УДК 004.415.52

Доказательное построение, верификация и синтез предикатных программ

В.И. Шелехов

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

Предикатная программа – вычислимый предикат. В предикатном программировании исповедуется стиль доказательного программирования: программа является математическим объектом и строится применением математических утверждений с использованием свойств исходной задачи, представленной спецификацией программы. Язык предикатного программирования P построен последовательным расширением, начиная с ядра P_0, для которого определена формальная операционная семантика. На базе формулы тотальной корректности предикатных программ разработаны методы дедуктивной верификации и программного синтеза, гарантирующие корректность предикатной программы относительно спецификации. Показано, что известные методы декомпозиции программы в процессе ее построения следуют из формулы тотальной корректности. Определяется набор базисных трансформаций, преобразующих предикатную программу в эффективную программу на языке C++.

Ключевые слова: формальная операционная семантика, дедуктивная верификация, программный синтез, доказательное программирование, онтология

1. Введение

В предикатном программировании программа – это вычислимый предикат или вычислимая логическая формула. Предикат всегда можно определить через функцию, а функцию – через предикат. Вроде не должно быть принципиальных различий с функциональным программированием: там программа является вычислимой функцией. Тем не менее, синтаксически разница ощутимая. Функциональный язык – это язык функций и выражений, а предикатный язык является языком операторов. Однако главное различие в следующем. В предикатном программировании программист сам оптимизирует создаваемую предикатную программу, чтобы применением оптимизирующих трансформаций получить из нее эффективную императивную программу. Иной менталитет в функциональном программировании. Программисту следует в элегантном стиле, просто и компактно написать функциональную программу, а получить из нее эффективный код – это задача транслятора.

В настоящей работе определяется процесс построения языка предикатного программирования P [10] из его ядра – простого языка P_0 [1]. Для языка P_0 построена формальная операционная семантика и доказана базисная теорема о тождестве любого оператора, рассматриваемого как логическая формула, с его операционной семантикой. Язык P строится из языка P_0 в виде цепочки расширяемых языков с сохранением тождества программы и ее операционной семантики.

Определяется формула тотальной корректности предикатной программы относительно спецификации в виде предусловия и постусловия. Разработан метод дедуктивной верификации предикатных программ. Реализована подсистема генерации формул корректности предикатных программ для системы автоматического доказательства PVS [17]. Разрабатываются методы синтеза предикатных программ.

На базе формулы корректности представлен метод доказательного программирования, при котором создаваемая предикатная программа становится корректной по построению.

Описывается набор оптимизирующих трансформаций, преобразующий предикатную программу в эффективную императивную программу на языке C++.

2. Построение языка предикатного программирования P

Язык предикатного программирования P [10] является языком доказательного программирования. *Предикатная программа* есть предикат (логическая формула) в форме вычислимого оператора. Множество предикатных программ есть вычислимое подмножество языка исчисления предикатов. Для предикатной программы используется обозначение H(x**:** y), где H – имя программы, x – возможно пустой набор аргументов, y – набор результатов вычисления программы H.

Минимальный полный *базис предикатных программ* определен в виде языка P_0 [1]. Для него построена формальная операционная семантика и доказана базисная теорема о тождестве любого оператора, рассматриваемого как логическая формула, с его операционной семантикой. Наша цель – расширить язык P_0 до удобного языка предикатного программирования P. Последовательным расширением определяются языки P_1, P_2, P_3 и P. Оптимизирующие трансформации переводят программу на язык императивного расширения P_{imp}, с которого программа легко конвертируется в язык C++.

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.

Таблица 1. Вычислимые предикаты и их программная форма

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)} }

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 не фиксируется и в принципе может быть произвольным.

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

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 $\mathcal{R}$(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) || ≅ $\mathcal{R}$(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, ( )).

