ВВЕДЕНИЕ

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, 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]. Описывается трансляция условий корректности на язык спецификаций системы PVS [8] и во внутреннее представление решателя CVC3 [18].

Парадигма предикатного программирования занимает промежуточное положение между парадигмами функционального и императивного программирования. В языке P [3] наряду с выражениями имеются предикаты в форме операторов, в первую очередь оператор присваивания, которого нет в функциональных языках. Кроме того, допускается модификация переменных. В отличие от императивных языков в языке P нет циклов типа **while** и указателей. Эффективность предикатных программ достигается применением при трансляции оптимизирующих трансформаций, заменяющих рекурсивные программы – циклами; объекты алгебраических типов (списков и деревьев) кодируются с помощью массивов и указателей.

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. 3 подробно описан метод дедуктивной верификации, используемый для доказательства корректности предикатных программ, приведен пример применения метода. В разд. 4 представлены алгоритмы генерации формул корректности для программы в целом и для конкретных операторов языка P. В разд. 5 описана трансляция полученных формул во внутреннее представление SMT решателя CVC3. В разд. 6 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] 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) (Рис. 1).

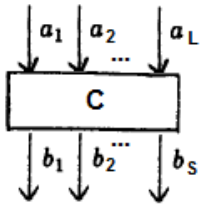


Рис. 1. Команда блок-схемы

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(**P**, **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 (Рис. 2), 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.

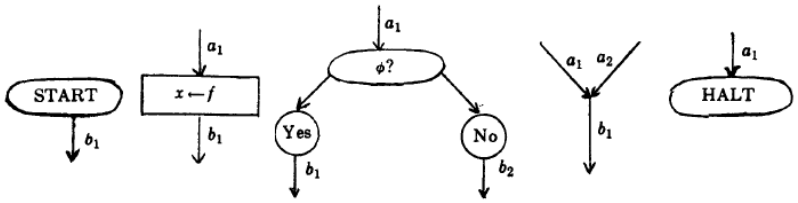


Рис. 2. Команды блок-схемы

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 $f: N \to X$ — 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 (Рис. 3).

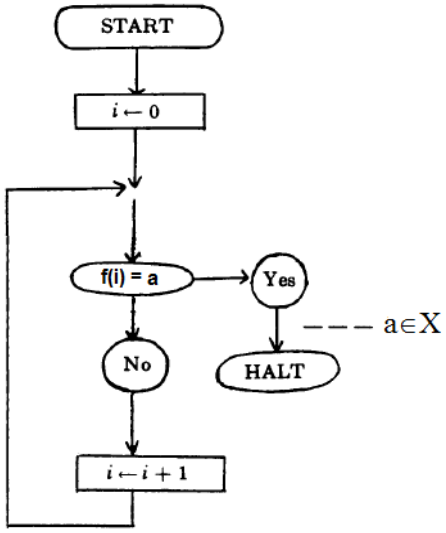


Рис. 3. Алгоритм распознование элементов множества X

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 (Рис. 4). Будем называть такие функции W-функциями.

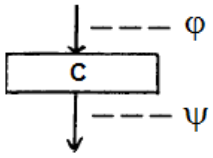


Рис. 4. Доказательство завершаемости программ

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 последних 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 C-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. ПОСТАНОВКА ЗАДАЧИ

Целью работы является разработка и реализация системы верификации программ на языке P. Система верификации разрабатывается как back-end в системе предикатного программирования.

Система предикатного программирования осуществляет трансляцию программы с языка P на язык C++ (Рис. 5). Основной компонентой системы верификации является генератор формул корректности программы, представленной во внутреннем представлении. Сгенерированные формулы должны быть доказаны в автоматическом или автоматизированном режиме.

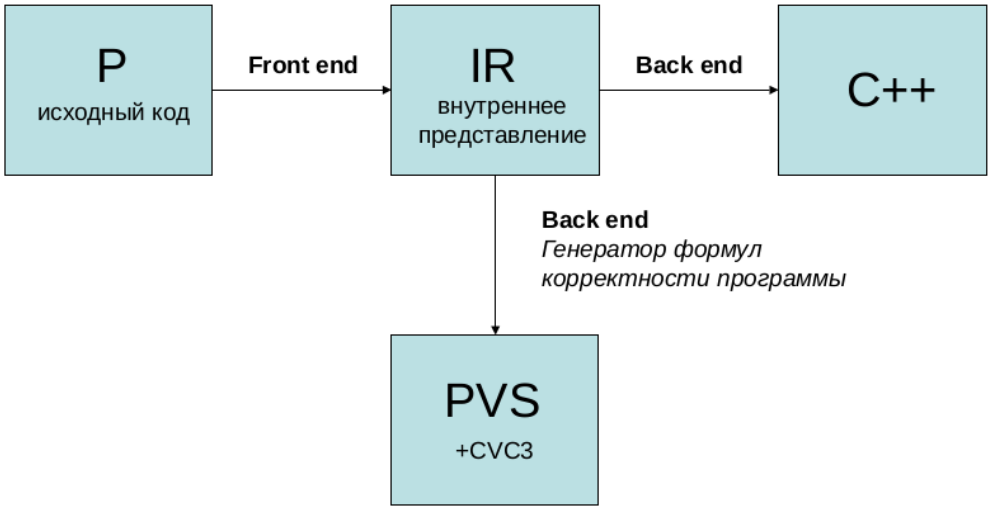


Рис. 5. Система предикатного программирования

Первой задачей работы является реализация автоматической генерации формул корректности для программы с исходным кодом на языке P в системе предикатного программирования.

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.

Такими инструментами являются всевозможные системы доказательства (proover'ы) и решатели (solver'ы). Таких инструментов несколько видов:

- *Системы интерактивного доказательства*. В данных системах пользователь, используя предоставленный системой язык команд, сам строит из этих команд

доказательство утверждения, а система автоматически проверяет его истинность. Примером такой системы служит PVS.

- *Системы автоматического доказательства.* Данные системы способны автоматически строить доказательство. Пример: система Russel.
- *SAT-решатели*. Системы подобного рода решают задачу выполнимости булевых формул. По завершению работы данная система дает ответ — выполнима ли заданная формула и если выполнима, то выдается набор значений, на котором она выполнима.
- *SMT-решатели*. Данный класс систем решает задачи из теорий, представленных в библиотеке SMT-LIB (англ. Satisfiability Modulo Theories), которая включает, например, теорию списков, массивов, линейной арифметики, неинтерпретируемых функций и т.д. Примеры: Yices, Z3, CVC3.

Значительная часть генерируемых формул являются простыми для доказательства. Истинность таких формул хотелось бы проверять автоматически, без использования сложных систем интерактивного доказательства. Инструмент для контроля динамической семантики [21] генерирует условия семантической корректности для программы на языке P. Их истинность также нужно проверить автоматически.

Для автоматической проверки полученных формул было решено использовать решатель CVC3. Вторая цель работы – трансляция полученных формул во внутреннее представление SMT решателя CVC3.

Для доказательства более сложных формул корректности нами было решено использовать систему PVS, обладающую языком спецификаций высокого уровня и мощным блоком доказательства. Третья задача работы - трансляция полученных формул на язык спецификаций системы PVS.

Ни одну работу нельзя считать завершенной, а ее результаты достоверными, если полученный в ходе работы инструмент не прошел практических испытаний. Инструмент для автоматизированной верификации было решено испытать в рамках курса “Формальные методы в описании языков и систем программирования”. Последняя задача работы – провести детальный анализ результатов апробации разработанной системы верификации.

3. КОРРЕКТНОСТЬ ПРЕДИКАТНЫХ ПРОГРАММ

3.1. Дерево вывода

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.

3.2. Логика программы

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

A(x: y) **pre** P(x) { S(x: y); } **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 ≠ 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 ∃y L(B(x: y)).

3.3. Корректность программы

Допустим, программа представлена предикатом A(x: y)

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

со спецификацией [P(x), Q(x, y)], где S(x: y) – оператор. 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.словие завершения исполнения оператора S(x: y) определяется утверждением: ∃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.

3.4. Корректность рекурсивной программы

3.4.1. Корректность рекурсивно определяемого предиката

Допустим, имеется определение рекурсивного предиката A:

A(x: y) **pre** P(x) { K(x: y) } **post** Q(x, y) **measure** m(x);

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

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 <.

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:

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

В итоге, формула тотальной корректности рекурсивного предиката A примет следующий вид:

$$\forall t\ \mathrm{Induct}(A, P, Q)(t) \Rightarrow \mathrm{Corr}(A, P, Q)(t) \qquad (1)$$

3.4.2. Корректность рекурсивного кольца предикатов

Рассмотрим программу языка P, представленную набором определений предикатов:

$A_i(X_i\ x_i\colon Y_i\ y_i)$ $\qquad i = 1, \ldots, N \quad (2)$

pre $P_i\ (x_i)$

{

$S_i(x_i\colon y_i)$

}

post $Q_i\ (x_i, y_i)$

measure $m_i(x_i)$;

P_i и Q_i — это пред- и пост- условия предиката A_i, m_i — функция меры, заданная на аргументах предиката A_i, а $S_i(x_i\colon y_i)$ — оператор, в теле которого могут встречаться вызовы предикатов $A_1, \ldots, A_n$.

3.4.2.1 Прямой метод

В качестве основного метода для формального определения корректности предлагается использовать трансляцию набора определений рекурсивного кольца в эквивалентный набор логических формул. Технически это достигается заменой операторов на их логики.

Более формально это выглядит следующим образом. Для каждого предиката A_i, принадлежащего кольцу (2), определяется следующая формула:

formula $a_i(X_i\ x_i, Y_i\ y_i) =$

$L(S_i(x_i\colon y_i))$

measure $m_i(x_i)$;

Мера в формуле a_i исполняет ту же роль, что и в предикате A_i – задает частичный порядок на множестве входных аргументов. Необходимость в ней возникнет во время доказательства завершаемости рекурсии в формуле a_i.

В полученной формуле вхождение логики раскрывается заменой на логики подоператоров. Раскрытие происходит в соответствие с определение логики оператора (пункт 2.2.).

