную императивную программу. Построение алгоритмов такой оптимизации – новая задача, характерная для предикатного, но не функционального программирования. Оптимизация среднего уровня принципиально отличается от традиционной оптимизации для императивных языков. Трансформации склеивания переменных и кодирования объектов алгебраических типов аналогов не имеют.

Фактически, набор применяемых трансформаций планируется при построении программы. И задача трансформатора – «угадать», найти этот набор по программе. Иначе говоря, ставится задача разработки алгоритмов трансформаций не для произвольной программы, а для программы, ориентированной на трансформацию. Реализация трансформаций в частности, использует информацию о времени жизни переменных, поставляемую потоковым анализатором предикатной программы.

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [3–6]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [8]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, совершаемых программистом вручную на стадии построения предикатной программы.

### 4.1. Определение трансформаций

**Подстановка тела программы на место ее вызова**. Пусть имеется предикатная программа B(x: y) { K } и вызов предиката B(g: z) внутри предикатной программы H. Здесь x, y, z обозначают списки переменных, g – список выражений, тело программы K – оператор на императивном расширении языка P. В общем случае *подстановка тела программы* K *на место вызова* B(g: z) есть замена вызова следующей композицией:

| x | = | g |; { K }; | z | = | y | .

Конструкции | x |, | z | и | y | являются мультипеременными, а | g | – мультивыражением. Переменные наборов x и y становятся локальными переменными предикатной программы H, в которой находился вызов B(g: z). В общем случае проведению подстановки предшествует предварительное систематическое переименование переменных внутри K во избежание коллизий с именами переменных программы H.

Оператор присваивания вида | x | = | g | называется *групповым оператором присваивания*. В соответствии с операционной семантикой (см. разд. 2.4) вычисление выражений, входящих в набор g, проводится параллельно. Эффективная реализация такого параллелизма возможна на процессорах с архитектурой широких команд. На других архитектурах параллельная реализация неэффективна, и поэтому проводится *раскрытие* группового оператора присваивания его заменой набором обычных операторов присваивания. В общем случае раскрытие группового оператора возможно при использовании дополнительных промежуточных переменных. Например, раскрытие оператора |a ,b| = |b, a| реализуется операторами t = a; a = b; b = t с использованием дополнительной переменной t. В большинстве случаев раскрытие возможно без использования дополнительных переменных; иногда достаточно поменять местами операторы присваивания. Например, раскрытие |a, b| = |c, a| реализуется присваиваниями: b = a; a = c.

Оператор | z | = | y | можно устранить, если провести замену в операторе K всех переменных из списка y на соответствующие переменные списка z. Допустим, переменная a встречается в списке выражений g и в списке x ей соответствует переменная b. В большинстве случаев переменную b можно будет заменить на переменную a в операторе K, удалив a из списка g и b из списка x. Замена корректна за исключением случая, когда b модифицируется внутри K (что возможно при склеивании b с другими переменными), а переменная a используется в программе H после вызова B(g: z). Еще одно ограничение: если переменная a встретилась в списке g повторно и для первого вхождения a была проведена замена, то замена для второго вхождения a недопустима.

Стратегия именования переменных предикатной программы должна быть такой, чтобы обеспечить минимальное изменение оператора K при его подстановке на место вызова B(g: z). С этой целью полезно использовать те же самые имена для заменяемых переменных.

**Замена хвостовой рекурсии циклом**. Рекурсивный вызов предиката определяет *хвостовую рекурсию*, если:

- имя вызываемого предиката совпадает с именем предикатной программы, в теле которой находится вызов;

- вызов является последней исполняемой конструкцией в программе, содержащей вызов.

Пусть last(K) обозначает множество последних исполняемых конструкций в операторе K. Тогда: last(**if** (E) B **else** C) = last(B) ∪ last(C), last(B; C) = last(C), last(B || C) = ∅. Вызов функции, за которой исполняется операция, отличная от присваивания не является хвостовым.

