Графы в предикатном программировании

Шелехов В.И., Янбулатов Д.Р., Ворожбитов Н.О., Каблуков И.В., Тумуров Э.Г.

A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia
Novosibirsk State University
vshel@iis.nsk.su

В предикатном программировании графы представлены множествами. Эффективность предикатных программ достигается применением оптимизирующих трансформаций.

Ключевые слова: алгоритмы на графах, предикатное программирование, GraphBLAS, дедуктивная верификация, система верификации Why3.

1 Введение

Эффективная и корректная реализация алгоритмов на графах является сложнейшей задачей, в частности, для высокоэффективных кластеров GPU. В последние семь лет для повышении эффективности алгоритмов на графах используется метод линейной алгебры для разреженных матриц на базе стандартного интерфейса GraphBLAS [23], базирующегося на математической формализации [10] и предоставляющего исчерпывающий набор примитивов для эффективной реализации алгоритмов на графах. На базе стандарта GraphBLAS [23] разработано более семи разных библиотек для работы с графами. Это SuiteSparse [34], GBTL [35], GraphBLAST [21] и другие [4].

Интерфейс GraphBLAS [23] весьма сложен, в частности потому, что ориентирован на высокую степень оптимизации графовых алгоритмов при реализации библиотеки на его базе. Однако во многих приложениях не требуется предельной оптимизации. Учитывая это, библиотека LAGraph [4] предоставляет существенно более простой интерфейс. Тем не менее, во всех существующих подходах реализация графовых алгоритмов является сложной. О дедуктивной верификации даже речи не идет. Конечно, в мире достаточно много исследований по дедуктивной верификации графовых алгоритмов, но не в практической реализации.

Объектно-ориентированная технология, применяемая в библиотеках графовых алгоритмов, призвана скрывать внутри библиотеки детали сложной реализации, предоставляя пользователю простой интерфейс. В действительности ровно наоборот. Потребности оптимизации алгоритмов вынуждают усложнять интерфейс.

В настоящей работе предлагается развитие технологии предикатного (функционального) программирования [1, !!!!!] для написания программ обработки графов на языке предикатного программирования P [1] с возможностью применения набора оптимизирующих трансформаций к предикатной программе для получения эффективной программы на императивном языке

типа Си. Язык P ориентирован на использование в качестве языка публикации алгоритмов на графах вместо псевдокода. Другое направление исследований – дедуктивная верификация предикатных программ обработки графов.

В языке P для представления графов используется множества. Аналогичный подход при построении типов данных используется в системах моделирования и верификации Event-B [15] и TLA+ [16]. Однако там не стоит задача получения эффективного кода. Эффективная трансляция программ, оперирующих множествами, проблематична. В предикатном программировании для этого используются оптимизирующие трансформации.

Эффективность предикатных программ достигается применением оптимизирующих трансформаций [2]. Результатом трансформаций является эффективная императивная программа в стиле языка Си. Трансформации предикатной программы проводятся в два этапа. На первом этапе предикатная программа транслируется на промежуточный язык gP. Здесь применяется набор трансформаций, ранее представленный на многих программах [!!!!!!!]. Операции над множествами на первой фазе остаются неизменными. Их трансфомация проводится на второй фазе и описана в разделе 4.

Во втором разделе данной работы описываются основные конструкции и расширения языка предикатов P [1]. Введены операции линейной алгебры для матриц и векторов на полукольцах для эффективной реализации на графических процессорах. Эти конструкции определены на математическом уровне GraphBLAS [10], как в языке PyGB [3]. В третьем разделе определяются основные структуры данных для представления графов в языке P. В четвертом разделе определены трансформации операций с множествами вершин и ребер. В пятом разделе метод предикатного программирования иллюстрируется на нескольких простых программах обработки графов. В шестом разделе описывается предикатная программа для алгоритма Косараджу и ее трансформация. В конце статьи обзор и заключение.

2 Предикатное программирование

Предикатное программирование [1, …..] – это чистое функциональное программирование с нотацией в стиле якыка Си. Наряду с функциями используются предикаты (операторы). Расширения предикатного программирования P [1] по отношению к функциональному программированию, в частности гиперфункции, в принципе могут быть представлены конструкциями классического функционального программирования:

2.1 Предикатная программа

Полная предикатная программа на языке P [1] состоит из набора предикатных программ вида:

<program name>**(**<argument declarations>**:** <result declarations>**)**
pre <precondition>
post <postcondition>
measure <expression>
{ <statement> }

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [3, 4, 8, 9, 15????]. Мера задается только для рекурсивных программ и используется для доказательства их завершения.

Ниже представлены основные конструкции языка P [1]: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

<variable> = <expression>
{<statement1>; <statement2>}
<statement1> || <statement2>
if (<boolean expression>**)** <statement1> **else** <statement2>
<program name> (<argument list>**:** <result list>)
<type> <space> <list of variable names>

Операторы, входящие в параллельный оператор, независимы по результатам. Циклы и указатели запрещены в языке P.

Допускается использование функциональной формы. Вызов предиката B(x**:** y), где y – список переменных, эквивалентен оператору присваивания |y| = B(x) или просто y = B(x), где B(x) интерпретируется как вызов функции. В частности, оператор суперпозиции вида B(x**:** z); C(x, z**:** y) можно записать так: y = C(x, B(x)). Оператор присваивания вида y = E в конце программы, где y – списсок результатов программы, может быть заменен выражением E.

Эффективность предикатных программ достигается применением оптимизирующих трансформаций [2]. Они определяют отличную от классической оптимизацию среднего уровня с преобразованием предикатной программы в эффективную императивную программу на языке подобном Си. Базовыми трансформациями являются:

- замена всех вхождений одной переменной на другую переменную (склеивание переменных);
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) с помощью массивов и указателей. В частности, через односвязные списки, строковые объекты и т.д.

Также могут применяться эквивалентные упрощения.

Предикатный программист сам оптимизирует создаваемую предикатную программу таким образом, чтобы применением оптимизирующих трансформаций получить из нее эффективную императивную программу. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи [!!!!!!]. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Иной менталитет в функциональном программировании. Программисту следует в элегантном стиле, просто и компактно написать функциональную программу, а получить из нее эффективный код – это задача транслятора; в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду.

С помощью оптимизирующих трансформаций можно получить программу предельной эффективности. Оптимизирующие трансформации применяются автоматически для

большинства программ. Для сложных программ используется корректируемая программистом трансформационная программа, в которой фиксируются применяемые трансформации.

2.2 Гиперфункции

Гиперфункция – это программа с несколькими ветвями результатов [1, разд. 3.1, 3.3]. Гиперфункция B(x: y: z) имеет две ветви результатов y и z. Выполнение гиперфункции заканчивается вычислением результатов одной из ветвей для этой ветви; результаты других ветвей не вычисляются. Ветвь может не иметь результатов. Вызов гиперфункции B(x: y: z) обычно используется в следующем операторе switch:

switch B(x) { **case** y: <statement1> **case** z: <statement2>}

Испульзуется также другая форма операторе switch:

switch B(x: y: z) { **case** <statement1> **case** <statement2>}

Ограничитель **switch** обычно опускается. Оператор switch в некотором роде является гибридом условного оператора и оператора суперпозиции. Использование гиперфункций делает программу быстрее и короче [8].

После каждого последнего исполняемого оператора программы-гиперфункции указывается, какой ветвью гиперфункции завершается программа. Оператор перехода #1 фиксирует завершение по первой ветви, а #2 – по второй.

Если одна из альтернатив **case** в операторе switch становится пустой, оператор switch вырождается. Взамен может появиться оператор вызова гиперфункции вида, например, B(x: y: z #1). При завершении второй ветвью вызова вырабатыаается значение z; далее программа, содержащая данный вызов, завершает исполнение первой ветвью. Если вызов B(x: y: z #1) завершается первой ветвью, то вырабатывается значение y, и далее программа продолжается со следующего оператора после данного вызова гиперфункции.

Возможна ситуация, когда заведомо известно, что одна из ветвей вызова гиперфункции будет недостижимой. В этом случае в позиции недостижимой ветви вставляется оператор ##. Напримкр, B(x: y: ##).

2.3 Типы

Типы **bool**, **nat**, **int**, **real** и **char** являются *примитивными*. Значение типа **struct**(T_1 f_1, ..., T_n f_n) - это *структура* из n значений, имеющих *поля* f_1, f_2,..., f_n и имеющих типы T_1, T_2,..., T_n, соответственно. Тип *кортеж* (T_1, T_2,..., T_n) определяет упорядоченный набор значений типов T_1, T_2,..., T_n.

Значение типа **array**(T_e, T_i) представляет собой *массив* с элементами типа T_e и индексами конечного типа T_i. Конструкция **for** (T_i i) <expression> определяет новый массив. Каждому элементу массива с индексом i, принадлежащему T_i, присваивается значение <expression>, зависящее от i. Для массива A конструкция A **with** (i: x) определяет новый массив, заменяя элемент массива A[i] значением выражения x. Конструкция A **with** (i: x, j: y) определяется как (A **with** (i: x)) **with** (j: y).

Тип **array**(T_e, T_i, T_j) определяет *матрицу* с индексами конечных типов T_i и T_j. Для матрицы A конструкции A[i,] и A[, j] определяют массивы (векторы), которые являются i-й строкой и

j-м столбцом матрицы A, соответственно. Операция A# определяет транспонирование матрицы A.

Пусть $A^{m\times n}$ (с m строками и n столбцами) и $B^{n\times k}$ - матрицы, c^n и d^m - векторы. Для *полукольца* с произвольными операторами сложения ⊕ и умножения ⊗ [10], вводятся операции умножения матрица на матрицу A ⊕.⊗ B, матрица на вектор A ⊕.⊗ c и вектор на матрицу d ⊕.⊗ A для компактной и эффективной реализации графовых алгоритмов на GPU.

Значением *алгебраического типа* **union**(K_1, …, K_n) является один из конструкторов K_1, …, K_n. Конструктор определяется именем конструктора и набором полей, подобно структуре; набор полей может быть пустым. Типичными объектами алгебраических типов являются списки и деревья. *Список* определяет последовательность элементов некоторого типа T. Описание типа списка выглядит следующим образом:

type list (**type** T) = **union**(nil, cons(T car, list(T) cdr));

Тип list имеет два конструктора: nil, обозначающий пустой список, и cons, определяющий список, первый элемент которого представлен полем car, а остальная часть списка определяется полем cdr. Обработка некоторого объекта s типа list реализуется оператором switch с конструкторами списков в качестве альтернатив:

switch s { **case** nil**:** <statement1> **case** cons(x, v)**:** <statement2>}

Здесь локальные переменные для элемента x и списка v используются в <statement2>. Ниже пример функции, вычисляющей длину списка.

```
len(list(int) s): nat
{  switch s {
        case nil: 0
        case cons(a, u): len(u)+1
}};
```

Любой элемент типа **set**(T) - это *множество*, которое является конечным подмножеством типа T. Тип T называется *базовым типом* для типа **set**. *Пустое множество* представляется как {}. Конструкция {x} - это множество с единственным элементом x. Результатом операции add(x, s) является множество s∪{x}. Операция s+u задает объединение множеств s и u. Операция s^u задает пересечение множеств s и u. Операция s\u определяет вычитание множества u из множества s. Конструкция |s| обозначает число элементов множества s. Гиперфункция pick(s **: :** x, s1) недетерминированно выбирает произвольный элемент x, если он существует. Множество s1 – это остальное множество без элемента x, то есть s = s1∪{x}. Первая ветвь - это результат выбора гиперфункции, когда множество s пусто. Заметим, что в случае непустого множества s гиперфункция pick быстрее, чем последовательное применение двух действий: проверка на пустоту и выбор произвольного элемента из s. Гиперфункци\ pick определяется следующей спецификацией:

pick(**set**(T) s **: :** T x, **set**(T) s1) **pre** 1: s=∅ **post** 2: s=s1∪{x} & x∈s & x∉s1

Предусловие по первой ветви: s=∅. Предусловие по второй ветви: s≠∅. Оно не указывается, поскольку является дополнением к предусловию первой ветви. Постусловие по первой ветви отсутствует, так как там нет результатов.

Выражение типа **set** может быть использовано в качестве типа переменных в предикатной программе. В частности, множество может быть типом индексов массива и коллекции вершин графа.

2.4 Конструкции в стиле императивного программирования

В классическом функциональном программировании внесение в программу императивных и других удобных для программирования конструкций реализуется неявно через аппарат монад. Без монад функциональные программы потеряли бы свою компактность и привлекательность. В предикатном программировании такого рода конструкции определены явно. Программа с ними легко приводима к правильной предикатной программе.