Особым образом раскрывается логика вызова предиката. Если это вызов предиката, принадлежащего рекурсивному кольцу, то логика заменяется на вызов соответствующей формулы. На примере это выглядит следующим образом:

$L(A_j(x'_j: y'_j)) \rightarrow a_j(x'_j, y'_j)$

В том случае, если предикат не принадлежит рекурсивного кольцу, то логика вызова заменяется на вызов постусловия этого предиката. Аргументы вызова постусловия при этом совпадают с аргументами вызова предиката.

$L(B(u': v')) \rightarrow Q_B(u', v')$

Используя полученные формулы, мы формально определим корректность рекурсивного кольца в виде следующих утверждений:

$P_i(x_i) \Rightarrow (\forall y_i\ a_i(x_i, y_i) \Rightarrow Q_i(x_i, y_i))\ \&\ \exists y_i\ a_i(x_i, y_i)$,

где i = 1, ..., N. Данная формула получаются из формулы для $Corr(A_i, P_i, Q_i)$ подстановкой $a_i(x_i, y_i)$ вместо логики $L(A_i(x_i: y_i))$.

3.4.2.2 Метод связующих формул

В качестве альтернативы предлагается использовать индуктивный метод [20]. Применяя N раз схему индукции с разными индуктивными предположениями ко всему рекурсивному кольцу (2), получаем набор формул, определяющих корректность этой программы:

$$\forall t_1, \ldots, t_N\ Induct(A_1, P_1, Q_1)(t_1) \wedge \ldots \wedge Induct(A_N, P_N, Q_N)(t_N) \Rightarrow$$
$$Corr(A_1, P_1, Q_1)(t_1) \wedge \ldots \wedge Corr(A_N, P_N, Q_N)(t_N),$$

где i = 1, …, N.

Преобразуем индуктивные предположения:

$$Induct(A_k, P_k, Q_k)(t_k) \cong F(t'_k, t_k) \wedge m(t'_k) < m(t_k) \Rightarrow Corr(A_k, P_k, Q_k)(t'_k),$$

где F — это некоторое выражение, связывающее переменные t'_k и t_k. В данном случае t_k – это формальные параметры предиката A_k, а t'_k – его всевозможные фактические параметры. Связь необходимо выразить в явной форме для того, чтобы наложить на вызов условие меры $m(t'_k) < m(t_k)$

В случае рекурсивного кольца из одного предиката в формуле F нет нужды. Цель формулы F – выразить фактические параметры вызова через формальные параметры рекурсивного предиката. Если предикат один, то эти параметры выражаются очевидным образом, и в точности соответствуют фактическим параметрам вызова. Например, для следующего рекурсивного предиката:

```
foo(nat a, b: nat c) {
    foo(a - 1; b - 1: c)
}
```

формула F будет выглядеть следующим образом:

```
formula F(nat a, b: nat a', b')
    = a' = a - 1 & b' = b - 1;
```

В случае рекурсивного кольца более чем из одного предиката построение подобных формул – отдельная задача [20]. В данной работе она не рассматривается. Использование взаимной рекурсии, т.е. рекурсивных колец более чем из одного предиката, – довольно редкое явление в программировании.

3.4.3. Корректность оператора, содержащего рекурсивный вызов

Допустим, оператор B находится в теле предиката A_k, принадлежащего программе (2). Поскольку внутри оператора B может находиться рекурсивный вызов предиката A_i, корректность оператора B определяется формулой:

$$Corr^*(B, P_B, Q_B)(x)$$

$$\cong \forall t_1\ Induct(A_1, P_1, Q_1)(t_1) \wedge \ldots \wedge \forall t_N\ Induct(A_N, P_N, Q_N)(t_N) \Rightarrow Corr(B, P_B, Q_B)(x)$$

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

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

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, доказательство корректности рекурсивного определения предиката A_i представляется следующей формулой:

$$Corr^*(K_i, P_i, Q_i)(x_i)$$

Рассмотрим нерекурсивный предикат A, с телом, представленным оператором K. В таком случае индуктивное предположение отсутствует, т.е. равно **true**, и тогда Corr*(K, P, Q)(x) тождественно Corr(K, P, Q)(x).

3.5. Система правил вывода условий корректности

Используя формулу тотальной корректности, можно автоматически построить формулу корректности для некоторого 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]. В дополнении к этому формальные доказательства корректности правил проведены в системе автоматического доказательства PVS; см. доказательства корректности правил в формате PVS [16]. Описание модели PVS приведено в приложении 1.

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 — параллельный оператор, S — оператор суперпозиции, C — условный оператор, B — оператор суперпозиции или оператор вызова, SB — оператор суперпозиции в общем виде.

3.5.1. Система правил вывода для общего случая

Предположим, что множества переменных x, y, z и v не пересекаются, а множества x и v могут быть пустыми. В таком случае, допустимы следующие правила:

QP: $$\frac{\mathrm{Corr}(B, P, Q_B)(x);\quad \mathrm{Corr}(C, P, Q_C)(x);}{\mathrm{Corr}(B(x: y) \parallel C(x: z), P, \lambda x,y,z.\ Q_B(x, y)\ \&\ Q_C(x, z))(x)}$$

$$\textbf{QS:}\ \frac{P(x) \to \exists z, v\ L(B(x: z, v));\quad \forall z\ Corr(C, \lambda x, z.(\ P(x)\ \&\ \exists z, v\ L(B(x: z, v))\ ), Q)(x, z);}{Corr(B(x: z, v); C(x, z: y), P, Q)(x)}$$

$$\textbf{QSB:}\ \frac{Corr^{*}(B, P_B, Q_B)(x);\quad P(x) \to P^{*}_{B}(x);\quad \forall z\ Corr(C, \lambda x, z.\ (\ P(x)\ \&\ Q_B(x, z)\ ), Q)(x, z)}{Corr(B(x: z, v); C(x, z: y), P, Q)(x)}$$

$$\textbf{QC:}\ \frac{Corr(B, \lambda x.P(x)\ \&\ E(x), Q)(x);\quad Corr(C, \lambda x.P(x)\ \ \&\ \neg E(x), Q)(x);}{Corr(\textbf{if}\ (E(x))\ B(x: y)\ \textbf{else}\ C(x: y))(x)}$$

Используя эти правила, доказательства корректности оператора суперпозиции, параллельного оператора и условного оператора можно свести к доказательству корректности их подоператоров B и C. Эти правила могут применяться до тех пор, пока не будут достигнуты элементарные операторы, такие как оператор присваивания.

Для доказательства корректности рекурсивных программ требуются другие правила.

Допустим, операторы B(x: z) и C(z: y) корректны относительно своих спецификаций [P_B, Q_B] и [P_C, Q_C]. В правилах, описанных ниже, если подоператор B (или C) является рекурсивным вызовом, то посылка $Corr^{*}(B, \ldots)$ (или $Corr^{*}(C, \ldots)$) опускается, а $P^{*}_{B}(x)$ (или $P^{*}_{C}(x)$) заменяется на $P^{*}_{B}(x)\ \&\ m(x) < m(z)$ (или $P^{*}_{C}(x)\ \&\ m(x) < m(z)$), где множеством z обозначаются аргументы предиката B(или C). Если же подоператор B(или C) не является рекурсивным вызовом, то $Corr^{*}$ и P^{*} обозначают просто Corr и P.

$$\textbf{RP:}\ \frac{Corr^{*}(B, P_B, Q_B)(x);\quad Corr^{*}(C, P_C, Q_C)(x);\quad P(x) \to P^{*}_{B}(x)\ \&\ P^{*}_{C}(x);\quad \forall y, z\ (Q_B(x, y)\ \&\ Q_C(x, z) \to Q(x, y, z))}{Corr(B(x: y)\ ||\ C(x: z), P, Q)\ (x)}$$

$Corr^*(B, P_B, Q_B)(x)$; $\forall z\ Corr^*(C, P_C, Q_C)(x, z)$;

$P(x) \rightarrow P^*_B(x)$; $\forall z, v\ (P(x) \& Q_B(x, z, v) \rightarrow P^*_C(x, z))$;

RS: $\dfrac{\forall z, v, y\ (P(x) \& Q_B(x, z, v) \& Q_C(x, z, y) \rightarrow Q(x, y))}{Corr(B(x: z, v); C(x, z: y), P, Q)(x)}$

$Corr^*(B, P_B, Q_B)(x)$; $Corr^*(C, P_C, Q_C)(x)$;

$P(x) \& E \rightarrow P^*_B(x)$; $P(x) \& \neg E \rightarrow P^*_C(x)$;

$\forall y\ (P(x) \& E \& Q_B(x, y) \rightarrow Q(x, y))$;

RC: $\dfrac{\forall y\ (P(x) \& \neg E \& Q_C(x, y) \rightarrow Q(x, y))}{Corr(\textbf{if}\ (E(x))\ B(x: y)\ \textbf{else}\ C(x: y), P, Q)(x)}$

Описанное ниже правило – это специализация правила RS. Вызов предиката B(x: z) может быть записан как z = B(x). Таким образом, мы можем использовать конструкцию C(B(x): y) вместо оператора суперпозиции B(x, z); C(z: y).

$\forall z\ Corr^*(C, P_C, Q_C)(z)$;

$P(x) \rightarrow P_B(x) \& P^*_C(B(x))$;

$\forall y\ (P(x) \& Q_C(B(x), y) \rightarrow Q(x, y))$;

RB: $\dfrac{\forall x, z_1, z_2\ P_B(x) \& L(B(x: z_1)) \& L(B(x: z_2)) \rightarrow z_1 = z_2;}{Corr(C(B(x): y), P, Q)(x)}$

3.5.2. 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 спецификацииP(x), Q(x, y)] определяются, соответственно, формулами:

$$\forall y_1, y_2.\ P(x) \,\&\, L(S(x\colon y_1)) \,\&\, L(S(x\colon y_2)) \Rightarrow y_1 = y_2$$

$$T(P, Q)(x) \ \cong\ P(x) \Rightarrow \exists y\ Q(x, y)$$

$$SV(P, Q)(x) \ \cong\ \forall y_1, y_2.\ P(x) \,\&\, Q(x, y_1) \,\&\, Q(x, y_2) \Rightarrow y_1 = y_2$$

Теорема тождества спецификации и программы. Рассмотрим предикат A(x: y) со спецификацией [P(x), Q(x, y)] и телом, представленным 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.

