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

М.С. Чушкин, В.И. Шелехов

ГЕНЕРАЦИЯ И ДОКАЗАТЕЛЬСТВО УСЛОВИЙ КОРРЕКТНОСТИ ПРЕДИКАТНЫХ ПРОГРАММ

Препринт
166

Новосибирск 2012

Разработана система правил доказательства корректности для различных видов операторов предикатных программ. Описывается реализация генератора условий тотальной корректности предикатных программ в системе предикатного программирования. Условия тотальной корректности программы транслируются в систему автоматического доказательства PVS.

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

M.S. Chushkin, V.I. Shelekhov

GENERATION AND PROOF OF CORRECTNESS CONDITIONS FOR PREDICATE PROGRAMS

Preprint
166

Novosibirsk 2012

Rules for proving the correctness conditions for different statement of a predicate program are developed. The implementation of a generator of the total correctness of a predicate program is described. The total correctness conditions of a program are translated to the specification language of the PVS proof system.

ВВЕДЕНИЕ[1]

Первые программные системы разрабатывались в рамках программ научных исследований или программ для нужд министерств обороны. Тестирование таких продуктов проводилось строго формализовано с записью всех тестовых процедур, тестовых данных и полученных результатов.

Однако данный подход не гарантировал выявление всех ошибок. Так, ошибка в системе управления космическим аппаратом Mariner 1 привела к потере этого аппарата 22 июля 1962 года. Ошибка заключалась в том, что в одном месте была пропущена операция усреднения скорости корабля по нескольким последовательно измеренным значениям. В результате колебания значения скорости, вызванные ошибками измерений, стали рассматриваться системой как реальные, и она попыталась предпринять корректирующие действия, которые привели к полной неуправляемости аппарата.

В начале 1970-х тестирование определялось как «процесс, направленный на демонстрацию корректности продукта». Однако ввиду невозможности достичь желаемого результата к концу 1970-х тестирование представлялось как выполнение программы с намерением найти ошибки, а не доказать, что она работает.

Тестирование – не единственный способ обнаружения ошибок. Комплекс различных методов обнаружения ошибок и контроля качества программ определяется как верификация программного обеспечения. Верификация включает в себя множество методов, таких как статический анализ, экспертиза и формальные методы. Одним из формальных методов является дедуктивная верификация. В отличие от тестирования с помощью дедуктивной верификации можно обнаружить все ошибки несоответствия программы своей формальной спецификации. Однако это весьма сложный и трудоемкий метод. Применение дедуктивной верификации оправдано лишь в приложениях с высокой ценой ошибки: в авиакосмической отрасли, энергетике, медицине и др.

Дедуктивная верификация реализует проверку правильности программы относительно ее спецификации, записанной на формальном языке спецификаций. Условия корректности программы генерируются автоматически по формулам логики и спецификации программы путем применения системы логических правил. Условие корректности программы обычно имеет вид $A \Rightarrow B$, где B – утверждение, определяемое спецификацией программы,

[1] Работа выполнялась в рамках гранта РФФИ № 12-01-00686.

а формула A, истинная для программы, извлекается из нее с помощью формальной (операционной, денотационной и др.) семантики [12, 13] языка программирования. Доказательство условий корректности проводится с помощью некоторой системы автоматического доказательства. Ее применение, в отличие от доказательства «вручную», гарантирует правильность программы относительно спецификации.

В данной работе описывается автоматическая генерация условий корректности для программ на языке предикатного программирования P [3] и трансляция условий корректности на язык спецификаций системы PVS [8].

Язык предикатного программирования P находится на границе между функциональными и логическими языками. Язык P определяет класс программ, не взаимодействующих с внешним окружением. Эти программы реализуют функции, отображающие значения входных переменных в значения результатов и не взаимодействуют с внешним окружением. Исполнение таких программ должно всегда завершаться, поскольку их бесконечное исполнение бессмысленно. Этот класс, по меньшей мере, включает программы для задач дискретной и вычислительной математики. Спецификация предикатных программ реализуется с помощью предусловия и постусловия.

Метод дедуктивной верификации, используемый в предикатном программировании, существенно отличается от классических методов Флойда и Хоара, описанных в разд. 1. В разд. 2 подробно описан метод дедуктивной верификации, используемый для доказательства корректности предикатных программ, приведен пример применения метода. В разд. 3 представлены алгоритмы генерации формул корректности для программы в целом и для конкретных операторов языка P.

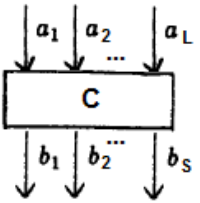
Доказательство условий корректности может быть выполнено вручную, но для практически значимых программ сам размер спецификации и реализации таков, что необходимо использование специализированных инструментов для автоматического построения доказательств или предоставляющих существенную помощь в их осуществлении.

Для доказательства полученных формул корректности используется система PVS [8], обладающая языком спецификаций высокого уровня и мощным блоком доказательства. В разд. 4 описана трансляция полученных формул корректности в язык спецификаций системы PVS.

1. ОБЗОР КЛАССИЧЕСИХ МЕТОДОВ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ

1.1. Подход Флойда

В 1967 году американский ученый в области теории вычислительных систем Роберт Флойд опубликовал статью «Assigning Meanings to Programs» [1] («Придание смысла программам»). В статье детально описан метод формальной верификации для алгоритмов, представленных блок-схемами.



Блок-схема является ориентированным графом с командами в качестве вершин и с дугами, соединяющими вершины. Дугами показывают возможность передачи управления от одной команды к другой. У каждой команды есть несколько входов (a_i) и выходов (b_j).

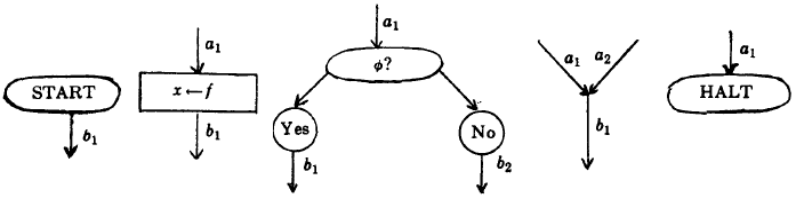
Идея подхода Флойда заключалась в прикреплении к каждой дуге т.н. ярлыка (tag) в виде логического утверждения, определяющего смысл перехода. Для примера рассмотрим команду C. В качестве ярлыка над входом a_i выступает предусловие P_i, а над выходом b_j постусловие Q_i. Для сокращения записей все предусловия собираются в вектор **P**, а постусловия в вектор **Q**.

Флойд не использовал терминов «предусловие» и «постусловие», они были введены несколько позже. Для обозначения входных и выходных ярлыков он использовал слова «antecedent» и «consequent» (посылка и заключение, соответственно).

Верификация блок-схемы определена следующим образом. Это доказательство для каждой команды C утверждения о том, что, если управление перешло к команде C по входу a_i с истинным предусловием P_i, то управление должно покинуть команду, причем, если управление покинуло команду по выходу b_j, то постусловие Q_j команды C истинно.

Это утверждение Флойд называл условием верификации (verification condition) и записывал как $V_C(\mathbf{P}, \mathbf{Q})$. Построение подобных условий верификации (позже названными условиями корректности) — реальная задача,

тесно связанная с языком программирования, на котором записан алгоритм. Вид этих условий уникален и зависит от конкретного оператора в языке программирования.



Блок-схемы содержат лишь 5 команд, благодаря которым можно записать практически любой алгоритм, в том числе и алгоритм любого оператора. В подтверждении этому Флойд тут же реализовал несколько операторов из языка ALGOL на блок-схемах и провел их верификацию.

Таким образом, задача, которую в дальнейшем решал Флойд — это построение условий верификации для пяти видов команд. В процессе построения этих условий Флойд пришел к выводу: для любой команды C условие верификации $V_C(\mathbf{P}, \mathbf{Q})$ может быть приведено к виду

$$T_C(\mathbf{P}) \vdash \mathbf{Q}$$

Под этой записью он подразумевал, что **Q** пропозиционально выводимо из $T_C(\mathbf{P})$. Формулу $T_C(\mathbf{P})$ была названа сильнейшим следствием (strongest verifiable consequent) на предусловие **P**.

Однако доказательство подобных условий верификации не гарантирует нам завершаемости программы. Доказательство завершаемости — это отдельная задача, которая не всегда имеет решение. Рассмотрим небольшой пример, подтверждающий это, но сначала вспомним несколько определений.

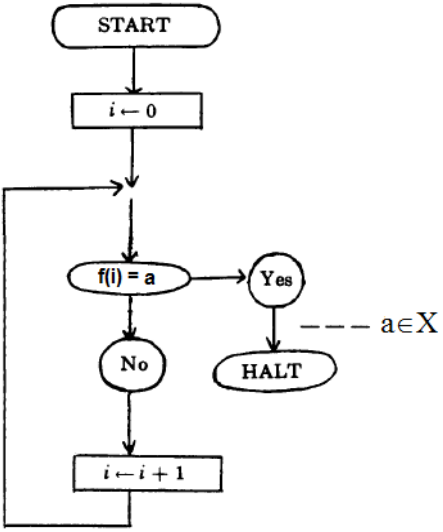
Перечислимое множество — это множество, элементы которого могут быть получены с помощью некоторого алгоритма, или X — перечислимо, если

$$\forall x \in X \quad \exists i \in N \quad f(i) = x$$

где $f: N \rightarrow X$ — некоторая алгоритмически вычислимая функция.

Разрешимое множество — множество, характеристическая функция которого вычислима.

Существуют примеры множеств, которые являются перечислимыми, но, увы, неразрешимы. Возьмем одно из таких множеств X и попробуем построить алгоритм распознавания его элементов, воспользовавшись функцией f, которая их перечисляет.



