Система дедуктивной верификации предикатных программ

Чушкин М.С.
Лаборатория Системного Программирования
Институт Систем Информатики им. А.П. Ершова
г. Новосибирск, Россия
chushkinm@rambler.ru

На базе аппарата формальной логической семантики разработан новый метод генерации формул тотальной корректности рекурсивных программ без циклов и указателей. Для языка предикатного программирования реализована система дедуктивной верификации, генерирующая формулы корректности. Разработана система правил для упрощения генерации формул корректности. Условия совместимости типов, которые трудно проверить при трансляции с языка предикатного программирования добавляются к формулам тотальной корректности. Сгенерированные формулы поступают в SMT-решатель CVC3. Оставшиеся недоказанные формулы оформляются в виде теорий для доказательства в системе PVS. Система верификации успешно использовалась магистрантами НГУ для дедуктивной верификации программ в рамках заданий по курсу "Формальные методы в описании языков и систем программирования".

Keywords — предикатное программирование, тотальная корректность программы, дедуктивная верификация, система автоматического доказательства PVS, SMT-решатель CVC3

I. ВВЕДЕНИЕ

Язык предикатного программирования P определяет класс программ, реализующих функции и не взаимодействующих с внешним окружением [1]. Исполнение невзаимодействующей программы обязано всегда завершаться. В языке P запрещены циклы и указатели.

На базе аппарата формальной логической семантики разработан метод дедуктивной верификации предикатных программ. Предложена формула тотальной корректности предикатной программы относительно своей спецификации, представленной предусловием и постусловием. На базе формулы тотальной корректности разработана система правил [3], позволяющая декомпозировать сложные формулы на более простые и короткие, упрощая тем самым последующее доказательство. Корректность правил доказана в системе PVS [4].

Система дедуктивной верификации включает: генератор формул корректности, транслятор формул во внутреннее представление SMT-решателя CVC3, транслятор формул на язык спецификаций системы PVS. Генератор формул корректности строит формулы корректности программы, используя систему правил.

В процессе трансляции программы для каждого вхождения переменной (или выражения) реализуется проверка соответствия типа переменной (или выражения) типу той позиции, в которую находится переменная. Контроль соответствия типов не всегда можно провести статически при трансляции. При наличии ошибки несоответствия типов исполнение программы может оказаться неверным даже в том случае, когда доказана тотальная корректность программы. По этой причине для гарантии корректности программы необходимо проводить полный контроль типов.

В трансляторе с языка P реализована статическая проверка совместимости типов. Условия совместимости типов, которые транслятор не может проверить статически, добавляются к формулам корректности программы.

Все формулы проходят проверку в SMT-решателе CVC3. Формулы, которые решатель не смог доказать, оформляются в виде теории для системы PVS.

Система верификации успешно прошла апробацию в рамках университетского курса "Формальные методы в программировании".

В разделе 2 представлен обзор работ. В разделе 3 приведено описание ядра языка P и системы предикатного программирования. В разделе 4 описан метод дедуктивной верификации предикатных программ. В разделах 5-10 описывается система верификации: общая схема работы системы, алгоритм генерации формул корректности, стратегия применения правил вывода. Опыт использования системы верификации представлен в разделе 11.

Работа выполнена при поддержке РФФИ, грант № 12-01-00686.

II. Обзор работ

A. *Методы верификации*

Дедуктивная верификация программных систем в основном базируется на логике Хоара [13], определяющей набор правил вывода (аксиом) для троек Хоара. Последовательное применение правил позволяет получить набор формул – условий частичной корректности императивной программы.

Доказательство завершения программы не реализуется в классической логике Хоара. Позже метод был расширен. Для доказательства завершения исполнения циклов и рекурсивных вызовов применяется аппарат вполне упорядоченных множеств [16]. Каждой рекурсивной функции и циклу приписывалась переменная из вполне упорядоченного множества, значение которой должно было строго убывать с каждой итерацией или вызовом.

Метод дедуктивной верификации предикатных программ отличается от метода Хоара. Он реализован для невзаимодействующих программ, исполнение которых должно всегда завершаться. Метод реализует доказательство тотальной корректности программ, а не частичной, как в методе Хоара.

В предикатном программировании нет циклов, вместо них используются рекурсивные предикаты. Верифицировать рекурсивный предикат проще, чем цикл. Построение инварианта цикла – нетривиальная задача. В предикатном программировании роль инварианта выполняет предусловие рекурсивного предиката, которое существенно проще инварианта цикла.

