Моделирование и анализ информационных систем. Т. 25, №4 (2018), с. 358–381
Modeling and Analysis of Information Systems. Vol. 25, No 4 (2018), pp. 358–381

Верификация программ

DOI: 10.18255/1818-1015-2018-4-358-381

УДК 004.052.42

Верификация программ со взаимной рекурсией на языке Пифагор

Ушакова М. С., Легалов А. И.

получена 23 марта 2018

Аннотация. В работе рассматривается верификация программ со взаимной рекурсией для языка функционально-потокового параллельного программирования Пифагор. В языке используется модель представления программы в виде графа потока данных (информационного графа), в котором нет дополнительных управляющих связей, а присутствуют только информационные зависимости. Это позволяет упростить процесс верификации, так как не требует анализа возникающих в традиционных архитектурах дополнительных ресурсных конфликтов.

Доказательство корректности программы опирается на удаление взаимных рекурсий посредством преобразования программы. Универсальным способом удаления взаимной рекурсии произвольного количества функций является построение универсальной рекурсивной функции, которая выполняет работу всех исходных функций и принимает, кроме аргумента выполняемой функции, натуральное число, являющееся номером выполняемой функции. В ряде случаев, когда присутствует косвенная рекурсия, можно использовать более простой способ преобразования — объединение кода функций, при котором происходит объединение тел вызывающих друг друга функций.

Для преобразования произвольной рекурсии в прямую предлагается построение графа всех связанных функций и последующая трансформация данного графа путём удаления функций, не связанных с рассматриваемой, объединения косвенно рекурсивных функций и построения универсальной рекурсивной функции. Доказывается, что изменение функции на языке Пифагор при объединении кода и построении универсальной рекурсивной функции не влияет на корректность исходной программы. Приводится пример доказательства частичной корректности программы на языке Пифагор, осуществляющей синтаксический разбор простого арифметического выражения. После построения графа всех связанных функций рассматриваются два способа доказательства: с использованием объединения кода функций и с построением универсальной рекурсивной функции.

Ключевые слова: функционально-потоковое параллельное программирование, язык программирования Пифагор, корректность рекурсий, удаление взаимной рекурсии, универсальная рекурсивная функция

Для цитирования: Ушакова М. С., Легалов А. И., "Верификация программ со взаимной рекурсией на языке Пифагор", *Моделирование и анализ информационных систем*, **25**:4 (2018), 358–381.

Об авторах: Ушакова Мария Сергеевна, orcid.org/0000-0003-4234-2714, аспирант,
Сибирский федеральный университет, Институт космических и информационных технологий
ул. Академика Киренского, 26, г. Красноярск, 660074 Россия, e-mail: ksv@akadem.ru

Легалов Александр Иванович, orcid.org/0000-0002-5487-0699, д-р техн. наук, профессор,
Сибирский федеральный университет, Институт космических и информационных технологий
ул. Академика Киренского, 26, г. Красноярск, 660074 Россия, e-mail: legalov@mail.ru

Благодарности:
Исследование выполняется при финансовой поддержке РФФИ в рамках научного проекта № 17-07-00288.

Введение

В связи с возрастающими требованиями к надежности программного обеспечения, наряду с традиционными методами тестирования, всё чаще стали использоваться методы формальной верификации программ. Формальная верификация — это доказательство корректности программы, которое заключается в установлении соответствия между программой и ее спецификацией, описывающей цель разработки [1]. Методы формальной верификации позволяют доказать отсутствие ошибок в программе, в то время как тестирование лишь выявляет ошибки, но не даёт гарантии их отсутствия.

Распараллеливание программ позволяет существенно увеличить их производительность на современных вычислительных системах. Однако параллелизм приводит к значительному усложнению разработки и особенно отладки. В основном программы пишутся на императивных языках программирования. По сравнению с последовательными, параллельные императивные программы могут содержать новые виды ошибок, которые трудно выявить при тестировании. Также резко увеличивается сложность формальной верификации параллельных императивных программ.

Вместе с тем всё большую популярность приобретает функциональное программирование, которое ориентируется на отношение между данными. В рамках этого направления формальные методы верификации развиваются достаточно интенсивно. Основополагающей работой в данной области является работа Бойера и Мура [2]. В системе NQTHM они реализовали метод автоматизированного доказательства функций, написанных на языке LISP. Другой пример — доказательство утверждений для программ на Haskell [3].

Одним из функциональных языков является язык функционально-потокового параллельного программирования Пифагор [5]. Его специфика — богатый набор операторов, обеспечивающий описание параллелизма с сохранением представления программы в виде зависимости функций и управлением по готовности данных. Это позволяет изначально создавать программы с максимальным параллелизмом за счёт его формирования на уровне операций. Данный подход также обеспечивает написание программы без учёта ресурсных ограничений, что позволяет упростить процесс верификации, так как не требует анализа возникающих в традиционных архитектурах дополнительных ресурсных конфликтов. После верификации возможны дальнейшие преобразования исходных программ в программы для реальных архитектур параллельных вычислительных систем путём «сжатия» параллелизма с учётом ограниченных ресурсов.

Для функционально-потоковых параллельных программ предложен метод формальной верификации, позволяющий доказывать корректность программ [6], а также разрабатывается инструментальное средство для поддержки этого процесса [7]. Для дальнейшего развития данной системы требуется включать дополнительные алгоритмы, расширяющие функциональность и упрощающие работу пользователя системы. Одной из задач, которую может выполнить система, является устранение взаимной рекурсии нескольких функций (например, функция f вызывает функцию g, а функция g вызывает функцию f). Существует два основных способа решения данной проблемы [8]. Первый способ — совместное доказательство корректности всех функций во взаимной рекурсии, которое обычно требует проведения доказа-

тельства с помощью одновременной индукции (simultaneous induction) [9]. Второй способ — удаление взаимной рекурсии посредством преобразования программы, в результате которого получается одна рекурсивная функция [10]. По сути, это два одинаковых метода, которые различаются только внешним представлением доказательства для пользователя. Недостаток первого способа — необходимость доказывать несколько утверждений одновременно, а недостаток второго способа — результирующая функция может получиться достаточно сложной.

В работе рассматривается второй способ решения проблемы взаимной рекурсии для программ на языке Пифагор. Основная причина выбора состоит в том, что этот способ делает доказательство более наглядным, так как не требуется одновременно доказывать утверждения для каждой из рекурсивных функций. В связи с тем, что проблема доказательства корректности программы является неразрешимой, то есть не существует универсального алгоритма решения этой проблемы, то невозможно сделать процесс доказательства полностью автоматическим. Поэтому пользователь должен иметь представление о теоремах, на которых основана работа алгоритма, и должен принимать участие в доказательстве, если система не может выполнить его автоматически.

Помимо доказательства корректности, необходимо доказывать завершение программы. Эта задача также является неразрешимой. Доказательство завершения функциональной программы сводится к построению порядка полной фундированности (порядок $\succ$ является отношением полной фундированности, если не существует бесконечной убывающей цепи $x_1 \succ x_2 \succ \ldots$) на множестве допустимых аргументов функции и проверке того, что аргумент каждого рекурсивного вызова меньше, чем входной аргумент [11, 12]. При наличии взаимной рекурсии необходим свой порядок для аргументов каждой из функций. Если провести преобразование и устранить взаимную рекурсию, то достаточно построить один порядок на множестве аргументов результирующей функции.

Данный метод может быть реализован в автоматическом режиме и интегрирован в инструментальное средство доказательства корректности программ на языке Пифагор.

1. Определения

В работе термин «функция» используется в нескольких смыслах. В математическом смысле — это однозначное отображение множества допустимых значений аргумента на множество значений функции. Когда речь идёт о программной реализации алгоритма, то термин «функция» или «функция на языке Пифагор» используется в рамках понимания функции в программировании — как именованного фрагмента кода на языке программирования, возвращающего значение по своему имени.

Под рекурсией понимается такой способ организации обработки данных, при котором функция может вызвать одну или несколько своих копий непосредственно или с помощью других функций [13]

Рекурсия может быть прямой или косвенной. *Прямая рекурсия* — это рекурсивная функция, в теле которой присутствует вызов самой себя. *Косвенная рекурсия* — это рекурсивная функция, в теле которой нет вызовов самой себя, но эта функ-

ция вызывается вновь через цепочку вызовов других функций [14]. Введем понятие *смешанной рекурсии* как функции, в которой присутствует прямой и косвенный рекурсивный вызов.

Также используем термины прямая и косвенная связь для разных функций. Функция A *прямо связана* с функцией B, если B вызывается непосредственно в теле A. Функция A *косвенно связана* с B, если вызов B опосредован вызовами других функций. A и B — *связанные* функции, если они связаны прямо или косвенно. Если функция A не связана с функцией B ни прямо, ни косвенно, то назовём A — *независимой* от B функцией.