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

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

Переформулируем теорему в виде правила доказательства корректности программы:

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

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

3.5.3. Система правил вывода для однозначной спецификации

Допустим, подоператоры B и C имеют спецификации и эти операторы корректны по отношению к своим спецификациям. Множества переменных x, y и z не пересекаются, а множество x может быть пустым. Приведенные ниже правила применимы только для однозначной спецификации.

$$\textbf{LP:}\ \frac{\begin{array}{l} T(P, Q)(x);\quad SV(P_B, Q_B)(x);\quad SV(P_C, Q_C)(x); \\ Corr^*(B, P_B, Q_B)(x);\quad Corr^*(C, P_C, Q_C)(x); \\ \forall y, z\ (P(x) \,\&\, Q(x, y, z) \rightarrow Q_B(x, y) \,\&\, Q_C(x, z)) \\ P(x) \rightarrow P^*_B(x) \,\&\, P^*_C(x); \end{array}}{Corr(B(x\colon y) \,||\, C(x\colon z), P, Q)(x)}$$

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

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

3.5.4. 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:
$$\frac{R(x, y, z) \rightarrow L(B(x: y));\quad R(x, y, z) \rightarrow L(C(x: z))}{R(x, y, z) \rightarrow L(B(x: y)\ ||\ C(x: z))}$$

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

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

Приведенные выше правила достаточно просты. Их можно применять многократно для декомпозиции вхождений $L(S(x: y))$ при доказательстве формул вида: $R(x, y) \rightarrow L(S(x: y))$. Таких формул большинство среди посылок в приведенных выше правилах. Однако правило FS использует посылки двух других видов: $R(x, y) \rightarrow \exists y\ L(S(x: y))$ и $R(x, y)\ \&\ L(S(x: y)) \rightarrow$

H(x, y), где R(x, y) и H(x, y) — произвольные формулы. Ниже приведены 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: $$\frac{R(x, y) \,\&\, E \to \exists y\, L(B(x: y));\quad R(x, y) \,\&\, \neg E \to \exists y\, L(C(x: y))}{R(x, y) \to \exists y\, L(\textbf{if}\ (E)\ B(x: y)\ \textbf{else}\ C(x: y))}$$

Пусть A(x: y) — нерекурсивный вызов предиката, а P(x) — предусловие этого предиката. Вхождение логики A(x: y) под квантором существования декомпозируется следующим правилом:

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

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

FLB: $$\frac{P_A(x);\quad R(x, y) \,\&\, Q_A(x, y) \to H(x, y);}{R(x, y) \,\&\, L(A(x: y)) \to H(x, y)}$$

3.6. 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 Р, 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.

4. ГЕНЕРАЦИЯ УСЛОВИЙ КОРРЕКТНОСТИ

В данном разделе будет описан алгоритм работы генератора условий корректности.

4.1. Общая схема

Генерация условий корректности проходит по схеме, представленной на рисунке 6.

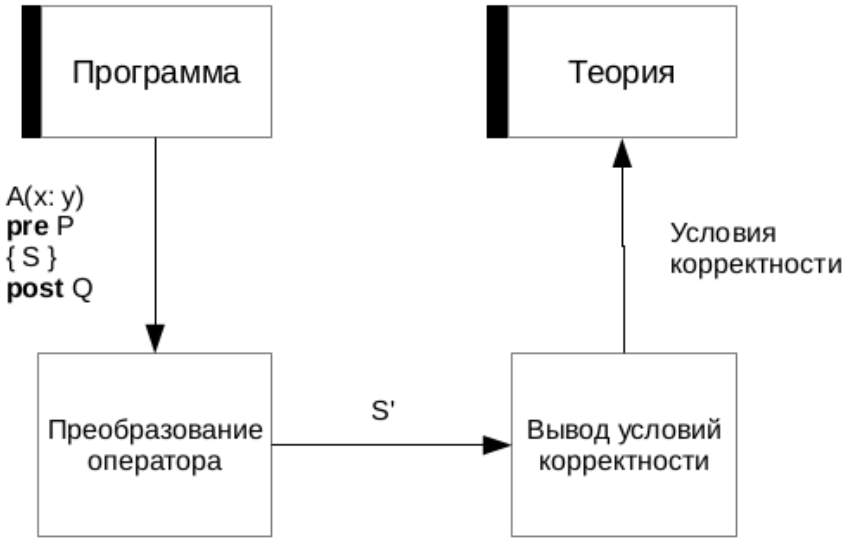


Рис. 6. Общая схема генерации условий корректности

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.

Из программы последовательно извлекаются определения предикатов:

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

Тело предиката, представленное оператором S преобразуется к канонической форме, определяемой языком P_2. Язык P построен из языка исчислений вычислимых предикатов в виде цепочки расширяющихся языков: $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$ [4]. Преобразование к канонической форме – это трансляция с языка P_4 на P_2, существенно упрощающая структуру исходной программы.

Далее для преобразованного оператора происходит вывод условий корректности.

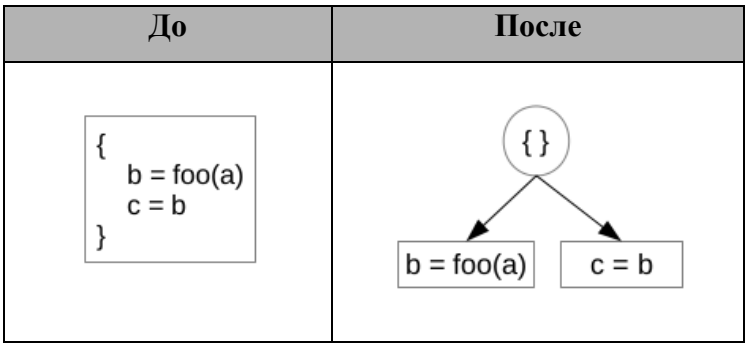
На основе полученных условий корректности строится теория, которая содержит в себе, помимо самих условий, еще и определение необходимых формул и типов, а также информацию об импортируемх теориях.

Далее будет более подробно описаны две задачи, возникшие во время генерации условий корректности: преобразование оператора и вывод условий корректности.

4.2. Преобразование оператора

Преобразование оператора состоит из четырех основных этапов. На первом этапе оператор представляется в виде дерева, узлами которого являются составные операторы, а листьям несоставные. Этот этап условно называется *“разверткой”*. На втором этапе происходит непосредственное *преобразование* дерева. На третьем этапе происходит *упрощение* полученного дерева, с целью сократить количество узлов. И на последнем этапе происходит построение оператора, на основе имеющегося дерева. Последний этап носит условное название *“свертка”*.

4.2.1. “Развертка”

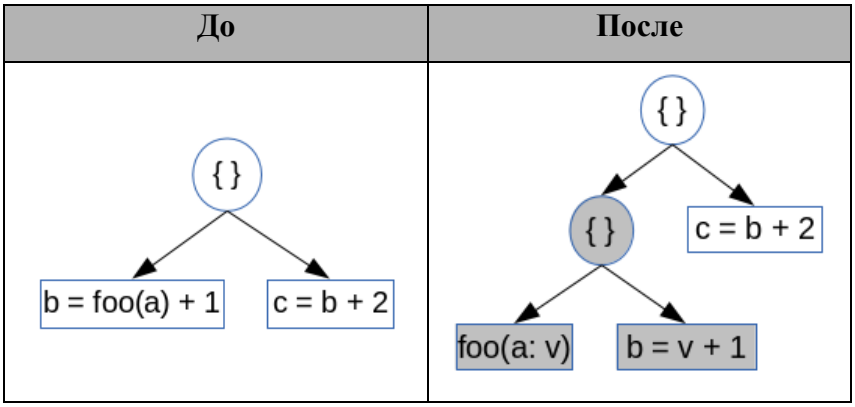


На этапе “развертки” из оператора строится дерево. Узлами этого дерева являются составные операторы. Листьями являются несоставные операторы. Отношение “оператор – подоператор” фиксируется направленным ребром, от оператора к подоператору. Ребра обладают строгим порядком. Этот порядок совпадает с порядком следования подоператоров.

“Развертка” осуществляется рекурсивно. Стартовое дерево содержит лишь один лист – начальный оператор. Затем, если лист является составным оператором, происходит его “развертка”: оператор становится узлом, а его подоператоры листьями. Анологичным образом происходит “развертка” каждого листа, до тех пор, пока все листья не окажуться несоставными операторами.

Составными операторами считаются: условный оператор, операtр суперпозиции, параллельный оператор, оператор выбора (switch).

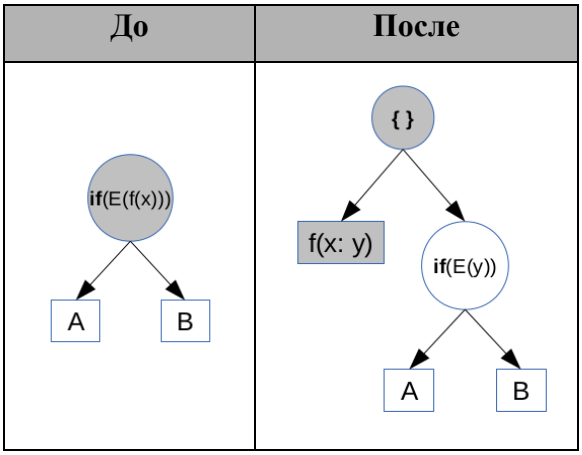
4.2.2. Преобразование



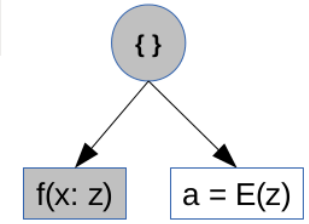
На этапе преобразования происходит модификация исходного дерева. Основные направления модификаций: вынесения вызова функций из выражений и исключение из дерева сложных составных операторов. Сложные составные операторы представляются более простыми операторами. Все эти модификации формально описаны рядом правил.

Правила для модификации вынесения вызова имеют схожую структуру. Корнем исходного дерева является оператор, содержащий в себе вызов функции. Строится дерево, корнем которого является оператор суперпозиции с двумя подоператорами. Первым подоператором является вызов функции, преобразованный в вызов предиката. Вторым подоператором является исходное дерево, с переменной в корневом узле, вместо вызова функции.