В методе Хоара для доказательства тотальности рекурсивных функций используется аппарат вполне упорядоченных множеств. В нашем подходе используется функция меры, строго убывающая на аргументах рекурсивных вызовов. Доказательство с использованием функции меры проще, чем доказательство с использованием переменных на упорядоченных множествах.

Для доказательства корректности последовательного оператора в логике Хоара используется правило композиции. Правила, используемые в дедуктивной верификации предикатных программ, точнее, чем правила Хоара, поскольку последовательный оператор разделен на оператор суперпозиции и параллельный оператор. Разработан простой универсальный механизм обобщения правил для рекурсивных вызовов. Правила для элиминации квантора существования значительно упрощают формулы корректности.

Классический метод Хоара неприменим для программ с указателями. В предикатном программировании нет указателей, вместо них используются алгебраические типы.

Метод separation logic [14] является расширением классического метода Хоара. В нем допускаются высказывания относительно программ, манипулирующих указателями на структуры данных. Утверждения separation logic описывают состояния, содержащие в себе статически и динамически выделяемую память. Одна из ключевых идей метода – это расширения стандартной логики Хоара оператором *, где P * Q означает, что динамическая память может быть разделена на две непересекающиеся части, для каждой из которых истинно одни из условий P или Q.

В работе Мейера [12] ставится задача доказательства тотальной корректности объектно-ориентированных программ. Интегрированы четыре разных техники. Для описания семантики программы используется composition logic. Для работы со специфическими конструкциями объектно-ориентированного программирования используются negative variables. Для сравнения значений указателей используется alias calculus. Для построения спецификаций сложных структур используется calculus of object structures.

B. *Система верификации WP*

Frama-C является платформой для проведения статического анализа программ, написанных на языке C. Основная задача, решаемая в Frama-C – это поиск ошибок и последующее предупреждение программиста об опасных участках кода. Платформа предоставляет продвинутый механизм для расширения собственной функциональности [17].

Одним из расширений платформы Frama-C является инструмент для дедуктивной верификации WP. WP проводит дедуктивную верификацию программ языка C, в которых соблюдаются принципы контрактного программирования. На вход WP поступает программа, аннотированная спецификациями на языке ACSL [18], для которой генерируется набор условий верификации [19].

Проведены эксперименты по верификации низкоуровневой библиотеки kLIBC с использованием WP и Frama-C. В работе [20] описан процесс построения спецификаций для нескольких функций из библиотек <stdio.h> и <string.h>. Для проверки полученных условий верификации авторы использовали решатели CVC3, Alt-ergo и Z3.

C. *Доказательство условий совместимости типов*

Язык предикатного программирования P не единственный, где условия семантической корректности формализуются в виде логических формул. Аналогичный метод используется в системе PVS, где семантический контроль нетривиальных конструкций осуществляется через доказательство соответствующих формул корректности (TCCs – type correctness conditions).

Семантический анализатор автоматически строит условие совместимости типов. В дальнейшем система пытается автоматически их доказать. Если автоматически это сделать не удается, то пользователь должен доказать эти условия средствами системы PVS [15].

III. Предикатное программирование

Предикатное программирования занимает промежуточное положение между функциональным и императивным программированием. Предикатная программа не взаимодействует с внешним окружением. Точнее, перед началом программы возможен ввод данных и лишь по ее завершению – вывод результатов. Программа обязана всегда завершаться, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Следовательно, программа определяет функцию, вычисляющую по набору аргументов некоторый набор результатов.

A. *Базисные конструкции языка предикатного программирования P*

Программа на языке предикатного программирования P (Predicate programming language) состоит из набора определений предикатов. Определение предиката имеет следующий вид:

```
<имя предиката>
(<список аргументов>: <список результатов>)
pre <предусловие>
{
        <оператор>
}
post <постусловие>
```

Необязательные конструкции предусловие и постусловие являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [7-10]. Для любых значений аргументов, удовлетворяющих предусловию, значения результатов, в случае завершения оператора, должны удовлетворять постусловию.