Если результат программы требуется сохранить в той же переменной, что и аргумент b, то для результата используется имя b′. При трансляции b′ будет заменено на b. Штрих в конце имени b′ означает, что b′ будет склеена с переменной b при трансляции.

Модификацию переменных удобнее записывать явно в стиле императивного программирования. Например, оператор b = b + 1 трактуется как b′ = b + 1. Оператор присваивания вида a[i] = x является эквивалентом a′ = a **with** (i**:** x). Допускается вызов вида G(...**:** a[i]), трактуемый как G(...**:** x); a′ = a **with** (i**:** x). Оператор **if** (E(x)) A(x**:** y) **else** y′ = y заменяется на укороченный условный оператор **if** (E(x)) A(x**:** y), поскольку оператор y′ = y удаляется.

Неизменяемые переменные, то есть константы, удобно трактовать как глобальные переменные, опуская их в списках аргументов предикатных программ. Это существенно укорачивает и упрощает предикатную программу. При этом желательно, чтобы имена констант нигде не использовались в программе для обозначения других объектов.

При последовательном сканировании множества вершин s вместо вызова pick(s **: :** x, s1) обычно используется pick(s **: :** x). Переменная x называется *итератором*. В случаях, когда s1 требуется в спецификации или в программе, используется x@ в качестве s1. Допускается вызов вида pick(x **: :** x1), трактуемый как pick(x@ **: :** x1). Параметр x, используемый как итератор, определяется описанием: **iter** X x, где X – итерируемое множество. Фактичекий параметр вида **iter** X в вызове программы определяет установку итератора по множеству X. Для переменной-итератора определены операции отношения с элементами множества. Отношение y<x истинно для элементов y, уже пройденных итератром. Однако x<y, если y принадлежит s1. Для сканирования множества ребер итератор не применяется

3 Графы в предикатном программировании

Граф определяется множеством *вершин* и множеством ребер. *Ребро* – это пара вершин графа. Вершины x и y ребра (x, y) называются *смежными* (соседними). Ребро соединяет x и y и *инцидентно* вершинам x и y. Вершины x и y также называются *концевыми* ребра (x, y). *Степень вершины* – это количество инцидентных ей рёбер.

В языке P [1] любая вершина имеет тип **node**. Множество вершин графа имеет следующий предопределенный тип:

type nodeset = **set**(**node**);

Здесь **node** – базисный тип для множества вершин. Ребра графа имеют тип:

struct(source: **node**, target: **node**) .

Для ребра e его поля e.source и e.target являются концевыми, то есть e = (e.source, e.target). Будем также писать e.1 и e.2 и оперировать с ребром e как с кортежем типа (**node**, **node**). Тип множества ребер является предопределенным типом:

type edgeset = **set**(**struct**(source: **node**, target: **node**));

Вершины и ребра графа могут быть снабжены *атрибутами* (или метками в терминах теории графов), являющимися объектами произвольных типов. Атрибуты – реальное содержимое графа, тогда как вершины и ребра определяют лишь скелет графа. Атрибуты определяются в программе как массивы, индексами которых являются все вершины или все ребра. Ниже приведен пример описания графа с набором вершин nodes и набором ребер edges, а также атрибутами: name – строковый атрибут вершин, size – целым атрибутом ребер.

```
nodeset nodes;
edgeset edges;
array(string, nodes) name;
array(int, edges) size;
```

По умолчанию тип **node** совпадает с типом **nat**. Если для конкретного приложения требуется другой базовый тип для набора вершин, отличный от **nat**, пользователю разрешается изменить тип вершины. Например, пользователь может ввести следующее описание:

```
type node = (int, 1..10);
```

Тип **set**(**node**), обозначаемый как nodeset, определяет наиболее общую и адекватную структуру для набора вершин. Любой подграф является частью структуры исходного графа. Извлечение подграфа не требует копирования из исходного графа. Решение использовать в качестве типа множества вершин диапазон вида m..n было бы хуже, так как извлечение подграфа и дальнейшая работа с ним как с независимым графом станет невозможным без копирования. Эффективная обработка на GPU графов, полученных в результате декомпозиции (без перемещения) исходного графа, вершины которого изначально расположены в диапазоне m..n, является новой задачей.

Для произвольного графа, определенного множеством вершин nodes и множеством ребер edges, необходимо дополнительное условие, определяемое предикатом is_graph: концевые вершины каждого ребра принадлежат множеству nodes. Кроме того, множества nodes и edges должны быть конечными.

```
formula is_graph(nodeset nodes, edgeset edges) =
                    finite(nodes) & finite(edges) &
                    ∀ node a, b. (a, b) ∈ edges => a ∈ nodes & b ∈ nodes;
```

Эффективность программ обработки графов достигается путем применения оптимизирующих трансформаций операций с множествами, учитывающих особенности конкретного приложения.

4 Трансформации операций над графами

Здесь определяется набор трансформаций различных операций над графами в программе на языке gP.

4.1 Трансформации операций с вершинами

4.2 Трансформации операций с ребрами

4.3 Представления матриц

5 Некоторые простые программы

Технология предикатного программирования иллюстрируется на простой программе вычисления степеней вершин. Также определяются представления графа в виде матрицы смежности и списков смежности.

5.1 Определение степеней вершин

В качестве первого примера рассмотрим алгоритм вычисления степеней вершин неориентированный графа G = (V, E) с множеством вершин V и множеством ребер E. *Неориентированный граф* определяется следующей спецификацией:

formula unordered (nodeset nodes, edgeset edges) =
    is_graph(nodes, edges) & ∀ **node** a, b. (a, b)∈edges => a ≠ b & (b, a)∉edges;

Ребро (a, b) совпадает с ребром (b, a). Однако в мнжестве ребер edges указывается только одно. В частности, не допускаются ребра вида (a, a).

Программа Degree по множеству ребер E строит массив d. Элемент d[j] есть степень вершины j. Предикат degree определяет степень вершины v.

```
formula degree(node v, nodeset V, edgeset E : nat) = | { V x | (x, v) ∈ E ∨ (v, x) ∈ E } |;
formula postDeg (nodeset V, edgeset E, array(nat, V) d) = ∀ V v. d[v] = degree(v, V, E);
Degree(nodeset V, edgeset E : array(nat, V) d)
    pre   unordered(V, E)
    post  postDeg (V, E, d)
{ NodesDegree(V, E, E, for (V k) 0) }
```

Главная программа Degree вызывает рекурсивную программу NodesDegree. Ее третий параметр eds – это оставшееся необработанное подмножество ребер. Таким образом, мы можем выразить E как E = eds0 ∪ eds, где eds0 – уже обработанное подмножество ребер. Массив deg, четвертый параметр, получен в результате обработки подмножества eds0. Вначале deg инициализируется нулевым массивом.

```
NodesDegree(nodeset V, edgeset E, eds, array(nat, V) deg: array(nat, V) d)
    pre  unordered(V, E) & eds ⊆ E & ∀ v ∈ V. deg[v]= degree(v, V, E\eds)
    post  postDeg (V, E, d)
  { switch pick(eds){ case : deg
                      case (node, node) x, edgeset eds1:
                         (node i, node j) = x;
                         NodesDegree( V, E, eds1, deg with (i: deg[i]+1, j: deg[j]+1) )
  }}
```

Программа NodesDegree недетерминированно выбирает следующее ребро из множества eds с помощью гиперфункции pick. При eds = ∅ срабатывает первая ветвь гиперфункции, и результатом работы NodesDegree становится исходный массив deg. В случае eds ≠ ∅ гиперфункция pick выбирает ребро (i, j) из множества eds. Остается набор ребер eds1 = eds \ {(i, j)}. Рекурсивный вызов NodesDegree применяется к модифицированному массиву deg для вершин i и j. В этой модификации элемент deg[i] заменяется на deg[i]+1, а deg[j] заменяется на deg[j]+1.

В дальнейшем ограничитель **switch** будет отсутствовать. Вместо переменной x в **case** части вставляются компоненты i и j из определения компонент кортежа (**node** i, **node** j) = x, которое далее становится ненужным. Наконец, параметры V и E сделаем глобальными. Данные изменения дают следующую программу.

```
nodeset V;
edgeset E;
Degree( : array(nat, V) d)
     pre   unordered(V, E)
     post  postDeg (V, E, d)
{ NodesDegree( E, for (V k) 0 ) }

NodesDegree(edgeset eds, array(nat, V) deg: array(nat, V) d)
     pre  unordered(V, E) & ∀ v ∈ V. deg[v] = degree(v, V, E \ eds)
     post  postDeg (V, E, d)
   { pick(eds){  case : deg
                 case (node i, node j), edgeset eds1:
                         NodesDegree(eds1, deg with (i: deg[i]+1, j: deg[j]+1) )
   }}
```

Определим последовательность трансформаций, применяемых к предикатной программе Degree для получения эффективной программы на языке gP. Первая трансформация – склеивание переменных по следующим правилам:

$$\begin{aligned} eds &\leftarrow eds1 \\ d &\leftarrow deg \end{aligned}$$

Все вхождения deg заменяются на d, eds1 заменяется на eds. Далее хвостовая рекурсия в программе NodesDegree заменяется циклом. Рекурсивный вызов заменяется присваиваниями аргументов рекурсивного вызова соответствующим формальным параметрам. Программа NodesDegree модифицируется следующим образом:

```
NodesDegree(edgeset eds, array(nat, V) d: array(nat, V) d)
{ loop { pick(eds){  case : return d
                     case (node i, j), eds:
                          | eds = eds, d = d with (i: d[i]+1, j: d[j]+1) |
       }}
}
```

Далее применяются упрощающие трансформации. Раскрывается мультиприсваивание. Устраняется тождественный оператор присваивания eds = eds. Раскрывается with-модификация для d.

```
NodesDegree(edgeset eds, array(nat, V) d: array(nat, V) d)
{ loop { pick(eds){ case : return d
                     case (node i, j), eds: d[i] := d[i]+1; d[j] := d[j]+1
       }}
}
```

Далее тело программы NodesDegree подставляется на место вызова в главной программе.

```
Degree( : array(nat, V) d)
{    | edgeset eds, array(nat, V) d |:= | E, for (V j) 0 |
     loop { pick(eds){  case : return d
                        case (node i, j), eds: d[i] := d[i]+1; d[j] := d[j]+1
          }}
}
```

Последняя трансформация – раскрытие мультиприсваивания. Получаем программу на языке gP:

```
Degree( : array(nat, V) d)
{ edgeset eds = E;
  array(nat, V) d = for (V j) 0;
  loop { pick(eds){   case : return d
                      case (node i, j), eds: d[i]++; d[j]++
       }}
}
```

Определим второй этап трансформации для полученной программы на языке gP. Множество вершин V представляется диапазоном 0..Vm-1, где Vm – число вершин. Множество ребер кодируется парным способом с использованием массивов Es и Et. Для реализации pick переменная eds заменяется индексом k по массивам Es и Et. Применим следующий набор трансформаций:

```
nodeset V → nat Vm
edgeset E → nat En, array(node, 0..En-1) Es, Et
d = for (V j) 0  → for (j=0; j<Vm; j++) d[j] = 0
edgeset eds = E → nat k = 0
pick(eds : : (node i, j), eds) → pick(k, En, Es, Et : : (node i, j), k)
```

Получим программу:

```
Degree( : array(nat, V) d)
{ nat k = 0;
  array(nat, V) d;  for (j=0; j<Vm; j++) d[j] = 0;
  loop { pick(k, En, Es, Et : : (node i, j), k){   case : return d case : d[i]++; d[j]++  }}
}
```

Для данного представления определим гиперфункцию pick.

```
pick(nat k, nat En, array(node, 0..En-1) Es, Et : : (node i, j), nat k)
{if (k< En) { |i, j| = |Es[k], Et[k]|; k=k+1 #2 } else #1}
```

Поставим тело pick на место вызова.

```
Degree( : array(nat, V) d)
{ nat k =0;
  array(nat, V) d;  for ( nat j=0; j<Vm; j++) d[j] = 0;
  loop {  if (k< En) { node i, j; |i, j| = |Es[k], Et[k]|; k=k+1; d[i]++; d[j]++  }
          else return d
}}
```

Заключительные действия: раскрытие групповых операторов присваивания, устранение переменных i и j, замена вхождений типа V на **nat**, оформление цикла и возврат итогового массива d через указатель. Отметим, что V должно бы быть заменено на тип диапазона 0..Vm-1, однако такого типа нет в языках типа Си.

```
array(nat, nat) *Degree(nat Vm, nat En, array(nat, nat) Es, Et)
{ array(nat, nat) d;
  for (nat j=0; j<Vm; j++) d[j] = 0;
  for (nat k =0; k < En; k++) { d[Es[k]]++; d[Et[k]]++ };
  return &d
}
```

5.2 Списки смежности неориентированного графа

В неориентированном графе *список смежности* вершины – это множество соседей вершины. Список смежности вершины можно построить по ее ребрам. Списки смежности для всех вершин определяются атрибутным массивом типа **array**(V, nodeset), где V – множество вершин графа. При трансформации программы списки смежности обычно кодируются через односвязные списки. Отметим, что общепринятый термин «список смежности» не вполне адекватен, так как здесь мы оперируем множеством смежности.