Формальная операционная семантика определена в работе [1]. Предикатная программа может быть рекурсивной. Допускается рекурсия относительно вызовов предикатов B и C в операторах суперпозиции, условном и параллельном. Для условного оператора **if** (e) B(x: y) **else** C(x: y) вхождение переменной e не может быть источником рекурсии. Не допускается сложная форма рекурсии, опосредованная через подстановку предиката C аргументом вызова B(...C...: y) в случае, когда C зависит от B. В частности, не допускается рекурсия через оператор каррирования.

Теорема 3 [1]. Для произвольной предикатной программы H(x: y) на языке P_0 истинно тождество $\mathcal{R}$(H) = H. Предполагается, что данное тождество истинно для всех базисных предикатов.

Здесь $\mathcal{R}$(H) определяется для программы предиката H, а вхождение H справа понимается в смысле предиката, определенного в левой колонке Таблицы 1. Теорема дает право использовать единое обозначение H(x: y) для программы и соответствующего предиката (логической формулы).

Теорема 3 справедлива также для неоднозначных программ при дополнительном условии. Для оператора суперпозиции B(x: z); C(z: y) в случае неоднозначного предиката B(x: z) необходимо дополнительное условие корректности ∀z. (B(x: z) ⇒ ∃u. C(z: u)), гарантирующее истинность всех лемм и теорем формальной операционной семантики [1]. Например, в случае, когда в качестве B(x: z) используется датчик случайных чисел. Отметим, что в соответствии с Теоремой 4 [1] гарантирована однозначность предикатных программ при условии, что все базисные предикаты являются однозначными.

2.2. Язык P_1: 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:

$$\|\{\ K(t{:}\ z)\ \}\| \cong \|K(t{:}\ z)\| .$$

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 $H^{\sim}$ реализуется тождество $\mathcal{R}(H^{\sim}) = H^{\sim}$.

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; 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)|| ≡ ∃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 стандартным арифметическим 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.

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

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. Модуль содержит набор предикатных программ и определяет полную программу или ее часть. Модуль также содержит описания глобальных переменных, типов и классов. *Глобальная переменная*, описанная в модуле, может встречаться в предикатных программах модуля в качестве аргумента или результата без упоминания в списках аргументов и результатов предикатных программ. *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 предикатной программы>(<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. 4.1). *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** [12], 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. Гиперфункции

Гиперфункция – программа с несколькими *ветвями* результатов. Гиперфункция B(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Ветви *вызова гиперфункции* выходят в разные места программы, содержащей вызов. Вызов гиперфункции записывается в виде B(x**:** y #M1**:** z #M2). Здесь M1 и M2 – *метки* программы; *операторы перехода* #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2. Вызов гиперфункции может комбинироваться с операторами обработки ветвей:

B(x**:** y #M1**:** z #M2) **case** M1: C(y**:** u) **case** M2: D(z**:** u) .

Вызов вида B(x**:** y #M1**:** z #M2); M1**: ...** может быть представлен оператором B(x**:** y**:** z #M2).

Формально гиперфункция определяется через предикатную программу следующего вида:

```
B(x: y, z, e)
pre P(x) post e = E(x) & (E(x) ⇒ S(x, y)) & (¬E(x) ⇒ R(x, z))
{ ... };
```

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), E(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида e = **true** и e = **false** – последние исполняемые операторы в теле предиката. Программа B может быть заменена следующей программой в виде *гиперфункции*:

```
B(x: y #1: z #2)
pre P(x)   pre 1: E(x) post 1: S(x, y) post 2: R(x, z)
{ ... };
```

В теле гиперфункции каждое присваивание e = **true** заменено оператором перехода #1, а e = **false** – на #2. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции.

Спецификация гиперфункции состоит из двух частей. Утверждение после "**pre** 1" есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после "**post** 1" и "**post** 2" есть постусловия для первой и второй ветвей, соответственно.

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Традиционные подходы в реализации обработки аварийных ситуаций предполагают заведение дополнительных структур, усложняющих программу. Этого удается избежать при использовании гиперфункций. Использование гиперфункций делает программу короче, быстрее и проще для понимания [3, 6].