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>

{ <оператор 1>; <оператор 2> }

<оператор 1> || <оператор 2>

if (<логическое выражение>)
    <оператор 1> else <оператор 2>

<имя предиката>
    (<аргументы>: <результаты>)

<тип> <список имен переменных>
```

В языке P запрещены такие языковые конструкции как циклы и указатели, серьезно усложняющие анализ программы. Полное описание языка P дано в работе [1].

B. *Императивное расширение языка P*

Императивное расширение языка P содержит дополнительные языковые конструкции: прерывание исполнения цикла (**break**), циклы **for** и **while**, метку и оператор безусловного перехода на метку. Семантика циклов **for** и **while** соответствует языку C++.

Эти конструкции возникают в программе в результате проведения трансформаций предикатной программы. Использование этих конструкций в исходной программе недопустимо.

C. *Формальная семантика языка предикатного программирования P*

Язык предикатного программирования P является результатом последовательного расширения цепочки языков [2]:

$$CPP = P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 = P$$

Базовым языком (P_0) является язык исчисления вычислимых предикатов (CCP - Calculus of Computable Predicates). Формальная семантика языка исчисления вычислимых предикатов определяется через операционную и логическую семантику.[11].

Каждая новая конструкция языка P_{i+1} определяется через некоторую композицию на языке P_i. Такой способ построения языка P_{i+1} дает возможность легко построить формальную семантику языка P_{i+1} на базе формальной семантики языка P_i.

D. *Система предикатного программирования*

Система предикатного программирования осуществляет трансляцию программы с языка P на язык C++ (Рис. 1). Синтаксический анализатор производит разбор кода программы на языке P и формирует ее образ во внутреннем представлении. Внутренним представлением является абстрактное синтаксическое дерево программы.

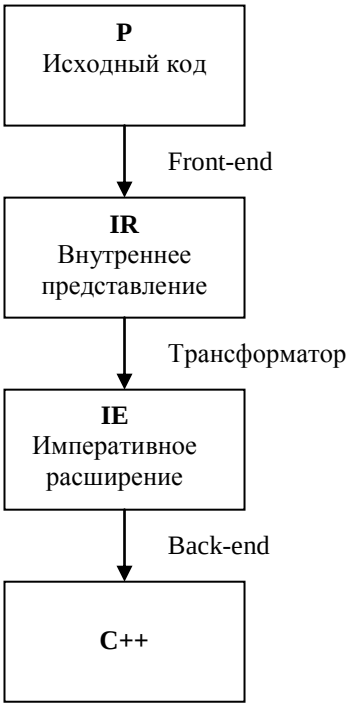


Рис. 1. Система предикатного программирования

К предикатной программы применяется набор оптимизирующих трансформаций**:**

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- открытая подстановка программ
- кодирование алгебраических типов (списков, деревьев) с помощью массивов и указателей.

Результатом трансформаций является программа на императивном расширении языка P. Далее генерируется код на языке C++.

IV. Метод дедуктивной верификации предикатных программ

A. *Логика программы*

Пусть имеется оператор S(x: y), где x и y – непересекающиеся наборы переменных, аргументы и результаты соответственно.

Логика L(S(x: y)) оператора S(x: y) — сильнейший[1] предикат, истинный при завершении его исполнения [3].

В дедуктивной верификации для доказательства истинности свойства X оператора S(x: y), достаточно доказать истинность формулы $L(S(x: y)) \Rightarrow X$.

[1] Предикат A сильнее предиката B, если истинно $A \Rightarrow B$

Корректность логики оператора определяется как согласованность логики с операционной семантикой для любого оператора языка программирования. Логика L(S(x: y)) согласована с операционной семантикой оператора S(x: y), если для любых наборов x и y формула L(S(x: y)) истинна тогда и только тогда, когда исполнение S(x: y) на аргументах x завершается с результатами y. Для языка CCP доказана теорема согласованности операционной и логической семантики [2]. Благодаря механизму построения цепочки языков теорема справедлива и для языка P.

Логика базисных операторов языка P определяется следующим образом:

$$L(B(x: z); C(z: y)) \cong \exists z\, L(B(x: z)) \,\&\, L(C(z: y))$$

$$L(B(x: y) \,||\, C(x: z)) \cong L(B(x: y)) \,\&\, L(C(x: z))$$

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

Для определения логики рекурсивно определяемых программ используется аппарат неподвижной точки [2].

B. *Корректность программы*

Рассмотрим определение предиката следующего вида:

```
A(X x:Y y)
pre P(x)
{ S(x: y) }                    (1)
post Q(x, y)
measure m(x)
```

Спецификация предиката (1) 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)].

Тотальная корректность программы (1) определяется истинностью следующей формулы:

$$Corr(S, P, Q)(x) \cong$$
$$P(x) \Rightarrow (\forall y\, (L(S(x: y)) \Rightarrow Q(x, y))) \,\&$$
$$\exists y\, L(S(x: y)) \qquad (2)$$

Здесь $\forall y\, (L(S(x: y)) \Rightarrow Q(x, y))$ является условием частичной корректности, т.е. условием того, что спецификация истинна при завершении программы.

Подформула $\exists y\, L(S(x: y))$ определяет условие завершение исполнения оператора S(x: y) в соответствии с теоремой тождества операционной и логической семантики [2].

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

C. Корректность рекурсивной программы

Для рекурсивной программы (1) используется следующая схема доказательства по индукции:

<индукционное предположение>

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

Индукционное предположение для программы (1) определяется формулой:

Induct(A, P, Q)(t) $\cong$

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

Функция m, называемая мерой, отображает набор значений аргументов предиката A(x: y) во множество натуральных чисел со стандартным отношением порядка <. Программист должен построить функцию меры, строго убывающую на аргументах рекурсивных вызовов.

D. Система правил вывода формул корректности

Используя формулу тотальной корректности, можно автоматически построить формулу корректности для некоторого оператора S(x: y). Итоговая формула корректности будет длинной и сложной даже для коротких программ; она будет намного 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.

Определяется система правил вывода условий корректности для различных операторов. Доказательства правил приведены в работах [2, 3]. В дополнении к этому формальные доказательства корректности правил проведены в системе автоматического доказательства PVS; см. доказательства корректности правил в формате PVS [4].

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

QP: $\dfrac{\text{Corr}(B, P, Q_B)(x);\quad \text{Corr}(C, P, Q_C)(x);}{\text{Corr}(B(x: y) \parallel C(x: z), P, Q_B(x, y)\ \&\ Q_C(x, z))(x)}$

QS: $\dfrac{P(x) \rightarrow \exists z, v\ L(B(x: z, v));\quad \forall z\ \text{Corr}(C, (P(x)\ \&\ \exists z, v\ L(B(x: z, v))), Q)(x, z);}{\text{Corr}(B(x: z, v); C(x, z: y), P, Q)(x)}$

QSB: $\dfrac{\text{Corr}^*(B, P_B, Q_B)(x);\quad P(x) \rightarrow P^*_B(x);\quad \forall z\ \text{Corr}(C, \lambda x, z.\ (P(x)\ \&\ Q_B(x, z)), Q)(x, z)}{\text{Corr}(B(x: z, v); C(x, z: y), P, Q)(x)}$

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

Используя эти правила, доказательства корректности оператора суперпозиции, параллельного оператора и условного оператора можно свести к доказательству корректности их подоператоров B и C.

В правилах, описанных ниже, если подоператор B является рекурсивным вызовом, то посылка $\text{Corr}^*(B, \ldots)$ опускается, а $P^*_B(x)$ заменяется на $P_B(x)\ \&\ m(x) < m(z)$, где z обозначает аргументы предиката B. Если же подоператор B не является рекурсивным вызовом, то Corr^* и P^* обозначают просто Corr и P. Вхождения Corr^* и P^* для подоператора C трактуются аналогично

RP: $\dfrac{\text{Corr}^*(B, P_B, Q_B)(x);\quad \text{Corr}^*(C, P_C, Q_C)(x);\quad \forall y, z\ (Q_B(x, y)\ \&\ Q_C(x, z) \rightarrow Q(x, y, z));\quad P(x) \rightarrow P^*_B(x)\ \&\ P^*_C(x);}{\text{Corr}(B(x: y) \parallel C(x: z), P, Q)(x)}$