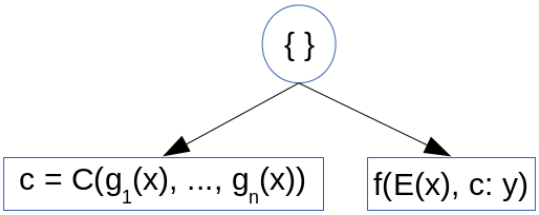
Правила для модификации вынесения вызова выглядят следующим образом. Правило для вынесения вызова из аргумента условного оператра:



Правило для вынесения вызова из правой части оператора присваивания:

До	После
a = E(f(x))	

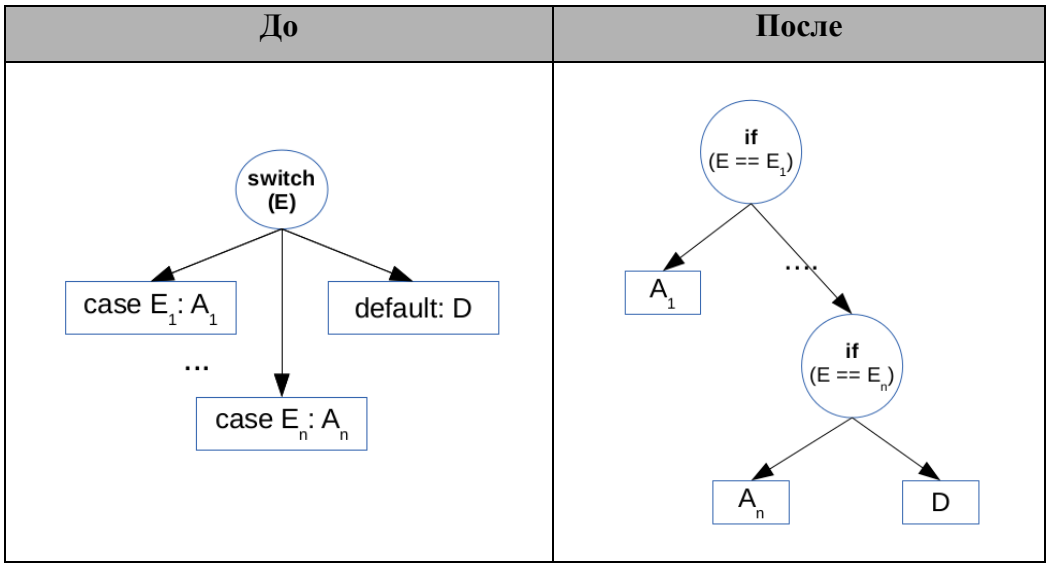
Правило для вынесения вызовов из аргументов оператора вызова предиката имеет более сложную структуру. Рассмотрим вызов предиката f, аргументы которого можно условно разбить на два множества: E и C, где C содержат вызовы предикатов, а E не содержат. В таком случае, необходимо вынести все множество аргументов C. Формальное правило выглядит следующим образом:

До	После
f(E(x), C(g_1(x), ..., g_n(x)): y)	

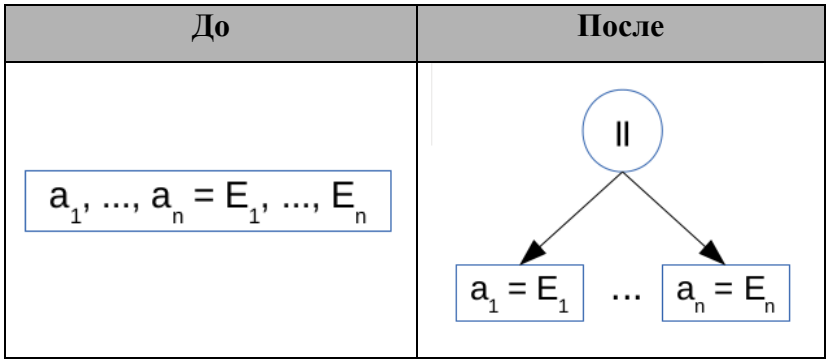
В дальнейшем первый подоператор будет приведен к параллельному оператору. Это реализуется правилом для оператора мультиприсваивания (см. ниже).

Правил для исключения сложных составных операторов всего два. Сложными составными операторами являются оператор выбора (switch) и оператор мультиприсваивания.

Оператор выбора представляется в виде последовательности условных операторов (if - else). Это реализуется следующим правилом:

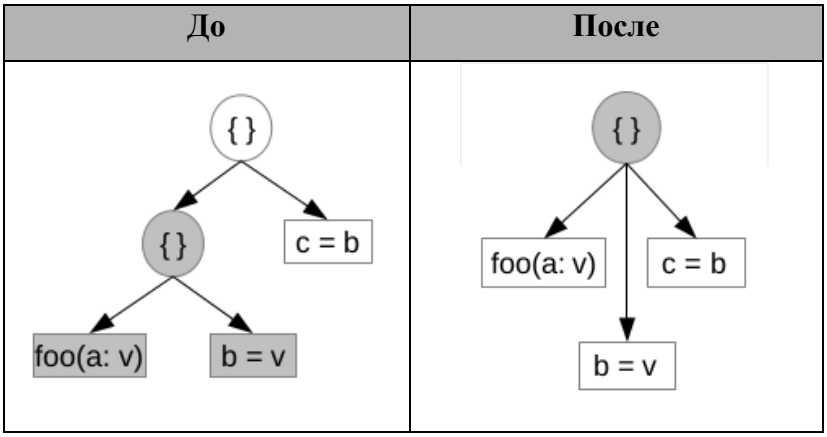


Оператор мультиприсваивание представляется в виде параллельного оператора с присваиваниями в качестве подоператоров. Это выражено в следующем правиле:



После преобразования дерева, в качестве его узлов могут выступать лишь три оператора: условный оператор, оператор суперпозиции и параллельный оператор.

4.2.3. Упрощение

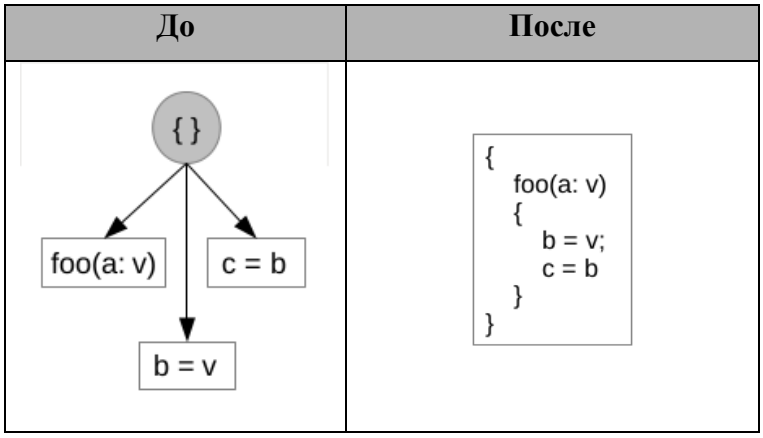


На этапе упрощения происходит сокращения количества узлов в дереве. Это достигается за счет объединения схожих узлов. Два узла схожи, если операторы, соответствующие им,

одинаковы. На данный момент одинаковыми считаются оператор суперпозиции и параллельный оператор.

Объединение происходит следующим образом. Если некоторый узел A является потомком узла B, и эти узлы схожи, то потомки узла A становятся потомками узла B, с сохранением порядка следования подоператоров. Узел A при этом удаляется.

4.2.4. “Свертка”



На этапе “свертки” из полученного дерева строится оператор. Это реализуется “сворачиванием” дерева сверху вниз, от листьев к корню. Процесс начинается от узлов, потомками которых являются лишь листья.

Основной задачей является “свертка” дерева, у которого есть лишь один узел и листья, в качестве потомков этого узла. Результатом свертки такого дерева является лист.

Если в качестве узла выступает условный оператор, то “свертка” происходит очевидным образом: создается лист с условным оператором, подоператорами которого являются потомки узла.

“Свертка” оператора суперпозиции чуть менее очевидна. Имеется узел, которым является оператор суперпозиции. У него есть n потомков: A_1, ..., A_n. Необходимо, чтобы у оператора суперпозиции было лишь два подоператора. В таком случае, результатом “свертки” будет следующий оператора: $\{A_1; \{A_2; ...\{A_{n-1}; A_n\}\}\}$.

“Свертка” параллельного оператора аналогична “свертке” оператора суперпозиции.

4.2.5. Резюме

Для преобразованного исходного оператора верны следующие утверждения. В нем не может быть операторов, отличных от оператора суперпозиции, параллельного оператора, условного оператора и оператора вызова. В нем не может быть “пустых” операторов

суперпозиции и параллельных операторов. У параллельного оператора и оператора суперпозиции могут быть лишь два подоператора. Никакое выражение не может содержать в себе вызов функции.

В будущем необходимо исследовать информационные связи между подоператорами операторов суперпозиции. В случае отсутствия информационной связи оператор суперпозиции нужно преобразовать к параллельному оператору.

4.3. Вывод условий корректности

После того, как тело предиката будет приведено к канонической форме, начинается вывод условий корректности. В данном разделе будет описана структура генератора: иерархия классов внутри генератора, общая схема работы генератора. Будет рассмотрен вопрос корректности генератора.

4.3.1. Иерархия классов

Генератор в процессе вывода оперирует рядом классов. Классы в генераторе организуют иерархию, представленную на рисунке 7.