2.7. 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. Дедуктивная верификация и программный синтез

Предикатная программа относится к классу *программ-функций* [19].

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.

Спецификацией предикатной программы $H(x\mathbf{:}\ y)$ являются два предиката: *предусловие* $P(x)$ и *постусловие* $Q(x, y)$. Спецификацию программы будем записывать в виде $[P(x), Q(x, y)]$. *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 $H(x\mathbf{:}\ y)$ истинно предусловие $P(x)$, то в случае завершения программы должно быть истинно постусловие $Q(x, y)$ в момент ее завершения. Поскольку предикатная программа является программой-функцией, обязательным является также *условие завершения программы*: если перед исполнением программы $H(x\mathbf{:}\ y)$ истинно предусловие $P(x)$, то программа обязана завершаться. Объединение условий частичной корректности и завершения программы определяет условие *тотальной корректности*.

Доказательство того, что некоторое свойство $W(x, y)$ истинно при завершении исполнения программы $H(x\mathbf{:}\ y)$, реализуется доказательством истинности формулы: $\mathcal{R}(H)(x, y) \Rightarrow W(x, y)$, где $\mathcal{R}$ – формальная операционная семантика. Эту формулу достаточно доказать при истинном предусловии. Поскольку $\mathcal{R}(H) = H$ [1], то формула для доказательства свойства W принимает следующий вид: $P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow W(x, y)$.

Условие частичной корректности программы $H(x\mathbf{:}\ y)$ записывается в виде формулы:

$$\forall x,y.\ P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y) \qquad (1)$$

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

$$\forall x.\ P(x) \Rightarrow \exists y.\ H(x\mathbf{:}\ y) \qquad (2)$$

Формула тотальной корректности программы относительно спецификации $[P(x), Q(x, y)]$ получается конъюнкцией формул (1) и (2).

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

Формулу тотальной корректности будем представлять в виде правила **COR**:

$$\textbf{COR}:\ \frac{\forall x,y.\ P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\mathbf{:}\ y)}{H(x\mathbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

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

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная система правил доказательства их корректности [21], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности. Корректность правил доказана [20] в системе PVS. В системе предикатного программирования реализован генератор формул корректности программы. Часть формул доказывается автоматически SMT-решателем CVC3. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [17]. Данный метод опробован для дедуктивной верификации более чем 50 небольших программ [3–6].

Задача *синтеза* программы $H(x\mathbf{:}\ y)$ из спецификации $[P(x), Q(x, y)]$ заключается в нахождении тотального предиката H на языке **P**, обеспечивающего истинность формулы $P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y)$. Вместо данной формулы можно использовать любое из правил доказательства корректности для одного из базисных операторов и реализовать синтез для посылок правила.

В настоящее время, автоматический синтез программ по формальным спецификациям, несмотря на значительные достижения в этой области, возможен лишь для простых коротких программ. Нашей целью является разработка методов синтеза небольших фрагментов предикатной программы. Синтез фрагментов реализуется в интеграции с дедуктивной верификацией [5, 6, 21] и обычным стилем конструирования программы в среде редактора Eclipse. Фрагменты программы, которые программист написал или модифицировал в редакторе Eclipse, автоматически проверяются на корректность подсистемой дедуктивной верификации. Фрагменты, отмечаемые в тексте комбинацией «[*]», программист поручает сгенерировать подсистеме синтеза, запускаемой в параллельном режиме. На примере синтеза операторов эффективной программы сортировки простыми вставками показано, что автоматический синтез с применением SMT-решателей возможен лишь при использовании 8 лемм; заранее предугадать их до начала процесса синтеза невозможно [22]. Таким образом, задача автоматического синтеза реальных программ оказывается принципиально нереализуемой. Чтобы программный синтез реальных программ стал возможным, необходимо проводить его в интеграции с интерактивной системой анализа генерируемых формул корректности, в рамках которой пользователь оперативно строит недостающие леммы.