RS: $\dfrac{\text{Corr}^*(B, P_B, Q_B)(x);\quad \forall z\ \text{Corr}^*(C, P_C, Q_C)(x, z);\quad \forall z, v, y\ (P(x)\ \&\ Q_B(x, z, v)\ \&\ Q_C(x, z, y) \rightarrow Q(x, y));\quad \forall z, v\ (P(x)\ \&\ Q_B(x, z, v) \rightarrow P^*_C(x, z));\quad P(x) \rightarrow P^*_B(x);}{\text{Corr}(B(x: z, v); C(x, z: y), P, Q)(x)}$

RC: $\dfrac{\text{Corr}^*(B, P_B, Q_B)(x);\quad \text{Corr}^*(C, P_C, Q_C)(x);\quad P(x)\ \&\ E \rightarrow P^*_B(x);\quad P(x)\ \&\ \neg E \rightarrow P^*_C(x);\quad \forall y\ (P(x)\ \&\ E\ \&\ Q_B(x, y) \rightarrow Q(x, y));\quad \forall y\ (P(x)\ \&\ \neg E\ \&\ Q_C(x, y) \rightarrow Q(x, y))}{\text{Corr}(\textbf{if}\ (E(x))\ B(x: y)\ \textbf{else}\ C(x: y), P, Q)(x)}$

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) как эквивалент оператора суперпозиции B(x, z); C(z: y).

Представленные выше правила принципиально упрощают формулы корректности рекурсивных программ.

RB: $\dfrac{\forall z\ \text{Corr}^*(C, P_C, Q_C)(z);\quad P(x) \rightarrow P_B(x)\ \&\ P^*_C(B(x));\quad \forall y\ (P(x)\ \&\ Q_C(B(x), y) \rightarrow Q(x, y));\quad \forall x, z_1, z_2\ P_B(x)\ \&\ L(B(x: z_1))\ \&\ L(B(x: z_2)) \rightarrow z_1 = z_2;}{\text{Corr}(C(B(x): y), P, Q)(x)}$

Применение вышеописанных правил порождает формулы вида $R(x, y) \Rightarrow L(S(x: y))$, где $R(x, y)$ — произвольная посылка. Ниже даны правила доказательства формулы для различных видов операторов в позиции оператора S(x: y):

FP: $\dfrac{R(x, y, z) \rightarrow L(B(x: y));\quad R(x, y, z) \rightarrow L(C(x: z))}{R(x, y, z) \rightarrow L(B(x: y) \parallel C(x: z))}$

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

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

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

$$\textbf{EP:}\ \frac{R(x) \rightarrow \exists y\ L(B(x: y));\quad R(x) \rightarrow \exists z\ L(C(x: z))}{R(x) \rightarrow \exists y\ L(B(x: y)\ \|\ C(x: z))}$$

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

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

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

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

Приведенная выше система правил позволяет элиминировать вхождение квантора существования. Вхождения логики оператора в левой части формулы, декомпозируются по следующим правилам:

$$\textbf{FLP:}\ \frac{R(x, y, z)\ \&\ L(B(x: y))\ \&\ L(C(x: z)) \rightarrow H(x, y, z)}{R(x, y, z)\ \&\ L(B(x: y)\ \|\ C(x: z)) \rightarrow H(x, y, z)}$$

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

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

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

E. Модель системы правил вывода в PVS

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

Для операторов языка P_2 в модели 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 оператора суперпозиции.

В модели определены теория для тотальности оператора и для однозначности спецификации. Определены теории для формул Corr. Каждое правило представляется в виде теоремы в теории на PVS.

V. Система верификации

Система верификации разрабатывалась как back-end в системе предикатного программирования. Основной компонентой системы верификации является генератор формул корректности программы, представленной во внутреннем представлении (рис. 2).

В трансляторе с языка P реализован статический контроль типов. Транслятор формирует условия совместимости и пытается автоматически их разрешить. В том случае, если условие совместимости типов не может быть разрешено статически при трансляции, это условие генерируется в виде логической формулы для последующего доказательства в CVC3 или PVS. Данный механизм ранее был реализован в инструменте для контроля динамической семантики [6]. Условия совместимости типов добавляются к сгенерированным формулам корректности программы.