Пусть A — рассматриваемая рекурсивная функция. Обозначим через $\Omega(A)$ *множество всех связанных функций* — функций, которые вызываются из A, как непосредственно в её теле, так и через цепочку вызовов других функций. Саму функцию A также отнесём к $\Omega(A)$. Для каждой функции в $\Omega(A)$ укажем её имя и после него в круглых скобках — список вызываемых непосредственно в её теле рекурсивных функций. Например, запись вида $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$ означает, что рассматриваемая функция A является косвенной рекурсией, в её теле присутствует вызов функции B и D; B — смешанная рекурсия, вызывающая себя и функции A, C; C — косвенная рекурсия, вызывающая функцию A; D — нерекурсивная функция, независимая от других функций. Множество связанных функций $\Omega(A)$ можно изобразить в виде *дерева возможных вызовов*. Это дерево, корень которого помечен именем рассматриваемой функции A; с каждым узлом, помеченным именем функции X, связаны дочерние узлы, соответствующие вызываемым в теле X функциям. Пример такого дерева для множества $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$ приведён на рис. 1a.

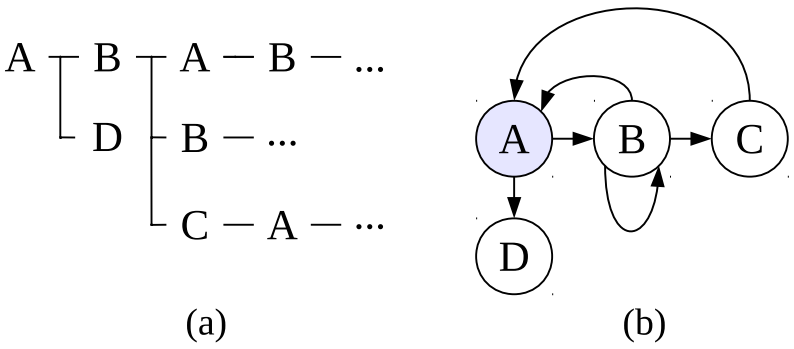

Рис. 1. Различные варианты представления множества связанных функций $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$; (a) — дерево возможных вызовов функций; (b) — граф всех связанных функций, серым отмечена вершина с именем рассматриваемой функции

Fig. 1. Different variants of representation of the set of connected functions $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$; (a) — the tree of possible function calls; (b) — the graph of all connected functions, the node with the name of the considered function is marked with gray

Бесконечное дерево возможных вызовов функций можно свернуть в *граф всех связанных функций*. Это ориентированный граф, вершины которого помечены именами функций $\Omega(A)$, а дуги направлены из вызывающей функции в вызываемую. Пример такого графа приведён на рис. 1b.

Функции A и B *взаимно рекурсивные*, если функция A связана с B и функция B связана с A. Обозначим отношение взаимной рекурсии $\leftrightarrow$. Если считать, что отношение взаимной рекурсии рефлексивно ($A \leftrightarrow A$), то отношение $\leftrightarrow$ будет отношением эквивалентности, которое разделяет множество всех связанных функций на непересекающиеся классы. На графе всех связанных функций эти классы будут соответствовать компонентам сильной связности [10].

2. Преобразование произвольной рекурсии в прямую

2.1. Универсальная рекурсивная функция

Одним из способов сведения произвольной рекурсии к прямой является построение универсальной рекурсивной функции [15]. Пусть $\mathfrak{F} = \{f_1, f_2, \ldots, f_k\}$ — множество функций $f_i : M \to M$, $i = 1, 2, \ldots, k$, M — произвольное множество. Каждой функции f_i приписано число $i \in \mathbb{N}$, называемое номером функции. Функция $F : \mathbb{N} \times M \to M$ называется *универсальной рекурсивной функцией* (УРФ) для множества $\mathfrak{F}$, если

$$F(n, x) = \begin{cases} f_1(x), & n = 1; \\ \ldots \\ f_k(x), & n = k. \end{cases}$$

Например, рассмотрим две связанные функции A и B. В общем случае функция A является смешанной рекурсией, вызывает функцию B, которая тоже является смешанной рекурсией. Это случай взаимной рекурсии двух функций. Множество связанных рекурсивных функций $\Omega(A)$ имеет вид: $\{A(A, B), B(A, B)\}$. Дерево возможных вызовов функций приведено в левой части рис. 2.

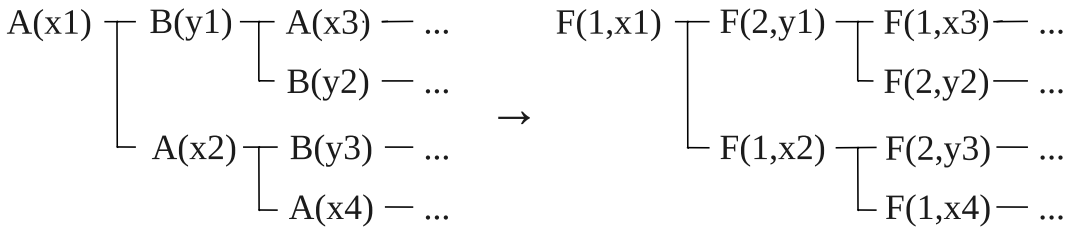

Рис. 2. Схема удаления взаимной рекурсии двух смешанно рекурсивных функций A и B с помощью универсальной рекурсивной функции F; после имени каждой функции в скобках указаны её аргументы

Fig. 2. The scheme of elimination of mutual recursion of functions A and B with the help of the universal recursive function F; after the name of each function its arguments are given in brackets

Устраним взаимную рекурсию с помощью универсальной рекурсивной функции. Зададим универсальную рекурсивную функцию $F(n, x)$, у которой n — номер рекурсивной функции, объединяемой в УРФ, а x — аргумент для функции с номером n; n будет принимать значение 1, когда надо выполнить код функции A, и 2, если

требуется выполнить код функции B. Дерево возможных вызовов универсальной рекурсивной функции F приведено в правой части рис. 2. Построение УРФ для большего числа функций аналогично.

2.2. Объединение функций

В простых случаях косвенную рекурсию можно свести к прямой рекурсии с помощью *объединения* тел связанных функций в одну функцию. Подобный метод используется в [10] для логических программ и называется развёрткой (unfolding).

Поясним данный способ преобразования на примере двух функций. Пусть дана функция A, которая, в общем случае, является смешанной рекурсией. В ней присутствует вызов функции B, которая является косвенной рекурсией и прямо связана с функцией A. То есть в функции B нет вызова самой себя, но присутствует вызов функции A. В этом случае косвенную рекурсию можно свести к прямой рекурсии объединением тел функций A и B в одну функцию AB. Таким образом, код функции B «встраивается» в код функции A в месте вызова B. Изменение графа двух связанных функций описанного случая приведено на рис. 3a. Исходный граф связанных функций для $\Omega(A) = \{A(A, B), B(A)\}$, после объединения кода функций A и B в функцию AB получаем $\Omega(AB) = \{AB(AB)\}$.

Если рассматриваемое множество связанных функций состоит из более чем двух функций, то объединение кода функций не приведёт к дублированию кода в том случае, если только одна функция связана с рассматриваемой косвенной рекурсией B. Это эквивалентно тому, что на графе всех связанных функций вершина B имеет только одну входную дугу. В остальных случаях косвенную рекурсию лучше устранять с помощью универсальной рекурсивной функции, описанной ранее.

Пример объединения функций для случая трёх связанных функций приведён на рис. 3b, граф связанных функций для исходного $\Omega(A) = \{A(B), B(A, B, C), C(A)\}$. Функция C является косвенной рекурсией и имеет одну входную дугу на графе связанных функций. Поэтому её код может быть «встроен» в функцию B, узел которой является смежным с узлом C по его единственной входной дуге. Полученной в результате объединения функции присваивается имя BC, изменённое $\Omega(A) = \{A(BC), BC(A, BC)\}$. Для дальнейшего преобразования полученной рекурсии в прямую необходимо построить УРФ.