Применением системы правил доказательства корректности задача синтеза отдельных фрагментов предикатной программы сводится к разрешению системы логических формул относительно неизвестных предикатов и термов. Логические формулы имеют вид $R(x)\ \&\ Z(x\mathbf{:}\ y) \Rightarrow S(x, y)$. Требуется найти неизвестный предикат $Z(x\mathbf{:}\ y)$, тотальный в области истинности предиката $R(x)$, так чтобы формула стала истинной.

4. 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)], т.е. удовлетворяющая формулам:

$$\forall x,y.\ P(x)\ \&\ H(x{:}\ y) \Rightarrow Q(x, y) \quad (1)$$

$$\forall x.\ P(x) \Rightarrow \exists y.\ H(x{:}\ y) \quad (2)$$

4.1. Формы декомпозиции предикатных программ

Операторы суперпозиции, условный и параллельный 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). Имеется правило **QP** доказательства корректности параллельного оператора:

$$\textbf{QP}:\ \frac{B(x{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)];\ C(x{:}\ z)\ \textbf{corr}\ [P(x), R(x, z)]}{\{B(x{:}\ y)\ ||\ C(x{:}\ z)\}\ \textbf{corr}\ [P(x), Q(x, y)\ \&\ R(x, z)]}$$

В соответствии с данным правилом, 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 в соответствии с правилом **QC**:

$$\textbf{QC}:\ \frac{\begin{array}{l}B(x{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x), Q(x, y)];\\ C(x{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x), Q(x, y)]\end{array}}{\{\textbf{if (}E(x))\ B(x{:}\ y)\ \textbf{else}\ C(x{:}\ y)\}\ \textbf{corr}\ [P(x), Q(x, y)]}$$

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).

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. Здесь рассмотрим лишь правило для частного случая – *сведения к более общей задаче* C(x, z: y).

$$\textbf{RBE}: \frac{\forall z\ C(x, z{:}\ y)\ \textbf{corr*}\ [P_C(x, z), Q(x, y)];\quad P(x) \Rightarrow \exists z.\ B(x{:}\ z)\ \&\ P_C{*}(x, B(x))}{C(x, B(x){:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

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 предикатном программировании сведение к более общей задаче часто применяется для приведения рекурсии к хвостовому виду. Истинность двух посылок правила **RBE** гарантирует корректность следующей программы:

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

В случае рекурсивного вызова C(x, B(x): y) обозначение **corr*** означает, что первая посылка опускается, а $P^*_C(x, B(x))$ заменяется на $P_C(x, B(x))$ & m(x) < m(v). Здесь m – натуральная функция *меры*, строго убывающая на аргументах рекурсивных вызовов, а v обозначает аргументы рекурсивной программы C.

Приведенные выше формы декомпозиции программы являются базисными. На практике используются самые разнообразные формы операторов, 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 (1) на конкретных примерах.

4.2. Пример. Вычисление наибольшего общего делителя

Рассмотрим программу GCD (great common divisor), вычисляющую наибольший общий делитель положительных чисел a и b.

Разработка программы начинается с построения теории Gcd для определения свойства gcd(a, b, c): значение c есть *наибольший общий делитель* чисел a и b. Сначала определим новый тип:

type nat1 = **subtype** (**nat** x: x > 0);

Все формулы в теориях должны быть определены для всех значений аргументов. Это вынуждает точно определять область допустимых значений переменных a и b.