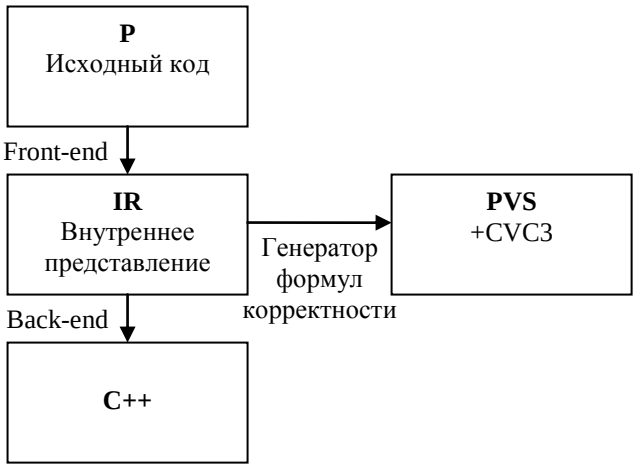


Рис. 2. Система верификации

Значительная часть генерируемых формул являются простыми для доказательства. Истинность таких формул проверяется автоматически с помощью решателя CVC3. Для доказательства более сложных формул корректности используется система интерактивного доказательства PVS.

VI. Генерация формул корректности

Генерация формул корректности проходит по схеме, представленной на рисунке 3.

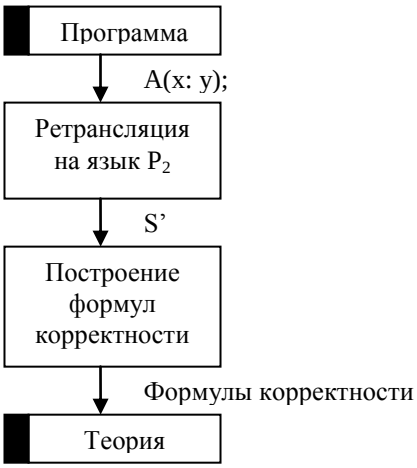


Рис. 3. Общая схема генерации условий корректности

Задачей алгоритма является автоматическое построение формул корректности предикатной программы. На языке P предикатная программа представлена набором определений предикатов. Корректность программы — это корректность всех определенных предикатов.

Из программы последовательно извлекаются определения предикатов вида (1). Для тела предиката, представленного оператором S, реализуется преобразование к канонической форме – это трансляция с языка P_4 на P_2, существенно упрощающая структуру исходной программы.

Для преобразованного оператора происходит построение условий корректности.

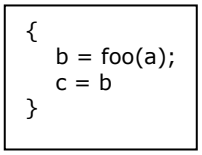
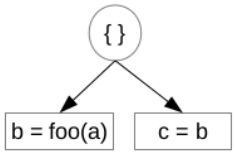
На основе полученных условий корректности строится теория, которая содержит в себе, помимо самих условий, еще и определение необходимых формул и типов, а также ссылки на импортируемые теории.

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

VII. Ретрансляция программы на язык P_2

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"*.

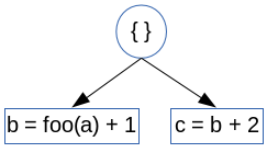
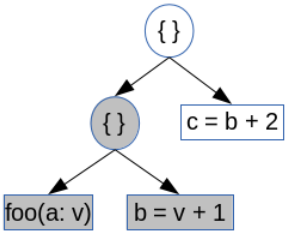
A. *Развертка*

До	После
	

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.

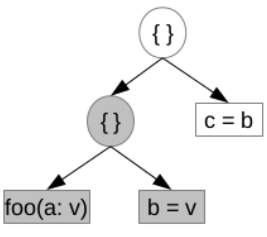
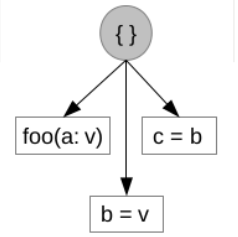
"Развертка" осуществляется рекурсивно. Стартовое дерево содержит лишь один лист – начальный оператор. Затем, если лист является составным оператором, происходит его "развертка": оператор становится узлом, а его подоператоры листьями. Аналогичным образом происходит "развертка" каждого листа, до тех пор,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

B. *Преобразование*

До	После
	

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.

C. *Упрощение*

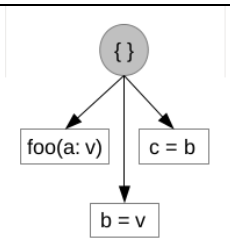
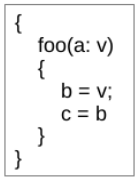
До	После
	

На этапе упрощения происходит сокращения количества узлов в дереве. Это достигается за счет

объединения схожих узлов. Два узла схожи, если оба являются либо операторами суперпозиции, либо параллельными операторами.

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.

D. Свертка

До	После
	

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.

VIII. Построение 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.

В процессе построения формул корректности возникает задача выбора правила вывода, которое необходимо применить на каждом следующем шаге. Этот выбор можно назвать стратегией вывода. Стратегия вывода схематически представлена на рисунке 4:

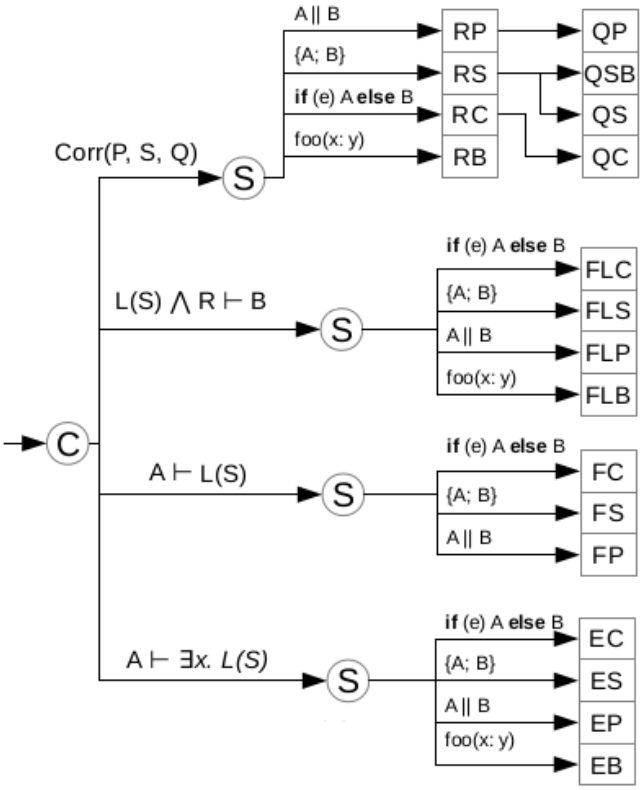


Рис. 4. Стратегия вывода формул корректности

IX. Пример

Дадим иллюстрацию работы метода дедуктивной верификации на примере программы умножения двух натуральных чисел a и b, с использованием лишь операций сложения и вычитания.

Для того чтобы получить программу в форме хвостовой рекурсии и автоматически преобразовать ее в программу с циклом **while** на этапе компиляции, мы должны рассмотреть более общую задачу mult(a, b, d: c) со спецификацией [a ≥ 0 & b ≥ 0 & d ≥ 0, c = a * b + d].

Ниже приведена предикатная программа умножения через сложение.