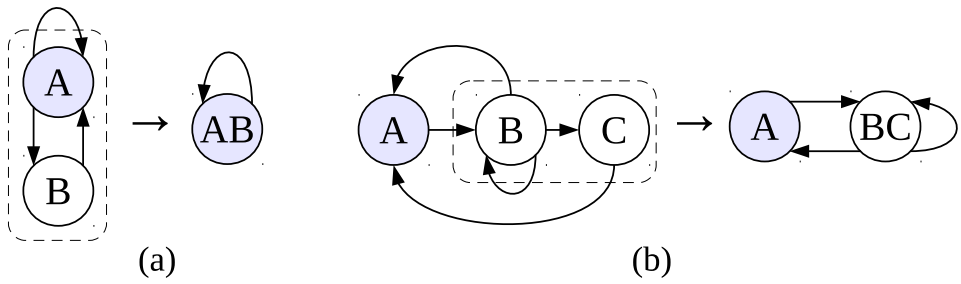

Рис. 3. Удаление косвенной рекурсии с помощью объединения кода функций

Fig. 3. The elimination of indirect recursion by merging of functions code

2.3. Алгоритм преобразования произвольной рекурсии в прямую

Резюмируя вышесказанное, можно предложить следующий алгоритм преобразования произвольной рекурсии в прямую.

Дана произвольная рекурсивная функция A со множеством связанных функций $\Omega(A)$. Требуется преобразовать её в прямую рекурсию.

1. На основе $\Omega(A)$ строим граф всех связанных функций $G(A)$.

2. В $\Omega(A)$ находим все нерекурсивные функции. В $G(A)$ таким функциям будут соответствовать вершины, не входящие ни в один цикл. То есть для вершины X, помеченной нерекурсивной функцией, не существует пути, который привёл бы опять в вершину X. Все найденные нерекурсивные функции являются независимыми от A и удаляются из множества $\Omega(A)$. В результате чего получаем новое множество связанных функций $\Omega_1(A)$ с графом связанных функций $G_1(A)$.

3. Удаляем из $\Omega_1(A)$ все рекурсивные функции, не связанные с A. Для этого в графе $G_1(A)$ находим все узлы, для которых любой цикл, проходящий через этот узел, не проходит через вершину A. Все независимые от A рекурсивные функции удаляются из множества $\Omega_1(A)$. В результате получаем множество связанных функций $\Omega_2(A)$ с графом связанных функций $G_2(A)$.

4. В $G_2(A)$ находим все функции, имеющие только одну входную дугу. Код этих косвенно рекурсивных функций может быть объединён с кодом функции смежной по входной дуге рассматриваемого узла. После объединения получаем $\Omega_3(A)$.

5. Задаём универсальную рекурсивную функцию F для всех функций из множества $\Omega_3(A)$.

Полученная в результате функция F будет прямой рекурсией.

Замечание 1. При доказательстве корректности функции A вначале необходимо доказать корректность всех независимых от A нерекурсивных и рекурсивных функций, которые удаляются из множества всех связанных функций в пунктах 2 и 3.

Замечание 2. Если узел нерекурсивной функции, найденной в пункте 2, имеет одну входную дугу, то код этой функции можно объединить с кодом функции, смежной по входной дуге.

Замечание 3. Если пункт 4 пропустить (после пункта 3 сразу переходить к пункту 5), то все функции из $\Omega_2(A)$ войдут в универсальную рекурсивную функцию.

В качестве иллюстрации работы алгоритма на рис. 4а изображен граф всех связанных функций для функции A со множеством

$$\Omega(A) = \{A(A,B,C,D,E), B(A,C,H,I), C(B), D(J), E(F,G), F(E,F,G), \\ G(), H(C), I(A), J()\}.$$

Три узла в графе (D, J и G, выделенные пунктиром) не лежат ни в каком цикле. Следовательно, функции, которыми помечены эти узлы, являются нерекурсивными и удаляются из множества $\Omega(A)$. Полученное $\Omega_1(A)$ имеет вид

$$\{A(A,B,C,E), B(A,C,H,I), C(B), E(F), F(E,F), H(C), I(A)\}.$$

На следующем этапе удаляются все рекурсивные функции, независимые от рассматриваемой функции A. Из графа связанных функций $\Omega_1(A)$ (рис. 4b) видно, что не существует цикла, проходящего через узлы E или F и узел A. Значит, рекурсивные функции E и F не зависят от A и удаляются из $\Omega_1(A)$, получается $\Omega_2(A) = \{A(A,B,C), B(A,C,H,I), C(B), H(C), I(A)\}$ (рис. 4c). Два узла I и H в графе для $\Omega_2(A)$ имеют по одной входной дуге, поэтому их код может быть объединён со смежным по их входной дуге узлу. Для обеих вершин это узел B. Назовём функцию, полученную в результате объединения, B_1, тогда $\Omega_3(A) = \{A(A,B_1,C), B_1(A,C), C(B_1)\}$ (рис. 4d). Далее, для трёх оставшихся функций формируется УРФ.

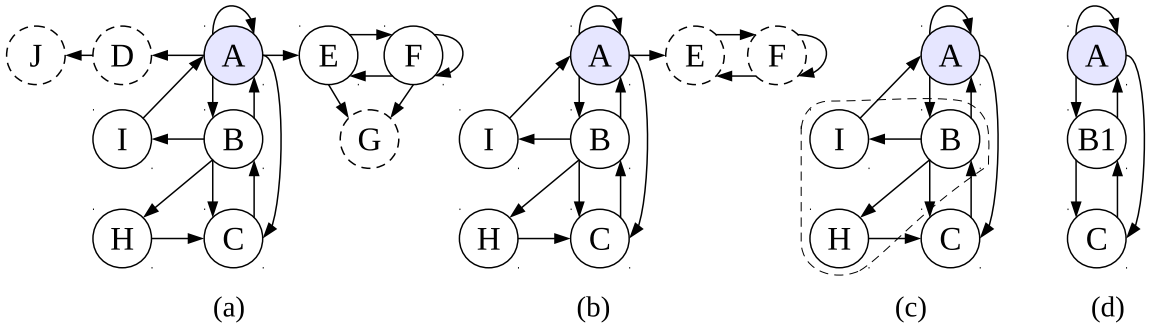

Рис. 4. Пример изменения графа связанных функций при преобразовании рекурсии A в прямую

Fig. 4. The example of modifications of the connected functions graph during transformations of the function A into the direct recursion

3. Преобразование рекурсивных функций на языке Пифагор

Язык программирования Пифагор хорошо подходит для проведения преобразования программы с удалением взаимной рекурсии. В языке присутствует только один оператор, применяющий функцию к аргументу, — оператор интерпретации. У оператора два входа, на которые поступают функция и аргумент функции. Оператор интерпретации имеет постфиксную и префиксную формы, которые обозначаются знаками «:» и «^» соответственно. Далее в тексте используется только постфиксная форма оператора. Например, X:F — применение функции F к аргументу X. Аргумент всегда один, но он может быть списком данных с достаточно сложной структурой, которая может интерпретироваться как передача нескольких аргументов. Таким образом, у каждой функции во взаимной рекурсии один аргумент, и после преобразования в результирующей УРФ также будет один аргумент.

Рассмотрим удаление различных видов рекурсий из программ на языке Пифагор.

3.1. Объединение кода функций

В начале рассмотрим случай косвенной рекурсии. Дано две функции A и B на языке Пифагор, с исходным кодом:

```
A << funcdef x
{
    (
        {x:g0},
        {x:g1:A},
        {x:g2:B}
    ):
    [(c0, c1, c2):?]:.:p >> return;

}
```

```
B << funcdef y
{
    (
        {y:e0},
        {y:e1:A}
    ):
    [(d0, d1):?]:.:q >> return;

}
```

В данном коде g_i, c_i, $i = 1, 2, 3$ и e_j, d_j, $j = 0, 1$, p, q — некоторые функции, независимые от функций A и B. Множество $\Omega(A) = \{A(A, B), B(A)\}$, соответствующее дерево всех связанных функций приведено на рис. 3а.

Для удаления косвенной рекурсии объединим код функций A и B в функцию AB, как показано ниже:

```
AB << funcdef x
{
    (
        {x:g0},
        {x:g1:AB},
        {
            (
                {x:g2:e0},
                {x:g2:e1:AB}
            ):
            [(d0, d1):?]:.:q
        }
    ):
    [(c0, c1, c2):?]:.:p >> return;

}
```

Покажем, что объединение кода не изменяет корректность исходной программы. Пусть условие корректности рассматриваемой функции A задано в виде тройки:

$$\boxed{P_A}\ \mathrm{A}\ \boxed{Q_A}.$$

При объединении кода получаем функцию AB с условием корректности

$$\boxed{P_{AB}}\ \text{AB}\ \boxed{Q_{AB}},$$

где $P_{AB} = P_A$, $Q_{AB} = Q_A$. Предположим, что корректность AB доказана, требуется показать, что A корректна. Очевидно, что из истинности $(P_{AB} \Rightarrow Q_{AB})$ следует истинность формулы $(P_A \Rightarrow Q_A)$, значит, функция A корректна.