Рассмотрим программу neighbors для построения списков смежности неориентированного графа. Результатом работы программы является массив списков смежности. Пусть имеется неориентированный граф G = (V, E) с множеством вершин V и множеством ребер E. Эти множества определяются глобальными константами. Формула neighborsP определяет необходимое и достаточное условие принадлежности вершины списку смежности.

```
nodeset V;
edgeset E;
type AdjV = array(nodeset, V);
formula neighborsP (nodeset V, edgeset E, AdjV adj) =
        ∀ node a, b. a∈V & b∈V => ( b∈adj[a] <=> (a, b)∈E ∨ (b, a)∈E )

neighbors( ): AdjV;
        pre unordered(V, E)
        post  neighborsP(V, E, result)
{ neighborsR (for (V j) {}, E) }
```

Программа neighbors обращается к рекурсивной программе neighborsR, которая сканирует ребра программы. Ее параметр eds – это оставшееся необработанное подмножество ребер. Для пройденной части множества ребер уже построен массив списков смежности, представленный первым параметром adjSet.

```
neighborsR(AdjV adjSet, edgeset eds): AdjV
        pre unordered(V, E) & eds ⊆ E & neighborsP(V, E\eds, adjSet)
        post  neighborsP(V, E, result)
        measure | eds |
{pick(eds){ case : adjSet
              case (node i, j), eds1 :
                  neighborsR(adjSet with (j: add(i, adjSet[j]), i: add(j, adjSet[i]) ), eds1 )
}}
```

Программа neighborsR через вызов pick выбирает следующее ребро (i, j) при непустом множестве ребер. Вершина i добавляется к множеству adjSet[j] для вершины j. Вершина j добавляется к множеству adjSet[i] для вершины i. Напомним, что результатом функции add(x, s) является множество $s \cup \{x\}$.

Определим последовательность трансформаций, применяемых к предикатной программе neighbors для получения программы на языке gP. Первая трансформация – склеивание переменных:

$$eds \leftarrow eds1$$

Все вхождения eds1 заменяется на eds. Далее хвостовая рекурсия в программе neighborsR заменяется на цикл. Рекурсивный вызов заменяется присваиваниями аргументов рекурсивного вызова соответствующим формальным параметрам. Программа neighborsR модифицируется следующим образом:

```
neighborsR(AdjV adjSet, edgeset eds): AdjV
{loop { pick(eds){ case : return adjSet
                   case (node i, j), eds :
        | adjSet = adjSet with (j: add(i, adjSet[j]), i: add(j, adjSet[i]) ), eds = eds1 |
      }}
}
```

Далее раскрывается мультиприсвоение. Устраняется тождественный оператор присваивания eds = eds1. Раскрывается with-модификация для adjSet.

```
neighborsR(AdjV adjSet, edgeset eds): AdjV
{loop { pick(eds){ case : return adjSet
                   case (node i, j), eds :
                        adjSet[j] := add(i, adjSet[j]); adjSet[i] := add(j, adjSet[i])
      }}
}
```

Далее тело программы neighborsR подставляется на место вызова в главной программе.

```
neighbors( ): AdjV;
{    | AdjV adjSet, edgeset eds | := | for (j in V) {}, E) |
     loop { pick(eds){ case : return adjSet
                       case (node i, j), eds :
                          adjSet[j] := add(i, adjSet[j]); adjSet[i] := add(j, adjSet[i])
      }}
}
```

Последняя трансформация – раскрытие мультиприсваивания. Получаем программу на языке gP:

```
neighbors( ): AdjV;
{ AdjV for (V j) adjSet[j] ={};
  edgeset eds = E;
  loop { pick(eds){  case : return adjSet
                     case (node i, j), eds :
                       adjSet[j] := add(i, adjSet[j]); adjSet[i] := add(j, adjSet[i])
         }}
}
```

Определим второй этап трансформации для полученной программы на языке gP. Для реализации pick переменная eds заменяется индексом x по массивам Es и Et. Применим следующий набор трансформаций:

```
nodeset V → nat Vm
edgeset E → nat En, array(node, 0..En-1) Es, Et
AdjV adjSet = for (j in V) {} → for (j=0; j<Vm; j++) adjSet[j] = {}
edgeset eds = E → nat x = 0
pick(eds : : (node i, j), eds) → pick(x, En, Es, Et : : (node i, j), x)
```

Получим программу:

```
neighbors( ): AdjV;
{ AdjV adjSet; for (j=0; j<Vm; j++) adjSet[j] = {};
  nat x = 0;
  loop { pick(x, En, Es, Et : : (node i, j) x'){ case : return adjSet
                                                 case (node i, j) :
                  adjSet[j] := add(i, adjSet[j]); adjSet[i] := add(j, adjSet[i])
  }}
}
```

Поставим тело pick на место вызова.

```
neighbors( ): AdjV;
{ AdjV adjSet; for (j=0; j<Vm; j++) adjSet[j] = {};
  nat x = 0;
  loop {if(x < En) { node i, j; |i, j| = |Es[x], Et[x]|; x = x + 1;
                     adjSet[j] := add(i, adjSet[j]); adjSet[i] := add(j, adjSet[i]) }
        else return adjSet
}}
```

Заключительные действия: раскрытие групповых операторов присваивания, устранение переменных i и j, оформление цикла и возврат итоговой матрицы adjSet через указатель.

```
AdjV *neighbors(nat Vm, nat En, array(node, 0..En-1) Es, Et)
{ AdjV adjSet; for (j=0; j<Vm; j++) adjSet[j] = {};
  for (nat x =0; x < En; x++) {
        adjSet[Et[x]] := add(Es[x], adjSet[Et[x]]);
        adjSet[Es[x]] := add(Et[x], adjSet[Es[x]])
  }
  return &adjSet
}
```

5.3 Списки соседей, упорядоченных по убыванию степеней вершин

В неориентированном графе программа Neighbors строит списки смежности (или списки соседей), упорядоченные по убыванию степеней вершин. Списки соседей для всех вершин определяются атрибутным массивом типа **array**(V, list(V)), где V – множество вершин графа. Программа Neighbors использует массив D степеней вершин.

```
nodeset V;
edgeset E;
array(V, nat) D = Degree(V, E);
type ArListV = array(V, list(V));

Neighbors( ): ArListV
{    NodesNeighbors(E, for (V j) nil ) }
```

В рекурсивной программе NodesNeighbors два дополнительных параметра. Параметр eds – это оставшееся необработанное подмножество ребер. Параметр S – массив списков соседей, построенный для пройденной части множества ребер E. Вначале S инициализируется пустыми списками.

Для очередного ребра (i, j), выбранного из множества E, к списку S[i] добавляется j, а к списку S[j] добавляется i. Это реализуется с помощью программы Insert, обеспечивающей упорядоченность модифицированного списка.

```
NodesNeighbors( edgeset eds, ArListV S): ArListV
{ pick(eds)  {  case : S
                case (node i, j), eds1:
                  NodesNeighbors(eds1, S with (j: Insert(nil, S[j], D, i), i: Insert(nil, S[i], D, j))
}}
```

Рекурсивная программа Insert вставляет вершину i в список s в порядке убывания степеней вершин. Исходный список есть конкатенация c + s, где c – просмотренная часть списка вершин со степенями, не меньше D[i], а s – оставшаяся часть списка.

```
Insert(list(V) c, s, array(V, nat) D, node i): list(V) s
{ switch s { case nil: c + i
             case cons(a, u) : if ( D[a]<D[i] )  c + i + s else Insert(c + a , u, i)
}}
```

Напомним, что аргументами конкатенации могут быть списки и элементы списка.

Определим последовательность трансформаций, применяемых к предикатной программе NodesNeighbors для получения эффективной программы на языке gP. Первая трансформация – склеивание переменных:

$$eds \leftarrow eds1$$

Все вхождения eds1 заменяются на eds. Далее хвостовая рекурсия в программе NodesNeighbors заменяется циклом. Рекурсивный вызов заменяется присваиваниями аргументов рекурсивного вызова соответствующим формальным параметрам. Программа NodesNeighbors модифицируется следующим образом:

```
NodesNeighbors(edgeset eds, ArListV S): ArListV
{ loop { pick(eds) { case : return S
                     case (node i, j), eds:
                  | eds = eds, S = S with ( j: Insert(nil, S[j], D, i), i: Insert(nil, S[i], D, j)) |
       }
}}
```

Далее применяются упрощающие трансформации. Раскрывается мультиприсвоение. Устраняется тождественный оператор присваивания eds = eds. Раскрывается with-модификация для S.

```
NodesNeighbors(eds, ArListV S): ArListV
{ loop { pick(eds) { case : return S
                     case (node i, j), eds:
                            S[j] := Insert(nil, S[j], D, i); S[i] := Insert(nil, S[i], D, j)
       }
}}
```

Далее тело программы NodesNeighbors подставляется на место вызова в главной программе.

```
Neighbors( ): ArListV
{ | edgeset eds, ArListV S | := | E, for (V j) nil |
  loop { pick(eds) { case : return S
                     case (node i, j), eds:
                            S[j] := Insert(nil, S[j], D, i); S[i] := Insert(nil, S[i], D, j)
       }
}}
```

Последняя трансформация – раскрытие мультиприсваивания. Получаем программу на языке gP:

```
Neighbors( ): ArListV
{ edgeset eds = E;
  ArListV S = for (V j) nil;
   loop { pick(k) { case : return S
                    case (node i, j), eds:
                            S[j] := Insert(nil, S[j], D, i); S[i] := Insert(nil, S[i], D, j)
        }
}}
```

Определим второй этап трансформации для полученной программы Neighbors на языке gP.

```
nodeset V → nat Vm
edgeset E → nat En, array(node, 0..En-1) Es, Et
edgeset eds = E → nat k =0
S = for (V j) nil  → for (j=0; j<Vm; j++) S[j] = 0
pick(eds : : (node i, j), eds) → pick(k, En, Es, Et : : (node i, j), k)
```

Получим программу Neighbors:

```
Neighbors( ): ArListV
{ nat k = 0;
  ArListV S; for (j=0; j<Vm; j++) S[j] = 0;
  loop { pick(k, En, Es, Et : : (node i, j), k) {
                    case : return S
                    case (node i, j), eds:
                            S[j] := Insert(nil, S[j], D, i); S[i] := Insert(nil, S[i], D, j)
}}}
```

Поставим тело pick на место вызова.

```
Neighbors( ): ArListV
{ nat k = 0;
  ArListV S; for (j=0; j<Vm; j++) S[j] = 0;
  loop {  if (k < En) { node i, j; |i, j| = |Es[k], Et[k]|; k=k+1;
                        S[j] := Insert(nil, S[j], D, i); S[i] := Insert(nil, S[i], D, j) }
          else return S
        }
}
```

Заключительные действия для программы Neighbors: раскрытие групповых операторов присваивания, устранение переменных i и j, оформление цикла и возврат итогового массива S через указатель. Отметим, что V должно бы быть заменено на тип диапазона 0..Vm-1, однако такого типа нет в языках типа Си.

```
ArListV *Neighbors(nat Vm, nat En, array(node, 0..En-1) Es, Et)
{ ArListV S; for (j=0; j<Vm; j++) S[j] = 0;
  for (nat k = 0; k < En; k++) { S[Et[k]] := Insert(S[Et[k]], D, i);
                                 S[Es[k]] := Insert(S[Es[k]], D, j) }
  return &S
}
```

Определим последовательность трансформаций, применяемых к предикатной программе Insert для получения эффективной программы на языке gP. Первая трансформация - склеивание переменных:

$$s \leftarrow u, s'$$

Хвостовая рекурсия в программе Insert заменяется циклом. Рекурсивный вызов заменяется присваиваниями аргументов рекурсивного вызова соответствующим формальным параметрам:

```
Insert(list(V) c, s, array(V, nat) D, node i): list(V) s
{ loop { switch s { case nil: return c + i
                    case cons(a, u) : if (D[a]<D[i]) return c + i + s
                                      else | c = c + a, s = u, i = i |
}}}
```

Далее применяются упрощающие трансформации. Раскрывается мультиприсвоение. Устраняется тождественный оператор присваивания i = i.

```
Insert(list(V) c, s, array(V, nat) D, node i): list(V) s
{ loop { switch s { case nil: return c + i
                    case cons(a, u) : if (D[a]<D[i]) return c + i + s
                                      else c = c + a; s = u
        }
}}
```

Применим трансформации для односвязного списка[?]. Заменим цикл **loop** на **while**.