Алгоритм довольно прост, и провести его верификацию по Флойду не составит труда. В доказательстве же завершаемости мы терпим неудачу, т.к. оно сводится к доказательству истинности одного единственного утверждения:

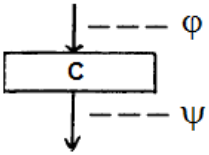
$$\chi_X(a)$$

где χ_X — характеристическая функция множества X**.** Однако это утверждение, ввиду невычислимости характеристической функции множества X, невычислимо.

Тем не менее, для большинства корректных программ мы можем доказать их завершаемость. Для этого Флойд воспользовался теорией вполне упорядоченных множеств (well-oredered set).

Вполне упорядоченное множество (Well-ordered set, W-множество) — это линейно упорядоченное множество, в любом непустом подмножестве

которого есть минимальный элемент. Иными словами, это множество, в котором не существует бесконечно убывающих последовательностей. Наиболее известный пример W-множества — это множество натуральных чисел.



Идея метода доказательства звершаемости довольно проста: изменим ярлык на каждой дуге в блок-схеме, добавив туда функцию для свободных переменных, заданную на W-множестве. Будем называть такие функции W-функциями.

Теперь, если для каждого исполнения команды C мы покажем, что значение W-функции, связанной с выходом, меньше значения W-функции, связанной с входом, то это будет значить, что значение W-функции постоянно убывает. Так как в W-множестве нет бесконечно убывающих последовательностей, то W-функция не может бесконечно убывать, а это значит, что программа рано или поздно завершится.

Более формально это выглядит как модификация предусловий и постусловий к командам. Пусть φ и ψ — это W-функции, связанные со входом и выходом, соответственно, а δ — это нигде не использованная ранее переменная. Новое утверждение верификации команды C будет выглядеть так:

$$V_C(\mathbf{P} \,\&\, \delta=\varphi,\ \mathbf{Q} \,\&\, \psi<\delta)$$

1.2. Подход Хоара

Как правило, в программе нас больше всего интересует то, соответствует ли она изначальному замыслу, или нет. Замысел – это та функция, которую, как мы полагаем, должна вычислять программа. Этот замысел может быть представлен в виде логического утверждения, описывающего значения, которые примут переменные после исполнения программы.

Например, рассмотрим программу целочисленного деления a на b:

```
int c = 0;
for (int i=0; i<a; ++i)
    if (i*b > a) {
        c = i - 1;
        break;
    }
```

Замысел данной программы может быть описан утверждением

$$a \geq b*c \ \& \ a < b*(c+1)$$

Однако в большинстве случаев результат работы программы (или части программы) зависит еще и от тех значений, которые она получила на входе. В частности, в нашем примере, если мы попытаемся поделить на 0, значение переменной c не изменится вовсе. Возможно, внимательный читатель заметит, что аналогичная ситуация возникает и для отрицательных чисел.

Правда, в нашем случае, переменная была предварительно инициализирована, и максимум что может произойти далее — мы просто будем работать с неверным значением. А вот если бы переменную не инициализировали, то результат дальнейшей работы программы был бы непредсказуем. Произойти могло бы все что угодно, вплоть до завершения работы программы с ошибкой.

Таким образом, на программу необходимо налагать некоторые начальные условия, которые могут быть записаны в виде логических утверждений. В нашем случае необходимо условие

$$a \geq 0 \ \& \ b > 0$$

Руководствуясь этими очевидными соображениями, английский ученый, специалист в области информатики и вычислительной техники, Чарльз Хоар [2], установил связь между предусловием (P), программой (S) и постусловием (Q). Эту связь он представил в виде обозначения:

$$P \{ S \} Q$$

называемого тройкой Хоара и обозначающего следующее утверждение: если предусловие P верно до начала исполнения программ S, то постусловие Q будет верно после его завершения. Была построена логика, называемая логикой Хоара, предназначенная для доказательства корректности программ.

Доказательство корректности программы осуществлялось приведением начальной тройки Хоара к одной из аксиом, коих в оригинальной работе было 2:

Аксиома для оператора присваивания

$$\vdash P_{f/x} \{ x := f; \} P$$

где $P_{f/x}$ означает выражение P, в котором все вхождения свободной переменной x заменены выражением f, и аксиома для пустого оператора

$$\vdash P \{ \textbf{skip;} \} P$$

Приведение осуществлялось за счет правила вывода:

$$\frac{P \{ S \} Q; \quad P' \vdash P; \quad Q \vdash Q'}{P' \{ S \} Q'}$$

и правила композиции:

$$\frac{P \{ S \} Q; \quad Q \{ T \} R}{P \{ S;T \} R}$$

Также были представлены правила вывода для условного оператора

$$\frac{B \,\&\, P \{ S \} Q; \quad \neg B \,\&\, P \{ T \} Q}{P \{ \textbf{if } (B) \textbf{ then } S \textbf{ else } T \textbf{ endif} \} Q}$$

и для оператора цикла

$$\frac{P \,\&\, B \{ S \} P}{P \{ \textbf{while } B \textbf{ do } S \textbf{ done} \} \neg B \,\&\, P}$$

Здесь утверждение P является инвариантом — условием, истинным до итерации, и после нее. Поиск инвариантов — это нетривиальная задача. Позже Хоаром и другими исследователями в дополнение к этим правилам были разработаны правила для многих конструкций (таких, например, как вызов подпрограммы).

Однако стандартный метод Хоара не дает ответ на вопрос о завершаемости программы, т.е. в стандартной логике Хоара может быть доказана

только частичная корректность. Завершение программы нужно доказывать отдельно.

1.3. Другие подходы

Систему правил Хоара принято называть логикой Хоара (или Флойда–Хоара). Подавляющее число работ по дедуктивной верификации базируется на логике Хоара. Из посдедних наиболее серьезными являются работы Microsoft Research по верификации программ операционных систем [9]. Для упрощения верификации вместо произвольных указателей в программе допускаются лишь двухсвязные кольцевые списки или структуры типа «лес деревьев». На базе смешанной аксиоматической семантики разработан метод верификации С-программ [17]. В последнее время популярен метод верификации программ с указателями на базе логики “separation logic” [14]. Среди других подходов следует отметить метод Э.Дейкстры построения формул тотальной корректности на базе слабейших предусловий [15].

2. КОРРЕКТНОСТЬ ПРЕДИКАТНЫХ ПРОГРАММ

2.1. Исчисление предикатов

Ниже будет приведено несколько основных определений, заимствованных из теории исчисления предикатов. Для более прозрачной связи с данной работой определения были слегка упрощены. Более подробно и полно этот материал описан в [11].

Правилом вывода называют формальную запись вида

$$\frac{\Phi_0, \Phi_1, \ldots, \Phi_n}{\Gamma}$$

где Φ_i и Γ — это формулы, причем Φ_i — это посылки, а Γ — это заключение.

Определим индуктивно понятие *дерева формул*:

1. Всякая формула является деревом.
2. Если $D_0, D_1, \ldots, D_n$ — деревья, и S — формула, то

$$\frac{D_0, D_1, \ldots, D_n}{S}$$

— также дерево. Формула S является корнем дерева.

Дерево формул D называется *деревом вывода* формулы C, если его переходы — это применение правил вывода, а корнем в D является C.

Истинность формул, являющихся листьями дерева, и корректность правил влечет истинность корневой формулы C.

2.1. Логика оператора

Программа на языке P – совокупность определений предикатов. Каждый предикат —вычислимая логическая формула, представленная в виде оператора. Определение предиката имеет вид

$$A(x: y)\ \textbf{pre}\ P(x)\ \{\ S(x: y);\ \}\ \textbf{post}\ Q(x, y)$$

где A — это имя определяемого предиката, S — оператор, а x и y — это непересекающиеся наборы переменных, аргументы и результаты соответственно.

Спецификация предиката A(x: y) задается двумя логическими формулами: предусловием P(x), ограничивающим область определения функции, реализуемой программой, и постусловием Q(x, y), связывающим значения аргументов и результатов. Спецификацию будем записывать в виде [P(x), Q(x, y)].

Логика оператора S(x: y) — сильнейший предикат L(S(x: y)), истинный при завершении его исполнения [6].

Определим логику для базисных операторов. В предположении, что выражение E, зависящее от x, не содержит переменную a, логика оператора присваивания a := E есть

$$L(a := E(x)) \cong\ R(x)\ \&\ a = E(x)$$

где R(x) — слабейший предикат, при истинности которого определено выражение E. Например, $L(c := a / b) = b \neq 0\ \&\ c = a / b$.

Построим логику L(B; C) оператора B; C, определяющего последовательное исполнение операторов B и C, через логики L(B) и L(C). Оказывается, необходимо отдельно рассматривать случай, когда вычисление оператора C от B не зависит. Для фиксации связей между операторами произвольный оператор A будем изображать в виде A(x**:** y), где наборы переменных x и y обозначают аргументы и результаты оператора, соответственно.

Вместо оператора B; C далее будем рассматривать оператор суперпозиции B(x**:** z); C(z**:** y) и параллельный оператор B(x**:** y) || C(x**:** z). Третьим рассматривается условный оператор **if** (E) B(x**:** y) **else** C(x**:** y), где логическое выражение E может зависеть от x. Предполагается, что наборы x, y и z не пересекаются, а набор x может быть пустым. Определим логики указанных операторов:

$L(B(x: z); C(z: y)) \cong \exists z\, L(B(x: z)) \,\&\, L(C(z: y))$
$L(B(x: y) \,||\, C(x: z)) \cong L(B(x: y)) \,\&\, L(C(x: z))$
$L(\textbf{if}\ (E)\ B(x: y)\ \textbf{else}\ C(x: y)) \cong (E \Rightarrow L(B(x: y))) \,\&\, (\neg E \Rightarrow L(C(x: y)))$

Логика программы должна быть *согласована* с формальной операционной семантикой языка P: для произвольного оператора $B(x: y)$ и фиксированных значений переменных наборов x и y его логика $L(B(x: y))$ истинна тогда и только тогда, когда любое исполнение оператора $B(x: y)$ на наборе значений x завершается, причем результатами исполнения являются значения набора y. Отметим, что условием завершения оператора $B(x: y)$ является формула $\exists y\, L(B(x: y))$.