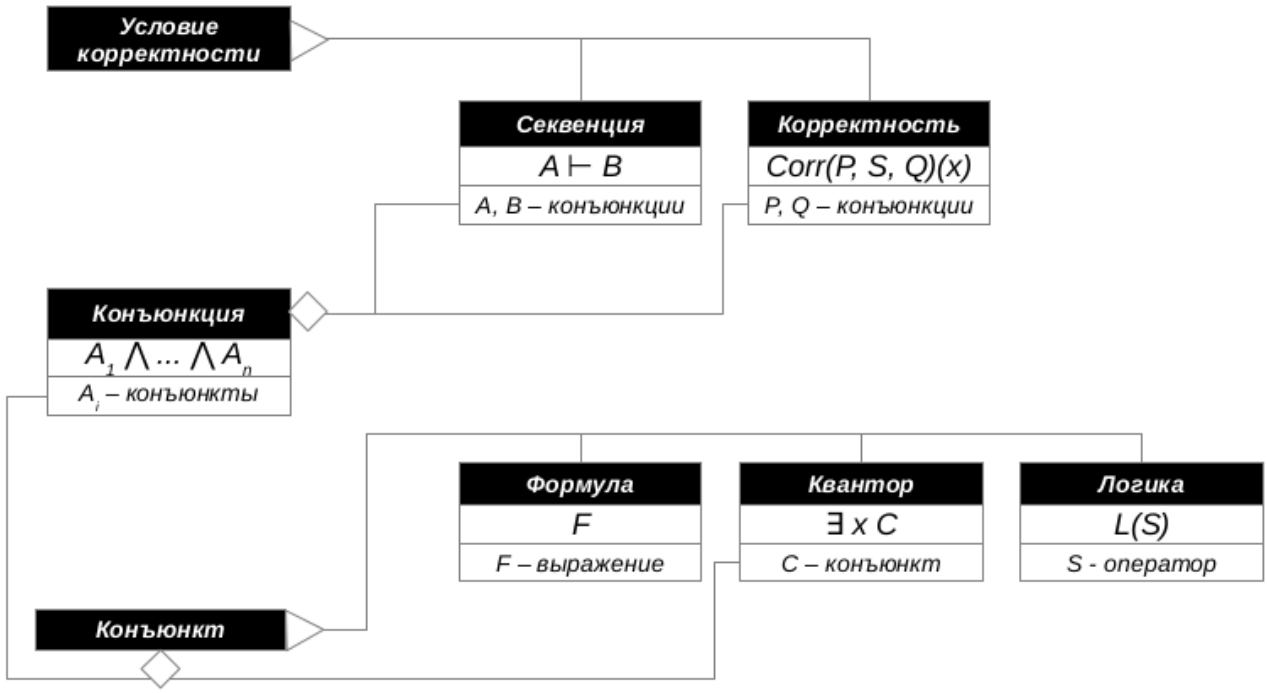


Рис. 7. Иерархия классов

Простейшим классом является “Конъюнкт”.

“Конъюнкт” имеет трех потомков. Первый потомок – это “Формула”, “Конъюнкт”, который является обычным булевым выражением. Второй потомок – это “Квантор”,

“Конъюнкт”, над которым установлен квантор существования. И последний потомок – это “Логика”, “Конъюнкт”, являющийся логикой некоторого оператора.

Не базе “Конъюнктов” строится “Конъюнкция”. “Конъюнкция” – самый важный класс в иерархии. Для “Конъюнкции” реализованы следующие операции: отрицание, импликация, разделения конъюнктов на разные группы по аргументам, упрощение, приведение к КНФ. Операции упрощают реализацию правил вывода.

На верхнем уровне иерархии находится “Условие корректности”. Именно “Условиями корректности” оперирует генератор в процессе вывода. “Условие корректности” имеет двух потомков. Первый потомок – это “Секвенция”. Вообще, секвенция – это термин из теории исчисления предикатов. В рамках данной работы его можно понимать как импликацию. У “Секвенции” два атрибута – левая и правая части, которые являются “Конъюнкциями”.

Второй потомок “Условия корректности” – это “Корректность” оператора. У него тоже есть два атрибута – это предусловие и постусловия. Они так же являются “Конъюнкциями”. В соответствие с определением, “Корректность оператора” может быть представлена двумя “Секвенциями”.

4.3.2. Алгоритм построения формул корректности

В начальный момент работы алгоритма имеется единственная *цель*: исходная формула $Corr(P, S', Q)(x)$, которую надо преобразовать в набор формул корректности, не содержащих вхождений функций Corr и логики L. На каждом шаге имеется несколько целей – формул, содержащих Corr или L. Для очередной цели применяется соответствующие правило, заменяющее цель на набор посылок правила. Посылки, содержащие Corr или L, дополняют набор целей. Посылки без Corr и L пополняют набор генерируемых формул корректности. Алгоритм завершается при отсутствии целей.

Построение формул корректности реализуется по следующей схеме, представленной на рисунке 8:

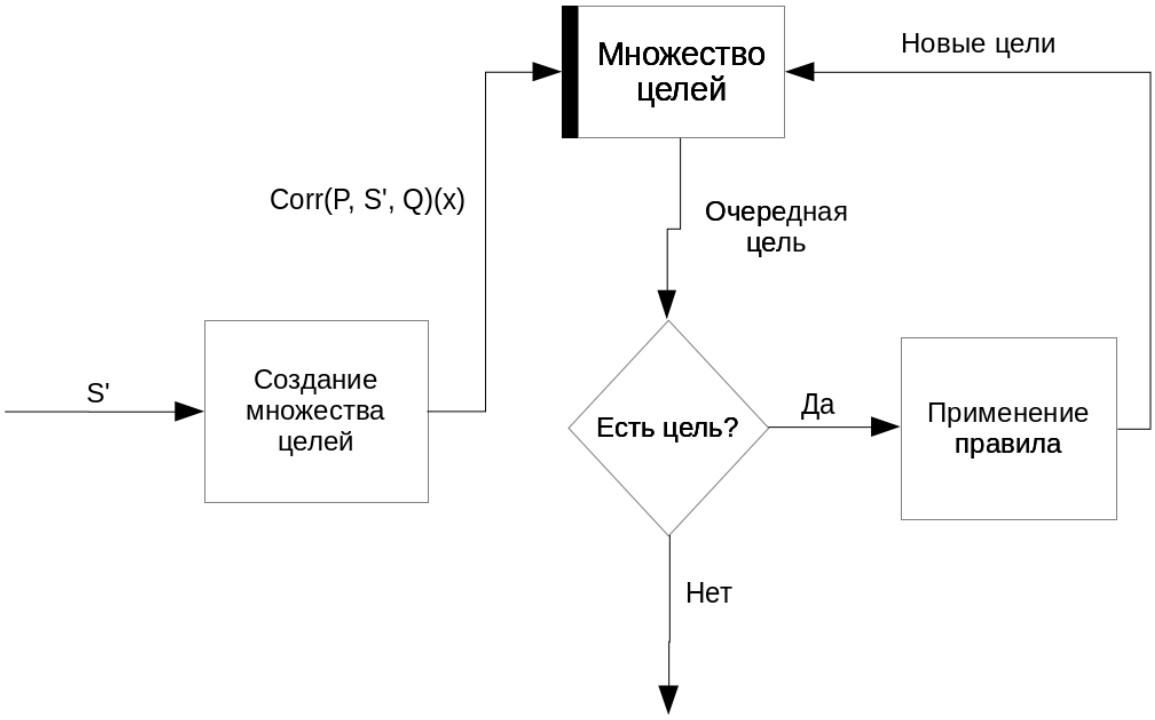


Рис. 8. Схема построения формул корректности

Цель – это условие корректности, которое либо формула Corr, либо содержит логику L. Построение формул корректности начинается с создания множества целей.

В цикле производится попытка извлечь цель из множества. Если это удается сделать, то к цели применяется одно из правил вывода. Результатом применения правила является два множества: множество новых целей и множество формул корректности. Новые цели помещаются в контекст. Построение завершается, если контекст становится пустым, т.е. не остается целей.

Возможен случай, когда построение будет остановлен, но не будет считаться завершенным. Это произойдет, если ни для какой цели в контексте не найдется правила вывода.

Результатом работы алгоритма является множество формул корректности.

4.3.3. Применение правил вывода

В процессе построения формул корректности возникает задача выбора правила вывода, которое необходимо применить к определенному виду “Условия корректности”. Этот выбор можно назвать стратегией вывода. Стратегия вывода схематически представлена на рисунке 9:

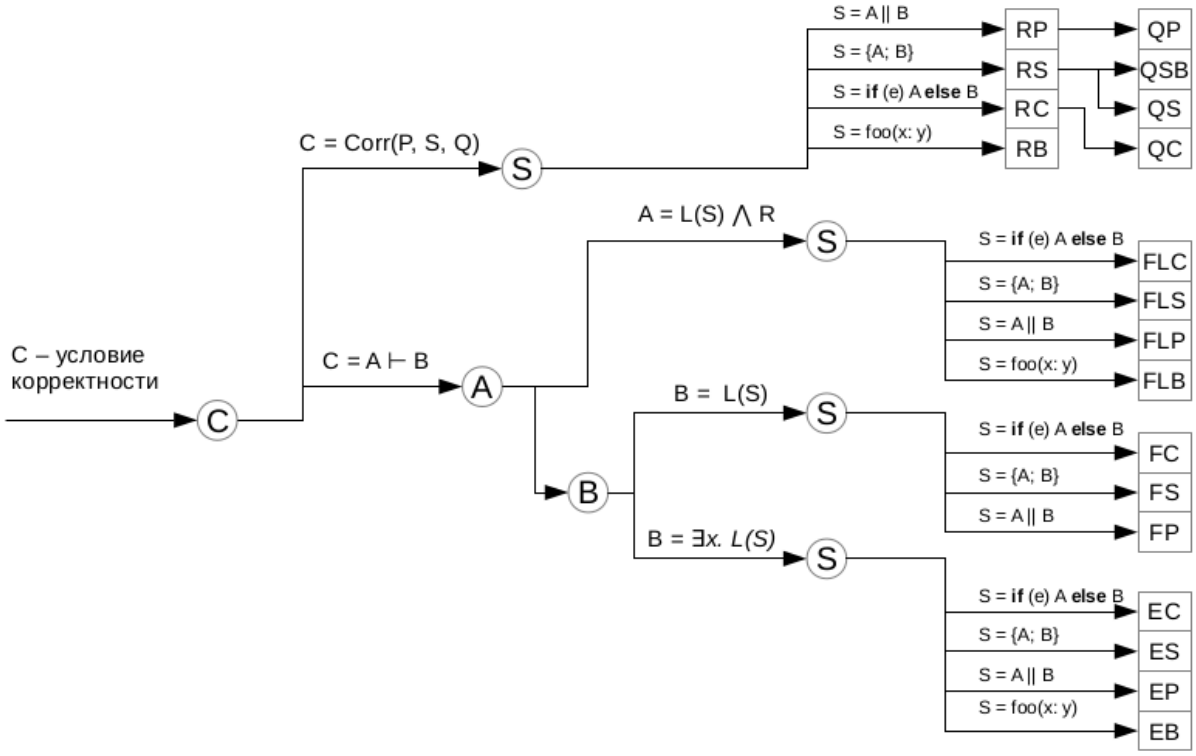


Рис. 9. Стратегия вывода

В том случае, если “Условие корректности” является “Корректностью” некоторого оператора, применяется либо система правил для общего случая (Q), либо система правил для случая корректных подоператоров (R). При этом система правил R более приоритетна, т.е. ее применение осуществляется в первую очередь.