3.2. Построение универсальной рекурсивной функции

Далее рассмотрим удаление взаимной рекурсии. Возьмём схему рекурсии, представленную на рис. 2. В общем случае исходный код функций A и B на языке Пифагор будет следующим:

```
A << funcdef x
{
    (
        {x:g0},
        {x:g1:A},
        {x:g2:B}
    ):
    [(c0, c1, c2):?]:.:p >> return;
}
```

```
B << funcdef y
{
    (
        {y:e0},
        {y:e1:A},
        {y:e2:B}
    ):
    [(d0, d1, d2):?]:.:q >> return;
}
```

где g_i, c_i, e_i, d_i, $i = 1, 2, 3$, p, q — некоторые функции, независимые от функций A и B.

Для удаления взаимной рекурсии введём универсальную рекурсивную функцию F следующим образом:

```
F << funcdef nx
{
    n<<nx:1;
    x<<nx:2;
    (
        {
            (
                {x:g0},
                {(1,x:g1):F},
                {(2,x:g2):F}
            ):
            [(c0, c1, c2):?]:.:p
        },
        {
            (
                {x:e0},
                {(1,x:e1):F},
                {(2,x:e2):F}
            ):
            [(d0, d1, d2):?]:.:q >> return;
```

```
        }
    ):
    [
        ((n,1):=,(n,2):=):?
    ]:. >> return;
}
```

Полученная функция F является прямой рекурсией.

Покажем, что объединение функций в УРФ не изменяет корректность исходной программы. Пусть условие корректности функции A задано в виде тройки $\boxed{P_A}$ A $\boxed{Q_A}$, а функции B — $\boxed{P_B}$ B $\boxed{Q_B}$. Зададим для функции $F(n, x)$ следующее предусловие и постусловие:

$$P_F = ((n = 1) \wedge P_A) \vee ((n = 2) \wedge P_B),$$

$$Q_F = ((n = 1) \wedge Q_A) \vee ((n = 2) \wedge Q_B).$$

Тогда, при верном указании номера функции $n = 1$, получаем, что из корректности функции F будет следовать корректность исходной функции A:

$$(P_F \wedge (n = 1)) \Rightarrow P_A,$$

$$(Q_F \wedge (n = 1)) \Rightarrow Q_A.$$

4. Пример доказательства корректности рекурсивной функции на языке Пифагор

Доказательство корректности рекурсивной программы рассмотрим на примере решения задачи распознавания простого арифметического выражения, порождаемого в соответствии со следующим правилом в нотации Бэкуса–Наура:

```
<выражение> ::= x | +<выражение><выражение> | *<выражение><выражение>      (1)
```

где x — терминальный операнд. Для упрощения примера введём следующие ограничения: функция должна принимать на вход строку с правильно построенным выражением и возвращать пустую строку, никакой обработки ошибок не требуется.

Распознавание данного выражения определяется следующими функциями на языке Пифагор:

```
parse<<funcdef str  // Основная функция parse, принимающая строку str
                    // с арифметическим выражением
{   s1   << str:1;  // Сохранение первого символа строки str в s1
    case << ( (s1,'+'):=, // Сравнение s1 с тремя допустимыми
              (s1,'*'):=, // вариантами значений
              (s1,'x'):=
            ):?;          // Выбор номера пути вычисления и
                          // сохранение номера в case
    act  << ( {str:fp},   // Формирование списка act трёх различных путей
```

```
            {str:fm},   // вычислений, зависящих от значения s1
            {str:fn}
          );
    act:case:.   // Выбор нужного пути вычисления и активация вычисления
    >> return;   // Возврат результата вычислений
}
fp<<funcdef str1 // Функция fp для разбора сложения, принимающая str1
{    str1:-1:parse:parse // Из строки str1 удаляется первый символ '+'
     >> return;          // и с помощью двух последовательных вызовов
                         // функции parse удаляется 1-е и 2-е слагаемое
}
fm<<funcdef str2 // Функция fm для разбора умножения, принимающая str2
{    str2:-1:parse:parse >> return;   } // Аналогична fp
fn<<funcdef str3 // Функция fn для разбора формального символа x
{    str3:-1 >> return; // Из входной строки str2 удаляется один символ
}
```

Основная функция, которая получает строку с арифметическим выражением, имеет имя *parse*. Также имеется три вспомогательные функции, которые отвечают за разбор одного из видов выражений: fp разбирает применение функции сложения, fm — функции умножения, а fn — формального символа «x». Функция *parse* определяет тип выражения по первому символу входной строки и передает эту строку соответствующей вспомогательной функции. И, в качестве ответа, возвращает результат работы вспомогательной функции. Функции fp, fm разбирают получаемую строку, удаляя из неё первый символ операции, а затем дважды применяя к ней функцию *parse*. Первый вызов функции *parse* удаляет из строки первое слагаемое (множитель), второй вызов — второе слагаемое (множитель), оставшийся «хвост» строки возвращается в качестве ответа. Функция fn разбирает получаемую строку удалением из неё первого символа и возвращает оставшийся «хвост» строки. Таким образом, если входное выражение для функции *parse* не содержит ошибок, то функция вернёт пустую строку, иначе результат непредсказуем.

На естественном языке спецификация к функции *parse* будет следующей. Входным аргументом для функции является строка символов *str*, в которой можно выделить две подстроки str_1 и *tail* таких, что $str = str_1 \circ tail$, где «$\circ$» обозначает объединение (конкатенацию) двух строк. Строка str_1 удовлетворяет правилу (1), а строка *tail* содержит произвольные символы, в том числе она может быть пустой. В результате работы функция *parse* возвращает строку *tail*.

Запишем спецификацию функции *parse* на формальном языке с помощью предусловия и постусловия [6,16]. Для записи формул используем язык исчисления предикатов с равенством [17]. Будем придерживаться следующих соглашений по записи формул и терминологии.

1. Для любой переменной указываем её тип.

   Простой (конкретный) тип — это множество значений этого типа вместе с совокупностью операций и отношений, в которых эти значения могут участвовать [18]. Из простых типов, описанных в [17], используем $\mathbb{N}$ — множество

натуральных чисел, $\mathbb{Z}$ — множество целых чисел, *char* — множество символов, *bool* — множество логических констант $\{true, false\}$.

Родовой (составной) тип — это (частичная) функция, которая по конкретным типам строит новый тип. Если применить эту функцию к нескольким конкретным типам, то получим составной конкретный тип. Родовым типом является декартово произведение n-множеств, его обозначаем как $T_1 \times T_2 \times \cdots \times T_n$ (или кратко $\prod_{i=1}^{n} T_i$), где $T_i,\ i = 1, \ldots, n$ — конкретные типы. Элементы декартова произведения n множеств $\prod_{i=1}^{n} T_i$ — упорядоченные n-ки или векторы. Пишем, что вектор $(t_1, ..., t_n) \in \prod_{i=1}^{n} T_i$, если $t_i \in T_i,\ i = 1, \ldots, n$.

2. Списки данных языка Пифагор также являются векторами, для удобства тип, которому принадлежит список данных из n-элементов, обозначаем как $(datalist \prod_{i=1}^{n} T_i)$. Строка в языке Пифагор является частным случаем списка данных, содержащим символы типа *char*. Строки записываем как совокупность символов, заключенную в кавычки «“» и «”». Например, строка *str*=“123text” длины $n = 7$ будет иметь тип $(datalist \prod_{i=1}^{7} char)$.

3. Функция (в формуле) — однозначное отображение одного вектора длины n в вектор длины m. Если f — функциональный символ для обозначении функции, $(x_1, \ldots, x_n)$ — исходный вектор типа $\prod_{i=1}^{n} X_i$, $(y_1, \ldots, y_m)$ — результирующий вектор типа $\prod_{i=1}^{m} Y_i$, то запись $f(x_1, \ldots, x_n)$ используется для обозначения результирующего вектора $(y_1, \ldots, y_m)$.

4. Множество функций, отображающих вектор размера n в вектор размера m, является декартовым произведением $n + m$ множеств, обозначаем $(X \to Y)$, где $X = \prod_{i=1}^{n} X_i$, $Y = \prod_{i=1}^{m} Y_i$.

5. Для краткости формулы на языке логики заменяем функциями (от свободных переменных формулы), которые принимают значения в булевском множестве *bool*: если формула истинна для данного значения переменных, то функция принимает значение $true$.

6. При записи формул упускаем скобки после кванторов, которые применяются ко всему выражению, следующему после них.

Опишем множество функций, аргументом которых является целое число n_1, а результатом — строка длиной $(2n_1 - 1)$, удовлетворяющая условию (1):

$$\forall n_1 \in \mathbb{N}\ \exists f \in (\mathbb{N} \to (datalist \prod_{i=1}^{2n_1-1} char))\ \left(f(1) = \text{“x”}\right) \wedge$$
$$\wedge \left(f(2) = \text{“+xx”} \vee f(2) = \text{“*xx”}\right) \wedge$$
$$\wedge \left(\bigvee_{i=1}^{n_1-1} \left(f(n_1) = \text{“+”} \circ f(i) \circ f(n_1 - i)\right) \vee \left(f(n_1) = \text{“*”} \circ f(i) \circ f(n_1 - i)\right)\right), \quad (2)$$

где символ «$\circ$» соответствует операции конкатенации строк.