В теории Gcd определим свойство gcd(a, b, c) вместе с набором лемм, которые потребуются в доказательствах.

```
theory Gcd {
   nat1 a, b;
   nat c, x;
   formula divisor(a, x) = ∃ nat1 z. x ∗ z = a;
   formula divisor2(a, b, c) = divisor(a, c) & divisor(b, c);
   formula gcd(a, b, c) = divisor2(a, b, c) & ∀x. (divisor2(a, b, x) ⇒ x ≤ c);
   gc1: lemma gcd(a, a, a);
   gc2: lemma gcd(a, b, c) ≡ gcd(b, a, c);
   gc3: lemma gcd(a, b, c) ≡ gcd(a + b, b, c);
   gc4: lemma a < b ⇒ gcd(a, b, c) ≡ gcd(a, b - a, c);
   gc5: lemma a > b ⇒ gcd(a, b, c) ≡ gcd(a - b, b, c);
};
```

Спецификацию программы GCD можно представить следующим образом:

```
GCD(nat1 a, b: nat c) post gcd(a, b, c);
```

Предусловие отсутствует. Фактически предусловием является: $a \in nat1$ & $b \in nat1$, т.е. $a > 0$ & $b > 0$.

В соответствии с формулой корректности (1) следует построить программу GCD, удовлетворяющую формуле:

$$GCD(a, b: c) \Rightarrow gcd(a, b, c) \qquad (4)$$

При этом решение ищется среди тотальных программ для $a \in nat1$ и $b \in nat1$.

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

В случае $a = b$ очевидным решением является оператор $c = a$. Формула корректности (4) истинна ввиду леммы gc1: gcd(a, a, a). Таким образом, получаем первый *вариант решения* задачи GCD:

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

Рассмотрим случай $a < b$. В соответствии с леммой gc4 истинна формула 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:

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

Аналогичным образом определяется третий вариант решения:

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

Объединение трех вариантов решения дает следующую программу:

```
type nat1 = subtype (nat x:  x > 0);
GCD(nat1 a, b:  nat c)
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)
};
```

Приведенная программа сконструирована с использованием лемм теории Gcd в обычном стиле предикатного программирования, представленном в публикациях [3–6, 12, 13, 16]. Чтобы гарантировать отсутствие ошибок, необходимо проведение

формального доказательства ее корректности. Проиллюстрируем метод дедуктивной верификации для данной задачи.

Исходная цель: доказать формулу GCD(a, b**:** c) **corr** [**true**, gcd(a, b, c)]. Применим дважды правило **QC** к телу программы. Получим три цели:

c = a **corr** [a = b, gcd(a, b, c)]
GCD(a, b – a**:** c) **corr** [a < b, gcd(a, b, c)]
GCD(a – b, b**:** c) **corr** [a > b, gcd(a, b, c)]

Для первой цели в соответствии с определением (3) получаем формулу корректности:

$$a = b \;\&\; c = a \;\Rightarrow\; gcd(a, b, c)$$

Формула доказывается с помощью леммы gc1.

Для второй цели используем правило **RBR** для случая, когда программа C рекурсивна:

$$\textbf{RBR:}\ \frac{P(x) \to P_B(x) \;\&\; P^*_C(B(x));\quad P(x) \;\&\; Q_C(B(x), y) \to Q(x, y);}{C(B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Здесь B(a, b) = (a, b – a). Правило конкретизируется следующим образом:

$$\textbf{RBR:}\ \frac{a < b \to\ \ b - a > 0 \;\&\; a + (b - a) < a + b;\quad a < b \;\&\; gcd(a, b - a, c) \to gcd(a, b, c)}{GCD(a, b - a\textbf{:}\ c)\ \textbf{corr}\ [a < b, gcd(a, b, c)]}$$

Первая посылка правила, очевидно, истинна; вторая определяет формулу корректности:

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

Данная формула доказывается с помощью леммы gc4.

Аналогичным образом реализуется генерация формул корректности и их доказательство для третьей цели.

Отметим, что для полной гарантии корректности программы относительно спецификации необходимо доказать все формулы корректности, а также все леммы теории Gcd.

На примере программы GCD дадим иллюстрацию процесса автоматического программного синтеза. Пусть имеется следующая программа с тремя фрагментами «[*]» для программного синтеза:

GCD(nat1 a, b**:** **nat** c) **post** gcd(a, b, c)
{ **if** (a = b) [*] **else** **if** (a < b) GCD([*]) **else** GCD([*]) };

На первом этапе в соответствии с контекстом фрагментов на их место вставляются неизвестные предикаты и термы.