Если же “Условие корректности” является “Секвенцией”, то необходимо проверить, в какой части содержится “Логика”. Если “Логика” содержится в левой части “Секвенции”, то применяется система правил FL.

Если правая часть “Секвенции” является “Логикой”, то необходимо применить систему правил F. Если же правая часть “Секвенции” – это “Логика”, над которой стоит “Квантор” существования, то следует применить систему правил E.

Случай, когда “Секвенция” не содержит “Логики” не рассматривается, т.к. в таком случае “Условие корректности” не является целью.

4.3.4. Резюме

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; см. теории и доказате PVS[16].

Проверка корректности реализации алгоритма обеспечивается тестированием. На данный момент корректность реализации алгоритма проверена на двух десятках тестов.

Иллюстрация работы алгоритма приведена в приложении 2.

5. ТРАНСЛЯЦИЯ НА CVC3

5.1. Система CVC3

CVC3 – это система автоматического доказательства (prover) теорем для задачи SMT (Satisfiability Modulo Theories): для формулы Φ, заданной на языке логики первого порядка, CVC3 пытается определить является ли Φ истинной (или выполнимой) по отношению к одной или нескольким теориям.

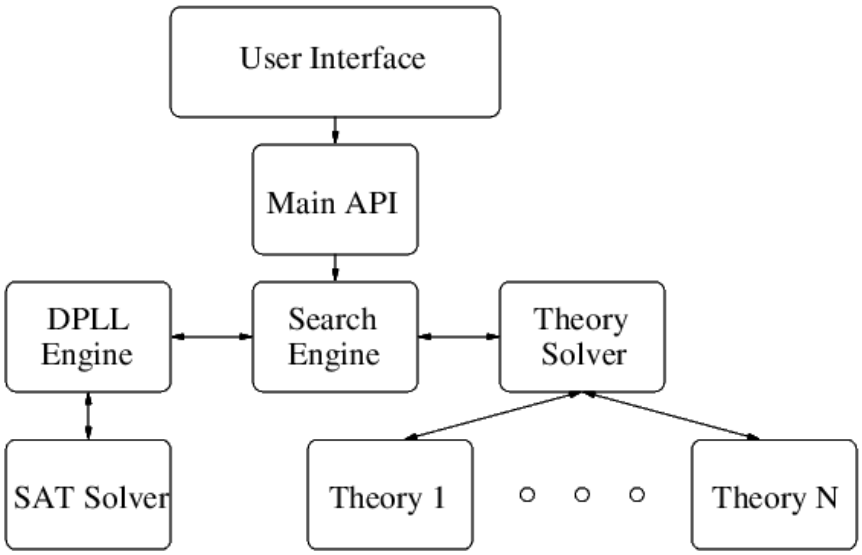


Рис. 10. Система CVC3

Общая схема работы решателя CVC3 представлена на рисунке 10. CVC3 предоставляет несколько пользовательских интерфейсов, включая высокоуровневое API для языков C и C++, интерфейс на основе интерактивных команд и файловый интерфейс. Основное API поддерживает 2 типа операций: создание формул и методы для проверки формул на истинность/выполнимость. Основной механизм вывода реализуется поисковым механизмом (Search engine). Его задача – это связывать DPLL-алгоритм с решателем теорий.

Алгоритм Дэвиса–Патнема–Логемана–Лавленда (DPLL) - это полный алгоритм поиска с возвратом для определения выполнимости булевых формул, записанных в конъюнктивной нормальной форме, т.е. для решения задачи CNF-SAT [23]. Алгоритм представляет собой усовершенствование более раннего алгоритма Дэвиса-Патнема [22].

5.2. Трансляция на CVC3

Программа на языке P состоит из набора модулей, в каждом из которых может содержаться ряд логических утверждений (лемм), которые необходимо проверить на истинность. Логическое утверждение транслируется во внутренне представление решателя CVC3 через примитивы API. Решатель осуществляет проверку на истинность, и в зависимости от результата выставляет лемме статус valid, invalid или unknow. Истинно, ложно и неизвестно соответственно.

Далее, описание структуры образа произвольной конструкции языка P в соответствующей конструкции языка CVC3 реализуется в следующей форме:

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

Образ будет представлен на языке CVC3, однако трансляция осуществляется непосредственно во внутреннее представление. Язык CVC3 описан в работе [19].

Большая часть образов конструкция приведена в приложении 3.

5.2.1. Трансляция булевых выражений

В CVC3 булев тип – это тип функций, а не двухэлементное множество, как в других языках. На практике это означает, что никакое значение не может быть булева типа, т.е. у функции не может быть параметров булева типа.

Образом типа **bool** является битовый вектор из одного элемента:

tr(**bool**) = BITVECTOR(1)

Логические константы транслируются соответственно в массив из одного бита:

tr(**true**) = 1

tr(**false**) = 0

Если выражение используется в операции, параметрами которой являются выражения типа **BOOLEAN**, либо в теле леммы, либо как подкванторное выражение, то его трансляция осуществляется следующим образом:

tr(**<**выражение типа **bool>**) = tr(**<**выражение типа **bool>) = 1;**

Если выражение получено из операции, результатом которой является тип **bool** (например, операции сравнения), то его трансляция осуществляется следующим образом:

tr(**<**выражение типа **bool>**)

= **IF** tr(**<**выражение типа **bool>) THEN** 1 **ELSE** 0 **ENDIF**

5.2.2. Трансляция типов

В CVC3 отсутствуют образы для типов **string**, **char**, **type** и для параметризованных типов.

5.2.2.1 Примитивные типы

Типа **nat** нет в CVC3, поэтому он отображается в подтип типа **INT**:

tr(**nat**) = **SUBTYPE**(**LAMBDA** (x: **INT**): x >= 0)

Трансляция остальных примитивных типов осуществляется следующим образом:

tr(**int**) = **INT**

tr(**real**) = **REAL**

5.2.2.2 Перечисления

Тип **enum** отображается в конструкцию **DATATYPE**:

tr(**enum**(<элемент 1>, ..., <элемент N>)) =

DATATYPE

<уникальный идентификатор> = <элемент 1> | ... | <элемент N>

END

6. ТРАНСЛЯЦИЯ НА PVS

6.1. Система PVS

Алгоритм, описанный выше, позволяет автоматически генерировать формулы тотальной корректности предикатной программы. В дальнейшем эти формулы 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.

6.2. Трансляция на PVS

Для предикатной программы во 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>

Большая часть образов конструкция пиведена в приложении 4.

6.2.1. Трансляция модуля

Образом модуля в PVS является следующая конструкция:

tr (module P; …) =

P : THEORY

BEGIN

<объявление переменных>

tr(<описание формул>)

tr(<описание лемм>)

END P

7. ЗАКЛЮЧЕНИЕ

7.1. Анализ работы системы верификации

В рамках учебного курса “Формальные методы в описании языков и систем программирования” магистрантам ФИТа НГУ было предложено задание по написанию формальной спецификации содержательно сформулированной задачи. Каждый магистрант выбрал одну из 40 предложенных задач. В качестве дополнительного задания для желающих (10 магистрантов) предложено написать предикатную программу и провести доказательство ее корректности в системе автоматического доказательства PVS. Семеро студентов выполнили задание по доказательству некоторых формул корректности в системе PVS; трое из них сумели обнаружить ошибки в спецификации и/или программе в процессе доказательства сгенерированных формул на PVS.

Построение формул корректности для 10 предикатных программ проводилось применением системы верификации, описанной в настоящей работе. Это первый опыт производственной эксплуатации данной системы верификации.

Ранее для системы предикатного программирования был реализован инструмент для контроля динамической семантики [21]. Он генерирует условия семантической корректности для программы на языке P. Условия семантической корректности были добавлены к сгенерированным формулам корректности программы.

В рамках текущего эксперимента было сгенерировано 363 формулы (табл. 1). Из них 260 – это условия тотальной корректности. Остальные 103 – это условия семантической корректности.

№	Семантика	Корректность	Всего
1	4	15	19
2	3	15	18
3	9	29	38
6	37	48	85
7	9	30	39
9	14	37	51
30	3	14	17
31	3	8	11

32	4	15	19
36	17	49	66
Итог			
	103	260	363

Таблица 1. Число формул корректности для 10 задач.

Некоторые из полученных формул тривиальны. Анализ соотношения между общим числом формул и числом тривиальных формул приведен в таблице 2:

№	Трив.	Всего
1	5	19
2	4	18
3	5	38
6	6	85
7	6	39
9	15	51
30	1	17
31	1	11
32	2	19
36	7	66
Итог		
	0.14	1

Таблица 2. Соотношения между общим числом формул
и числом тривиальных формул

В результате анализа было выявлено, что 14% полученных формул являются тривиальными. Это некритически больше число, т.е. большая часть сгенерированных формул вполне адекватна

Интересно также проследить зависимость между объемом программы и числом сгенерированных формул. Для верификации адекватной оценкой объема программы является число операторов. Зависимость числа формул тотальной корректности от числа операторов отображена на рисунке 11:

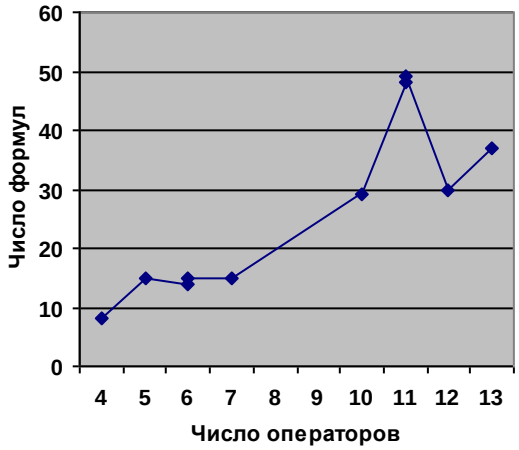


Рис.11. Зависимость числа формул тотальной корректности от числа операторов

По графику видно, что зависимость почти линейная. Зависимость числа формул семантической корректности от числа операторов отображена на рисунке 12:

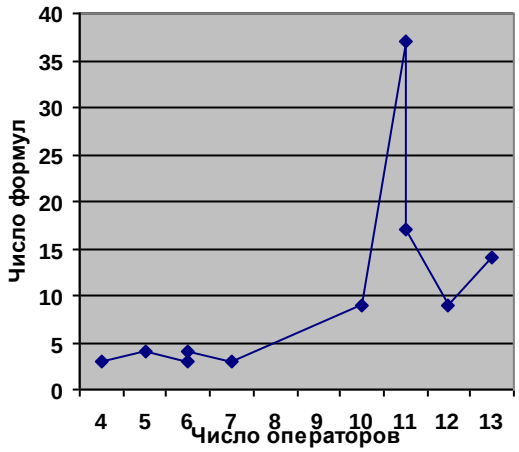


Рис. 12. Зависимость числа формул семантической корректности от числа операторов

По графику также видно, что зависимость почти линейная. В обоих графиках присутствуют точки, исключив которые можно убедиться в линейности зависимости. Конечно, стоит отметить, что для подобного анализа объем выборки маловат.