Обозначим (2) с помощью функции $F(f) \in \left(\mathbb{N} \to (datalist \prod_{i=1}^{2n_1-1} char)\right) \to bool$. Тогда предусловие P к функции $parse$ можно записать на языке логики следующим образом:

$$F(f) \wedge \left(\exists n \in \mathbb{N}\ \exists m \in \mathbb{Z}\ \left((m > 0) \wedge \forall str \in (datalist \prod_{i=1}^{2n-1+m} char)\right.\right.$$
$$sublist(str, 1, 2n-1) = f(n) \wedge \exists tail \in (datalist \prod_{i=1}^{m} char)$$
$$\left.\left. tail = sublist(str, 2n, 2n-1+m)\right)\right),$$

где $sublist(s, p, q)$ — функция, возвращающая подстроку строки s, начиная с позиции p до q; возвращает пустую строку, если $p = q$.

Предусловие P описывает структуру множества функций F, и требования к входному аргументу str: любой входной аргумент str является строкой длины $2n - 1 + m$, в которой можно выделить подстроку, длиной $2n - 1$, удовлетворяющую условию (1) (совпадает с $f(n)$ — результатом, возвращаемым одной из функций f множества F), оставшаяся часть строки $tail$ имеет длину m.

Постусловие Q функции $parse$ зададим следующей формулой:

$$return = tail,$$

где $return$ обозначает результат вычислений программы.

Для доказательства корректности косвенно рекурсивной функции $parse$ преобразуем её в прямую рекурсию. Множество всех связанных функций для $parse$ имеет вид:

$$\Omega(parse) = \{parse(fp, fm, fn), fp(parse), fm(parse), fn()\},$$

граф всех связанных функций приведён на рис. 5.

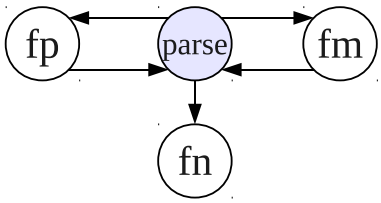

Рис. 5. Граф всех связанных функций для *parse*

Fig. 5. The graph of all connected functions for the function *parse*

4.1. Преобразование рекурсии *parse* в прямую при объединении кода

Связи функций из $\Omega(parse)$ допускают объединение кода функций для удаления косвенной рекурсии. В $\Omega(parse)$ функция fn является нерекурсивной, воспользуемся замечанием 2 алгоритма преобразования рекурсий из раздела 2.3. и объединим код функции fn с кодом *parse*. Полученную функцию обозначим $parsefn$,

$$\Omega(parsefn) = \{parsefn(fp, fm), fp(parsefn), fm(parsefn)\}.$$

В $\Omega(parsefn)$ отсутствуют независимые рекурсивные функции. Функции fp и fm имеют одну входную дугу, поэтому, согласно пункту 4 алгоритма преобразования рекурсий, объединим их код с кодом функции $parsefn$. Полученной функции дадим имя $parseUn$. Эта функция является прямой рекурсией и не требует задания УРФ. Ниже приведён исходный код $parseUn$:

```
parseUn<<funcdef str // Функция parseUn, принимающая строку str с
                     // арифметическим выражением
{   s1   << str:1;   // Сохранение первого символа строки str в s1
    case << ( (s1,'+'):=, // Сравнение s1 с тремя допустимыми
              (s1,'*'):=, //  вариантами значений
              (s1,'x'):=
            ):?;          // Выбор номера пути вычисления и
                          //сохранение номера в case
    act << (  // Формирование списка act трёх различных путей вычислений
            {str:-1:parseUn:parseUn }, //Результат объединения кода с fp
            {str:-1:parseUn:parseUn }, //Результат объединения кода с fm
            {str:-1}                   //Результат объединения кода с fn
          );
    act:case:.           // Выбор нужного пути и запуск вычисления
    >> return;           // Возврат результата вычислений
}
```

Докажем частичную корректность $parseUn$ с помощью метода, описанного в [6, 7]. Суть метода заключается в разметке дуг информационного графа программы [5] формулами, при использовании аксиом встроенных функций и теорем для функций

с ранее доказанной корректностью, а также преобразовании графа. В результате получается несколько полностью размеченных информационных графов с разметкой (ИГР), каждый из которых сворачивается в формулу. Если все, полученные формулы, тождественно истинны, то программа частично корректна.

Предусловие и постусловие *parseUn* совпадают с предусловием и постусловием *parse* (см. раздел 2.3.). Информационный граф *parseUn* приведён на рис. 6а. Для удобства все дуги графа пронумерованы. Считаем, что номер выходной дуги задаёт и номер оператора (вершины).

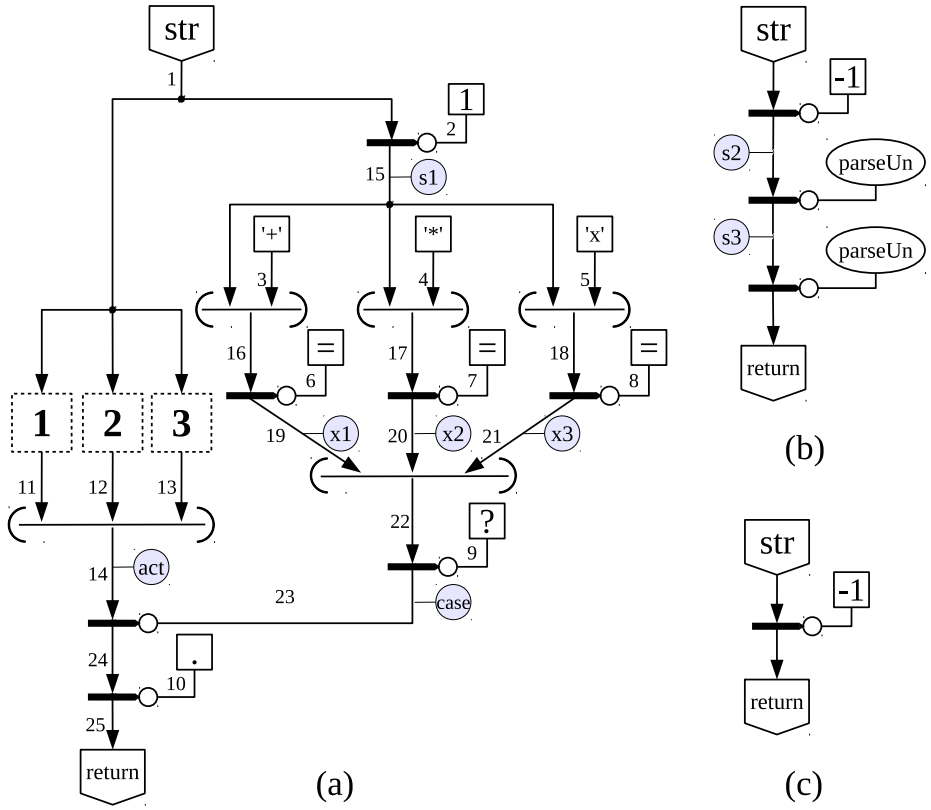

Рис. 6. Информационные графы функции *parseUn*, к некоторым дугам приписаны идентификаторы, обозначенные серыми кружками; (a) — исходный информационный граф функции *parseUn*, все дуги графа пронумерованы, задержанные списки обозначены пунктиром и пронумерованы; (b), (c) — информационные графы функции *parseUn* после расщепления и раскрытия задержанных списков

Fig. 6. Data flow graphs of the function *parseUn*, some arcs have identifiers that are denoted by gray circles; (a) — the initial data flow graph of the function *parseUn*, all arcs are enumerated, delay lists are marked with the dashed line and enumerated; (b), (c) — data flow graphs of the function *parseUn* after splitting and delay release

Дуги графа *parseUn* размечаются в порядке готовности данных: если все входные дуги узла размечены, можно размечать выходной узел. Узел 1 размечен предусловием, дуги 2–10 являются выходными дугами констант и получают автоматическую разметку, дуги 11–13 — выходные дуги задержанных списков, которые до снятия задержки являются константами. Дуга 14 с идентификатором *act* — выходная дуга списка данных также размечается автоматически. Дуга 15 с идентификатором

s_1 может быть размечена на основе аксиом функции выбора элемента из списка. В результате к дуге s_1 приписывается формула $(s_1 \in char \wedge s_1 = select(str, 1))$, где $select(x, i)$ — функция выбора i-го элемента из списка x. Дуги 16–18 являются выходными дугами списков данных и размечаются автоматически, после разметки дуги s_1. Далее, на основе аксиом для функции «=», можно разметить дуги 19–21. В результате к дуге x_1 припишутся две формулы:

$$(x_1 \in bool) \wedge (x_1 = true) \wedge (s_1 = \text{`}+\text{'}),$$

$$(x_1 \in false) \wedge (x_1 = false) \wedge \neg(s_1 = \text{`}+\text{'}),$$

где одинарные кавычки «'» используются для записи символьной константы типа $char$. К дугам x_2 и x_3 будут приписаны аналогичные формулы, в которых заменён идентификатор дуги и символьная константа на '$*$' и 'x' соответственно.