Используемые трансформации выписать явно.

```
list(nat) → struct list { list *next; nat data; }
```

```
Insert(list(V) c, s, array(V, nat) D, node i): list(V) s
{ while (s != NULL) {
      if (D[s→data]<D[i]) { list(nat) temp = s;
                            s→data = i;
                            s→next = temp;
                            return c; }
      else s = s→next;
  }
  if (c == NULL) return new list(i);
  else { c→next = new list(i); return c }}
```

Заключительные действия: оформление возвращаемого итогового листа s через указатель, замена V на **nat**.

```
list(nat) *Insert(list(nat) c, s, array(nat, nat) D, node i)
{ while (s != NULL) {
      if (D[s→data]<D[i]) { list(nat) temp = s;
                            s→data = i;
                            s→next = temp;
                            return c; }
      else s = s→next;
  }
 if (c == NULL) return new list(i);
 else { c→next = new list(i); return c }}
```

5.4 Матрицы смежности

Для графа с множеством вершин $V = \{v_1, ..., v_n\}$ *матрица смежности* – это квадратная $n \times n$ матрица A такая, что ее элемент A_{ij} равен единице, когда есть ребро из вершины v_i в вершину v_j, и равен нулю, когда ребра нет. Для *мультиграфа*, в котором несколько ребер могут иметь одну и ту же пару концевых вершин, A_{ij} – это количество ребер, соединяющих v_i и v_j. Матрицы смежности используются в эффективно реализуемых операциях линейной алгебры [9, 10].

Ниже приведена предикатная программа BuildAdjMatrix, которая строит матрицу смежности по множеству ребер ориентированного графа. В формулу adjMatrixP единичные элементы матрицы смежности соответствуют ребрам графа.

```
nodeset V;
edgeset E;
type Matrix(nodeset vs) = array(nat, vs, vs);
formula adjMatrixP(edgeset E, Matrix(V) adjM) =
     ∀ node a, b. adjM[a, b] = ((a, b)∈E)? : 1 : 0;

BuildAdjMatrix( ): Matrix(V)
     pre is_graph(nodes, edges)
     post  adjMatrixP(edges, result)
{ BuildAdjMatr(for (V i, j) 0, E) }

BuildAdjMatr( Matrix(V) M, edgeset eds): Matrix(V)
     pre is_graph(nodes, edges) & eds ⊆ E & adjMatrixP(E\eds, M)
     post  adjMatrixP(edges, result)
     measure  | eds |
{ pick(eds){ case : M
               case  (i, j), eds1 : BuildAdjMatr( M with (i, j: 1), eds1 )
}}
```

Здесь каждое ребро (i, j), извлеченное из множества E, отображается в элемент матрицы M[i, j] = 1.

Определим последовательность трансформаций, применяемых к предикатной программе BuildAdjMatr для получения эффективной программы на языке gP. Первая трансформация – склеивание переменных:

$$eds \leftarrow eds1$$

Все вхождения eds1 заменяется на eds. Далее хвостовая рекурсия в программе BuildAdjMatr заменяется циклом. Рекурсивный вызов заменяется присваиваниями аргументов рекурсивного вызова соответствующим формальным параметрам. Программа BuildAdjMatr модифицируется следующим образом:

```
BuildAdjMatr( Matrix(V) M, edgeset eds): Matrix(V)
{ loop { pick(eds){ case : return M
                    case  (i, j), eds: | M = M with (i, j: 1), eds = eds1 |
  }}
 }
```

Далее раскрывается мультиприсваивание. Устраняется тождественный оператор присваивания eds = eds1. Раскрывается with-модификация для M.

```
BuildAdjMatr( Matrix(V) M, edgeset eds): Matrix(V)
{ loop { pick(eds){ case : return M
                case  (i, j), eds: M[i][j] := M[i][j] = 1
  }}
}
```

Далее тело программы BuildAdjMatr подставляется на место вызова в главной программе.

```
BuildAdjMatrix( ): Matrix(V)
{ | Matrix(V) M, edgeset eds | := | for (V i, j) 0, E |
  loop { pick(eds){  case : return M
                     case  (i, j), eds: M[i][j] := 1
  }}
}
```

Раскрываем мультиприсваивание. Получаем программу на языке gP:

```
BuildAdjMatrix( ): Matrix(V)
{ Matrix(V) M = for (V i, j) 0;
  edgeset eds = E;
  loop { pick(eds)){ case : return M
                     case  (i, j) , eds: M[i][j] = 1
  }}
}
```

Определим второй этап трансформации для полученной программы на языке gP. Применим следующий набор трансформаций:

```
nodeset V → nat Vm
edgeset E → nat En, array(node, 0..En-1) Es, Et
edgeset eds = E → nat x =0
pick(eds : : (node i, j), eds) → pick(x, En, Es, Et : : (node i, j), x)
M = for (V i, j) 0  → for (i=0; i<Vm; i++) for (j=0; j<Vm; j++) M[i][j] = 0
```

Получим программу:

```
BuildAdjMatrix( ): Matrix(V)
{  Matrix(V) M;
   for (i=0; i<Vm; i++) for (j=0; j<Vm; j++) M[i][j] = 0;
   nat x;
   loop { pick(x, En, Es, Et : : (node i, j), x'){  case : return M
                                                    case  (i, j): M[i][j] = 1
   }}
}
```

Поставим тело pick на место вызова.

```
BuildAdjMatrix( ): Matrix(V)
 { Matrix(V) M;
   for (i=0; i<Vm; i++) for (j=0; j<Vm; j++) M[i][j] = 0;
   nat x;
   loop {  if (x < En) { node i, j; |i, j| = |Es[x], Et[x]|; x=x+1; M[i][j] = 1 }
           else return M
   }
 }
```

Заключительные действия: раскрытие групповых операторов присваивания, устранение переменных i и j, оформление цикла и возврат итогового массива d через указатель. Отметим, что V должно бы быть заменено на тип диапазона 0..Vm-1, однако такого типа нет в языках типа Си.

```
Matrix(V) *BuildAdjMatrix(nat Vm, nat En, array(node, 0..En-1) Es, Et)
 { Matrix(V) M;
   for (i=0; i<Vm; i++) for (j=0; j<Vm; j++) M[i][j] = 0;
   for ( nat x = 0; x < En; x++) { M[Es[x]][Et[x]] = 1 }
   return &M
}
```

6 Алгоритм построения списка максимальных клик

Пусть G = (V, E) – неориентированный граф с множеством вершин V и множеством ребер E. Граф G является *полным*, если каждая вершина соединена ребром с любой другой вершиной. Пусть R – подмножество вершин графа G, то есть $R \subseteq V$. Подграф, *порожденный* подмножеством вершин R, – это другой граф с множеством вершин R и подмнжеством всех ребер из E, соединяющих пары вершин из R.

Подмножество R вершин неориентированного графа G называется *кликой*, если любые две вершины из R соединены ребром в графе G.

```
formula  is_click(nodeset  R, V, edgeset E) =
                    R ⊆ V & ∀ node  a, b. a ∈ R & b ∈ R  => (a, b) ∈ E;
```

Это эквивалентно утверждению, что подграф, порождённый R, является полным.

Максимальная клика — это клика, которую нельзя расширить добавлением в неё вершин. Иначе говоря, в графе нет клики большего размера, в которую она входит.

```
formula  maximal_click(nodeset  R, V, edgeset E) =
          is_click( R, V, E) & ∀ nodeset  U. R ⊂ U ⊆ V => ¬ is_click( U, V, E);
```

Эквивалентное определение максимальной клики представлено следующей леммой.

```
lemma  maximal_click(R, V, E) ⇔ is_click( R, V, E) & ∀ a ∈ V \ R. ∃ b ∈ R.  (a, b) ∉ E
```

Наибольшая клика — это клика максимального размера в данном графе.

Представим разработку предикатной программы для алгоритма Брона-Кербоша (Bron–Kerbosch) [38] нахождения всех максимальных клик в неориентированном графе с улучшениями алгоритма [39].

Исходный граф определим описаниями:

```
nodeset V;
edgeset E;
```

Результатом программы является список максимальных клик. Каждая клика имеет тип nodeset. Для списка клик введем тип listV:

```
type listV = list (nodeset);
```

В алгоритме необходимо будет использовать соседей вершины.

```
array(V, edgeset) N = neighbors(V, E);
```

Здесь N[i] – множество соседей i-ой вершины в графе G.

Сначала представим простой алгоритм BK0. В дальнейшем определим улучшения этого алгоритма: BK1 и BK2.

Аргументами программы BK0 являются множество вершин V и множество ребер E , определенные выше глобальными константами. Результат программы – список максимальных клик графа с вершинами V и ребрами E.

```
BK0( ): listV
{ if (V={}) nil else Clicks0({}, V, nil) }
```

Программа BK0 вызывает программу Clicks0 с аргуметами: R – текущая клика, расширяемая до максимальной, P – множество вершин-кандидатов для включения в текущую клику, S – список ранее построенных максимальных клик. Результатом программмы Clicks0 является полный список максимальных клик. В начале работы алгоритма текущая клика пуста, а кандидатами для включения в текущую клику являются все вершины графа. Это означает, что для каждой вершины строится по крайней мере одна максимальная клика.

```
Clicks0(nodeset R, P, listV S): listV
{ if (P={}) S + [[R]]
  else Clicks0C(R, P, iter P, S)
}
```

Если множество кандидатов P пусто, то клика R является максимальной. Она присоединяется к списку клик S и это становится результатом программы Clicks0. Заметим, что [[R]] обозначает список из единственного элемента R. Иначе, при непустом множестве P, запускается рекурсивная программа Clicks0C.

Если клика R непуста, каждая вершина из множества кадидатов является соседней для любой вершины в клике, т.е. $\forall\ x{\in}R\ \forall\ y{\in}P.\ (x, y) \in E$. Поэтому присоединение любого кандидата из P к клике образует новую клику большего размера. В программе Clicks0C множество кандидатов сканируется; очередной кандидат присоединяется к клике.

Далее представлена программа Clicks0C. Третий параметр – итератор k фиксирует множество оставшихся кандидатов из P для присоединения к клике R.

```
Clicks0C(nodeset R, P, iter P k, listV S: listV S')
{ pick(k){ case : S' = S              (* k@={} *)
           case node v :              (*k@={v}+ k'@ *)
                Clicks0(R + {v}, P ^ N[v], S: listV S1);
                Clicks0C(R, P, k', S1: S')
}}
```

Гиперфункция pick(k) выбирает очередную вершину v из множества оставшихся кандидатов k@. Множество k'@ получается из k@ удалением вершины v. При пустом множестве k@ программа завершается.

Выбранная вершина v присоединяется к клике R. Рекурсивно запускается программа Clicks0 на ограниченном множестве кандидатов $P{\cap}N[v]$, по размеру меньшим чем P, поскольку $v{\notin}N[v]$. Результатом этого вызова является список максимальных клик S1. Наконец, рекурсивно запускается программа Clicks0C для множества оставшихся кандидатов k'@.

Любая максимальная клика графа G = (V, E) будет одним из элементов списка, являющегося результатом программы BK0. Однако в списке возможны повторы. Если клика R непуста и имеются два разных кандидата x и y таких, что $(x, y) \in E$, то клики, построенные на x и на y, будут совпадать.