```
mult(nat a, b, d: nat c)
pre a ≥ 0 & b ≥ 0 & d ≥ 0
{
   if (a = 0)
      c = d
   else
      mult(a − 1, b, d + b: c)
}
post c = a * b + d
measure a;
```

Определим предусловие, постусловие и меру в виде отдельных формул:

formula P_mult(**nat** a, b, d) =

a ≥ 0 & b ≥ 0 & d ≥ 0;

formula Q_mult(**nat** a, b, d, c) = c = a * b + d;

formula m(**nat** a: **nat**) = a;

Корректность предиката mult, по определению (2), выражается формулой:

Corr(mult, P_mult, Q_mult)(a, b, d) (3)

Таким образом, предикат mult будет корректен, если нам удастся доказать истинность формулы (3). В рамках эксперимента, можно раскрыть эту формулу по определению и попытаться доказать. Это будет намного сложнее по сравнению с нашим методом, описанным ниже.

Разложим формулу (3), заменив в ней вхождение предиката mult на его тело, представленное условным операторам. В соответствии со стратегией выводы (рис. 4) применим правило RC. Это порождает две новые цели:

Corr(c = d, λa, b, d. P_mult(a, b, d) & a = 0 ,

Q_mult) (4)

Corr(mult(a – 1, b, d + b: c),

λa, b, d. P_mult(a, b, d) & ¬ a = 0,

Q_mult) (5)

Формула (4) раскрывается по определению (2) в виде следующей леммы:

lemma P_mult(a, b, d) & a = 0 &

c = d ⇒ Q_mult(a, b, d, c);

К формуле (5) применяется правило RB, где B(a, b, d) = |a – 1, b, d + b| и PB = a - 1 ≥ 0. Первая посылка правила RB опускается ввиду рекурсивности предиката mult. Вторая посылка тождественно истинна. Третья и четвертая посылки записываются в виде следующих лемм:

lemma P_mult(a, b, d) & ¬ a = 0 ⇒

a - 1 ≥ 0 & m(a - 1) < m(a);

lemma P_mult(a, b, d) & ¬ a = 0 &

Q_mult(a - 1, b, d + b, c) ⇒

Q_mult(a, b, d, c);

Доказательство корректности предиката mult сводится к доказательству истинности трех описанных выше лемм.

Для оценки эффективности метода можно сравнить описанный процесс верификации предиката mult с классическим процессом верификации по Хоару следующей императивной программы, полученной приведением хвостовой рекурсии в предикате mult к циклу **while**:

c := d; **while** a != 0 **do** a := a – 1; c := c + b **end**

X. Доказательство истинности формул корректности

Алгоритм, описанный выше, позволяет автоматически генерировать формулы тотальной корректности предикатной программы. Для каждого определения предиката строиться теория, содержащая в себе определения формул корректности и типов. В дальнейшем необходимо доказать истинность этих формул.

Все формулы проходят проверку на SMT-решателе CVC3 [5]. Формулы транслируется во внутреннее представление решателя. Решатель осуществляет проверку на истинность, и в зависимости от результата выставляет формуле статус valid, invalid или unknown.

Для доказательства истинности формул, которые решатель не смог проверить, используется система PVS. Теории, построенные генератором, транслируются на язык спецификаций системы. После чего пользователю необходимо в интерактивном режиме, построить доказательство на языке команд системы PVS.

XI. Опыт применения системы верификации

В рамках учебного курса “Формальные методы в описании языков и систем программирования” магистрантам Факультета Информационных Технологий Новосибирского Государственного Университета 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.

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

Суммарно по всем задачам было сгенерировано 363 формулы. Из них 260 – это условия тотальной корректности. Остальные 103 – это условия совместимости типов.

Программы небольшие. Число операторов не более 13. Однако, количество формул корректности и семантики весьма значительно. В связи с этим от работы решателя CVC3 требуется показать приемлемый результат. Условия совместимости типов желательно проверить полностью автоматически.