Дуга 22 размечается автоматически. К 23 дуге с идентификатором $case$, при разметке на основе аксиомы для функции «?», приписываются три формулы:

$$(case \in int) \wedge (case = 1) \wedge (x_1 = true) \wedge (x_2, x_3 = false),$$

$$(case \in int) \wedge (case = 2) \wedge (x_2 = true) \wedge (x_1, x_3 = false),$$

$$(case \in int) \wedge (case = 3) \wedge (x_3 = true) \wedge (x_1, x_2 = false).$$

Применение $case$ как функции к аргументу arg при выполнении оператора интерпретации 24 приведёт к расщеплению исходного графа на три, в каждом из которых произойдёт раскрытие задержанного списка при применении функции «.». Пусть номера полученных графов совпадают с номерами их задержанных списков. Первый и второй полученные графы одинаковые и приведены на рис. 6b, а третий — на рис. 6c.

Рассмотрим первый граф. Его предусловие P_1 будет следующим: $P \wedge (s_1 \in char) \wedge s_1 = select(str, 1) \wedge (s_1 = \text{`}+\text{'})$, а постусловие $Q_1 = Q$. То есть предусловие P_1 отличается от исходного предусловия P, тем, что в нём присутствуют формулы, которые описывают причины выбора данного пути вычисления: первый символ строки str равен символу '+'. Для разметки дуги s_2 используются аксиомы функции выбора элемента из списка, которая при отрицательном входном аргументе удаляет элемент из списка. В результате к дуге s_2 приписывается формула:

$$s_2 \in (datalist \prod_{i=0}^{2n-2+m} char) \wedge s_2 = sublist(str, 2, 2n - 1 + m).$$

Далее к результату s_2 применяется функция $parseUn$, то есть производится рекурсивный вызов. Для доказательства частичной корректности программы достаточно предположить, что все рекурсивные вызовы функции возвращают верный результат, если их аргумент удовлетворяет предусловию. Покажем, что s_2 удовлетворяет предусловию $parseUn\ P$. Так как строка str удовлетворяет предусловию и первый её символ $s_1 = \text{`}+\text{'}$ (следует из P_1), значит, $f(n)$ (из P) по правилу (2) будет иметь вид $f(n) = \text{“}+\text{”} \circ f(k) \circ f(n-k)$, для некоторого k. На языке логики это утверждение можно выразить следующей формулой:

$$\exists k \in \mathbb{N}\ (k < n) \wedge f(n) = \text{“}+\text{”} \circ f(k) \circ f(n-k).$$

А следовательно, при удалении первого символа из $str = f(n) \circ tail$, получим $s_2 = f(k) \circ f(n-k) \circ tail$. Таким образом, в начале строки s_2 стоит подстрока $f(k)$, удовлетворяющая (1), а «хвост» $tail_1 = f(n-k) \circ tail$ имеет длину $m_1 = 2(n-k) - 1 + m$. Тогда по индуктивному предположению s_3 — результат применения функции $parseUn$ к s_2, будет удовлетворять постусловию: $s_3 = tail_1$. И к дуге s_3 будет приписана формула:

$$\exists k \in \mathbb{N}\ (k < n) \wedge (\exists m_1 \in \mathbb{Z}\ (m_1 > 0) \wedge sublist(s_2, 1, 2k-1) = f(k) \wedge \\ \wedge s_3 = sublist(s_2, 2k, 2k-1+m_1))\,.$$

Таким образом, $s_3 = f(n-k) \circ tail$, а значит, тоже удовлетворяет предусловию функции $parseUn$. Тогда по индуктивному предположению имеем, что после применения $parseUn$ к s_3, $return = tail$. Постусловие Q всегда следует из этой формулы, следовательно, первый ИГР эквивалентен тождественно истинной формуле.

Второй граф (рис. 6b) полностью совпадает с первым. Отличие затрагивает только его предусловие P_2, в котором символ ‘+’, заменён на ‘$*$’. Остальные формулы разметки полностью совпадают с соответствующими формулами первого графа, поэтому второй ИГР также эквивалентен тождественно истинной формуле.

Рассмотрим третий граф (рис. 6c). Его предусловие P_3 имеет вид: $(P \wedge (s_1 \in char) \wedge s_1 = select(str, 1) \wedge (s_1 = \text{‘}x\text{’}))$, а постусловие $Q_3 = Q$. К входной строке str применяется функция удаления первого элемента, в результате к дуге $return$ приписывается формула $return = sublist(str, 2, 2n-1+m)$. Исходя из (2) и того, что $s_1 = \text{‘}x\text{’}$, $str = \text{‘}x\text{’} \circ tail$. Поэтому при удалении первого символа str получаем, что $return = tail$. Следовательно, третий ИГР эквивалентен тождественно истинной формуле. Из истинности всех трёх ИГР следует частичная корректность программы $parseUn$, а значит, функция $parse$ тоже частично корректна.

4.2. Преобразование рекурсии *parse* в прямую с помощью универсальной рекурсивной функции

Метод объединения кода функций применим в достаточно простых случаях, в сложных ситуациях необходимо использовать более сложный, но универсальный метод построения универсальной рекурсивной функции. Данный метод можно применить и для удаления взаимной рекурсии в функции $parse$. Для этого применим алгоритм преобразования произвольной рекурсии в прямую из раздела 2.3. к $\Omega(parse)$ ещё раз. В $\Omega(parse)$ функция fn является нерекурсивной, удалим её из $\Omega(parse)$ (её корректность докажем отдельно), получим

$$\Omega_1(parse) = \{parse(fp, fm), fp(parse), fm(parse)\}.$$

В $\Omega_1(parse)$ отсутствуют независимые от $parse$ рекурсивные функции. Пропустим шаг 4 алгоритма преобразования рекурсий и сразу зададим УРФ $pURF$. Ниже приведён исходный код $pURF$:

```
pURF<<funcdef sn{ // Функция pURF принимающая список (N, str),
                  // N - номер функции, объединённой в УРФ, str - строка
    N<<sn:1;      // Сохранение первого аргумента в N
```

```
    s<<sn:2;      // Сохранение второго аргумента в s
    csU << (
        (N,1):=,
        (N,2):=,
        (N,3):=   // Сравнение N с тремя допустимыми вариантами значений
    ):?;          // Выбор номера пути вычисления и его сохранение в csU
    actU << (
         {  block{ // Тело функции parse с заменой всех вызовов функций
                   // на вызов УРФ
                s1<<s:1;
                case << ( (s1,'+'):=, (s1,'*'):=, (s1,'x'):= ):?;
                act<< ( {(2,s):pURF}, {(3,s):pURF}, {s:fn} );
                act:case:.>>break;
            }
         },
         { // Тело функции fp с заменой всех вызовов функций на вызов УРФ
            ( 1, (1, s:-1):pURF ):pURF
         },
         { // Тело функции fm с заменой всех вызовов функций на вызов УРФ
            ( 1, (1, s:-1):pURF ):pURF
         }
    ); // Формирование списка actU 3-х различных путей вычислений
       // в зависимости от N
    actU:csU:. // Выбор нужного пути вычисления и активация вычисления
    >> return; // Возврат результата вычислений
}
```

Информационный граф функции $pURF$ приведён на рис. 7. Согласно разделу 3. предусловие и постусловие функции $pURF$ формируются из предусловий и постусловий входящих функций. Зададим предусловие и постусловие для fp (для fm всё аналогично). Функция fp должна принимать строку str_1, которая начинается с символа '+', далее идут две подстроки, удовлетворяющие (1), а оставшаяся часть строки $tail$ может содержать любые символы. В результате работы функция должна вернуть $tail$. На формальном языке предусловие P_{fp} и постусловие Q_{fp} соответственно выражаются следующими формулами:

$$P(str_1) \wedge (\exists k \in \mathbb{N}\ (k < n) \wedge str_1 = \text{“}+\text{”} \circ f(k) \circ f(n-k) \circ tail)\,,$$

$$return = tail,$$

где $P(str_1)$ — предусловие функции $parse$, в котором имя переменной str заменено на str_1. Тогда предусловие функции $pURF$ с входным аргументом $sn = (N, s)$ будет следующим:

$$((N = 1) \wedge P(s)) \vee ((N = 2) \wedge P_{fp}(s)) \vee ((N = 3) \wedge P_{fm}(s)).$$