Число операторов в программах, которые участвовали в тестировании, невелико. Оно не превосходит 13. Однако, количество формул корректности и семантики весьма значительно. В связи с этим от работы решателя CVC3 требуется показать приемлемый результат. Условия семантической корректности желательно проверить полностью автоматически. В таблице 3 приведен анализ работы решателя на наборе формул корректности, сгенерированных для 10 студенческих задач.

№	Семантика		Корректность		Итог	
	CVC3	Всего	CVC3	Всего	CVC3	Всего
1	2	4	7	15	9	19
2	1	3	5	15	6	18
3	7	9	13	29	22	38
6	27	37	35	48	62	85
7	9	9	14	30	23	39
9	13	14	18	37	31	51
30	1	3	2	14	3	17
31	1	3	2	8	3	11
32	1	4	3	15	4	19
36	13	17	18	49	31	66
Итог						
	75	103	117	260	192	363
%						
	0.72	1	0.45	1	0.53	1

Таблица 3. Анализ работы решателя.

В последней строчке таблицы приведено процентное соотношение между общим числом формул и числом формул, которые решатель смог проверить. Решатель смог проверить 72% формул семантической корректности и 45% формул тотальной корректности. Данные результаты можно считать приемлемыми, хотя и неидеальными. Тем более, что статистику портят задачи 1, 2, 30, 31 и 31, т.к. в них используется операция взятия остатка от деления, которая не представлена в CVC3.

Полные результаты апробации системы верификации на 10 студенческих задачах приведены в приложении 5.

7.2. Результаты

В работе описан подход, позволяющий доказывать тотальную корректность предикатных программ. Описаны этапы разработки и реализации системы верификации программ на языке P. Результаты работы следующие:

1) 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. Автором работы было разработано два метода для доказательства корректности рекурсивных программ. Система правил была расширена на рекурсивный случай. Было разработано несколько новых правил.

2) Для системы правил была построена модель на языке спецификаций системы PVS. В системе PVS было проведено доказательство корректности правил. Автором работы была определена значительная часть правил в моделей. Была доказана корректность нескольких правил.

3) 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 построение предусловий для выражений [21], 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.

4) Реализован транслятор формул языка P в язык SMT-решателя CVC3. Формулы транслируется во внутренне представление решателя CVC3 через примитивы API. Решатель осуществляет проверку истинности формулы. Транслятор реализован как back-end системы предикатного программирования.

5) Реализован транслятор формул с языка P на язык спецификаций системы PVS. В рамках транслятора реализована трансляция типов и выражений. Транслятор реализован как back-end системы предикатного программирования.

6) Проведена апробация разработанной системы верификации на студенческих задачах в рамках курса “Формальные методы в описании языков и систем программирования”. Для 10 программ сгенерированы формулы корректности. Определены направления дальнейшего развития системы верификации.

7) На протяжении всей работы велась разработка системы предикатного программирования. Реализовано следующее: линейный порядок на структурах внутреннего представления, поиск/замена подвыражений по шаблону, “ad-hoc” полиморфизма для вызова предикатов, упорядочивание объявлений внутри модуля, механизм импортирования модулей, упорядочивание бинарных выражений, запрет переопределения ранее объявленных идентификаторов, вынесение структурных типов. Реализовано сравнение и вычисление экстремумов для следующих типов: массивов, параметризованных типов, подтипов, объединений. Реализована утилита pretty-printer, отображающая программу на внутреннем представлении в текст на язык P. Реализован вывод типов следующих выражений: конструктора массива, итератора массива, объединения массивов.

В рамках проекта опубликовано 6 работ. Полный список работ представлен в разделе “Публикации”. Работа [1] получила диплом третьей степени.

Дальнейшие планы. Необходимо разработать метод доказательства корректности предикатов с переменными предикатного типа в качестве параметров; разработать правила для доказательства корректности гиперфункций. Необходимо снабдить результирующую теорию подробными комментариями.

ПУБЛИКАЦИИ

1. Чушкин М. С. Дедуктивная верификация программ в системе предикатного программирования // Материалы 52-й юбилейной международной научной студенческой конференции «Студент и научно-технический прогресс»: Программирование / Новосиб. гос. ун-т. — Новосибирск, 2014. — С. 189.
2. Чушкин М. С. Генерация условий корректности предикатных программ с взаимной рекурсией // XIV Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям. Тезисы докладов. — Томск, 2013. — С. 48-49.
3. Чушкин М. С. Генерация условий корректности предикатных программ с взаимной рекурсией // XIV Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям. — Томск, 2013. — 8с. http://conf.nsc.ru/files/conferences/ym2013/fulltext/175121/176936/article.pdf
4. Чушкин М. С. Дедуктивная верификация предикатных программ // XIII Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям. — Новосибирск, 2012. — 7с. http://conf.nsc.ru/files/conferences/ym2012/fulltext/137963/139438/Chushkin_article.pdf
5. Чушкин М. С., Шелехов В.И. Генерация и доказательство формул корректности предикатных программ. — Новосибирск, 2012. — 34с. — (Препр. / ИСИ СО РАН; N 166).
6. Чушкин М. С. Генерация формул корректности предикатной программы // Материалы 50-й юбилейной международной научной студенческой конференции «Студент и научно-технический прогресс»: Программирование и вычислительные системы / Новосиб. гос. ун-т. — Новосибирск, 2012. — С. 24.

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. Версия 0.12 — Новосибирск, 2013. — 52с. http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
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.
18. Clark Barrett, Cesare Tinelli. CVC3. // In Werner Damm and Holger Hermanns, editors, Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07), volume 4590 of Lecture Notes in Computer Science, pages 298-302. Springer, July 2007. Berlin, Germany.
19. The CVC3 User's Manual // http://www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html#user_doc_pres_lang_expr_rec_tup
20. Чушкин М.С. Генерация условий корректности предикатных программ с взаимной рекурсией // XIV Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям, 2013
21. Каблуков И.В., Шелехов В.И. Контроль динмаической семантики предикатной программы // Новосибирск, 2012. — 28с. — (Препр. / ИСИ СО РАН; N 162).
22. Davis, Martin; Putnam, Hilary. A Computing Procedure for Quantification Theory // Journal of the ACM 7 (3): 201–215.
23. Davis, Martin; Logemann, George, Loveland, Donald (1962). A Machine Program for Theorem Proving // Communications of the ACM 5 (7): 394–397

ПРИЛОЖЕНИЕ

Приложение 1. Модель системы правил вывода в PVS

Для доказательства корректности описанных выше правил была разработана модель в системе PVS.

Для операторов языка P определены теории. Внутри теорий определены входные и выходные параметры операторов, определены логики операторов. В модели представлены теории для условного оператора, параллельного оператора, для четырех видов операторa суперпозиции.

В модели определены теория для тотальности оператора и для однозначности спецификации. Определены теории для формулл Corr и Corr*. Последние, помимо непосредственного определения соответствующих формул, содержат еще и определение теоремы тождества спецификации и программы.

Следом, на базе уже определенных теорий, определяются теории для каждого существующего правила. Например, рассмотрим теорию для правила ES. Определение теории начинается с указанием зависимости от теории, в которой определена логика оператора суперпозиции.

```
IMPORTING SuperposStat
```

Сначала в теории для оператора суперпозиции B(x: z); C(z: y) определены типы переменных x, y, z как произвольные неинтерпретированные типы:

```
X: TYPE;
Y, Z: TYPE+;
```

Логика предиката определена тождественной самому предикату. Логики операторов B и C, а так же используемая в правиле формула R, определены как произвольные неитерпретированные предикаты. Первый параметр предиката определяет аргументы, а второй – результаты.

```
R: [X, Y -> bool]
B: [X, Z -> bool]
C: [[X, Z], Y -> bool]
```

Само правило определено в теории в виде следующей леммы.

```
ES: LEMMA
    FORALL (x: X, y: Y):
        (R(x, y) IMPLIES EXISTS (z: Z): B(x, z)) &
        (FORALL (z: Z): R(x, y) & B(x, z)
            IMPLIES EXISTS (y: Y): C((x, z), y))
        IMPLIES
            (R(x, y)
                IMPLIES EXISTS (y: Y): o[X, Y, Z](B, C)(x, y))
```

Большая часть определенных правил была доказана магистрантами НГУ в рамках курса “Формальные методы в описании языков и систем программирования”.

Приложение 2. 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 новых цели:

```
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:

$P\_GCD(a, b) \& a = b \& c = a \Rightarrow \forall c\ Q\_GCD(a, b, c)$

$P\_GCD(a, b) \& a = b \Rightarrow \exists c\ c = a$

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 цели:

```
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:

$P\_GCD(a, b) \& a != b \& a < b \Rightarrow$

$b - a >= 0 \& P(a, b - a) \& m(a, b - a) < m(a, b)$

$P\_GCD(a, b) \& a != b \& a >= b \Rightarrow$

$a - b >= 0 \& P(a - b, b) \& m(a - b, b) < m(a, b)$

$P\_GCD(a, b) \& a != b \& a < b \& Q\_GCD(a, b - a, c) \Rightarrow Q\_GCD(a, b, c)$

$P\_GCD(a, b) \& a != b \& a >= b \& Q\_GCD(a - b, b, c) \Rightarrow Q\_GCD(a, b, c)$

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.