Допустим, имеется рекурсивная программа B(x: y) { K } и вызов B(g**:** y) с хвостовой рекурсией внутри оператора K. Подставим тело программы B на место вызова B(g**:** y). Результатом подстановки является композиция: | x | = | g |; { K }. Обозначим через K′ оператор, полученный заменой в K вызова B(g**:** y). Очевидно, можно заменить в K′ второе вхождение K передачей управления на начало оператора K′. В итоге предикатная программа B преобразуется к виду: B(x: y) {M: K″ }, где K″ получается из K заменой вызова B(g**:** y) парой операторов: | x | = | g |; **goto** M. Данное преобразование есть трансформация *замены хвостовой рекурсии циклом*.

Если в рекурсивном определении предиката B все рекурсивные вызовы имеют вид хвостовой рекурсии, то применение трансформации ко всем этим вызовам преобразует тело предиката в цикл, а рекурсивную программу B – в нерекурсивную. Предикатную программу B будем представлять в виде B(x: y) {**loop** { K‴ }}, где K‴ получается из K заменой всякого хвостового вызова B(g**:** y) оператором | x | = | g |, а в конце каждой ветви оператора K, не завершающейся рекурсивным вызовом, вставляется оператор выхода из цикла **break**.

После данной трансформации становится эффективной постановка тела программы B на место вызова B в другой программе.

**Склеивание переменных**. Трансформация *склеивания переменных* a ← x есть замена в предикатной программе всех вхождений каждой переменной из списка переменных x на переменную a. Например, склеивание переменных a ← b, c представляет замену всех вхождений в программе имен b и c на имя a.

Склеиванию подлежат результаты программы с аргументами или локальные переменные с результатами, между которыми имеется информационная связь. Задача склеивания переменных не актуальна для оптимизации функциональных программ. Эта задача не возникает также для императивных программ, поскольку практически все рассматриваемые здесь склеивания обычно реализуются программистом в императивной программе.

Склеивание переменных может задаваться в предикатной программе. Если имеется результирующая переменная, имя которой завершается штрихом, при наличии аргумента с тем же именем, то результирующая переменная должна быть склеена с аргументом. Например, результат a′ при наличии аргумента a неявно определяет трансформацию a ← a′. Если аргумент определяет начальное значение результата и должен быть с ним склеен, то рекомендуется именование вида: b – результат, b0 – аргумент. Это облегчит определение трансформации b ← b0.

Эквивалентность программы до и после проведения трансформации склеивания достигается при выполнении ряда условий. Одно из них: аргумент a, склеенный с результатом b, не может использоваться после присваивания переменной b.

Склеивание переменных рассматривалось ранее в рамках задачи экономии памяти при трансляции программ 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 [9]. Подобная задача возникает при оптимизации регистровой памяти.

**Кодирование алгебраических типов**. В императивном расширении языка P нет алгебраических типов. Трансформация *кодирования алгебраических типов* (списков, деревьев и других) реализует их представление посредством структур более низкого уровня, таких как массивы и указатели. Операции с объектами алгебраических типов кодируются с учетом выбранного представления типов.

В основном способе кодирования всех алгебраических типов, за исключением строк, используются указатели, которыми снабжаются элементы для связи с элементами-потомками. Имеется другой более эффективный способ представления списков: список кодируется в виде вырезки массива. Строковые объекты кодируются начальной вырезкой массива с нулевым элементом в качестве завершителя. Для списков и строк введены средства их сканирования, аналогичные итераторам в императивных языках; имеется возможность определить объем памяти для них [12].

Реализация операции, результатом которой является объект алгебраического типа, в общем случае требует отведения новой памяти для создаваемого объекта. Во многих случаях удается избежать отведения новой памяти, используя значения других объектов или реализуя данную операцию в составе последующей операции присваивания. Корректность такой реализации обеспечивается при выполнении определенных условий, подтверждаемых с использованием информации, получаемой потоковым анализом программы.

Особенности трансформации предикатных программ покажем на примерах программ.