После упрощения получаем предусловие P_{URF}:

$$P(s) \wedge ((N = 1) \vee (N = 2 \wedge select(s, 1) = \text{‘}+\text{’}) \vee (N = 3 \wedge select(s, 1) = \text{‘}*\text{’}))\,.$$

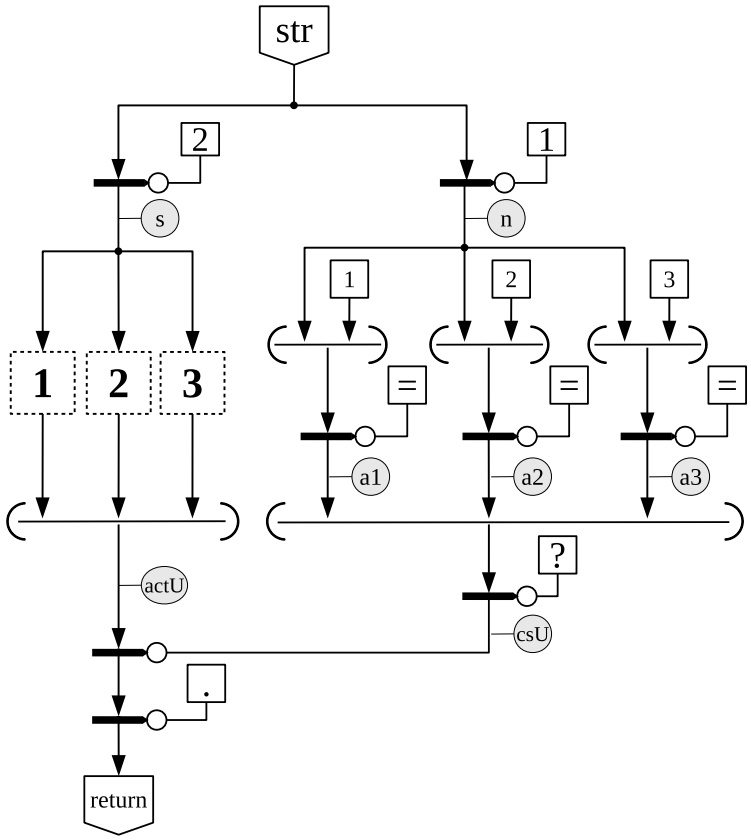

Рис. 7. Информационный граф функции $pURF$; к некоторым дугам приписаны идентификаторы, обозначенные серыми кружками; задержанные списки обозначены пунктиром и пронумерованы

Fig. 7. Data flow graph of the function $pURF$; some arcs has identifiers that are denoted by gray circles; delay lists are marked with the dashed line and enumerated;

Это означает, что строка s удовлетворяет $P(s)$, то есть представима в виде $f(n) \circ tail$; при $N = 2$ первый символ s равен ‘+’, а при $N = 3$ — ‘$*$’.

Постусловие функции $pURF$ имеет вид $((N = 1) \wedge Q) \vee ((N = 2) \wedge Q_{fp}) \vee ((N = 3) \wedge Q_{fm})$, и после упрощения приводится к формуле $return = tail$, то есть $Q_{URF} = Q$.

После автоматической разметки выходных дуг константных операторов и списков данных в графе $pURF$ размечаются дуги a_i, $i = 1, 2, 3$. К каждой дуге a_i приписывается пара формул:

$$(a_i \in bool) \wedge (a_i = true) \wedge (N = i),$$

$$(a_i \in bool) \wedge (a_i = false) \wedge \neg(N = i).$$

Далее, к дуге csU приписывается три формулы:

$$(csU \in int) \wedge (csU = i) \wedge (a_i = true) \bigwedge_{j \neq i} (a_j = false),\ i = 1, 2, 3.$$

Применение csU как функции к аргументу $argU$ приводит к расщеплению исходного графа на три, в каждый из которых попадёт один задержанный список.

При применении функции «.» произойдёт раскрытие задержанных списков. Назовём полученные в результате снятия задержки графы G_1, G_2 и G_3, где индекс совпадает с номером задержанного списка, попавшего в граф. Граф G_1 совпадёт с исходным графом функции $parseUn$, приведённым на рис. 6a, если в нём заменить идентификатор входного аргумента str на s (в этих графах отличаются константы задержанных списков, но на рисунке это не отражено). Графы G_2 и G_3 одинаковые и приведены на рис. 8a.

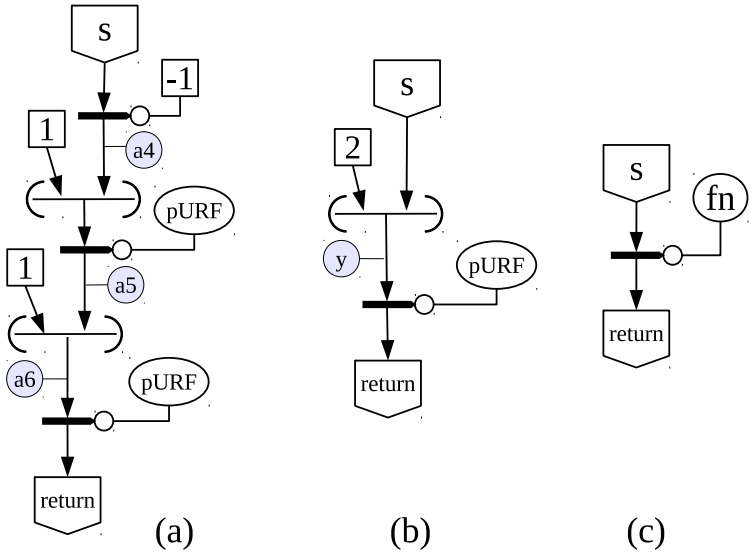

Рис. 8. Модифицированные информационные графы функции $pURF$

Fig. 8. Modified data flow graphs of the function $pURF$

Рассмотрим граф G_1. Его предусловие P_{URF1} имеет вид: $P_{URF} \wedge (N = 1)$. А после упрощения может быть записано как $P(s) \wedge (N = 1)$. Граф G_1 размечается так же, как исходный граф $parseUn$. Применение $case$ к act приводит к расщеплению графа на три, и после снятия задержки получаются графы, которые назовём G_{11}, G_{12} и G_{13}, где вторая цифра в индексе соответствует номеру попавшего в граф задержанного списка. Граф G_{11} приведен на рис. 8b. Граф G_{12} такой же, как G_{11}, только константа «2» заменяется на «3». Граф G_{13} приведён на рис. 8c.

Разметим граф G_{11}. Предусловие G_{11} имеет вид:

$$P_{URF} \wedge (N = 2) \wedge (select(s, 1) = \text{`}+\text{'}).$$

Значит, для аргумента (N, s), $N = 2$, а строка s представима как $f(n) \circ tail$ и первый её символ равен '+'. Это эквивалентно тому, что существует натуральное число k, такое что s представимо как $\text{`}+\text{''} \circ f(k) \circ f(n-k) \circ tail$.

В графе G_{11} константа «2» и аргумент s формируют список данных, которому присваивается идентификатор y. Далее к y применяется рекурсивный вызов функции $pURF$. Покажем, что y удовлетворяет предусловию функции $pURF$. Действительно, из предусловия G_{11} следует $P_{URF}(2, s)$ — предусловие функции $pURF$, для которого входной аргумент $(N, s) = (2, s)$. Тогда по индуктивному предположению рекурсивный вызов $pURF(y)$ вернёт верный ответ и к дуге $return$ припишется

формула: $return = tail$. Следовательно, постусловие выполнено и полностью размеченный граф G_{11} эквивалентен тождественно истинной формуле.

Для G_{12} всё аналогично, поэтому после разметки он также будет соответствовать тождественно истинной формуле.

Рассмотрим граф G_{13} (рис. 8c). Предусловие G_{13} имеет вид:

$$P_{URF} \wedge (N = 3) \wedge (select(s, 1) = \text{`}x\text{'}).$$

К аргументу s применяется функция fn, а результат её работы присваивается $return$.

Докажем корректность функции fn. Зададим предусловие P_{fn} и постусловие Q_{fn} для функции fn:

$$P(str_3) \wedge (str_3 = \text{“}x\text{”} \circ tail)\,,$$

$$return = tail.$$

В теле функции fn единственная функция, которая применяется ко входному аргументу, — это функция удаления первого элемента из списка. В результате из строки str_3 удаляется первый символ ‘x’, поэтому результат $return = sublist(s, 2, 1 + m) = tail$, что и требуется в постусловии. Следовательно, функция fn корректна.