Приложение 3. Образ конструкций языка P в CVC3

3.1. Структуры

Структуры транслируются следующим образом:

tr(**struct**(<определение полей>)) = [# tr(<определение полей>) #]

tr(<определение полей>)
= tr(<имя поля 1>): tr(<тип поля 1>), ..., tr(<имя поля N>): tr(<тип поля N>)

3.2. Объединения

Аналогом типа **union** в CVC3 является тип **DATATYPE**. Трансляция типа **union** осуществляется следующим образом:

tr(**union**(<определение конструктора 1>, ..., <определение конструктора n>)) =
DATATYPE
<уникальный идентификатор> =
<определение конструктора 1> | ... | <определение конструктора n>
END

tr(<определение конструктора>)
= tr(<имя конструктора>)
(tr(<имя поля 1>): tr(<тип поля 1>), ..., tr(<имя поля N>): tr(<тип поля N>))

3.3. Массивы

Трансляция массива осуществляется следующим образом:

tr(**array**(<тип элементов>, <тип размерности>))
= **ARRAY** tr(<тип размерности>) **OF** tr(<тип элементов>)

3.4. Списки и множества

Для трансляции списков и множеств используются бесконечные массивы. Выглядит эта конструкция следующим образом:

type InfinityArray(**type** BaseType) = **struct**(**int** size, **array**(BaceType , **nat**));

Трансляция списков и множеств осуществляется следующим образом:

tr(**list**(<базовый тип>)) = tr(InfinityArray(<базовый тип>))
tr(**set**(<базовый тип>)) = tr(InfinityArray(<базовый тип>))

3.5. Подтипы

Трансляция типа **subtype** осуществляется следующим образом:

tr(**subtype**(<определение параметра>: <выражение типа **bool**>)) =
SUBTYPE(**LAMBDA**(tr(<определение параметра>)):
tr(<выражение типа **bool**>) = 1bin)

3.6. Диапазоны

Диапазон транслируется в эквивалентный ему тип:

tr(<левая граница> ... <правая граница>)
= tr(<левая граница>) .. tr(<правая граница>)

3.7. Предикатные типы

Тип **predicate** транслируется в функциональный тип. Множество выходных ветвей транслируются как **DATATYPE**.

tr(**predicate**(<входные параметры>: <выходные параметры>))
= tr(<входные параметры>) -> tr(<выходные параметры>)

tr(<входные параметры>) = (tr(<имя типа 1>), ..., tr(<имя типа N>))

tr(<выходные параметры>) =

DATATYPE

<уникальный идентификатр> =

tr(<имя ветви 1>)(tr(<имя типа 1.1>), ..., tr(<имя типа 1.N>)) |

...

tr(<имя ветви N>)(tr(<имя типа N.1>), ..., tr(<имя типа N.N>)) |

END

3.8. Литералы

Трансляция числовых и строковых литералов осуществляется очевидным образом:

tr(<число>) = <число>

tr(<строка>) = <строка>

3.9. Унарные операции

Трансляция унарных выражений осуществляется следующим образом:

tr(+<выражение>) = tr(<выражение>)

tr(-<выражение>) = -(tr(<выражение>))

tr(!<выражение>) = ~(tr(<выражение>))

tr(~<выражение>) = ~(tr(<выражение>))

3.10. Бинарные операции

Трансляция бинарных выражений осуществляется следующим образом:

tr(<a>+<b>) = tr(<a>)+tr(<b>)

tr(<a>-<b>) = tr(<a>)-tr(<b>)

tr(<a>*<b>) = tr(<a>)*tr(<b>)

tr(<a>/<b>) = tr(<a>)/tr(<b>)

tr(<a><<b>) = tr(<a>)<tr(<b>)

tr(<a>><b>) = tr(<a>)>tr(<b>)

tr(<a><=<b>) = tr(<a>)<=tr(<b>)

tr(<a>>=<b>) = tr(<a>)>=tr(<b>)

tr(<a>=<b>) = tr(<a>)=tr(<b>)

tr(<a>!=<b>) = !(tr(<a>)=tr(<b>))

tr(<a> and <b>) = tr(<a>) & tr(<b>)

tr(<a> or <b>) = tr(<a>) | tr(<b>)

tr(<a> xor <b>) = BVXOR(tr(<a>), tr(<b>))

tr(<a>=><b>) = tr(<a>)=>tr(<b>)

tr(<a><=><b>) = tr(<a>)<=>tr(<b>)

3.11. Условные выражения

Трансляция условного выражения осуществляется следующим образом:

tr(<if> ? <then> : <else>)

= **IF** tr(<if>) **THEN** tr(<then>) **ELSE** tr(<else>) **ENDIF**

3.12. Компоненты

Трансляция вырезки массива осуществляется следующим образом:

tr(<переменная типа **array**>[<индекс 1>, ..., <индекс N>])

= tr(<переменная типа **array**>)[tr(<индекс 1>)]...[tr(<индекс N>)]

В CVC3 в позиции индекса не может находиться диапазон.

Образ для обращения к полю структуры следующий:

tr(<переменная типа **struct**>.<поле >)

= tr(<переменная типа **struct**>).tr(<поле>)

Трансляция обращения к элементу ассоциативного массива осуществляется следующим образом:

tr(<переменная типа **map**>[<индекс>])

= tr(<переменная типа **map**>)[tr(<индекс>)]

Ввиду кодирования списка бесконечным массивом, трансляция обращения к его элементу осуществляется следующим образом:

tr(<переменная типа **list**>[<индекс>])
= tr(<переменная типа **list**>).elements[tr(<индекс>)]

3.13. Модификаторы

Трансляция модификации массива осуществляется следующим образом:

tr(<переменная типа **array**> **with**
[<индекс 1>: <значение 1>, ..., <индекс N>: <значение N>])
= (...(tr(<переменная типа **array**>)
WITH [tr(<индекс 1>)] := tr(<значение 1>)) ...
WITH [tr(<индекс N>)] := tr(<значение N>)

Аналогично транслируется модификатор для ассоциативного массива.

Трансляция модификатора полей структуры осуществляется следующим образом:

tr(<a> **with**
[<поле 1>: <значение 1>, ..., <поле N>: <значение N>])
= (# tr(<поле 1>) := tr(<значение 1>), ..., tr(<поле N>) := tr(<значение N>),
tr(<немодифицированное поле a i>) :=
tr(<a>). tr(<немодифицированное поле a i>) #)

3.14. Формулы

Объявление формулы транслируется следующим образом:

tr(**formula** <имя формулы>(<параметры>:<тип результата>) = <тело>)
= tr(<имя формулы >):(tr(<параметры>)) -> tr(<тип результата>)
= **LAMBDA** (tr(<параметры >)) : tr(<тело>)

Вызов формулы транслируется следующим образом:

tr(<вызываемая формула>(<аргументы>))
= tr(<вызываемая формула >)(tr(<аргументы>))

3.15. Кванторные выражения

Трансляция кванторных выражений:

tr(<квантор> <переменные>. <выражение>)
= tr(<квантор>)(tr(<переменные>)): tr(<выражение >)

tr(**forall**) = **FORALL**
tr(**exsists**) = **EXISTS**

3.16. Конструкторы

В языке P нет кортежей. Их заменой является конструктор структуры, в котором не указаны имена полей. Но в CVC3 есть кортежи. Трансляция кортежей осуществляется следующим образом:

tr((<значение 1>, ...,< значение N>))
= (tr(<значение 1>), ..., tr(<значение N>))

Трансляция конструктора структуры:

tr((<поле 1>:<значение 1>, ..., <поле N>:< значение N>))
= (# tr(<поле 1>) := tr(<значение 1>), ..., tr(<поле N>) := tr(<значение N>) #)

Трансляция конструктора массива:

tr([<индекс 1>:<значение 1>, ..., <индекс N>:< значение N>])
= (...(<уникальный идентификатор>

WITH [tr(<индекс 1>)] := tr(<значение 1>))

...

WITH [tr(<индекс N>)] := tr(<значение N>)

Трансляция конструктора множества:

tr({<значение 1>, ...,< значение N>})

= (# elements = (...(<уникальный идентификатор>

WITH [1] := tr(<значение 1>))

...

WITH [N] := tr(<значение N>),

size = N #)

Аналогичным образом осуществляется трансляция конструктора для списков.

Приложение 4. Образ конструкций языка P в PVS

4.1. Трансляция констант и переменных

Объявление переменной транслируется следующим образом:

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

Образом констант являются сами константы.

tr (<константа>) = <константа>

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. 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(<=>) = <=>

4.3. 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

4.4. 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>)

Приложение 5. Таблица с анализом работы системы верификации

Задача №	Программа		Семантика					Корректность					Итог				
	Опер.	Строк	Формулы		CVC3			Формулы		CVC3			Формулы		CVC3		
			Трив.	Всего	Valid	Invalid	Всего	Трив.	Всего	Valid	Invalid	Всего	Трив.	Всего	Valid	Invalid	Всего
1	5	23	2	4	1	1	2	3	15	4	3	7	5	19	5	4	9
2	7	27	1	3	0	1	1	3	15	0	5	5	4	18	0	6	6
3	10	48	2	9	4	3	7	3	29	8	5	13	5	38	12	8	22
6	11	56	1	37	27	0	27	5	48	33	2	35	6	85	60	2	62
7	12	69	0	9	9	0	9	6	30	14	0	14	6	39	23	0	23
9	13	76	3	14	11	2	13	12	37	16	2	18	15	51	27	4	31
30	6	21	1	3	0	1	1	0	14	2	0	2	1	17	2	1	3
31	4	24	1	3	0	1	1	0	8	2	0	2	1	11	2	1	3
32	6	27	1	4	0	1	1	1	15	3	0	3	2	19	3	1	4
36	11	55	1	17	8	5	13	6	49	13	5	18	7	66	21	10	31
Итог:																	
			13	103	60	15	75	39	260	95	22	117	52	363	155	37	192
%																	
			0.13	1	0.58	0.14	**0.72**	0.15	1	0.37	0.08	**0.45**	0.14	1	0.43	0.10	**0.53**