Решатель смог проверить 72% формул семантической корректности и 45% формул тотальной корректности. Данные результаты можно считать приемлемыми, хотя и неидеальными.

Ниже приведен полный результат анализа работы системы верификации. В таблице 1 для каждой задачи указано количество формул, сгенерированных системой, и количество формул, которые решатель смог проверить.

ТАБЛИЦА I. РЕЗУЛЬТАТЫ ПРОВЕРКИ РАБОТЫ РЕШАТЕЛЯ CVC3

№	Типы	Корректность	Итог
1	2/4	7/15	9/19
2	1/3	4/15	6/18
3	7/9	13/29	22/38
6	27/37	25/48	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
%	72	45	53

XII. ЗАКЛЮЧЕНИЕ

В работе описан метод, позволяющий доказывать тотальную корректность предикатных программ. На основании данного метода была разработана система дедуктивной верификации предикатных программ, которая автоматически генерирует формулы корректности для программ с исходным кодом на языке Р. Проведены результаты апробация разработанной системы в рамках курса “Формальные методы в программировании”.

Разработанный метод дедуктивной верификации предикатных программ имеет существенные преимущества по сравнению с классическим методом Хоара. Верификация предикатной программы ориентировочно в 2-4 раза превосходит верификацию аналогичной императивной программы при сравнении по трудоемкости и времени верификации. Это определяет целесообразность разработки и верификации сложных и критических фрагментов кода программной комплекса в системе предикатного программирования с получением итогового кода таких фрагментов на языке C++.

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

[1] 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

[2] Предиктное программирование. Учебное пособие / Под ред. Шелехова В.И. НГУ. Новосибирск, 2009. 111 С.

[3] 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

[4] http://www.iis.nsk.su/persons/vshel/files/rules.zip

[5] 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.

[6] Каблуков И.В., Шелехов В.И. Контроль динмаической семантики предикатной программы // Новосибирск, 2012. — 28с. — (Препр. / ИСИ СО РАН; N 162).

[7] Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements. Automatic Control and Computer Sciences. Vol. 45, No. 7, 421–427.

[8] Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С.14-21.

[9] В.А. Вшивков, Т.В. Маркелова, В.И. Шелехов. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т.4(33), с. 79-94, 2008.

[10] Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164)

[11] Shelekhov V. The language of calculus of computable predicates as a minimal kernel for functional languages // BULLETIN of the Novosibirsk Computing Center. Series: Computer Science. IIS Special Issue. – 2009. – 29(2009). – P.107-117.

[12] Bertrand Meyer. Towards a Calculus of Object Programs // ETH Zurich, ITMO & Eiffel Software.

[13] C. A. R. Hoare. «An axiomatic basis for computer programming». Communications of the ACM, 12(10):576—580,583 October 1969. DOI:10.1145/363235.363259

[14] Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // IEEE Symposium on Logic in Computer Science. 2002.

[15] . Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert. PVS Language Reference.

[16] Manna, Pnueli, Axiomatic Approach to Total Correctness of Programs, 1973

[17] Correnson. L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual (June 2013)

[18] Baudin, P., Cuoq, P., Filliatre, J.-C., Marche, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (June 2013)

[19] Baudin, P., Correnson, L., Dargaye, Z.: WP Plug-n Manual (June 2013)

[20] Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto, Aaron Tomb: Formal Verification of kLIBC with the WP Frama-C Plug-in