На основе доказанной теоремы (корректности fn) можно разметить выходную дугу графа G_{13}: $return = tail$. Из последней формулы следует постусловие Q_{URF}. Значит, полностью размеченный граф G_{13} сворачивается в тождественно истинную формулу.

Остаётся разметить графы G_2 и G_3 (рис. 8a). Рассмотрим граф G_2, для G_3 всё аналогично. Предусловие P_{URF2} имеет вид: $P_{URF} \wedge (N = 2)$. После упрощения P_{URF2} можно записать $P(s) \wedge (N = 2) \wedge (select(s, 1) = \text{`}+\text{'})$, откуда следует, что для некоторого натурального k строку s можно представить в виде “ $+$ ” $\circ\, f(k) \circ f(n-k) \circ tail$.

В начале к аргументу s применяется функция удаления первого элемента из списка, и результату $f(k) \circ f(n-k) \circ tail$ присваивается идентификатор a_4. Список данных $(1, a_4)$ — допустимый аргумент для рекурсивного вызова функции $pURF$, поэтому по индуктивному предположению результат a_5 удовлетворяет постусловию $pURF$ и равен $f(n-k) \circ tail$. Новый формируемый список данных $(1, a_4)$ также удовлетворяет предусловию $pURF$, поэтому по индуктивному предположению $return$ будет равен $tail$. Откуда следует постусловие Q_{URF}. Значит, полностью размеченный G_2 (и аналогичный ему G_3) эквивалентен тождественно истинной формуле.

Таким образом, корректность $pURF$ доказана, из этого следует корректность функции $parse$.

5. Заключение

В работе рассмотрен алгоритм преобразования произвольной рекурсии в прямую при помощи универсальной рекурсивной функции. Для функций на языке Пифагор показано, как изменяется предусловие и постусловие программы при построении УРФ и объединении кода функций. Доказано, что из корректности УРФ следует корректность исходных функций. Рассмотрен пример доказательства корректности программы на языке Пифагор при объединении кода функций и построении УРФ.

Преобразование рекурсий может проводиться автоматически, поэтому описанный алгоритм можно внедрить в инструментальное средство поддержки доказательства программ на языке Пифагор.

Список литературы / References

[1] Непомнящий В. А., Рякин О. М., *Прикладные методы верификации программ*, Радио и связь, М., 1988, 255 с.; [Nepomnyashchiy V. A., Ryakin O. M., *Prikladnye metody verifikatsii programm*, Radio i svyaz, M., 1988, 255 pp., (in Russian).]

[2] Boyer R.S., Moore J.S., *A computational logic*, Academic Press, New York, 1979, 420 pp.

[3] Vazou N., Seidel E.L., Jhala R., Vytiniotis D., Peyton-Jones S., "Refinement Types for Haskell", *SIGPLAN Not.*, **49**:9 (2014), 269–282.

[4] Giesl J., "Termination of nested and mutually recursive algorithms", *Journal of Automated Reasoning*, **19**:1 (1997), 1–29.

[5] Легалов А. И., "Функциональный язык для создания архитектурно-независимых параллельных программ", *Вычислительные технологии*, **10**:1 (2005), 71–89; [Legalov A. I., "Functional language for creating of architectural independent parallel programs", *Computational Technologies*, **10**:1 (2005), 71–89, (in Russian).]

[6] Кропачева М. С., Легалов А. И., "Формальная верификация программ, написанных на функционально-потоковом языке параллельного программирования", *Моделирование и анализ информационных систем*, **19**:5 (2012), 81–99; English transl.: Kropacheva M. S., Legalov A. I., "Formal Verification of Programs in the Functional Data-Flow Parallel Language", *Automatic Control and Computer Sciences*, **47**:7 (2013), 373–384.

[7] Ushakova M. S., Legalov A. I., "Automation of Formal Verification of Programs in the Pifagor Language", *Modeling and Analysis of Information Systems*, **22**:4 (2015), 578–589.

[8] Giesl J., "Termination of nested and mutually recursive algorithms", *Journal of Automated Reasoning*, **19**:1 (1997), 1–29.

[9] Bevers E., Lewi J., "Proving termination of (conditional) rewrite systems", *Acta Informatica*, **30**:6 (1993), 537–568.

[10] Plumer L., *Termination Proofs for Logic Programs*, Lecture Notes in Artificial Intelligence, **446**, 1990, 142 pp.

[11] Giesl J., "Termination Analysis for Functional Programs Using Term Orderings", *International Static Analysis Symposium*, LNCS, **983**, 1995, 154–171.

[12] Walther C., "On Proving the Termination of Algorithm by Machine", *Artificial Intelligence*, **71**:1 (1994), 101–157.

[13] Кушниренко А. Г., Лебедев Г. В., *Программирование для математиков*, Наука, М., 1988, 384 с.; [Kushnirenko A. G., Lebedev G. V., *Programmirovanie dlya matematikov*, Nauka, M., 1988, 384 pp., (in Russian).]

[14] Головешкин В. А., Ульянов М. В., *Теория рекурсии для программистов*, Физматлит, М., 2006, 296 с.; [Goloveshkin V. A., Ul'yanov M. V., *Teoriya rekursii dlya programmistov*, Fizmatlit, M., 2006, 296 pp., (in Russian).]

[15] Мальцев А. И., *Алгоритмы и рекурсивные функции*, Наука, М., 1986, 368 с.; [Mal'tsev A. I., *Algoritmy i rekursivnye funktsii*, Nauka, M., 1986, 368 pp., (in Russian).]

[16] Hoare C. A. R., "An axiomatic basis for computer programming", *Communications of the ACM*, **10**:12 (1969), 576–585.

[17] Ушакова М. С., "Семантика типов данных функционально-потокового языка параллельного программирования Пифагор", *Образовательные ресурсы и технологии*, 2016, № 2(14), 263–269; [Ushakova M. S., "Semantics of program data types for functional data-flow parallel programming language Pifagor", *Educational resources and technologies*, 2016, № 2(14), 263–269, (in Russian).]

[18] Шилов Н. В., *Основы синтаксиса, семантики, трансляции и верификации программ*, Издательство СО РАН, Новосибирск, 2009, 340 с.; [Shilov N. V., *Osnovy sintaksisa, semantiki, translyatsii i verifikatsii programm*, Izdatelstvo SO RAN, Novosibirsk, 2009, 340 pp., (in Russian).]

Ushakova M. S., Legalov A. I., "Verification of Programs with Mutual Recursion in the Pifagor Language", *Modeling and Analysis of Information Systems*, **25**:4 (2018), 358–381.

DOI: 10.18255/1818-1015-2018-4-358-381

Abstract. In the article, we consider verification of programs with mutual recursion in the data driven functional parallel language Pifagor. In this language the program could be represented as a data flow graph, that has no control connections, and has only data relations. Under these conditions it is possible to simplify the process of formal verification, since there is no need to analyse resource conflicts, which are present in the systems with ordinary architectures. The proof of programs correctness is based on the elimination of mutual recursions by program transformation. The universal method of mutual recursion of an arbitrary number of functions elimination consists in constructing the universal recursive function that simulates all the functions in the mutual recursion. A natural number is assigned to each function in mutual recursion. The universal recursive function takes as its argument the number of a function to be simulated and the arguments of this function. In some cases of the indirect recursion it is possible to use a simpler method of program transformation, namely, the merging of the functions code into a single function. To remove mutual recursion of an arbitrary number of functions, it is suggested to construct a graph of all connected functions and transform this graph by removing functions that are not connected with the target function, then by merging functions with indirect recursion and finally by constructing the universal recursive function. It is proved that in the Pifagor language such transformations of functions as code merging and universal recursive function construction do not change the correctness of the initial program. An example of partial correctness proof is given for the program that parses a simple arithmetic expression. We construct the graph of all connected functions and demonstrate two methods of proofs: by means of code merging and by means of the universal recursive function.

Keywords: data driven functional parallel programming, Pifagor programming language, correctness of recursions, elimination of mutual recursion, universal recursive function

On the authors:
Mariya S. Ushakova, orcid.org/0000-0003-4234-2714, graduate student,
Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo str., Krasnoyarsk 660074, Russia, e-mail: ksv@akadem.ru

Alexander I. Legalov, orcid.org/0000-0002-5487-0699, doctor of science,
Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo str., Krasnoyarsk 660074, Russia, e-mail: legalov@mail.ru

Acknowledgments:
The research is supported by RFBR (research project No 17-07-00288).