2.2. Корректность программы

Допустим, программе в целом соответствует главный предикат A(x: y) со спецификацией [P(x), Q(x, y)], оператором которого является S(x: y). Тогда корректность программы определяется следующими условиями:

1) постусловие Q(x, y) должно быть истинным после исполнения оператора S(x: y);
2) исполнение оператора S(x: y) всегда завершается.

Утверждение L(S(x: y)) становится посылкой при доказательстве первого условия корректности, поскольку после исполнения оператора S(x**:** y) оно становится истинным. Условие завершения исполнения оператора S(x: y) определяется утверждением: $\exists y\, L(S(x: y))$. Кроме того, предусловие P(x) необходимо использовать в качестве посылки в обоих условиях корректности, определяемых ниже следующими формулами:

$P(x) \,\&\, L(S(x: y)) \Rightarrow Q(x, y)$
$P(x) \Rightarrow \exists y\, L(S(x: y))$

Первое утверждение называется условием частичной корректности, а второе — условием завершения программы. Их конъюнкция определяет условие *тотальной (или полной) корректности* оператора $S(x: y)$ относительно спецификации [P(x), Q(x, y)]:

$Corr(S, P, Q)(x) \cong P(x) \Rightarrow (\forall y\, (L(S(x: y)) \Rightarrow Q(x, y))) \,\&\, \exists y\, L(S(x: y))$

Далее термин «корректность» будем использовать в смысле тотальной корректности.

2.3. Корректность рекурсивно определяемого предиката

Используется следующая схема доказательства по индукции для некоторого произвольного утверждения W(z):

$$\forall t \in X\ [\ (\forall u \in X\ m(u) < m(t) \Rightarrow W(u)\ ) \Rightarrow W(t)\ ] \Rightarrow \forall z \in X\ W(z)$$

Функция m, называемая мерой, отображает X во множество натуральных чисел со стандартным отношением порядка <.

Допустим, имеется определение рекурсивного предиката A:

$$A(x: y)\ \textbf{pre}\ P(x)\ \{\ K(x: y);\ \}\ \textbf{post}\ Q(x, y)$$

Внутри оператора K(x: y) имеется рекурсивный вызов предиката A.

В соответствии со схемой индукции корректность рекурсивного предиката может быть определена следующей формулой:

$$\forall t\ (\ \forall u\ (m(u) < m(t) \Rightarrow Corr(A, P, Q)(u)\ ) \Rightarrow Corr(A, P, Q)(t)$$

Введем формулу, которая будет обозначать индуктивное предположение для набора аргументов t:

$$Induct(A, P, Q)(t) \cong \forall u\ (m(u) < m(t) \Rightarrow Corr(A, P, Q)(u))$$

Допустим, оператор B находится в теле рекурсивного предиката A. Поскольку внутри оператора B может находиться рекурсивный вызов A, корректность оператора B определяется формулой:

$$Corr(A, P, Q, B, P_B, Q_B)(t, x) \cong Induct(A, P, Q)(t) \Rightarrow Corr(B, P_B, Q_B)(x),$$

где P_B и Q_B — предусловие и постусловие оператора B.

Для рекурсивного вызова проверку по мере m(u) < m(t) присоединим к предусловию. Введем обозначение:

$$P^*(x) \cong m(x) < m(t)\ \&\ P(x)$$

Здесь t — формальные параметры рекурсивного определения предиката, а x — фактические параметры рекурсивного вызова. В случае нерекурсивного вызова запись P*(x) эквивалентна P(x).

В итоге условие корректности рекурсивного определения предиката A, с телом в виде оператора K, имеет следующий вид:

$$Corr(A, P, Q, K, P, Q)(x, x)$$

В случае нерекурсивного предиката A индуктивное предположение отсутствует, т.е. равно **true**, и тогда $Corr(A, P_A, Q_A, K, P, Q)(x, x)$ тождественно Corr(K, P, Q)(x).

2.4. Система правил вывода условий корректности

Используя формулу тотальной корректности, можно автоматически построить формулу корректности оператора S(x: y) при условии, что для языка программирования построена логика программы. Итоговая формула корректности будет длинной и сложной даже для коротких программ; она будет длиннее программы S(x: y). Специализация формулы тотальной корректности для разных видов операторов позволяет декомпозировать длинную формулу корректности к нескольким более коротким и простым формулам.

В данном разделе определяется система правил вывода условий корректности для различных операторов. Доказательства правил приведены в работах [4, 5, 6]. Кроме того, формальные доказательства корректности правил проведены в системе автоматического доказательства PVS [16].

В качестве аргументов предикатов в предлагаемых ниже правилах используются переменные. Нетрудно показать, что в позициях аргументов можно использовать выражения, при условии, что предусловия выражений выводимы из предусловий предикатов.

Каждое правило обладает уникальным именем, состоящим из нескольких заглавных латинских букв. Имя строится по следующему принципу. Первые буквы обозначают группу правил, к которой принадлежит конкретное правило. Групп несколько: R — группа правил для общего случая, Q — для случая отсутствия спецификации у подоператоров, L — для случая однозначной спецификации, F — группа правил для декомпозиции логики в правой части, FL — для декомпозиции логики в левой части, E — для декомпозиции квантора существования в правой части. Далее идет буква, обозначающая оператор, к которому применяется правило: P — параллель-

ный оператор, S — оператор суперпозиции, C — условный оператор, R — оператор суперпозиции или оператор вызова, RB — оператор суперпозиции в общем виде.

2.4.1. Система правил вывода для общего случая

Допустим, операторы B(x: z) и C(z: y) корректны относительно своих спецификаций $[P_B, Q_B]$ и $[P_C, Q_C]$. Для этого случая используются следующие правила:

RP:
$$\frac{\begin{array}{l}\mathrm{Corr}(A, P_A, Q_A, B, P_B, Q_B)(t, x);\quad \mathrm{Corr}(A, P_A, Q_A, C, P_C, Q_C)(t, x);\\ P(x) \to P^*_B(x) \,\&\, P^*_C(x);\quad \forall y, z\,(Q_B(x, y) \,\&\, Q_C(x, z) \to Q(x, y, z))\end{array}}{\mathrm{Corr}(A, P_A, Q_A, B(x: y) \,||\, C(x: z), P, Q)\,(t, x)}$$

RS:
$$\frac{\begin{array}{l}\mathrm{Corr}(A, P_A, Q_A, B, P_B, Q_B)(t, x);\\ \forall z\ \mathrm{Corr}(A, P_A, Q_A, C, P_C, Q_C)(t, (x, z));\\ P(x) \to P^*_B(x);\quad \forall z, v\,(P(x) \,\&\, Q_B(x, z, v) \to P^*_C(x, z));\\ \forall z, v, y\,(P(x) \,\&\, Q_B(x, z, v) \,\&\, Q_C(x, z, y) \to Q(x, y))\end{array}}{\mathrm{Corr}(A, P_A, Q_A, B(x: z, v); C(x, z: y), P, Q)(t, x)}$$

RC:
$$\frac{\begin{array}{l}\mathrm{Corr}(A, P_A, Q_A, B, P_B, Q_B)(t, x);\\ \mathrm{Corr}(A, P_A, Q_A, C, P_C, Q_C)(t, x);\\ P(x) \,\&\, E \to P^*_B(x);\quad P(x) \,\&\, \neg E \to P^*_C(x);\\ \forall y\,(P(x) \,\&\, E \,\&\, Q_B(x, y) \to Q(x, y));\\ \forall y\,(P(x) \,\&\, \neg E \,\&\, Q_C(x, y) \to Q(x, y))\end{array}}{\mathrm{Corr}(A, P_A, Q_A, \textbf{if}\ (E(x))\ B(x: y)\ \textbf{else}\ C(x: y), P, Q)(t, x)}$$

Допустим, необходимо доказать тотальную корректность оператора вызова C(B(x): y), где C — вызываемый предикат, а B(x) — набор аргументов в виде выражений, не содержащих других вызовов. Для этого необходимо воспользоваться следующим правилом:

RB:
$$\frac{\begin{array}{l}\forall z\ \mathrm{Corr}(A, P_A, Q_A, C, P_C, Q_C)(t, z);\\ P(x) \to P_B(x) \,\&\, P^*_C(B(x));\\ \forall y\,(P(x) \,\&\, Q_C(B(x), y) \to Q(x, y));\\ \forall x, z_1, z_2\ P_B(x) \,\&\, L(B(x: z_1)) \,\&\, L(B(x: z_2)) \to z_1 = z_2;\end{array}}{\mathrm{Corr}(A, P_A, Q_A, C(B(x): y), P, Q)(t, x)}$$

В случае если вызов C(B(x): y) рекурсивен (C совпадает с A), то первая посылка правила совпадает с индуктивным предположением и поэтому от-

сутствует в правиле. Отметим, что для всех предыдущих и последующих правил действует аналогичное соглашение: если посылка правила имеет вид: Corr(A, P_A, Q_A, A, P_A, Q_A), то она отсутствует в правиле.