### 4.2. Трансформация программы вычисления наибольшего общего делителя

Рассмотрим программу вычисления наибольшего общего делителя (см. разд. 3.1):

```
GCD(nat a, b:  nat c)
pre a > 0 & b > 0 post gcd(a, b, c) measure a + b
{       if (a = b) c = a
        else   if (a < b) GCD(a, b – a: c)
               else GCD(a – b, b: c)
};
```

Каждый из двух рекурсивных вызовов программы GCD имеет хвостовой вид. Результатом применения трансформации замены хвостовой рекурсии циклом является следующая программа.

```
GCD(nat a, b: nat c) {
M:    if (a = b) c = a
        else   if (a < b) {|a, b| = |a, b - a|; goto M}
        else  {|a, b| = |a - b, b|; goto M}
}
```

Раскроем групповые операторы присваивания, а также заменим фрагмент с операторами перехода на цикл **loop**. Получим итоговую программу:

```
GCD(nat a, b: nat c) {
      loop {
        if (a = b) {c = a; break; };
        if (a < b) b = b – a
        else a = a – b
      }
}
```

Замена на цикл **loop** является трансформацией оформления. Она не оптимизирует программу. Ее цель – привести программу к виду, более удобному для восприятия. Другим примером применения трансформации оформления является программа:

```
GCD(nat a, b: nat c) {
      while (a != b) { if (a < b) b = b – a else a = a – b };
      c = a
}
```

### 4.3. Трансформация программы обмена элементов массива

Программа обмена элементов массива состоит из следующих двух предикатных программ (разд. 3.2):

```
Swap(a: a') pre pSwap(a) post qSwap(a, a')
{ Swap1(1, a: a') };

Swap1(p, a: a')
pre pSwap1(p ,a) post qSwap(a, a') measure n – p
{           if (p = n) a' = a
            else   if (a[p+1] = 0) Swap1(p+1, a: a')
                   else a' = a with (1: a[p+1], p+1: 0)
};
```

Начальной трансформацией, применяемой к двум программам, является трансформация склеивания $a \leftarrow a'$, реализующая замену всех вхождений переменной $a'$ на $a$. Получим:

```
Swap(a: a) { Swap1(1, a: a) };
Swap1(p, a: a)
{           if (p = n) a = a
            else   if (a[p+1] = 0) Swap1(p+1, a: a)
                   else a = a with (1: a[p+1], p+1: 0)
};
```

В программе Swap1 применим трансформацию замены хвостовой рекурсии циклом с раскрытием группового оператора присваивания:

```
Swap1(p, a:  a)
{ loop {
        if (p=n) { a = a; break;}
        else   if (a[p+1] = 0) p = p+1
               else {a = a with (1: a[p+1], p+1: 0); break;}
  }
};
```

Применяется трансформация упрощения, заменяющая оператор a = a пустым оператором. Далее условие p=n втягивается в заголовок цикла **while**.

```
Swap1(p, a:  a)
{while (p!=n) {
        if (a[p+1] = 0) p = p+1
        else {a = a with (1: a[p+1], p+1: 0) ; break;}
  }
};
```

Следующим действием является вынесение оператора p = p+1 перед условным оператором. При этом модифицируется условный оператор – это трансформация оформления. Раскрытие операции **with** в операторе a = a **with** (1: a[p], p: 0) реализуется бесконфликтно.

```
Swap1(p, a:  a)
{while (p!=n) { p = p+1; if (a[p]!=0) { a[1] = a[p]; a[p] = 0; break } } };
```

Отметим, что вынесение оператора p = p+1 можно было бы реализовать при построении предикатной программы следующим образом:

```
   Swap1(p, a: a')
   {    if (p = n) a' = a
        else   {  nat p1 = p+1;
                    if (a[p1] = 0) Swap1(p1, a: a') else a' = a with (1: a[p1], p1: 0)
                }
   };
```

Такое рекомендуется в случаях, когда возможно дальнейшее улучшение программы.

Поскольку программа Swap1 более не рекурсивна, подставим ее внутрь программы Swap.

```
Swap(a:  a)
{ nat p; | p, a | = | 1, a|;
  while (p!=n) { p = p+1; if (a[p]!=0) { a[1] = a[p]; a[p] = 0; break } }
}
```

Раскроем групповой оператор присваивания и втянем оператор p = 1 в заголовок цикла. Получим итоговую программу.

```
Swap(a:  a)
{ for(nat p = 1; p!= n; ) { p = p+1; if (a[p]!=0) { a[1] = a[p]; a[p] = 0; break } } }
```

### 4.4. Трансформация программы суммирования элементов списка

Рассмотрим программу sum(s: c) суммирования элементов списка s. Сумма элементов списка s определяется следующей формулой:

SUM(s) = s = nil ? 0 : s.car + SUM(s.cdr) .