В программе BK1 множество P заменяется на X∪C, где X – уже обработанные кандидаты, а C – оставшиеся кандидаты для присоединения к клике R. При рекурсивном вызове Clicks1 множество кандидатов представлено множеством C∩N[v], где уже нет ранее обработанных кандидатов X, что препятствует повторному включению максимальной клики в список.

```
BK1( ): listV
{ if (V={}) nil else Clicks1({}, V, {}, nil) }

Clicks1(nodeset R, C, X, listV S): listV
{ if (C={} & X={}) S + [[R]]
  else Clicks1C(R, C, X, S)
}

Clicks1C(nodeset R, C, X, listV S: listV S')
{ pick(C){ case : S' = S                    (* C={} *)
           case node v, nodeset C1 :    (* C={v}+C1 *)
                Clicks1(R + {v}, C ^ N[v], X ^ N[v], S: listV S1);
                Clicks1C(R, C1, X + {v}, S1: S')
}}
```

Программа BK2 улучшает программу BK1 введением в программе Clicks1C опорного элемента u из полного множества кандидатов X∪C. В основном цикле сканируется множество кандидатов C \ N[u] вместо C.

```
formula  maxClickList(nodeset V, edgeset E, listV S) =
             ∀ nodeset R. mem(R, S) => maximal_click(R, V, E)
formula  postBK2(nodeset V, edgeset E, listV S) = maxClickList(V, E, S) &
                                                  плюс полная и без повторов
BK2( : listV S)
   pre   unordered(V, E)
   post  postBK2(V, E, S)
{ if (V={}) nil else Clicks2({}, V, {}, nil) }


Clicks2(nodeset R, C, X, listV S : listV S')
   pre  unordered(V, E) & is_click(R, V, E) & maxClickList(V, E, S) &
        C+X = Closests(R) & X∩C = Ø &
   post  postBK2(V, E, S')
{ if (C={} & X={}) S + [[R]]
  else { pick(C) { case : S' = S
                   case node u, nodeset C1 : Clicks2C(R, C \ N[u], X, S')
}}}
```

Программа Clicks2 аналогична Clicks1 и имеет тот же набор параметров. Вводится опорный элемент u, выбираемый из множества C. Множество соседей вершины u удаляются из множества C, что позволяет ускорить алгоритм. Программа Clicks2C фактически тождественна Clicks1C.

```
Clicks2C(nodeset R, C, X, listV S: listV S')
    pre  is_click(R, nodes, edges) & C=nodes\R ∩ neighbors(E, v) &
      X=nodes\R ∩ neighbors(E, v) & X∩C = Ø  & ∀ mem W S. is_maxclick(W, nodes, edges)
    post ∀ mem C S'. is_maxclick(C, nodes, edges)
{ pick(C){ case : S' = S                    (* C={} *)
           case node v, nodeset C1 :    (* C={v}+C1 *)
                Clicks2(R + {v}, C ^ N[v], X ^ N[v], S: listV S1);
                Clicks2C(R, C1, X + {v}, S1: S')
}}
```

7 Алгоритм нахождения изоморрфного подграфа

Имеются два графа: граф-образец $G = (V, E)$ и целевой граф $H = (U, F)$, где V и U – вершины, а E и F – ребра. Граф G не пустой и число вершин в нем не больше, чем в графе H.

```
nodeset V, U;
edgeset E, F;
axiom |V| > 0;
axiom |V| ≤ |U|;
```

Множество соседних вершин для вершины i в графе G обозначим через $N_G(i)$. Тогда степень i-ой вершины есть $|N_G(i)|$.

Обычный изоморфизм (точнее, гомоморфизм) образца G и цели H определяется инъективной (взаимно-однозначной) функцией $f: V \to U$, такой что

$$(i, j) \in E \Rightarrow (f(i), f(j)) \in F$$

Если функция f биективна (при $|V| = |U|$), графы G и H являются изоморфными.

Нахождение изоморфного подграфа графа H для образца G является одной из важнейших задач. Алгоритм, представленный здесь, использует некоторые улучшения [41] базового алгоритма Ульмана [40].

В алгоритме изоморфизм $f: V \to U$ представляется битовой матрицей M размерности $|V| \times |U|$. Определим тип матрицы:

```
type bit = {0, 1};
type Isom = array(bit, V, U);
```

Матрица M типа Isom кодирует изоморфизм $f: V \to U$: условие $M[i, j] = 1$ эквивалентно $f(i) = j$. Ввиду инъективности функции f в каждой строке матрице должна быть ровно одна единица, а в каждом столбце – не более одной единицы.

Программа matchIso находит изоморфный подграф перебором значений матрицы M. Вторая ветвь гиперфункции реализуется, если в графе H нет подграфов, изоморфных графу G.

```
matchIso( : Isom M : ){
    array(V, nat) D = Degree(V, E);
    array(U, nat) L = Degree(U, F);
    array(V, list(V)) X = Neighbors(V, E, D);
    array(U, list(U)) Y = Neighbors(U, F, L);
    initMatr( : Isom M0);
    findIso(M0 : M : #2)
};
```

Здесь D[i] – степень i-ой вершины в графе G, L[i] – степень j-ой вершины в графе H, X[i] – список соседей i-ой вершины в графе G, Y[j] – список соседей j-ой вершины в графе H. Списки соседних вершин в массивах X и Y упорядочены по убыванию степеней вершин – элементов списка. Степени вершин вычисляются программой Degree. Множества соседей, упорядоченные по убыванию степеней, строятся программой Neighbors. Переменные D, L, X и Y в дальнейшем используются как глобальные в программах initMatr и findIso.

Программа initMatr заполняет матрицу M0 единицами, за исключением тех элементов M[i, j], заполняемых нулями, для которых легко обнаруживается невозможность построить изоморфный подграф. Например, если степень i-ой вершины в графе G оказалось больше степени j-ой вершины в графе H, т.е. если $|N_G(i)|>|N_H(j)|$, то невозможно все соседние вершины для i-ой вершины отобразить на соседние вершины для j-ой вершины в графе H.

Программа findIso в каждом ряду матрицы M0 выбирает по одному ненулевому элементу, чтобы в итоге получить требуемое решение. После выбора очередного элемента некоторые элементы зануляются, поскольку они становятся недостижимыми. При отсутствии единичных элементов в текущем ряду происходит возврат в предыдущий ряд, где выбираетя следующий элемент.

7.1 Программа initMatr начального заполнения матрицы M

Программа initMatr для каждого элемента матрицы M[i, j] проверяет условие $|N_G(i)|\le|N_H(j)|$. Также проверяется, что соседи i-ой вершины графа G соответствуют соседям j-ой вершины в графе H по своим степеням.

```
initMatr( : Isom M0){
    M0 = for(V i, U j) initElem(i, j)
};
```

В параллельном цикле **for** для каждого элемента матрицы M[i, j] формируется начальное битовое значение с помощью программы initElem, проверяющей условие $|N_G(i)|\le|N_H(j)|$ для элемента матрицы, а также для списков соседей.

```
initElem(V i, U j): bit
{ if (D[i]<=L[j] & lessDegrees(X[i], Y[j])) 1 else 0 };
```

На входе программы lessDegrees два упорядоченных по убыванию степеней списка вершин из графов G и H. Если обнаруживается, что какая-то вершина из списка x не отображается на очередной элемент из списка y, то реализуется присваивание M[i, j]:=0.

```
lessDegrees(list(V) x, list(U) y): bool
{ if (x=nil) true
  else   if (D[x.car]>L[y.car]) false
         else lessDegrees(x.cdr, y.cdr)
};
```

7.2 Программа findIso поиска изоморфного подграфа

Программа-гиперфункция findIso находит изоморфный подграф перебором значений матрицы M, являющейся аргументом. Значение матрицы M получено вызовом программы initMatr. Результат гиперфункции findIso по первой ветви – матрица M1 – определяет изоморфизм f. Напомним, что условие M1[i, j] = 1 эквивалентно f(i) = j. Вторая ветвь гиперфункции, без результата, реализуется при отсутствии изоморфного подграфа в графе H.

```
findIso(Isom M : Isom M1 : ){
    pick(V: ##: i);
    pick(U:## : j);
    findIsoRow(i, j, M : M1 : #2)
};
```

Итератор i устанавливается на первый ряд матрицы M, а итератор j – на первый столбец. По первому ряду запускается программа-гиперфункция findIsoRow с теми же параметрами, что и программа findIso.

```
findIsoRow(iter V i, iter U j, Isom M : Isom M1 : ){
    nextElem(i, j : U j1 : #2);
    M2 := M;
    findIsoCur(i, j1, M2 : M1 #1 : );
    M[i, j1] = 0;
    findIsoRow( i, j1, M #1: M1 : #2)
};
```

Программа findIsoRow реализует просмотр i-го ряда матрицы M, начиная с j-ой позиции. Находится ближайшая позиция j1 такая, что M[i, j1] = 1. По этому элементу запускается программа-гиперфункция findIsoCur, которая пытается достроить матрицу M для следующих рядов. Если это удается, то результат – матрица M1 по первой ветви гиперфункции, а переход #1 завершает программу findIsoRow первой ветвью. Иначе реализуется вторая ветвь вызова findIsoCur, и просмотр i-го ряда матрицы M продолжается через рекурсивный вызов гиперфункции findIsoRow. Оператор M2 := M реализует копирование матрицы M, необходимое по-

сле выхода по второй ветви из findIsoCur; отметим, что M2 модифицируется внутри findIsoCur. При оптимизирующей трансформации M2 не склеивается с M.

```
findIsoCur(iter V i, iter U j, Isom M : Isom M1 : ){
    nullingEls(i, j, M:  M2);
    pick(i){ case : M1 = M2 #1
            case  i1:
                pick(U:## :  j1);
                findIsoRow(i1, j1, M2 : M1 #1 : #2);
}};
```

Программа findIsoCur, запускаемая для текущего выбранного элемента M[i, j] = 1, продолжает построение матрицы M. Сначала программа nullingEls обнуляет те элементы, которые оказываются недоступными после выбора элемента M[i, j]. Далее вызов pick(i) реализует переход к просмотру следующего ряда i1 программой findIsoRow. Однако если ряд i оказался последним (по первой ветви pick), то программа findIsoCur завершается первой ветвью с матрицей M2 в качестве конечного результата.

7.3 Программа nullingEls обнуления недостижимых элементов

Выбор элемента M[i, j] делает невозможным последующий выбор других элементов. Обнуление таких элементов может существенно ускорить алгоритм.

```
nullingEls(V i, U j, Isom M : Isom M1: ){
    nullingRowColumns(i, j, M:  M2);
    nullingForNeighbors(i, j, M2: M3)
    nullingUnreachableEls(i, j, M3: M1: #2)
};
```

После выбора элемента M[i, j] становятся недостижимыми остальные элементы i-ой строки и j-го столбца. Их обнуление реализуется программой nullingRowColumns.

```
nullingRowColumns(V i, U j, Isom M : Isom M1){
    M[i, ] = for (U) 0;
    M[ , j] = for (V) 0;
    M[i, j] = 1;
};
```

Соседи i-ой вершины в графе G могут быть отображены только на соседей j-ой вершины в графе H. Следовательно, для соседей i-ой вершины становятся недоступными другие вершины, не являющиеся соседями j-ой вершины. Обнуление соответствующих элементов проводит программа nullingForNeighbors.

```
nullingForNeighbors (V i, U j, Isom M : Isom M1){
    nullingFN(i, X[i], (U \  j) \ listtoset(Y[ j]), M : M1)
};
```

Программа nullingFN для каждого элемента списка X[i] и каждого элемента множества z обнуляет элемент матрицы M.

```
nullingFN(iter V i,  list(V) x, set U z, Isom M : Isom M1){
    if (x=nil) M1 = M
    else {     if (x.car > i) { for (z k) M[x.car, k] = 0 };
               nullingFN(i, x.cdr, z, M : M1)
         }
};
```

Условие x.car > i означает, что вершина i находится не среди уже пройденных рядов матрицы M, где не следует обнулять элементы матрицы.

Поскольку M[i, j]=1, для вершин i и j их соседи X[i] и Y[j] должны удовлетворять следующему условию: для всякой вершины k из X[i] существует вершина l из Y[j] такая, что M[k, l]=1. Если условие нарушается, то можно обнулить элемент M[i, j]. В алгоритме Ульмана [40] данное действие реализуется для всех элементов матрицы многократно пока сохраняется возможность обнуления элементов. В работе [41] предлагается ограничить применение данной операции только для пар соседей вершин i и j текущего элемента M[i, j].