Не всегда известны спецификации для подоператоров. В этом случае следует применять другие правила:

$$\textbf{QP:}\ \frac{\mathrm{Corr}(A, P_A, Q_A, B, P, Q_B)(t, x);\quad \mathrm{Corr}(A, P_A, Q_A, C, P, Q_C)(t, x);}{\mathrm{Corr}(A, P_A, Q_A, B(x: y) \,||\, C(x: z), P, Q_B \,\&\, Q_C)(t, x)}$$

$$\textbf{QS:}\ \frac{\begin{array}{l}P(x) \to \exists z, v\ L(B(x: z, v));\\ \forall z\ \mathrm{Corr}(A, P_A, Q_A, C, \lambda x, z.(\ P(x)\ \&\ \exists z, v\ L(B(x: z, v))\ ), Q)(t, (x, z));\end{array}}{\mathrm{Corr}(A, P_A, Q_A, B(x: z, v); C(x, z: y), P, Q)(t, x)}$$

$$\textbf{QSB:}\ \frac{\begin{array}{l}\mathrm{Corr}(A, P_A, Q_A, B(), P_B, Q_B)(t, x);\quad P(x) \to P^*_B(x);\\ \forall z\ \mathrm{Corr}(A, P_A, Q_A, C, \lambda x, z.\ (\ P(x)\ \&\ Q_B(x, z)\ ), Q)(t, (x, z))\end{array}}{\mathrm{Corr}(A, P_A, Q_A, B(x: z, v); C(x, z: y), P, Q)(t, x)}$$

$$\textbf{QC:}\ \frac{\begin{array}{l}\mathrm{Corr}(A, P_A, Q_A, B, \lambda x.P(x)\ \&\ E(x), Q)(t, x);\\ \mathrm{Corr}(A, P_A, Q_A, C, \lambda x.P(x)\ \ \&\ \neg E(x), Q)(t, x);\end{array}}{\mathrm{Corr}(A, P_A, Q_A, \textbf{if}\ (E(x))\ B(x: y)\ \textbf{else}\ C(x: y))(t, x)}$$

2.4.2. Теорема тождества спецификации и программы

Формула $L(S(x: y)) \Rightarrow Q(x, y)$, являющаяся главной частью формулы тотальной корректности, определяет вывод спецификации из программы, точнее, из ее логики. Существуют подходы (в частности, программного синтеза), в которых, наоборот, программа выводится из спецификации.

Известные понятия тотальности и однозначности функции f: x → y, где x и y непересекающиеся наборы переменных, естественным образом переносятся на формулу H(x, y), рассматриваемую как функция из x в y. Формула H(x, y) является однозначной в области X для набора переменных x, если истинно

$$\forall x \in X \quad \forall y1, y2 \quad H(x, y1)\ \&\ H(x, y2) \Rightarrow y1 = y2$$

Формула H(x, y) является тотальной в области X для набора переменных x, если

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

Однозначность оператора S(x: y), тотальность и однозначность спецификации [P(x), Q(x, y)] определяются, соответственно, формулами

$$\forall y_1, y_2.\ P(x) \,\&\, L(S(x{:}\ y_1)) \,\&\, L(S(x{:}\ y_2)) \Rightarrow y_1 = y_2$$
$$T(P, Q)(x) \ \cong\ P(x) \Rightarrow \exists y\ Q(x, y)$$
$$SV(P, Q)(x) \ \cong\ \forall y_1, y_2.\ P(x) \,\&\, Q(x, y_1) \,\&\, Q(x, y_2) \Rightarrow y_1 = y_2$$

Теорема тождества спецификации и программы. Рассмотрим предикат A(x: y) со спецификацией [P(x), Q(x, y)] и оператором S(x: y). Допустим, оператор S(x: y) является однозначным, а спецификация [P(x), Q(x, y)] является тотальной. Предположим, логика оператора L(S(x: y)) выводима из спецификации, т. е.

$$P(x) \,\&\, Q(x, y) \Rightarrow L(S(x{:}\ y))$$

Тогда предикат A(x: y) корректен относительно спецификации [4, 16].

Переформулируем теорему в виде правила доказательства корректности программы:

$$\textbf{T1:}\ \frac{T(P, Q)(x);\quad \forall y.\ P(x) \,\&\, Q(x, y) \rightarrow L(S(x: y))}{Corr(S, P, Q)(x)}$$

Здесь и далее в правилах условие однозначности оператора опускается, хотя и подразумевается.

2.4.3. Система правил вывода для однозначной спецификации

Допустим, подоператоры B и C имеют спецификации и эти операторы корректны по отношению к своим спецификациям. Приведенные ниже правила применимы только для однозначной спецификации.

LP:
$$\frac{\begin{array}{l} T(P, Q)(x);\ \ SV(P_B, Q_B)(x);\ \ SV(P_C, Q_C)(x); \\ Corr(B, P_B, Q_B)(x);\ \ Corr(C, P_C, Q_C)(x); \\ \forall y, z\ (P(x)\ \&\ Q(x, y, z) \to Q_B(x, y)\ \&\ Q_C(x, z)) \\ P(x) \to P_B(x)\ \&\ P_C(x); \end{array}}{Corr(B(x: y) \parallel C(x: z), P, Q)(x)}$$

LS:
$$\frac{\begin{array}{l} T(P, Q)(x);\ \ SV(P_C, Q_C)(x, z); \\ Corr(B, P_B, Q_B)(x);\ \ \ \forall z.\ Corr(C, P_C, Q_C)(x, z); \\ \forall y, z\ (\ P(x)\ \&\ Q(x, y)\ \&\ Q_B(x, z) \to P_C(x, z)\ \&\ Q_C(x, z, y)\ ) \\ P(x) \to P_B(x) \end{array}}{Corr(B(x: z); C(x, z: y), P, Q)(x)}$$

LC:
$$\frac{\begin{array}{l} T(P, Q)(x);\ \ SV(P_B, Q_B)(x);\ \ SV(P_C, Q_C)(x); \\ Corr(B, P_B, Q_B)(x);\ \ Corr(C, P_C, Q_C)(x); \\ \forall y\ (P(x)\ \&\ Q(x, y)\ \&\ E(x) \to P_B(x)\ \&\ Q_B(x, y)); \\ \forall y\ (P(x)\ \&\ Q(x, y)\ \&\ \neg E(x) \to P_C(x)\ \&\ Q_C(x, y)) \end{array}}{Corr(\textbf{if}\ (E(x))\ B(x: y)\ \textbf{else}\ C(x: y), P, Q)(x)}$$

2.4.4. Система правил для декомпозиции L(S(x: y))

Теорема тождества спецификации и программы сводит доказательство корректности оператора к формуле $P(x)\ \&\ Q(x, y) \Rightarrow L(S(x: y))$. Декомпозиция доказательства этой формулы реализуется для вхождения $L(S(x: y))$. Таким образом, имеется задача доказательства формулы вида $R(x, y) \Rightarrow L(S(x: y))$, где $R(x, y)$ — произвольная посылка. Решением задачи являются правила доказательства формулы для различных видов операторов в позиции оператора $S(x: y)$:

FP:
$$\frac{R(x, y, z) \to L(B(x: y));\ \ R(x, y, z) \to L(C(x: z))}{R(x, y, z) \to L(B(x: y) \parallel C(x: z))}$$

FS:
$$\frac{R(x, y) \to \exists z\ L(B(x: z));\ \ R(x, y)\ \&\ L(B(x: z)) \to L(C(z: y))}{R(x, y) \to L(B(x: z); C(z: y))}$$

FC:
$$\frac{R(x, y)\ \&\ E \to L(B(x: y));\ \ R(x, y)\ \&\ \neg E \to L(C(x: y))}{R(x, y) \to L(\textbf{if}\ (E)\ B(x: y)\ \textbf{else}\ C(x: y))}$$

Приведенные выше правила достаточно просты. Их можно применять многократно для декомпозиции вхождений $L(S(x: y))$ при доказательстве

формул вида: $R(x, y) \to L(S(x: y))$. Таких формул большинство среди посылок в приведенных выше правилах. Однако правило FS использует посылки двух других видов: $R(x, y) \to \exists y\, L(S(x: y))$ и $R(x, y)\ \&\ L(S(x: y)) \to H(x, y)$, где $R(x, y)$ и $H(x, y)$ — произвольные формулы. Необходимо разработать правила для декомпозиции вхождений $L(S(x: y))$ в этих новых видах формул.

EP: $$\frac{R(x, y, z) \to \exists y\, L(B(x: y));\quad R(x, y, z) \to \exists z\, L(C(x: z))}{R(x, y, z) \to \exists y\, L(B(x: y) \parallel C(x: z))}$$

ES: $$\frac{R(x, y) \to \exists z\, L(B(x: z));\quad R(x, y)\ \&\ L(B(x: z)) \to \exists y\, L(C(z: y))}{R(x, y) \to \exists y\, L(B(x: z); C(z: y))}$$

EC: $$\frac{R(x, y)\ \&\ E \to \exists y\, L(B(x: y));\quad R(x, y)\ \&\ \neg E \to \exists y\, L(C(x: y))}{R(x, y) \to \exists y\, L(\textbf{if}\ (E)\ B(x: y)\ \textbf{else}\ C(x: y))}$$

В том случае, если $A(x: y)$ — нерекурсивный вызов предиката, а $P(x)$ — предусловие этого предиката, то логика такого вызова декомпозируется следующим образом:

EB: $$\frac{Corr(A, P, Q)(x);\quad \forall y\, (R(x, y) \to P(x))}{R(x, y) \to \exists y\, L(A(x: y))}$$

Утверждения, вхождение логики в которых находится в левой части формулы, декомпозируются по следующим правилам:

FLP: $$\frac{R(x, y, z)\ \&\ L(B(x: y))\ \&\ L(C(x: z)) \to H(x, y, z)}{R(x, y, z)\ \&\ L(B(x: y) \parallel C(x: z)) \to H(x, y, z)}$$

FLS: $$\frac{R(x, y)\ \&\ L(B(x: z))\ \&\ L(C(z: y)) \to H(x, y)}{R(x, y)\ \&\ L(B(x: z); C(z: y)) \to H(x, y)}$$

FLC: $$\frac{\begin{array}{l} R(x, y)\ \&\ E\ \&\ L(B(x: y)) \to H(x, y); \\ R(x, y)\ \&\ \neg E\ \&\ L(B(C(x: y)) \to H(x, y); \end{array}}{R(x, y)\ \&\ L(\textbf{if}\ (E)\ B(x: y)\ \textbf{else}\ C(x: y)) \to H(x, y)}$$

2.5. Резюме

Метод дедуктивной верификации предикатных программ кардинально отличается от метода Хоара и др. подходов. Он базируется на понятии логики программы. Это метод тотальной корректности невзаимодействующих программ, которые обязаны завершаться, а не частичной, как у Хоара. Используемые правила точнее, чем соответствующие правила Хоара, поскольку последовательный оператор разделен на оператор суперпозиции и параллельный оператор. Нет циклов, вместо них используются рекурсивные процедуры. Разработан простой универсальный механизм обобщения правил для рекурсивных вызовов. Для доказательства завершаемости рекурсивных программ используется функция меры, вставляемая пользователем в код программы. Построить меру проще, чем инвариант цикла. Указателей нет, вместо них используются алгебраические типы. Генерируемый набор формул корректности в целом проще и лучше структурирован, чем для метода Хоара. Однако автоматическое доказательство формул корректности остается чрезвычайно трудоемким и сложным.