GCD(nat1 a, b**:** **nat** c) **post** gcd(a, b, c)
{ **if** (a = b) X(a, b**:** c) **else** **if** (a < b) GCD(A, B**:** c) **else** GCD(C, D**:** c) };

Неизвестные термы A, B, C и D зависят от переменных a и b.

Для данной программы генерируются формулы корректности с помощью тех же правил, примененных выше при дедуктивной верификации. Приведем лишь формулы корректности, в которых встречаются неизвестные предикаты и термы.

a = b & X(a, b: c) ⇒ gcd(a, b, c)
a < b & gcd(A, B, c) ⇒ gcd(a, b, c)
a < b ⇒ m(A, B) < m(a, b)
a > b & gcd(C, D, c) ⇒ gcd(a, b, c)
a > b ⇒ m(C, D) < m(a, b)

Здесь m – неизвестная функция меры, которая должна быть синтезирована таким образом, чтобы аргументы рекурсивного вызова строго убывали по этой мере.

Рассмотрим первую формулу корректности. В этой формуле необходимо найти предикат X(a, b: c), при котором gcd(a, b, c) станет истинным. Очевидно, что в качестве X(a, b: c) можно было бы использовать gcd(a, b, c), однако система синтеза должна «знать», что для рекурсивной программы такое решение является плохим. Сначала система синтеза ищет терм gcd(a, b, c) в текущем контексте, которым является набор лемм теории Gcd. В первой лемме gc1 встречается терм gcd(a, a, a). *Унификация* с термом gcd(a, b, c) дает подстановку b = a и c = a. Отношение b = a встречается в посылке первого требования. Отношение c = a становится искомым предикатом для X(a, b: c).

Рассмотрим вторую формулу корректности. Тривиальное решение A = a и B = b здесь не годится. Попытка использовать лемму gc1 приводит к противоречию с первой посылкой a < b. Применение лемм gc2 и gc3 не дает решения. Использование посылки a < b в формуле для леммы gc4 упрощает ее до gcd(a, b, c) ≡ gcd(a, b - a, c). Это позволяет использовать gcd(a, b - a, c) в качестве решения. И тогда A = a и B = b - a.

Аналогичным образом для четвертой формулы корректности получим решение C = a – b и D = b. Подставляем решения для A, B, C и D в третью и пятую формулы:

a < b ⇒ m(a, b - a) < m(a, b)
a > b ⇒ m(a – b, b) < m(a, b)

Решение для функции m ищется перебором термов, зависящих от a и b: 0, a, b, a+b и т.д.

4.3. 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.

Сначала определим типы данных.

```
nat n; // n>0
type natn = 1..n;
type Arn = array(real, natn);
```

Здесь n описывается глобальной переменной. Фактически n является константой.

Предусловие и постусловие определяются следующими формулами.

```
formula pSwap(Arn a) = n > 0 & a[1] = 0;
formula qSwap(Arn a, a') =
    ∃ natn j. (a[j] ≠ 0 & a' = a with (1: a[j], j: 0)) ∨ a' = a & ∀ natn j. 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. 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.

Спецификация программы представлена следующим определением:

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

Отметим, что спецификация неоднозначна. Особенность программы в том, что массивы – объекты логики второго порядка. Данная программа считается тривиальной в императивном программировании и реализуется перебором элементов массива в цикле. 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:

formula pSwap1(natn p, Arn a) = n > 0 & ∀ j=1..p. a[j] = 0;

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

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 правилом **RBE** задача Swap сводится к более общей задаче Swap1:

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

{ Swap1(1, a**:** a′) };

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] ≠ 0.

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

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

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:

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

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)
};
```