Простейшая программа суммирования является реализацией данной формулы.

```
 type listR = list (real);
 sum(listR s: real c) post c = SUM(s)
        { if (s = nil ) c = 0 else c = s.car + sum(s.cdr) };
```

В данной программе рекурсия не является хвостовой, так как после вызова sum реализуется операция сложения. Такая программа неэффективна. Чтобы привести рекурсию к хвостовому виду, применяется метод обобщения исходной задачи.

Введем накопитель d. Рассмотрим обобщенную задачу суммирования sumG, в которой вычисляется значение SUM(s) + d, со следующей спецификацией:

sumG(listR s, **real** d**: real** c) **post** c = SUM(s) + d

Истинно тождество: sum(s**:** c) ≡ sumG(s, 0**:** c). Новая программа суммирования представлена ниже.

```
sum(listR s: real c) post c = SUM(s)
        { sumG(s, 0: c) };
sumG(listR s, real d: real c) post c = SUM(s) + d
        { if (s = nil ) c = d else sumG(s.cdr, d + s.car : c) }
```

Хвостовая рекурсия в программе sumG дает возможность провести следующую серию оптимизирующих трансформаций. Проведем склеивание c ← d.

```
sumG(listR s, real c: real c) { if (s = nil ) c = c else sumG(s.cdr, c + s.car : c) }
```

Заменим хвостовую рекурсию циклом.

```
sumG(listR s, real c: real c)
{ loop { if (s = nil ) {c = c; break; } else | s, c | = |s.cdr, c + s.car | } }
```

Заменим c = c пустым оператором и втянем условие s = nil в заголовок цикла. При раскрытии группового оператора присваивания необходимо будет поменять порядок присваивания параметрам s и c.

```
sumG(listR s, real c: real c) { while (s! = nil) { c = c + s.car; s = s.cdr }
```

Подставим данную программу на место вызова в программу sum и раскроем групповой оператор.

```
sum(listR s: real c) { c = 0; while (s != nil) { c = c + s.car; s = s.cdr } }        (8)
```

Программа (8) использует три операции со списками. Непосредственная реализация оператора s = s.cdr приводит к копированию большей части списка. Такая программа неэффективна. Для программ со списками применяется два вида трансформаций: кодирование списка вырезкой массива и кодирование списка через указатели.

**Кодирование списка вырезкой массива.** Список s представляется вырезкой некоторого массива S достаточной длины для размещения всех списков, являющихся значением переменной s. В программе (8) переменная s модифицируется, поэтому следует определить ее начальное и текущее значение. Пусть S[m..n] – начальное значение, а S[j..n] – текущее. Массив S и величины j, m, n должны быть определены в программе, причем j – переменная, а m и n могут быть константами.

```
type BUF = array (real, M..N);
BUF S;
int j = m;
```

Значения границ M и N должны быть достаточными, т. е. M ≤ m, n, j ≤ N. Правила кодирования операций со списками, используемых в программе (8), представляются следующим образом:

| | | |
|---|---|---|
| s! = nil | → | j <= n |
| s.car | → | S[j] |
| s = s.cdr | → | j = j + 1 |

Применение правил к программе (8) дает программу:

```
sum(int m, n: real c) { int j = m; c = 0; while (j <= n) { c = c + S[j]; j = j + 1 } }
```

Замена цикла **while** на **for** приводит к итоговой программе:

```
sum(int m, n: real c) { c = 0; for (int j = m; j <= n; j = j + 1) c = c + S[j] }
```

**Кодирование списка односвязным списком с использованием указателей.** Каждый элемент списка снабжается указателем на следующий элемент. Элемент списка представляется структурой из двух полей: первое поле – значение элемента, второе – указатель на следующий элемент. Список s представляется указателем S на первый элемент. Пустой список кодируется нулевым указателем NULL.

```
type ListP = struct (real car, ListP *cdr);
ListP *S;
```

Для операций со списками, используемых в программе (8), применяются следующие правила кодирования:

| | | |
|---|---|---|
| s! = nil | → | S != NULL |
| s.car | → | S -> car |
| s = s.cdr | → | S = S -> cdr |

Применение правил к программе (8) дает программу:

```
sum(ListP *S: real c) { c = 0; while (S != NULL) { c = c + S -> car; S = S -> cdr } }
```

Технология предикатного программирования позволяет воспроизвести любую реализацию в императивном программировании.