Для класса программ языка P, не содержащих гиперфункций и рекурсивных колец более чем из одного предиката, система правил полна, т.е. для каждой программы из этого класса можно построить условия корректности.

3. ГЕНЕРАЦИЯ ФОРМУЛ КОРРЕКТНОСТИ ДЛЯ ОПЕРАТОРОВ ЯЗЫКА P

В данном разделе для каждого оператора представлены его синтаксис и примеры использования; описывается алгоритм генерации формул корректности. Синтаксис описывается на расширенном языке бэкусовских нормальных форм со следующими особенностями:

- Терминальные символы выделены жирным шрифтом;
- [фрагмент] означает возможное отсутствие в синтаксическом правиле заключенного в квадратные скобки фрагмента; фрагмент определяет последовательность терминальных и нетерминальных символов;
- (фрагмент)* определяет повторение фрагмента нуль или более раз; круглые скобки могут быть опущены, если фрагмент состоит из одного символа;
- (фрагмент)+ определяет повторение фрагмента один или более раз.

Более подробно синтаксис описан в работе [3].

3.1. Алгоритм генерации условий корректности для программы в целом

Задачей алгоритма является автоматическое построение условий корректности предикатной программы. На языке P предикатная программа представлена набором определений предикатов. Корректность подобной программы — это корректность каждого определенного предиката. Поэтому задача сводится к построению условий корректности для определения предиката:

$$A(x: y)\ \textbf{pre}\ P(x)\ \{\ S(x: y);\ \}\ \textbf{post}\ Q(x, y)$$

Доказательство формулы тотальной корректности предиката:

$$Corr(A, P, Q, S, P_S, Q_S)(x, x)$$

реализуется построением дерева вывода с применением правил вывода, описанных в разделе 2. Генерируемыми условиями корректности являются листья этого дерева. Истинность формул, являющихся листьями дерева, и истинность правил влечет истинность формулы тотальной корректности.

В процессе вывода могут встретиться два вида формул: это условие тотальной корректности оператора

$$Corr(S, P, Q)(x)$$

и логическая формула

$$A_1\ \&\ A_2\ \&\ \ldots\ \&\ A_n \Rightarrow B$$

где в качестве A_i и B могут выступать как простые логические утверждения, так и логики операторов L(S(x: y)).

В том случае, если формула имеет первый вид, применяется либо система правил для общего случая (Q), либо система правил для случая корректных подоператоров (R). При этом система правил Q более приоритетна, т.е. ее применение осуществляется в первую очередь.

Если же формула имеет второй вид, то необходимо проверить, содержит ли она в себе логики операторов L(S(x: y)). Если не содержит, то это заключительная формула (одно из условий корректности), и она оформляется в виде леммы. Но если в ней присутствует логика, то вывод этого утверждения необходимо продолжить за счет системы правил декомпозиции

L(S(x: y)) (F). Дальнейшая цель вывода — это исключение из формулы логики оператора.

Корректность алгоритма генерации условий корректности – это корректность каждого применяемого правила. Корректность правил доказана в работах [4, 5, 6] и подтверждена также автоматическим доказательством в системе PVS [16]. Проверка корректности реализации алгоритма обеспечивается тестированием. На данный момент корректность реализации алгоритма проверена на двух десятках тестов.

3.2. Оператор суперпозиции

ОПЕРАТОР-СУПЕРПОЗИЦИИ ::= ОПЕРАТОР (**;** ОПЕРАТОР)+

Пример:

```
nat a, b;
a = 0; b = 0;
nat c, d;
c = a + 1; d = b + c
```

Доказательство корректности оператора суперпозиции разбивается на два возможных варианта. В том случае, если первый подоператор – это вызов предиката, то доказательство реализуется применением правила QSB ввиду наличия спецификации у вызова. В других случаях доказательство реализуется применением правила QS.

3.3. Параллельный оператор

ПАРАЛЛЕЛЬНЫЙ-ОПЕРАТОР ::= ОПЕРАТОР (**||** ОПЕРАТОР)+

Пример:

```
nat a, b;
{ a = 0 || b = 0 }
nat c, d;
{ c = a + 1 || d = b + 1 }
```

Доказательство корректности параллельного оператора реализуется применением правила QP. Нетривиальной задачей данного правила является разделение исходного постусловия на конъюнкцию двух постусловий.

$$Q(x, y, z) \cong Q_B(x, y) \,\&\, Q_C(x, z)$$

Изначально необходимо определить состав наборов x, y и z. Набор x – это аргументы параллельного оператора, y и z — это результаты первого и второго подоператора соответственно.

Дальнейшая задача разделения постусловия была решена лишь для частного случая: когда постусловия представлено в конъюнктивной форме.

$$Q(x, y, z) \cong Q_1(x, y) \,\&\, Q_2(x, z) \,\&\, Q_3(x) \,\&\, \ldots$$

В таком случае, в постусловие $Q_B(x, y)$ войдут лишь те конъюнкты, которые содержат в себе только наборы x и y, а в постусловие $Q_C(x, z)$ лишь те, что содержат x и z. Стоит отметить, что конъюнкты, содержащие в себе только набор x, входят в оба постусловия.

В том случае, если постусловие невозможно разделить на две части, а сам оператор не содержит рекурсивных вызовов, доказательство корректности реализуется формулированием двух условий тотальной корректности.

3.4. Оператор присваивания

ОПЕРАТОР-ПРИСВАИВАНИЯ ::= ПЕРЕМЕННАЯ **=** ВЫРАЖЕНИЕ

Пример:

```
nat a;
a = 0
```

Доказательство корректности оператора присваивания разбивается на два возможных варианта. В том случае, если в выражении не встречается вызова предиката, условие тотальной корректности эквивалентно условию частичной корректности, т.к. оператор присваивания, очевидно, завершается.

$$Corr(a := E, P, Q)(x) \cong \forall y\,(P(x) \,\&\, a = E \Rightarrow Q(x, y))$$

Если выражение содержит вызов предиката, то исходный оператор представляется в виде оператора суперпозиции.

a := E(f(x)); ≅ f(x: δ); a := E(δ); ,

где δ – это новая, не использованная ранее переменная. Дальнейшее доказательство реализуется с помощью правила QSB, ввиду наличия спецификации у оператора вызова.

3.5. Групповой оператор присваивания

ГРУППОВОЙ-ОПЕРАТОР-ПРИСВАИВАНИЯ ::=
| СПИСОК-ПЕРЕМЕННЫХ **| = |** СПИСОК-ВЫРАЖЕНИЙ **|**

Пример:

```
nat a, b c;
| a, b, c | = | 0, 1, 2 |;
```

Доказательство корректности группового оператора присваивания реализуется заменой оператора эквивалентным ему параллельным оператором

```
{ a = 0 || b = 1 || c = 2 }
```

и последующим применением правила QP.

3.6. Условный оператор и оператор выбора

УСЛОВНЫЙ-ОПЕРАТОР ::=
if (ВЫРАЖЕНИЕ **)** ОПЕРАТОР **else** ОПЕРАТОР

Доказательство корректности условного оператора реализуется применением правила QC.

ЗАГОЛОВОК-ОПЕРАТОРА-ВЫБОРА ::= **switch (**ВЫРАЖЕНИЕ**)**
ОПЕРАТОР-ВЫБОРА ::=
        ЗАГОЛОВОК-ОПЕРАТОРА-ВЫБОРА
        { ОПЕРАТОР-АЛЬТЕРНАТИВЫ+
          [**default** : ОПЕРАТОР]
        }
ОПЕРАТОР-АЛЬТЕРНАТИВЫ ::=
        case ВЫРАЖЕНИЕ (**,** ВЫРАЖЕНИЕ+) **:** ОПЕРАТОР

Пример:

```
bool a, b, c;
switch (a + b) {

    case 0: c = 1;
    case 1: c = 0;
    case 2: c = 1;
    default: c = 0;
}
```

Доказательство корректности оператора вызова реализуется приведением его к условному оператору

```
if (a + b = 0)
    c= 1;
else if (a + b = 1)
    c= 0;
else if (a + b = 2)
    c= 1;
else
    c= 0;
```

и последующим применением правила QC. В будущем планируется подробное описание всех трансформаций, используемых в генераторе, и формальное доказательство сохранения семантики при трансформации.

3.7. Оператор вызова

ВЫЗОВ-ПРЕДИКАТА ::=
ИДЕНТИФИКАЦИЯ-ПРЕДИКАТА **(** [АРГУМЕНТЫ] **:** РЕЗУЛЬТАТЫ **)**

Пример:

```
nat a;
foo(a - 1: b);
```

В случае если аргументы оператора вызова не содержат вызовов предикатов или переменных предикатного типа, доказательство корректности такого оператора реализуется применением правила RB. Для того чтобы применить данное правило необходимо вначале сформировать предусловие для аргументов и перенести аргументы вызова предиката на вызов его предусловия и постусловия, а в рекурсивном случае еще и на вызов меры.