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

Программа обмена элементов массива может быть определена в виде гиперфункции Swap(a**:** **:** a’), где первая ветвь соответствует полностью нулевому массиву a. Представим спецификацию гиперфункции Swap.

```
formula pSwap(Arn a) = n > 0 & a[1] = 0;
formula nulAr(Arn a) = ∀ natn j. a[j] = 0
formula qSwap2(Arn a, a’) =∃ natn j. (a[j] ≠ 0 & a’ = a with (1: a[j], j: 0));
Swap(Arn a: : Arn a’) pre pSwap(a) pre 1: nulAr(Arn a) post 2: qSwap2(a, a’);
```

Предусловие по первой ветви определяет, что массив a – нулевой. Постусловие по первой ветви отсутствует, поскольку эта ветвь не имеет результатов. Постусловие qSwap2 является первой альтернативой формулы qSwap.

Построение программы в виде гиперфункции реализуется по той же схеме, описанной выше. Результатом является программа:

```
Swap(Arn a: : Arn a’) pre pSwap(a) pre 1: nulAr(Arn a) post 2: qSwap2(a, a’);
      { Swap1(1, a: : a’) };

Swap1(nat p, Arn a: : Arn a’)
pre pSwap1(p, a) pre 1: nulAr(Arn a) post qSwap2(a, a’) measure n - p
{           if (p = n) #1
            if (a[p+1] = 0) Swap1(p+1, a: #1: a’#2)
            else { a’ = a with (1: a[p+1], p+1: 0)  #2 }
};
```

Здесь оператор перехода #1 реализует выход из программы Swap1 по первой ветви.

5. 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.

5.1. Определение трансформаций

Подстановка тела программы на место ее вызова. Пусть имеется предикатная программа B(x: y) { K } и вызов предиката B(g: z) внутри другой 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 [1] вычисление выражений, входящих в набор g, проводится параллельно. Реализация такого параллелизма на процессорах с обычной архитектурой не эффективна, 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. 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 предикатной программы определяет *хвостовую рекурсию*, если:

- имя вызываемой программы совпадает с именем 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 проведены 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 $a \leftarrow a'$.

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.

5.2. Трансформация программы вычисления наибольшего общего делителя

Рассмотрим программу вычисления наибольшего общего делителя (разд. 4.2):

```
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
}
```

5.3. Трансформация программы обмена элементов массива

Программа обмена элементов массива состоит из следующих двух предикатных программ (разд. 4.3):

```
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 ← 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 } } }
```

5.4. Трансформация программы суммирования элементов списка

Рассмотрим программу sum(s: c) суммирования элементов списка s; c – результат суммирования.

Тип списка представлен описанием:

```
type listR = list (real);
```

Сумма элементов списка s определяется следующей формулой:

```
formula SUM(listR s: real) = (s = nil) ? 0 : s.car + SUM(s.cdr);
```

Простейшая программа суммирования является реализацией данной формулы.

```
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
```

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 } }        (5)
```

Программа (5) 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 (5) переменная s модифицируется, поэтому следует определить ее начальное и текущее значение. Пусть вырезка 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, 0..N);
BUF S;
int j = m;
```

Должны выполняться ограничения: $0 \le m,\ n, j \le N$. Правила кодирования операций со списками, используемых в программе (5), представляются следующим образом:

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

Применение правил к программе (5) 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 (5), применяются следующие правила кодирования:

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

Применение правил к программе (5) дает итоговую программу:

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

6. Обзор работ

Имеется много различных исследований по использованию языка логики в качестве языка программирования. Большая их часть связана с логическим программированием. В этом русле значительное число работ по семантическому программированию [2]. В семантическом и предикатном программировании программа – это предикат; имеется сходство по многим другим позициям. Различие в способе исполнения программы. В семантическом программировании применяется логический вывод. Исполнение предикатной программы ближе к императивному и функциональному программированию.