В нашем алгоритме эти действия реализуются сначала для пар соседей, а затем для самого элемента M[i, j]. Второй выход гиперфункции реализуется в случае, если для исходной пары вершин i и j нет правильного отображения для их соседей.

```
nullingUnreachableEls(V i, U j, Isom M : Isom M1: ){
    checkNeighborPairs(X[i], Y[j], M: M1);
    if (badPair(i, j, M) #2 else #1
};
```

Программа checkNeighborPairs для всевозможных пар из списков X[i] и Y[j] запускает проверку соответствия их соседей, обнуляя соответствующий элемент матрицы если соответствия нет.

```
checkNeighborPairs(list(V) x, list(U) y, Isom M : Isom M1){
    if (x=nil) M' = M
    else { checkNeighbors(x.car, y, M : M2);
           checkNeighborPairs(x.cdr, y, M2: M1)
         }
};
```

В программе checkNeighbors текущая вершина k из списка x сопоставляется со всеми вершинами списка y. Для каждой пары k и l, для которой M[k, l] = 1, запускается программа badPair(k, l, M) проверки соответствия их соседей. Если соответствия нет, элемент M[k, l] обнуляется.

```
checkNeighbors(V k, list(U) y, Isom M : Isom M1){
   if (y=nil) M1 = M
   else { if (M[k, y.car]=1 & badPair(k, y.car, M)) {M[k, y.car]=0};
          checkNeighbors (k, y.cdr,  M: M1)
        }
};
```

Программа badPair проверяет на соответствие соседей i-ой вершины в графе G с соседями j-ой вершины в графе H. Результат **true**, если соответствия нет.

```
badPair(V i, U j, Isom M : bool){
   badNeighborPairs(X[i], Y[j], nil, M)
};
```

Программа badNeighborPairs проверяет список вершин x из графа G на соответствие со списком вершин y из графа H. При этом каждой вершине из списка x должна быть сопоставлена некоторая вершина из списка y, причем соответствующий элемент матрицы M должен быть единичным. Вершина из списка y не может быть сопоставлена повторно. С этой целью заводится дополнительный список z.

```
badNeighborPairs(list(V) x, list(U) y, z, Isom M : bool){
   if (x=nil) false
   else { badNeighbors(x.car, y, z, M) {
            case list(U) z1 : badNeighborPairs(x.cdr, y, z1, M)
            case  :  true
        }}
};
```

Для каждой вершины k из списка x запукается гиперфункция badNeighbors. Вторая ветвь гиперфункции реализуется, если для вершины k не удается найти соответствия. В первой ветви гиперфункции сопоставленная вершина добавляется в список z. Если для пары k и l=y.car выполняется условие M[k, l] = 1, то проверяется отсутствие вершины l в списке z. Просмотр списка y продолжается до обнаружения соответствия.

```
badNeighbors(V k, list(U) y, z, Isom M : list(U) z' : ){
   if (y=nil)  #2
   else if (M[k, y.car]=0 or inList(y.car, z))
                    badNeighbors(k, y.cdr, z, M : z' #1 : #2)
   else  z' = z + y.car #1
};
```

8 Алгоритм Косараджу нахождения сильно связных компонент

8.1 Сильно связные компоненты

Ориентированный граф (кратко *орграф*) – граф, ребра которого имеют направление. Орграф G = (V, E) состоит из множества вершин V и множества направленных ребер E. Направленное ребро (u, v) – упорядоченная пара вершин u, v∈ V; u – *начальная вершина* ребра, а v – *конечная вершина*.

Путь в графе определяется последовательностью вершин, в которой каждая вершина соединена ребром со следующей вершиной. Формально, путь p между вершинами a и b определяется следующим образом:

```
formula path(nodeset V, edgeset E) (V a, list(V) p, V b) =
    a ∈ V & b ∈ V  & ( a = b & p = nil ∨
                          ∃ V c, list(V) q. (a, c) ∈ E & path(c, q, b) & p = cons(a, q) ) ;
```

Внешние параметры V и E в дальнейшем будем опускать. Они будут определены глобальными переменными.

Вершина b *достижима* из вершины a тогда и только тогда, когда существует путь между a и b.

```
formula reachable(V a, b) = ∃ list(V) p. path(a, p, b) ;
```

Орграф является *сильно связным*, если между каждой парой вершин графа существует путь в любом направлении. *Сильно связная компонента* (SCC – *strongly connected component*) орграфа G – это подграф, который является сильно связным и максимальным: никакие дополнительные ребра или вершины из G не могут быть включены в подграф без нарушения его свойства быть сильно связным.

```
formula in_same_scc(V a, b) = reachable(a, b) & reachable(b, a) ;
formula is_subscc(nodeset s) =
                          s ⊆ V & ∀a, b ∈ s. in_same_scc(a, b) ;
formula is_scc(nodeset s) =  s ≠ nil & is_subscc(s) &
                          ∀nodeset w. ( s ⊆ w & is_subscc(w) ⇒  s = w ) ;
```

Тривиальная сильно связная компонента включает только одну вершину, которая не принадлежит ни одному циклу графа. Орграф является *ациклическим* тогда и только тогда, когда он не имеет нетривиальных сильно связных компонент. Чтобы упростить обработку больших графов, предварительно применяется их декомпозиция. В частности, каждая сильно связная компонента заменяетсяодной вершиной. В результате получается ориентированный ациклический граф.

Нахождение сильно связных компонент часто применяется для анализа больших графов. Алгоритмы нахождения сильно связных компонент используют поиск в глубину.

8.2 Входящие и исходящие списки смежности

Список смежности вершины – это множество соседей вершины.

Для каждой вершины орграфа используются два списка смежности, отдельно для входящих и исходящих ребер вершины. Адекватной структурой для списка смежности в языке P [1] является множество. Список смежности – это атрибутивный массив типа **array**(vs, vs), где vs – множество вершин графа.

Рассмотрим программу InOutEdges для построения входящих и исходящих списков смежности для орграфа. Результатом работы программы являются два массива атрибутов.

```
type Adj(nodeset vs) = array(vs, vs);
nodeset V;
edgeset E;

InOutEdges( ): (Adj(V), Adj(V));
{ InOutEdges1 ( E, for (j in V) {}, for (j in V) {}) ) }

InOutEdges1(edgeset eds, Adj(V) inEdg, outEdg): (Adj(V), Adj(V))
{ pick(eds){ case : (inEdg, outEdg)
            case (node i, j), eds1: InOutEdges1(eds1,
                                        inEdg with (j: add(i, inEdg[j]) ),
                                        outEdg with (i: add(j, outEdg[i]) ) )
}}
```

Элементы массивов атрибутов inEdg и outEdg изначально пусты. Рекурсивная программа InOutEdges1 выбирает следующее ребро (i, j). Вершина i добавляется к входящему списку inEdg[j] для вершины j. Вершина j добавляется к исходящему списку inEdg[i] для вершины i. Напомним, что функция add(x, s) вычисляет множество $s \cup \{x\}$.

8.3 Поиск в глубину

Поиск в глубину (DFS – *depth-first search*) – это алгоритм для обхода графа. Алгоритм DFS начинает с некоторой произвольной вершины x и проходит как можно дальше по каждой ветви, прежде чем вернуться назад. Используется атрибутый массив vis. При прохождении некоторой вершины y в DFS устанавливается vis[y] = **true**. Для обхода графа используются исходящие списки смежности outEdg, получаемые вызовом программы InOutEdges. Представленная ниже программа DFS запускается для вершины x и получает в результате массив vis, в котором отмечены пройденные вершины. В процессе обхода графа программа DFS обычно реализует некоторую работу, которая здесь отсутствует.

```
nodeset V;
edgeset E;
type Vis = array(bool, V);
| Adj(V) inEdg, outEdg | = InOutEdges(V, E);

DFS(Vis vis, node x): Vis
{ if (vis[x] ) vis else DfsEdges(vis with (x: true), outEdg[x]) }
```

Программа DfsEdges запускается при vis[x] = **false** для всех ребер, исходящих из x, и рекурсивно вызывает DFS для конечных вершин ребер.

```
DfsEdges(Vis vis, nodeset outN): Vis
{ switch pick(outN){  case : vis
                        case V y, outN1: Vis vis1 = DFS(vis, y);
                                              DfsEdges(vis1, iter y)
}}
```

Исполнение каждого вызова DFS(vis, x) сканирует множество вершин, являющегося деревом с корнем в вершине x.

Программа DFS на базе исходящих списков смежности outEdg – это обычный (прямой) поиск в глубину. Поиск в глубину в обратном направлении базируется на входящих списках смежности inEdg.

8.4 Алгоритм Косараджу

Алгоритм Косараджу [11] является наиболее простым среди алгоритмов нахождения сильно связных компонент. Для орграфа G = (V, E) алгоритм Косараджу находит все нетривиальные сильно связные компоненты. Каждая компонента – это множество более чем из одной вершины. Произвольный набор компонент образует множество типа:

```
type SCCs = set nodeset;
```

Представим предикатную программу алгоритма Косараджу.

```
nodeset V;
edgeset E;
KosarajuSCC( ): SCCs
{    list(node) S = Back();
     Forward(S)
}
```

Стек S хранит вершины из множества V. Программа Back переписывает все вершины из V в стек S, применяя обратный поиск по глубине в исходном графе. Для каждого дерева, пройденного в DFS, вершина, корень дерева, записывается в стек последней, когда все остальные вершины дерева уже записаны в стек S. Программа Forward последовательно считывает вершины из стека S и для каждой необработанной вершины строит сильно связную компоненту, применяя прямой поиск в глубину. Набор нетривиальных сильно связных компонент является результатом работы программы Forward.

Прямой и обратный поиск в глубину используют входящие и исходящие списки смежности inEdg и outEdg, полученные вызовом программы InOutEdges (см. раздел 8.1):

```
| Adj(V) inEdg, outEdg | = InOutEdges(V, E);
```

Программа Back реализуется через вызов более общей программы BackDfs:

```
Back( : list (node) S )
{ BackDfs(V, nil, for (j in V) false : S, _ )  }
```

Конструкция "_" означает, что второй результат вызова BackDfs не используется в дальнейших вычислениях.

Представленная ниже программа BackDfs имеет три аргумента: набор вершин nds, которые еще не обработаны, стек S и атрибут vis для пометки уже обработанных вершин. Изначально nds = V, стек S пуст, и все вершины не обработаны. Для множества вершин nds программа BackDfs расширяет стек S и набор посещенных вершин vis. Расширенный стек и множество обработанных вершин – два результата работы программы.

```
type Vis = array(bool, V);
BackDfs(nodeset nds, list (node) S, Vis vis : list (node), Vis)
{ pick(nds){ case : S
            case node x, nodeset nds1 :
                        | list (node) S1, Vis vis1 | =
                                    if (vis[x] ) ( S, vis )
                                    else BackDfs(inEdg[x], S, vis with (x: true));
                        BackDfs(nds1, S1, vis1)
}}
```

Программа BackDfs выбирает следующую вершину x из множества nds, если nds не пусто. Для вершины x рекурсивный вызов BackDfs запускает обратный поиск в глубину на множестве inEdg[x], чтобы обойти дерево с корнем x и получить модифицированный стек S1 и атрибут vis1, которые используются во втором рекурсивном вызове BackDfs для оставшегося множества nds1.

Программа Forward вызывает более общую программу ForwardDfs с дополнительными параметрами: атрибутом vis для маркировки обработанных вершин и набором сильно связных компонент sccs. Изначально все вершины не обработаны, а sccs – пустое множество.

```
Forward(list (node) S): SCCs
{  ForwardDfs(S, for (j in V) false), {}) }
```

В программе ForwardDfs оператор switch выбирает вершину x из стека S, если S не пуст; стек S1 – это S после выборки x. Если вершина x все еще не обработана, вызывается программа ForwardNode, которая строит сильно связную компоненту scc, начиная с вершины x. Нетривиальная компонента scc добавляется к множеству sccs.

```
ForwardDfs(list (node) S, Vis vis, SCCs sccs): SCCs
{ switch S{ case nil: sccs
            case cons(node x, list (node) S1):
             SCCs sccs1; Vis vis1;
             if (vis[x]) sccs1 = sccs || vis1 =vis
             else { ForwardNode(vis, {x}, x : vis1, nodeset scc);
                    sccs1 = if (size(scc)>1) add(scc, sccs) else sccs;
             }
             ForwardDfs (S1, vis1, sccs1)
}}
```

Программа ForwardNode вызывает программу ForwardEdges для вершины x и исходящего списка смежности outEdg[x]. Вершина x помечается как обработанная.

```
ForwardNode(Vis vis, nodeset scc, node x): (Vis, nodeset)
{ ForwardEdges(vis with (x: true), outEdg[x], scc) }
```

В программе ForwardEdges параметр outN – это подмножество всех вершин, доступных из вершины x через исходящие ребра. Следующая вершина y извлекается из подмножества outN, если она существует. Необработанная вершина y добавляется к компоненту scc. Программа ForwardNode рекурсивно запускается для обхода дерева с корневой вершиной y.

```
ForwardEdges(Vis vis, nodeset outN, scc): (Vis, nodeset)
{ pick(outN){ case : (vis, scc)
              case node y, nodeset outN1:
                        | Vis vis1, nodeset scc1| =
                                  if (vis[y]) | vis, scc |
                                  else ForwardNode(vis, add(y, scc), y);
                        ForwardEdges(vis1, outN1, scc1)
}}
```

Отметим, что программа ForwardNode может быть устранена подставкой ее тела на место двух ее вызовов.