$P_B(a) \cong a >= 1;$
$[\ P_{foo}(a - 1), Q_{foo}(a - 1, b)\ ],\quad m(a - 1)$

В случае если аргументы оператора вызова содержат вызовы других предикатов, доказательство корректности такого оператора реализуется заменой исходного оператора на оператор суперпозиции и последующим применением правила QSB:

$foo(bar(x): y) \cong bar(x: \delta); foo(\delta: y);$

В том случае, если в аргументах вызова предиката встречается переменная предикатного типа (например, идентификатор другого предиката), то ее корректность должна быть доказана отдельно, в то время как доказательство оператора вызова также производится по правилу QSB (или RB, в случае отсутствия вызовов).

Допустим, в наборе формальных параметров предиката, или в теле предиката, объявлена переменная предикатного типа, и далее производится ее вызов. Будем называть такую ситуацию формальным вызовом предиката.

```
foo(pre p(a) pred(nat a : nat b) post q(a, b) : nat c)
{
    pred(1: c);
}
```

Если в вызове предиката A(x: y) в качестве A используется переменная предикатного типа, то предусловие и постусловие вызаемого предиката берутся из спецификации типа предикатной переменной. Для вызова `pred(1: c)` используется спецификация [p, q].

3.8. Корректность предиката

Доказательство корректности программы реализуется доказательством корректности каждого определяемого предиката.

A(x: y) **pre** P(x) { S(x: y) } **post** Q(x, y)

Доказательство корректности предиката A(x: y) реализуется доказательством корректности оператора S(x: y) относительно спецификации [P(x), Q(x, y)]:

Corr(A, P, Q, S, P, Q)(x, x)

3.9. Алгоритм генерации формул корректности на примере

Дадим иллюстрацию работы алгоритма на примере программы нахождения наибольшего общего делителя двух чисел:

```
GCD(nat a, b : nat c)
pre a >= 1 & b >= 1
{
    if (a = b)
        c = a
    else if (a < b)
        GCD(a, b - a : c)
    else
        GCD(a - b, b : c)
}
post isGCD(c, a, b)
measure a + b;
```

Для удобства генерации формул корректности программы введем обозначения для формул предусловия, постусловия и функции меры:

```
formula P_GCD(nat a, b) = a >= 1 & b >= 1;
formula Q_GCD(nat a, b, c) = isGCD(c, a, b);
formula m(nat a, b : nat) = a + b;
```

Программа представлена условным оператором, следовательно, наша цель — доказать следующее утверждение:

```
Corr(GCD, P_GCD, Q_GCD
      if (a = b)
            c = a
      else if (a < b)
            GCD(a, b - a : c)
      else
            GCD(a - b, b : c),
      P_GCD, Q_GCD
)((a, b), (a, b))
```

Доказательство этого утверждения проводится по правилу **QC**, ввиду отсутствия спецификации у подоператоров. В соответствии с этим правилом формируются две посылки:

```
Corr(
      c := a,
      P_GCD(a, b) & a = b,
      Q_GCD
)((a, b))

Corr(GCD, P_GCD, Q_GCD
      if (a < b)
            GCD(a, b - a : c)
      else
            GCD(a - b, b : c),
      P_GCD(a, b) & a != b,
      Q_GCD
)((a, b), (a, b))
```

Первая посылка — это условие тотальной корректности оператора присваивания c := a, которое распадается на два условия корректности:

P_GCD(a, b) & a = b & c = a ⇒ ∀c Q_GCD(a, b, c)
P_GCD(a, b) & a = b ⇒ ∃c c = a

В данном случае второе утверждение (утверждение о завершаемости оператора присваивания) можно не доказывать, т.к. оно очевидно истинно. Доказательство же второй посылки, по тому же правилу QC, распадается на две новых посылки:

```
Corr(GCD, P_GCD, Q_GCD
      GCD(a, b - a : c),
      P_GCD(a, b) & a != b & a < b,
      Q_GCD
)((a, b), (a, b))

Corr(GCD, P_GCD, Q_GCD
      GCD(a - b, b : c),
      P_GCD(a, b) & a != b & a >= b,
      Q_GCD
)((a, b), (a, b))
```

Далее по правилу RB формируются следующие условия корректности:

P_GCD(a, b) & a != b & a < b ⇒
b – a >= 0 & P(a, b - a) & m(a, b - a) < m(a, b)
P_GCD(a, b) & a != b & a >= b ⇒
a - b >= 0 & P(a - b, b) & m(a - b, b) < m(a, b)

P_GCD(a, b) & a != b & a < b & Q_GCD(a, b - a, c) ⇒ Q_GCD(a, b, c)
P_GCD(a, b) & a != b & a >= b & Q_GCD(a - b, b, c) ⇒ Q_GCD(a, b, c)

В данном случае в правиле RB были опущены первая и последняя посылки. Первая посылка была опущена ввиду того, что оператор вызова является рекурсивным. Вторая посылка была опущена, т.к. однозначность бинарной операции устанавливается формальной семантикой языка P.

4. ТРАНСЛЯЦИЯ НА PVS

4.1 Система PVS

Алгоритм, описанный выше, позволяет автоматически формировать условия тотальной корректности предикатной программы. В дальнейшем эти условия необходимо автоматически доказать, что невозможно без системы автоматического доказательства. PVS является одной из наиболее эффективных систем подобного рода.

Система PVS обладает языком спецификаций высокого уровня. Язык спецификаций – это формальный язык, предназначенный для декларативного описания структуры, связей, свойств данных и способов их преобразований (в отличие от императивных и функциональных языков) без явного упоминания порядка выполняемых действий и использования конкретных значений данных.

Язык спецификаций PVS базируется на классической логике высших порядков. Он имеет развитую систему типов, которая может абсолютно точно представлять требуемую сущность.

Интерактивный модуль доказательства теорем системы PVS имеет обширную систему команд в виде набора мощных примитивных процедур логического вывода. Команды содержат правила логики высказываний, квантификаторов, индукции, подстановок и процедуры логики линейной арифметики.

4.2. Трансляция на PVS

В процессе реализации генератора формул корректности возникла задача – трансляция полученных условия корректности на язык спецификаций системы PVS.

Для предикатной программы на внутреннем представлении и спецификаций (в виде предусловий и постусловий предикатов) генерируется набор теорий с формулами корректности предикатной программы. Для каждого определения предиката строится теория, имя которой совпадает с именем предиката.

Во внутреннем представлении теория кодируется конструкцией МОДУЛЬ. Теория во внутреннем представлении состоит из объявлений типов, формул и утверждений.

```
module P;

   type A = …
   type B = …

   formula a(…) = …
   formula b(…) = …

   lemma …
   lemma …
```

Далее, описание структуры образа произвольной конструкции языка P в соответствующей конструкции языка спецификаций PVS реализуется в следующей форме:

tr(<языковая конструкция>) = <образ конструкции>

4.2.1. Трансляция модуля

Образом модуля в PVS является следующая конструкция:

tr (module A; ...) =

P : THEORY
BEGIN

<объявление переменных>

tr(<описание формул>)
tr(<описание лемм>)

END P

4.2.2. Объявление переменных

tr(<имя типа> <идентификатор>) =
tr(<идентификатор>): VAR tr(<имя типа>)

В системе PVS весьма специфичный способ описание формул. В частности, прежде чем поместить переменную в формальные параметры, ее предварительно можно описать.

```
a: VAR nat
P (a) : …
```

Ввиду этого задача трансляции идентификаторов усложняется. Рассмотрим пример:

```
formula f1(nat a) = …
formula f2(bool a) = …
```

При попытке вынести объявление переменных за пределы формальных параметров формулы возникнет конфликт: у нас будет две переменные с одним именем, но разных типов. Для разрешения подобных конфликтов был реализован т.н. mangling – искажение имен идентификаторов, с целью придания им уникальности.

tr(<идентификатор>) = <идентификатор>_<mangling>

В итоге вот так будет выглядеть блок объявление переменных для данного примера:

```
a_n: VAR nat
a_b: VAR bool
```

4.2.3. Трансляция констант и переменных

Образом констант являются сами константы.

tr (<константа>) = <константа>

В том случае, если в качестве переменной выступает только ее идентификатор, образом будет являться этот же идентификатор, но с учетом искажения.

tr(<идентификатор>) = <идентификатор>_<mangling>

Массив в системе PVS представляет собой функцию от индексов, поэтому, если в качестве переменной выступает элемент массива, то его образ таков:

tr(<выражение>[<список индексов>]) =
tr(<выражение>)(tr(<список индексов>))

Если в качестве переменной выступает элемент поля, то его образ будет следующим:

tr(<выражение>.<имя поля>) = tr(<выражение>) ' <имя поля>

Мультипеременная и мультивыражение отображаются следующим образом:

tr(|<список выражений>|) = (tr(<список выражений>))

4.2.4. Трансляция унарных и бинарных выражений

Для унарной op a и бинарной a op b операции, где op — операция, a a и b — выражения, реализуется отображение:

tr(op a) = tr(op) (tr(a))
tr(a op b) = (tr(a)) tr(op) (tr(b))

Операнды a и/или b обрамляются круглыми скобками после их перевода на язык PVS.