Парадигма предикатного программирования до недавнего времени не имела в мире близких аналогов. Впервые появился язык, похожий на язык предикатного программирования P. Функциональный язык доказательного программирования Smart разработан французской компанией Prove&Run[2]. Корректность программ, а также отсутствие уязвимостей обеспечиваются дедуктивной верификацией. Язык Smart содержит конструкции, аналогичные гиперфункциям. Программа может быть оттранслирована на языки C и Java. Полное описание языка недоступно. Некоторые особенности языка Smart изложены в работах [14, 15].

[2] www.provenrun.com

Среди других систем доказательного программирования, в которых программа корректна по построению, следует упомянуть известную систему автоматического интерактивного доказательства Coq [24]. В ней функциональная программа определяется как элемент типа, отображающего область истинности предусловия в область истинности постусловия, и необходимо доказать принадлежность программы этому типу.

Формализованное описание языка программирования в виде онтологии встречается в ряде работ, например, в виде совместного онтологического описания [18] пяти языков стандарта IEC 61131-3 для программируемых логических контроллеров и для проецирования на единое внутреннее представление семейства реализуемых языков [23]. Однако применение онтологического аппарата для описания языков пока не оказало заметного влияния на индустрию разработки языковых процессоров. Формальная операционная семантика является существенно более сложной и глубокой в сравнении с онтологическими формализациями. Формальная семантика является обязательным базисом для дедуктивной верификации и программного синтеза.

7. Заключение

Разработан язык предикатного программирования P [10], в котором любая программа – предикат, тождественный ее формальной операционной семантике. Общее требование о соответствии программы и спецификации конкретизируется для класса программ-функций [19] в виде формулы тотальной корректности (3). На ее базе разработаны методы дедуктивной верификации и программного синтеза, гарантирующие корректность предикатной программы относительно спецификации. Формула (3) определяет универсальный закон построения алгоритмов в классе программ-функций не только в предикатном программировании (разд. 4.1). Применение дедуктивной верификации и программного синтеза дает абсолютную гарантию корректности программы относительно спецификации, что чрезвычайно важно в приложениях с высокой ценой ошибки.

Эффективность предикатных программ достигается тем, что программист проводит необходимые оптимизации и строит эффективный алгоритм, который применением набора оптимизирующих трансформаций преобразуется в эффективную императивную программу. Технология предикатного программирования позволяет воспроизвести любую реализацию в императивном программировании.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

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. Гончаров С.С., Ершов Ю.Л., Свириденко Д.И. Методологические аспекты семантического программирования. *Научное знание: логика, понятия, структура*. Новосибирск, 1987. С.154-184.
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. Andreescu O.F., Jensen T., Lescuyer S. Dependency Analysis of Functional Specifications with Algebraic Data Structures // ICFEM'15. LNCS 9407, 2015. P. 116-133.
15. Lescuyer S. Towards a verified isolation micro-kernel // MILS: Architecture and Assurance for Secure Systems, Amsterdam, 2015. 8 P.
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. Будаговский Д. А., Дубинин В. Н. Онтология языков программирования стандарта IEC 61131-3, *Сб. статей XVIII Международной научно-методической конференции "Университетское образование"*. - Пенза, 2014. С. 164-166.
19. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538. http://persons.iis.nsk.su/files/persons/pages/prog.pdf
20. http://www.iis.nsk.su/persons/vshel/files/rules.zip
21. Чушкин М.С. Система дедуктивной верификации предикатных программ // «Программная инженерия», № 5, 2016. – С. 202-210. http://persons.iis.nsk.su/files/persons/pages/paper.pdf
22. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы '2017», Ростов-на-Дону — 2017 — С.258-262. http://persons.iis.nsk.su/files/persons/pages/prog.pdf
23. Князева М.А., Тимченко В.А. Модель онтологии проекций языков программирования высокого уровня на единое представление программ. *Тр. 11-й конф. по искусственному интеллекту КИИ-08*. Москва, 2008. Т. 1. С. 124 - 132.

24. Chlipala, A. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. — MIT Press, 2013. — 440 p.