8.5 Transformations for the KosarajuSSC program

Для получения эффективной программы для программы KosarajuSCC применяется последовательность трансформаций. Первые трансформации – склеивание переменных по следующим правилам:

nds ← nds1; vis ← vis1; S ← S1; inN ← inN1; sccs ← sccs1; outN ← outN1;

Далее хвостовая рекурсия заменяется циклом в программах BackDfs, ForwardDfs и ForwardEdges. Присваивания аргументов рекурсивного вызова соответствующим формальным параметрам вставляются вместо рекурсивного вызова. Модифицированные программы представлены ниже.

```
BackDfs(nodeset nds, list (node) S, Vis vis : list (node), Vis)
{ pick(nds){ case : S
             case node x, nds :
                       | S, vis | =
                                 if (vis[x] ) | S, vis |
                                 else BackDfs(inEdg[x], S, vis with (x: true));
                       BackDfs(nds, S, vis)
}}
```

```
ForwardDfs(list (node) S, Vis vis, SCCs sccs): SCCs
{ loop { switch S{ case nil: return sccs
                   case cons(S, node x):
                        if (vis[x]) sccs = sccs || vis =vis
                        else { ForwardNode(vis, {x}, x : vis, nodeset scc);
                               sccs = if (size(scc)>1) add(scc, sccs) else sccs;
                        }
                        | S, vis, sccs | = | S, vis, sccs |
       }}
}


ForwardEdges(Vis vis, nodeset outN, scc): (Vis, nodeset)
{ loop {pick(outN){   case : return | vis, scc |
                      case node y, outN :
                                    | vis, scc | =
                                          if (vis[y]) | vis, scc |
                                          else ForwardNode(vis, add(y, scc), y);
                                    | vis, outN, scc | = | vis, outN, scc |
       }}
}
```

Далее применяются упрощающие трансформации. Устраняются присваивания вида vis = vis. Тела нерекурсивных программ подставляются на место их вызовов. Реализуется следующая цепочка подстановок:

ForwardEdges → ForwardNode; ForwardDfs → Forward; Back, Forward → KosarajuSCC;

После подставновки тел программ применяются дополнительные склеивания переменных. Параметы S, vis и x программы BackNode склеиваются с одноименными параметрами программы BackEdges. Аналогичные трансформации склеивания применяются для других пар программ. Раскрываются мультиприсваивания. Применяются другие упрощения.

Итоговая программа представлена ниже. Описания склееных переменных вынесены перед основной программой.

```
nodeset V;
edgeset E;
| Adj(V) inEdg, outEdg | = InOutEdges(V, E);
type SCCs = set nodeset;
list (node) S = nil;
type Vis = array(bool, V);
Vis vis = for (j in V) false;
SCCs sccs = {};
nodeset scc;
KosarajuSCC( ): SCCs
{ BackDfs(V);
  vis = for (j in V) false;
  loop { switch S{  case nil:  break
                    case cons(node x, S):
                              if (not vis[x]) {
                                    scc = {x};
                                    ForwardNode(x);
                                    if (size(scc)>1) sccs = add(scc, sccs)
                              }
        }}
}

BackDfs(nodeset nds)
{ pick(nds){ case : S
             case node x, nds :
                       if (not vis[x]) { vis[x] = true; BackDfs(inEdg[x]) }
                       BackDfs(nds)
}}

ForwardNode(node x)
{ vis = vis with (x: true);
  nodeset outN = outEdg[x];
  loop { pick(outN){  case : break
                      case node y, outN:
                              if (not vis[y]) { scc = add(y, scc); ForwardNode(y) }
}}
```

8.6 Замечание

Программа KosarajuSCC на языке gP, полученная в результате трансформаций, значительно короче, чем исходная программа в разд. 8.4. Вполне возможно, что язык gP проще для большинства программистов, чем исхлдный язык P. Тем не менее, исходная программа в разд. 8.4 более подходит для дедуктивной верификации.

9 Specification of some properties of graph processing predicate programs

It is necessary to specify the main properties of the basic data structures associated with graphs. The predicate is_graph(V, E) postulates that the graph defined by vertices V and edges E is correct.

formula is_graph(nodeset V, edgeset E) =
$\forall$**node** a, b. (a, b) $\in$ E $\Rightarrow$ a $\in$ V & b $\in$ V ;

Any subgraph formed by a subset of edges of the original correct graph is correct.

The predicate isAdjMatrix(V, E, M) defines a relation between a graph and its adjacency matrix M.

formula isAdjMatrix(nodeset V, edgeset E, Matrix(V) M) =
$\forall$ V a, b. M[a, b] = 1 $\Leftrightarrow$ (a, b) $\in$ E ;

The predicate path(a, p, b) defines that a path p from vertex a to vertex b is composed of edges of the graph (V, E).

formula path(nodeset V, edgeset E) (V a, list(V) p, V b) =
a $\in$ V & b $\in$ V &
(a = b & p = nil $\vee$
$\exists$ V c, list(V) q. (a, c) $\in$ E & path(c, q, b) & p = cons(a, q)) ;

The external parameters V and E are omitted in further declarations. We consider them as global constants.

Vertex b is *reachable* from vertex a iff there exists a path between a and b.

formula reachable(V a, b) = $\exists$ list(V) p. path(a, p, b) ;

Specification of strongly connected component is adapted from [12].

formula in_same_scc(V a, b) = reachable(a, b) & reachable(b, a) ;
formula is_subscc(nodeset s) =
s $\subseteq$ V & $\forall$V a, b. in_same_scc(a, b) ;
formula is_scc(nodeset s) = s $\neq$ nil & is_subscc(s) &
$\forall$nodeset w. (s $\subseteq$ w & is_subscc(w) $\Rightarrow$ s = w) ;

It appears that our KosarajuSCC program in Section 7.3, after substituting ForwardNode instead of its two calls, is identical to the Kosaraju program in [12][1]. A few differences between two programs are insignificant.

Deductive verification of graph processing predicate programs is quite feasible. This is demonstrated by the example of verification of Kosaraju program [12].

[1] http://pauillac.inria.fr/~levy/why3/graph/abs/scck/1/scc.html – Abstract Kosaraju 1978 Strongly Connected Components in Graph

10 Related work

Set type. A set is an unordered collection of elements, unlike arrays, lists, and sequences. In some programming languages, such as Pascal, a set object is implemented as a bit vector. In many languages (Java, C++, etc.) a set type is defined as a class, that allows to choose different representations for set objects in the body of the class. In deductive verification systems, set type is used in a mathematical sense and is supported by a library of theories that define the properties needed for the proof. In the Event-B [15] and TLA+ [16] verification systems, the type system is based on the mathematical notion of the set.

In predicate software engineering, objects of type **set** get their representation only in the second phase of the predicate program transformations. This ensures simplicity of the predicate program while preserving the ability of its efficient implementation.

Programming languages and libraries for graph processing. In multi-paradigm language Leda [26], graphs are defined completely in an object-oriented style. A set of vertices and a set of edges, which are fields of the Graph class, are defined as instances of class Set. In predicate language P [1], sets of vertices and edges are also defined as sets, but as values of structural type **set**. The efficiency of operations on sets is provided by transformations.

One promising approach to accelerate graph processing is based on the language of sparse linear algebra. GraphBLAS is an open standard for graph frameworks. The mathematical formalization of the GraphBLAS [10] defines a few standard operations on sparse matrices and vectors for expressing graph algorithms. The adjacency and incidence matrices are used in that operations. GraphBLAS C specification [23] defines common building blocks for efficient graph processing. Several high performance graph libraries based on GraphBLAS standard [23] have been developed. They are SuiteSparse [34], GBTL [35], GraphBLAST [21], and others.

GraphBLAS standard [23] is implemented in an object-oriented style, hiding the detailed representation of graph structures from the user. Graph processing programs based on libraries implemented according to GraphBLAS standard are efficient, but not simple. Deductive verification of such programs would be problematic. Predicate graph processing programs are significantly simpler, allowing for deductive verification.

Other approach to accelerate graph processing is vertex-centric large-scale based on synchronous computations in functional domain-specific language Fregel [30].

Improvements to the original FW-BW-Trim parallel algorithm for detection of strongly connected components are proposed [20].

Deductive verification. This year was the 48th International Workshop on Graph-Theoretic Concepts in Computer Science (WG2022). The proof of correctness of hundreds of graph algorithms proposed at these 48 workshops was done mathematically, i.e. by paper-and-pencil proof. Today, it would be impossible to prove the correctness of the vast majority of these algorithms through deductive verification. Nevertheless, deductive verification of an extensive number of algorithms on graphs has been successfully performed; see, for example, review in [24].

Deductive verification of Kosaraju program in the verification tool Why3 [13] was successfully performed in [12], with the vast majority of the correctness formulas being proved automatically using SMT solvers and only a small part being proved interactively, including via Coq [14]. This excellent

result of non-trivial verification task was achieved also due to several **assert** statements inserted inside the Kosaraju WhyML program.

There are two works on deductive verification of Kosaraju's algorithm [17, 18] in Coq proof assistant [14].

There are two complete libraries to support proofs on graphs: PVS [37] graph theory library [27] and graph library [28] for Isabelle/HOL [36]. Numerous collections of graph theories exist for other automatic proof systems. They are not yet integrated with each other to form a standard library for a particular proof system. There are four different equivalent definitions of strongly connected components in PVS graph theory [27].

Formal proofs of Tarjan's algorithm [39] for constructing strongly connected components have been performed in Why3, Coq and Isabelle/HOL proof assistants [19]. These proof assistants are mature, and each one has its own advantages w.r.t. readability, expressiveness, stability, ease of use, automation, length of proof, etc.

There are works on deductive verification of graph processing programs in C, for example [24, 25], usually applying separation logic [40]. Verification of programs with pointers is extremely difficult. It has been shown that pointers cause significant complication of program logic [41]. In our work on the verification of programs from the OS Linux kernel, the transformations of programs from C to the intermediate functional language cP without pointers are performed before verification [33,41-44][2]. Intermediate languages gP and cP will be integrated in the future.

Verification of graph processing concurrent programs [29, 39] is based on the extension of separation logic.

11 Conclusion

A new approach to develop the effective and correct graph processing algorithms in the framework of predicate software engineering is introduced. Vertices and edges of a graph are represented as values of **set** type in predicate language P [1]. To obtain efficiency, a graph processing predicate program is transformed to the intermediate language gP, in which objects of **set** type are not modified. Optimizing transformations for operations on objects of **set** type in the gP program is not considered in our paper.

A predicate graph processing program is simpler than the similar program in other programming languages, and therefore is more suitable for conducting deductive verification. A predicate program can be close in complexity to a similar program in a specification language, say WhyML [13], for some proof assistans. The verification of our Kosaraju program (Section 7.3) in the Why3 system would be approximately the same in complexity as the verification of Kosaraju's algorithm in [12].

Future work. A GraphBLAS library in predicate language P [1] will be developed on the base of mathematical formalization of the GraphBLAS [10]. It will be a simple adaptation of [10] for objects of **set** type, not in object-oriented style and not as DSL.

[2] OS Linux kernel programs verified: https://elixir.bootlin.com/linux/v5.1.4/source/lib/sort.c, https://elixir.bootlin.com/linux/v5.9/source/lib/llist.c#L79, https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c, https://elixir.bootlin.com/linux/v5.9/source/drivers/base/bus.c#L952

For programs in the intermediate language gP, several kinds of efficient optimizing transformations for operations on set and array objects will be developed.

Development and verification of concurrent distributed graph algorithms are challenging. Graph processing algorithms in P can be translated into automata-based programs [6] where special automata optimizing transformations can be applied [22]. Automata-based programs are easily converted to specifications in the language Event-B [31]. Note that in Event-B graph objects are naturally defined as relations. An automata-based distributed graph processing program can be successfully verified on the Rodin toolkit [32]. This verification will be significantly easier and will not require the use of separation logic, since EventB defines an adequate framework for representing distributed algorithms.

Research on transformations of sequential programs into distributed ones is also planned.

References

[1] N. Karnaukhov, D. Perchine, V. Shelekhov, The predicate programming language P. Novosibirsk, ISI SB RAN, 2018. 45p. (in Russian). URL: https://persons.iis.nsk.su/files/persons/pages/plang14.pdf

[2] I. Kablukhov, V. Shelekhov, "Implementation of optimizing transformations in the predicate programming system," Sistemnaya informatika, no. 11. Novosibirsk, 2017, pp. 21-48. (In Russian) URL: https://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf

[3] J. Chamberlin, M. Zalewksi, S. McMillan, A. Lumsdaine, "PyGB: GraphBLAS DSL in Python with dynamic compilation into efficient C++," in Graph Algorithms Building Blocks (GABB) Workshop at IEEE Intl. Parallel and Distributed Processing Symposium, 2018, pp. 310-319.

[4] T. Mattsonz, T.A. Davis, M. Kumar, A. Buluc, S.McMillanx, J. Moreira, C. Yang, "LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS," in 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 276-284.

[5] S. Owre, J. M. Rushby, N. Shankar, **"**PVS: specification and verification system," in CADE-11: Automated Deduction. LNCS, v. 607. 1992, pp. 748-752.