Отображения унарных операций следующие:

tr(^) = ^
tr(+) - унарный плюс отсутствует
tr(-) = -
tr(!) = NOT

Отображения бинарных операций следующие:

tr(*) = *
tr(/) = /
tr(a % b) = rem(b)(a)
tr(+) = +
tr(x **in** a) = member(x, a)
tr(<) = <
tr(>) = >
tr(<=) = <=
tr(>=) = >=
tr(=) = =
tr(!=) = /=
tr(&) = & для логических операндов
tr(a & b) = intersection(a, b) для множеств
tr(a **xor** b) = a XOR b для логических
tr(a **xor** b) = symmetric_difference(a, b) для множеств
tr(a **or** b) = a OR b для логических
tr(a **or** b) = union(a, b) для множеств
tr(=>) = =>
tr(<=>) = <=>

4.2.5. Трансляция кванторных выражений

Образ кванторного выражение определяется следующим образом:

tr(<квантор> <список переменных>. <выражение>) =
    tr(<квантор>) tr(<список переменных>) : tr(<выражение>)

Образы кванторов следующие:

tr(forall) = FORALL
tr (exists) = EXISTS

4.2.6. Трансляция формул и лемм

Образы формул и лемм определяются следующим образом:

tr(
    formula <имя формулы>(
        <описание формальных параметров>
        : <имя типа результата>)

```
        = <выражение>
) =
        <имя формулы> ( tr(<описание формальных параметров>) )
        : tr(<имя типа результата>) = tr(<выражение>)

tr(
        <тип параметра 1> <идентификатор 1>
        , <тип параметра 2> <идентификатор 2>
        , …
) =
        tr(<идентификатор 1>), tr(<идентификатор 2>), …

tr( lemma <выражение> ) = L<номер леммы>: LEMMA
tr(<выражение>)
```

ЗАКЛЮЧЕНИЕ

В работе описан подход, позволяющий доказывать тотальную корректность предикатных программ. Построена система правил вывода условий корректности. Ранее правила, входящие в эту систему, существовали разрозненно и были описаны в разных работах, а порой и под разными именами. Некоторые правила нуждались в обобщении на рекурсивный случай.

На основании данной системы правил был разработан и реализован генератор формул корректности в системе предикатного программирования. Генератор позволяет автоматически формировать условия корректности для программ с исходным кодом на языке P. В рамках генератора также было реализовано последующее упрощение полученных условий корректности с использованием простейших законов логики и булевой алгебры. Генератор является частью системы предикатного программирования и может быть вызван автоматически.

Реализован конвертор, транслирующий полученные условия корректности на язык спецификаций системы PVS. В рамках конвертора реализована транслитерация, трансляция типов, выражений и искажение идентификаторов с целью придания им уникальности. Конвертор реализован как back-end системы предикатного программирования.

Дальнейшие планы. Необходимо собрать из разных документов полные доказательства правил и доказать те правила, доказательства которых отсутствуют; разработать метод доказательства корректности предикатов со сложной рекурсией, а также для переменных предикатного типа в качестве параметров предикатов; разработать правила для доказательства корректности гиперфункций.

СПИСОК ЛИТЕРАТУРЫ

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.
3. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2010. 42с. (Препр. / ИСИ СО РАН; N 153).
4. Предиктное программирование. Учебное пособие / Под ред. Шелехова В.И. НГУ. Новосибирск, 2009. 111 С.
5. Предиктное программирование. Лекции / Под ред. Шелехова В.И. ИСИ СО РАН. Новосибирск, 2011.
6. Шелехов В.И. Методы доказательства корректности программ с хорошей логикой // Межд. конф. "Современные проблемы математики, информатики и биоинформатики", посвященная 100-летию со дня рождения А.А. Ляпунова. — 2011. — 17с. http://conf.nsc.ru/files/conferences/Lyap-100/fulltext/74974/75473/Shelekhov_prlogic.pdf
7. Кулямин В.В. Методы верификации программного обеспечения // Институт системного программирования РАН. — 2008.
8. S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert. PVS Language Reference.
9. Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. *VCC: A Practical System for Verifying Concurrent C* // LNCS, 5674, P. 1–22. 2009.
10. Ball T., Hackett B., Lahiri S.K., Qadeer S., and Vanegue J. Towards Scalable Modular Checking of User-Defined Properties // LNCS, 6217, P. 1-24. 2010.
11. Ершов Ю.Л., Палютин Е.А. Математическая логика: Учебное пособие для вузов – 2-е изд., испр. и доп. – М.: Наука. Гл. ред. физ.-мат. лит., 1987. – 336 с.
12. Mosses P.D. The Varieties of Programming Language Semantics And Their Uses. Perspectives of System Informatics. Lecture Notes in Computer Science, Springer, 2001, v. 2244, p. 165-190.

13. Shilov N.V. Make Formal Semantics Popular and Useful. Bulletin of the Novosibirsk Computing Center (Series: Computer Science, IIS Special Issue), v.32, 2011, p. 107-126.
14. Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // IEEE Symposium on Logic in Computer Science. 2002.
15. Дейкстра Э. Дисциплина программирования = A discipline of programming. — 1-е изд. — М.: Мир, 1978. — С. 275.
16. http://www.iis.nsk.su/persons/vshel/files/rules.zip
17. Ануреев И.С., Марьясов И.В., Непомнящий В.А. Верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем, Ярославский государственный университет, т. 17, № 3, 2010, с. 5-28.

ПРИЛОЖЕНИЕ

В приложение приведено несколько примеров, демонстрирующих результат работы генератора условий корректности. Примеры имеют следующую структуру: код на языке P, условия корректности на языке P и те же условия, но на языке спецификаций системы PVS.

logic.psrc

Пример, демонстрирующий вывод логики:

```
bar(nat a: nat b)
pre a > 0
{
    nat c, d;
    {
        if (a > 5)
            c = 1 + bar(a - 1)
        else
            c = 0
        || d = 0
    }
    b = c + d + a;
}
post b = 5
measure a;
```

Полученные условия корректности:

```
module bar;
formula P_bar ( nat a : bool ) = a > 0 ;
formula Q_bar ( nat b : bool ) = b = 5 ;
formula m_bar ( nat a : nat ) = a ;
L1: lemma P_bar(a) => false;
L2: lemma P_bar(a) => false;
L3: lemma P_bar(a) => P_bar(a - 1);
L4: lemma P_bar(a) => (exists nat b. Q_bar(b));
L5: lemma (P_bar(a) & Q_bar(b))
    => ((m_bar(a - 1) < m_bar(a)) &
         P_bar(a - 1) & Q_bar(b));
L6: lemma P_bar(a) =>
     ((m_bar(a - 1) < m_bar(a)) & P_bar(a - 1));
L7: lemma (P_bar(a) & Q_bar(b) & (a > 5))
    => (exists nat c. c = (1 + b));
L8: lemma (P_bar(a) & Q_bar(b) & (a <= 5))
    => (exists nat c. c = 0);
L9: lemma (P_bar(a) & Q_bar(b))
    => (exists nat d. d = 0);
L10: lemma (P_bar(a) & (b = (c + d + a)) & (d = 0)
    & (a > 5) & (c = (1 + b))) => Q_bar(b);
L11: lemma (P_bar(a) & (b = (c + d + a)) & (d = 0)
    & (a <= 5) & (c = 0)) => Q_bar(b);
L12: lemma (P_bar(a) & (d = 0) & (a > 5) &
                                    (c = (1 + b)))
    => (exists nat b. b = (c + d + a));
L13: lemma (P_bar(a) & (d = 0) & (a <= 5) & (c = 0))
    => (exists nat b. b = (c + d + a));
```

Теория на языке PVS:

```
bar : THEORY
BEGIN

% Declarations.
a_n, b_n: VAR nat

% Formulas.
```

```
P_bar (a_n) : bool = a_n > 0
Q_bar (b_n) : bool = b_n = 10
m_bar (a_n) : nat = a_n

% Lemmas.
L1: LEMMA
   (P_bar(a_n) AND (a_n > 0)) IMPLIES P_bar(a_n - 1)
L2: LEMMA
    P_bar(a_n) IMPLIES (EXISTS b_n: Q_bar(b_n))
L3: LEMMA
    (P_bar(a_n) AND Q_bar(b_n)) IMPLIES
    ((m_bar(a_n - 1) < m_bar(a_n)) AND
                   P_bar(a_n - 1) AND Q_bar(b_n))
L4: LEMMA
 (P_bar(a_n) AND (a_n <= 0)) IMPLIES
                       (EXISTS b_n: b_n = 0)
L5: LEMMA
    (P_bar(a_n) AND (b_n = (a_n + 10)) AND
     (a_n > 0)) IMPLIES
     ((m_bar(a_n - 1) < m_bar(a_n)) AND
        P_bar(a_n - 1))
L6: LEMMA
    (P_bar(a_n) AND (b_n = (a_n + 10)) AND
       (a_n > 0) AND Q_bar(b_n)) IMPLIES Q_bar(b_n)
L7: LEMMA
    (P_bar(a_n) AND (b_n = (a_n + 10)) AND
      (a_n <= 0) AND (b_n = 0)) IMPLIES Q_bar(b_n)
L8: LEMMA
    (P_bar(a_n) AND (a_n > 0)) IMPLIES
 ((m_bar(a_n - 1) < m_bar(a_n)) AND P_bar(a_n - 1))
L9: LEMMA
    (P_bar(a_n) AND (a_n > 0) AND
  Q_bar(b_n)) IMPLIES (EXISTS b_n: b_n = (a_n + 10))
L10: LEMMA
    (P_bar(a_n) AND (a_n <= 0) AND (b_n = 0))
         IMPLIES (EXISTS b_n: b_n = (a_n + 10))

END bar
```

split.psrc

Пример, демонстрирующий вывод условий корректности для параллельного оператора:

```
formula g(nat a, b: bool) = a < 12 & g(3, 2);
formula f(nat a, b: bool) =
            a < 5 & b < 6 & f(1, 2) & g(a, b);

bar(nat a: nat b, c)
pre a > 0
{
    b = 0 || c = 0
}
post f(b, c) & b = 0;
```