## Заключение

## Литература


1. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. — Новосибирск, 2015. — 13с. http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
2. Manna Z., Waldinger R. A deductive approach to program synthesis // ACM Transactions on Programming Languages and Systems, 2. – 1980. – P.90–121.
3. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. — Новосибирск, 2004. — 52с. — (Препр. / ИСИ СО РАН; N 115).
4. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements. *Automatic Control and Computer Sciences*. Vol. 45, No. 7, 421–427.
5. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. — С. 14-21.
6. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164 ).
7. Шелехов В.И. Язык и технология автоматного программирования // «Программная инженерия», №4, 2014. – С. 3-15. http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf
8. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design // Computer, Vol. 42, no. 9. – 2009. – P.56-63.
9. Ершов А. П. Введение в теоретическое программирование. М.: Наука, 1977. 288 с.
10. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2010. 42с. (Препр. / ИСИ СО РАН; N 153). http://persons.iis.nsk.su/files/persons/pages/plang12.pdf

11. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная информатика, №4, 2014. — ИСИ СО РАН, Новосибирск. — С. 1-29. http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf
12. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, №3, 2014. — ИСИ СО РАН, Новосибирск. — С. 25-43. http://persons.iis.nsk.su/files/persons/pages/String1.pdf
13. Шелехов В.И. Предикатная программа вставки в АВЛ-дерево. — Новосибирск, 2015. — 22с. —. http://persons.iis.nsk.su/files/persons/pages/avl_insert.pdf
14. Методы предикатного программирования / Под ред. Шелехова В.И. Вып.1. ИСИ СО РАН. Новосибирск, 2003. 62 С.
15. Методы предикатного программирования / Под ред. Шелехова В.И. Вып.2. ИСИ СО РАН. Новосибирск, 2006. 116 С.
16. В.А. Вшивков, Т.В. Маркелова, В.И. Шелехов. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т.4(33), с. 79-94, 2008.
17. PVS Specification and Verification System. SRI International. http://pvs.csl.sri.com/
--18. Kneuss E., Kuncak V., Kuraj I., Suter P. On Integrating Deductive Synthesis and Verification Systems // EPFL Report 186043, 2013.

==========

Shelekhov V. Validation of Rules for Deductive Verification of Predicate Programs // Verification and Assurance (VeriSure 2013), Workshop associated with 25th International Conference on Computer-Aided Verification (CAV-2013). July 14, May 2013. – Saint Petersburg, Russia, 2013. – 7p.
10.
11. Шелехов В.И. Предикатное программирование. Учебное пособие. Новосибирск: НГУ, 2009. 109с.

4. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. — Новосибирск, 2012. — 6с. — (Препр. / ИСИ СО РАН; N 167).

3. Чушкин М. С., Шелехов В.И. Генерация и доказательство формул корректности предикатных программ. — Новосибирск, 2012. — 34с. — (Препр. / ИСИ СО РАН; N 166).

4. Чушкин М.С. Система дедуктивной верификации предикатных программ // Тр. 2-й межд. конф. «Инструменты и методы анализа программ». – Кострома, Костромской государственный технологический университет. — 14-15 ноября 2014г. — С. 205-214.

7. Hehner E.C.R. Predicative programming, parts I and II // Communications of the ACM 27(2). – 1984. – P. 134-151.

1. *Floyd R. W.* Assigning meanings to programs // Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science. AMS, 1967. P. 19–32.

2. *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.

13. *Hoare C. A. R.* Communicating Sequential Processes. Prentice-Hall. 1985.

17. *Backus J.* Can programming be liberated from the von Neumann style? A. Functional Style and Its Algebra of Programs // Communications of the ACM. 1978. Vol. 21 (8). P. 613–641.

В обзор. Такой метод соответствует стилю программного синтеза [2], в котором программа извлекается посредством унификации из конструктивного доказательства теоремы существования: $\forall x.\ P(x) \Rightarrow \exists y.\ Q(x, y)$. В нашем случае извлекаемая программа составляет часть доказанной формулы тотальной корректности (4), поэтому синтез предикатной программы является явным и более простым.