[6] V.I. Shelekhov, "Automata-based software engineering with Event-B," Software engineering, vol. 13, no. 4, 2022, pp. 155-167. (In Russian).

[7] R. Tarjan, "Depth first search and linear graph algorithms," SIAM Journal on Computing, 1972.

[8] V. Shelekhov, " Development and verification of heapsort algorithms in predicate programming paradigm," *Preprint 164*, Institute of Informatics Systems. Novosibirsk, 2012. (In Russian).

[9] P. O'Hearn, J. Reynolds, H. Yang, "Local reasoning about programs that alter data structures," CSL 2001. LNCS 2142, pp. 1–19.

[10] J. Kepner, etc. "Mathematical foundations of the GraphBLAS," in 2016 IEEE High Performance Extreme Computing Conference (HPEC). 2016, pp. 1–9.

[11] M. Sharir, "A strong-connectivity algorithm and its applications to data flow analysis," Computers and Mathematics with Applications 7(1), 1981, pp. 67–72.

[12] R. Chen, J. Levy, Formal proofs of two algorithms for strongly connected components in graphs. 2016, 19p. URL: https://hal.inria.fr/hal-01422216

[13] Why 3. Where Programs Meet Provers. URL: http://why3.lri.fr

[14] The Coq Proof Assistant. URL: http://coq.inria.fr

[15] J.-R. Abrial, Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010. 586p.

[16] L. Lamport, Specifying systems: The TLA+ language and tools for hardware and software engineers. Addison-Wesley, 2021. 382p.

[17] F. Pottier, "Depth-First Search and Strong Connectivity in Coq," in Journées Françaises des Langages Applicatifs (JFLA), 2015.

[18] L. Théry, "Formally-proven Kosaraju's algorithm," Inria report, Hal-01095533, 2015

[19] R. Chen, C. Cohen, J.-J. Lévy, S. Merz, L. Théry, "Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabell,". Schloss Dagstuhl -Leibniz-Zentrum für Informatik, vol.141, pp.1-13, 2019.

[20] S. Hong, N. C. Rodia, K. Olukotun, “On fast parallel detection of strongly connected components (SCC) in small-world graphs,” in SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1-11.

[21] C. Yang, A. Buluç, J.D. Owens, “GraphBLAST: a highPerformance linear algebra-based graph framework on the GPU,” in CoRR, 2019. Vol. abs/1908.01407. 1908.01407.

[22] V. Shelekhov, “Automata-based program optimization by applying requirement transformations,” Software engineering, 2015, no. 11, pp. 3-13. (in Russian).

[23] A. Buluc, T. Mattson, S. McMillan, J. Moreira, C. Yang, “Design of the GraphBLAS API for C,” in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 643-652.

[24] S. Wang, Q. Cao, A. Mohan, A. Hobor, “Certifying graph-manipulating C programs via localizations within data structures,” in Proceedings of the ACM on Programming Languages 3 (OOPSLA), pp. 1-30.

[25] A. Mohan, W.X. Leow, A. Hobor, “Functional correctness of C implementations of Dijkstra’s, Kruskal’s, and Prim’s algorithms,” Computer Aided Verification (CAV 2021). LNCS 12760, 2021, pp. 801–826.

[26] T. Butt, Multiparadigm programming in Leda. Reading, Massachusetts: Addison Wesley, 1995. 394 p.

[27] R.W. Butler, J.A. Sjogren, “A PVS graph theory library,” Tech. rep., NASA Langley. 1998.

[28] L. Noschinski, “A graph library for Isabelle”, Math.Comput.Sci. 9. 2015, pp. 23–39.

[29] S. Krishna, A.J. Summers, T. Wies, “Local reasoning for global graph properties,” Programming Languages and Systems. ESOP 2020. LNCS 12075. 2020, pp. 308-335.

[30] H. Iwasaki, K. Emoto, A. Morihata, K. Matsuzaki, Z. Hu, “Fregel: A functional domain-specific language for vertex-centric large-scale graph processing,” Journal of Functional Programming, 32 (2), 2022.

[31] J.-R. Abrial. Modeling in Event-B: System and software engineering. Cambridge University Press, 2010. 586p.

[32] Rodin User's Handbook. Version 2.8. Jastram M. (editor). 2014. 184p.

[33] V. Shelekhov, “Applying Program Transformations for Deductive Verification of the List Reverse Program,” Software Engineering, vol. 12, no. 3, 2021, pp. 127-139. (In Russian). https://persons.iis.nsk.su/files/persons/pages/listspi.pdf

[34] T. A. Davis, “Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra,” ACM Trans. Math. Softw. 2018, 24p.

[35] GraphBLAS Template Library (GBTL), https://github.com/cmu-sei/ gbtl.

[36] T. Nipkow, L.C. Paulson, M.Wenzel, “Isabelle/HOL — A proof assistant for higher-order logic,” Springer, 2002, 211p.

[37] J. Chen, H. Sung, N.R. Tallent, K.J. Barker, X. Shen, A Li. “Bit-GraphBLAS: Bit-Level Optimizations of Matrix-Centric Graph Processing on GPU.” 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 515-525.

[38] C. Bron, J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph", Communications of the ACM,. 16 (9), 1973. pp. 575–577. doi:10.1145/362342.362367

[39] E. Tomita, A. Tanaka, H. Takahashi. "The worst-case time complexity for generating all maximal cliques and computational experiments", Theoretical Computer Science, 363 (1), 2006. pp. 28–42, doi:10.1016/j.tcs.2006.06.015

[40] Ullmann, J R. An Algorithm for Subgraph Isomorphism. J. Assoc. for Computing Machinery, 23, 1976. pp. 31-42.

[41] Čibej U., Mihelič J.. Improvements to Ullmann’s algorithm for the subgraph isomorphism problem. International Journal of Pattern Recognition and Artificial Intelligence, 29(07):1550025, 2015.

References

1. Karnaukhov N., Perchine D., Shelekhov V. The predicate programming language P. Novosibirsk, ISI SB RAN, 2018. 45p. (in Russian). URL: https://persons.iis.nsk.su/files/persons/pages/plang14.pdf

2. Kablukhov I., Shelekhov V. Implementation of optimizing transformations in the predicate programming system, Sistemnaya informatika, no. 11. Novosibirsk, 2017, pp. 21-48. (In Russian) URL: https://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf
3. Ekanadham K., Horn W. P., Kumar M., Jann J., Moreira J., Pattnaik P., Serrano M., Tanase G., Yu H. Graph Programming Interface (GPI): A linear algebra programming model for large scale graph computations. ACM Intl. Conference on Computing Frontiers, ser. CF '16. 2016, pp. 72–81.
4. Mattsonz T., Davis T.A., Kumar M., Buluc¸A., McMillanx S., Moreira J., Yang C. LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS. 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 276-284.
5. Shelekhov V.I. Program classification in software engineering. Software engineering, vol. 7, no. 12, 2016, pp. 531–538. (In Russian). URL: https://persons.iis.nsk.su/files/persons/pages/prog.pdf
6. Shelekhov V.I. Automata-based Software Engineering with Event-B. Software engineering, vol. 13, no. 4, 2022, pp. 155-167. (In Russian).
7. Liskov B., Zilles S. Programming with abstract data types. Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages. SIGPLAN Notices. 9. 1974. P. 50–59.
8. Shelekhov, V. Development and verification of heapsort algorithms in predicate programming paradigm. *Preprint 164*, Institute of Informatics Systems. Novosibirsk, 2012. (In Russian).
9. J. Kepner and J. Gilbert. Graph Algorithms in the Language of Linear Algebra. SIAM, Philadelphia, PA. 2011. 348p.
10. Kepner J., etc. Mathematical foundations of the GraphBLAS. 2016 IEEE High Performance Extreme Computing Conference (HPEC). 2016, pp. 1–9.
11. Sharir M. A strong-connectivity algorithm and its applications to data flow analysis. Computers and Mathematics with Applications 7(1), pp. 67–72, 1981.
12. Chen R., Levy. J. Formal proofs of two algorithms for strongly connected components in graphs. 2016, 19p. URL: https://hal.inria.fr/hal-01422216
13. Why 3. Where Programs Meet Provers. http://why3.lri.fr
14. The Coq Proof Assistant. http://coq.inria.fr
15. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010. 586p.
16. Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2021. 382p.
17. Pottier F. Depth-First Search and Strong Connectivity in Coq. In Journées Françaises des Langages Applicatifs (JFLA), 2015.
18. Théry L. Formally-proven Kosaraju's algorithm. Inria report, Hal-01095533, 2015
19. Chen R., Cohen C., Lévy J.-J., Merz S., Théry L. Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle. Schloss Dagstuhl -Leibniz-Zentrum für Informatik, vol.141, pp.1-13, 2019.
20. Hong S., Rodia N. C., Olukotun K. On fast parallel detection of strongly connected components (SCC) in small-world graphs. SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1-11.
21. Yang C., Buluç A., Owens J.D. GraphBLAST: A HighPerformance Linear Algebra-based Graph Framework on the GPU // CoRR. — 2019. — Vol. abs/1908.01407. — 1908.01407.
22. Shelekhov V. Automata-based program optimization by applying requirement transformations. Software engineering, 2015, no. 11, pp. 3-13. (in Russian).
23. Buluc A., Mattson T., McMillan S., Moreira J., Yang C. Design of the GraphBLAS API for C, 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 643-652.
24. Wang S., Cao Q., Mohan A., Hobor A. Certifying graph-manipulating C programs via localizations within data structures. Proceedings of the ACM on Programming Languages 3 (OOPSLA), pp. 1-30.
25. Mohan A., Leow W.X., Hobor A. Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms. Computer Aided Verification (CAV 2021). LNCS 12760, 2021, pp. 801–826.

26. Butt T. Multiparadigm programming in Leda. Reading, Massachusetts: Addison Wesley, 1995. 394 p.
27. Butler R.W., Sjogren J.A. A PVS graph theory library. Tech. rep., NASA Langley. 1998.
28. Noschinski L. A Graph Library for Isabelle. Math.Comput.Sci. 9. 2015, pp. 23–39.
29. Krishna S., Summers A.J., Wies T. Local Reasoning for Global Graph Properties. Programming Languages and Systems. ESOP 2020. LNCS 12075. 2020, pp. 308-335.
30. Iwasaki H., Emoto K., Morihata A., Matsuzaki K., Hu Z. Fregel: A functional domain-specific language for vertex-centric large-scale graph processing. Journal of Functional Programming, 32 (2), 2022.
31. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010. 586p.
32. Rodin User's Handbook. Version 2.8. Jastram M. (editor). 2014. 184p.
33. Shelekhov V. Transformation and verification of the OS program sorting devices in a computer bus. System informatics, no. 18. Novosibirsk, 2021. pp. 1-34. https://persons.iis.nsk.su/files/persons/pages/bus_sort3.pdf
34. Davis T. A. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. ACM Trans. Math. Softw. 2018, 24p.
35. GraphBLAS Template Library (GBTL), https://github.com/cmu-sei/ gbtl.
36. Nipkow T., Paulson L.C., Wenzel M. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer, 2002, 211p.
37. Owre S., Rushby J. M., Shankar N. PVS: Specification and Verification System. CADE-11: Automated Deduction. LNCS, v. 607. 1992, pp. 748-752.
38. Raad A., Hobor A., Villard J., Gardner P.. Verifying Concurrent Graph Algorithms. APLAS 2016, LNCS 10017, 2016, pp. 314–334.
39. Tarjan R.. Depth first search and linear graph algorithms. SIAM Journal on Computing, 1972.
40. O'Hearn, P, Reynolds J., Yang H. Local reasoning about programs that alter data structures. CSL 2001. LNCS 2142, pp. 1–19.
41. Shelekhov V. Deductive verification of a string concatenation program using a reverse transformation, ZONT-19. Novosibirsk, 2019, pp. 369-378. (In Russian). URL: http://persons.iis.nsk.su/files/persons/pages/logcflc1.pdf
42. Shelekhov V. Verification of the heapsort predicate program using inverse transformations, System informatics, no.16. Novosibirsk, 2020, pp. 75-102. (In Russian). URL: https://persons.iis.nsk.su/files/persons/pages/sort9.pdf
43. Shelekhov V. Applying Program Transformations for Deductive Verification of the List Reverse Program. Software Engineering, vol. 12, no. 3, 2021 pp. 127-139. (In Russian). https://persons.iis.nsk.su/files/persons/pages/listspi.pdf
44. Shelekhov V. Verification of a string to integer conversion program. System informatics, no. 17. Novosibirsk, 2020, pp. 43-90. (In Russian). URL: https://persons.iis.nsk.su/files/persons/pages/kstr2.pdf