Полученные условия корректности:

```
module Formulas;
formula g ( nat a, nat b : bool ) =
                    (b < 12) & g(3, 2) ;
formula f ( nat a, nat b : bool )=
         (b < 5) & (c < 6) & f(1, 2) & g(b, c) ;

module bar;
formula P_bar ( nat a : bool ) = a > 0 ;
formula Q_bar ( nat b, nat c : bool ) =
                             f(b, c) & (b = 0) ;
L1: lemma (P_bar(a) & (b = 0))=>
 ((b < 5) & f(1, 2) & (b < 12) & g(3, 2) & (b = 0));
L2: lemma P_bar(a) => (exists nat b. b = 0);
L3: lemma (P_bar(a) & (c = 0)) =>
                ((c < 6) & f(1, 2) & g(3, 2));
L4: lemma P_bar(a) => (exists nat c. c = 0);
```

Теории на PVS:

```
Formulas : THEORY
BEGIN
```

```
% Declarations.
a_n, b_n, c_n: VAR nat

% Formulas.
g (a_n, b_n) : bool = (b_n < 12) AND g(3, 2)
f (a_n, b_n) : bool = (b_n < 5) AND
        (c_n < 6) AND f(1, 2) AND g(b_n, c_n)

END Formulas

bar : THEORY
BEGIN

% Declarations.
a_n, b_n, c_n: VAR nat

% Formulas.
P_bar (a_n) : bool = a_n > 0
Q_bar (b_n, c_n) : bool = f(b_n, c_n) AND (b_n = 0)

% Lemmas.
L1: LEMMA
    (P_bar(a_n) AND (b_n = 0)) IMPLIES
     ((b_n < 5) AND f(1, 2) AND (b_n < 12) AND
                          g(3, 2) AND (b_n = 0))
L2: LEMMA
    P_bar(a_n) IMPLIES (EXISTS b_n: b_n = 0)
L3: LEMMA
    (P_bar(a_n) AND (c_n = 0)) IMPLIES
            ((c_n < 6) AND f(1, 2) AND g(3, 2))
L4: LEMMA
    P_bar(a_n) IMPLIES (EXISTS c_n: c_n = 0)

END bar
```

switch.psrc

Пример, демонстрирующий вывод условий корректности для оператора выбора:

```
bar(nat a: nat b)
pre a > 0
{
    switch (a) {
        case 0..3, 7: b = 0
        case 4..5, 8: b = 1
        default: b = 2
    }
}
post b < 10;
```

Полученные условия корректности:

```
module bar;
formula P_bar ( nat a : bool ) = a > 0 ;
formula Q_bar ( nat b : bool ) = b < 10 ;
L1: lemma (P_bar(a) & ((a >= 0) & (a <= 3))
   or (a = 7) & (b = 0)) => Q_bar(b);
L2: lemma (P_bar(a) & ((a >= 0) & (a <= 3))
   or (a = 7)) => (exists nat b. b = 0);
L3: lemma (P_bar(a) & (!(((a >= 0) & (a <= 3)) or
    (a = 7))) & ((a >= 4) & (a <= 5)) or
    (a = 8) & (b = 1)) => Q_bar(b);
L4: lemma (P_bar(a) & (!(((a >= 0) & (a <= 3)) or
    (a = 7)))& ((a >= 4) & (a <= 5)) or (a = 8)) =>
             (exists nat b. b = 1);
L5: lemma (P_bar(a) & (!(((a >= 0) & (a <= 3)) or
             (a = 7)))& (!(((a >= 4) & (a <= 5)) or
              (a = 8))) & (b = 2)) => Q_bar(b);
L6: lemma (P_bar(a) & (!(((a >= 0) & (a <= 3)) or
 (a = 7))) & (!(((a >= 4) & (a <= 5)) or (a = 8))))
                 => (exists nat b. b = 2);
```

Теория на PVS:

```
bar : THEORY
BEGIN
```

```
% Declarations.
a_n, b_n: VAR nat

% Formulas.
P_bar (a_n) : bool = a_n > 0
Q_bar (b_n) : bool = b_n < 10

% Lemmas.
L1: LEMMA
    (P_bar(a_n) AND (((a_n >= 0) AND (a_n <= 3)) OR
       (a_n = 7)) AND (b_n = 0)) IMPLIES Q_bar(b_n)
L2: LEMMA
    (P_bar(a_n) AND (((a_n >= 0) AND (a_n <= 3))
      OR (a_n = 7))) IMPLIES (EXISTS b_n: b_n = 0)

L3: LEMMA
    (P_bar(a_n) AND ((((a_n >= 0) AND (a_n <= 3)) OR
  (a_n = 7))) AND (((a_n >= 4) AND (a_n <= 5)) OR
  (a_n = 8)) AND (b_n = 1)) IMPLIES Q_bar(b_n)
L4: LEMMA
    (P_bar(a_n) AND ((((a_n >= 0) AND (a_n <= 3)) OR
     (a_n = 7))) AND (((a_n >= 4) AND (a_n <= 5)) OR
     (a_n = 8))) IMPLIES (EXISTS b_n: b_n = 1)
L5: LEMMA
    (P_bar(a_n) AND ((((a_n >= 0) AND (a_n <= 3)) OR
    (a_n = 7))) AND ((((a_n >= 4) AND (a_n <= 5)) OR
    (a_n = 8))) AND (b_n = 2)) IMPLIES Q_bar(b_n)
L6: LEMMA
    (P_bar(a_n) AND ((((a_n >= 0) AND (a_n <= 3)) OR
    (a_n = 7))) AND ((((a_n >= 4) AND (a_n <= 5)) OR
    (a_n = 8)))) IMPLIES (EXISTS b_n: b_n = 2)

END bar
```

fibonacci.psrc

Пример предиката, вычисляющего i-ое число Фибоначчи:

```
fib(nat _i: nat _result)
```

```
pre _i >= 0
{
    switch (_i) {
        case 0: _result = 0
        case 1: _result = 1
        default: _result = fib(_i - 1) + fib(_i - 2)
    }
}
post _result >= 0
measure _i;
```

Полученные условия корректности:

```
module fib;
formula P_fib ( nat _i : bool ) = _i >= 0 ;
formula Q_fib ( nat _result : bool ) = _result >= 0
;
formula m_fib ( nat _i : nat ) = _i ;
L1: lemma (P_fib(_i) & (_i = 0) & (_result = 0)) =>
                                    Q_fib(_result);
L2: lemma (P_fib(_i) & (_i = 0)) =>
             (exists nat _result. _result = 0);
L3: lemma (P_fib(_i) & (_i != 0) & (_i = 1) &
                  (_result = 1))=> Q_fib(_result);
L4: lemma (P_fib(_i) & (_i != 0) & (_i = 1)) =>
             (exists nat _result. _result = 1);
L5: lemma (P_fib(_i) & (_i != 0) & (_i != 1)) =>
     ((m_fib(_i - 1) < m_fib(_i)) & P_fib(_i - 1));
L6: lemma (P_fib(_i) & (_i != 0) & (_i != 1) &
          Q_fib(b))=> Q_fib(_result);
L7: lemma (P_fib(_i) & (_i != 0) & (_i != 1) &
       Q_fib(b))=>
     ((m_fib(_i - 2) < m_fib(_i)) & P_fib(_i - 2));
L8: lemma (P_fib(_i) & (_i != 0) & (_i != 1) &
          Q_fib(b) & Q_fib(c)) => Q_fib(_result);
L9: lemma (P_fib(_i) & (_i != 0) & (_i != 1) &
      Q_fib(b)& Q_fib(c) & (_result = (b + c))) =>
          Q_fib(_result);
```

```
L10: lemma (P_fib(_i) & (_i != 0) & (_i != 1) &
         Q_fib(b) & Q_fib(c)) =>
           (exists nat _result. _result = (b + c));
```

Теория на PVS:

```
fib : THEORY
BEGIN

% Declarations.
_i_n, _result_n, b_n, c_n: VAR nat

% Formulas.
P_fib (_i_n) : bool = _i_n >= 0
Q_fib (_result_n) : bool = _result_n >= 0
m_fib (_i_n) : nat = _i_n
% Lemmas.
L1: LEMMA
    (P_fib(_i_n) AND (_i_n = 0) AND (_result_n = 0))
     IMPLIES Q_fib(_result_n)
L2: LEMMA
    (P_fib(_i_n) AND (_i_n = 0)) IMPLIES
        (EXISTS _result_n: _result_n = 0)
L3: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n = 1) AND
     (_result_n = 1)) IMPLIES Q_fib(_result_n)
L4: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n = 1))
         IMPLIES (EXISTS _result_n: _result_n = 1)
L5: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n /= 1))
     IMPLIES ((m_fib(_i_n - 1) < m_fib(_i_n)) AND
         P_fib(_i_n - 1))
L6: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n /= 1) AND
     Q_fib(b_n)) IMPLIES Q_fib(_result_n)
L7: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n /= 1) AND
     Q_fib(b_n)) IMPLIES
      ((m_fib(_i_n - 2) < m_fib(_i_n)) AND
```

```
                                        P_fib(_i_n - 2))
L8: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n /= 1) AND
         Q_fib(b_n) AND Q_fib(c_n)) IMPLIES
                                  Q_fib(_result_n)
L9: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n /= 1) AND
     Q_fib(b_n) AND Q_fib(c_n) AND
        (_result_n = (b_n + c_n))) IMPLIES
                                  Q_fib(_result_n)
L10: LEMMA
    (P_fib(_i_n) AND (_i_n /= 0) AND (_i_n /= 1) AND
     Q_fib(b_n) AND Q_fib(c_n)) IMPLIES
       (EXISTS _result_n: _result_n = (b_n + c_n))
END fib
```

М.С. Чушкин, В.И. Шелехов

ГЕНЕРАЦИЯ И ДОКАЗАТЕЛЬСТВО УСЛОВИЙ КОРРЕКТНОСТИ ПРЕДИКАТНЫХ ПРОГРАММ

Препринт
166

Рукопись поступила в редакцию 10.10.2012
Редактор Т. М. Бульонкова
Рецензент Н.В. Шилов

Подписано в печать 15.11.2012
Формат бумаги 60 × 84 1/16 Объем 2.86 уч.-изд.л., 3.1 п.л.
Тираж 50 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42