International Journal of Open Information Technologies ISSN: 2307-8162 vol. 9, no. 5, 2021

Бесконечные деревья в алгоритме проверки условия эквивалентности итераций конечных языков. Часть II

Б. Ф. Мельников, А. А. Мельникова

Аннотация—**В настоящей статье мы возвращаемся к тематике, связанной с одним важным бинарным отношением на множестве формальных языков (рассматриваемом в первую очередь на множестве итераций непустых конечных языков) – отношению эквивалентности в бесконечности. Прежде всего мы рассматриваем примеры применения этого отношения (как примеры необходимости его выполнения, так и примеры использования) в различных областях теории формальных языков, дискретной математики и абстрактной алгебры. Для упрощения рассмотрения эквивалентности в бесконечности мы формулируем более простое бинарное отношение на множестве языков – отношение покрытия, двойное применение которого равносильно применению отношения эквивалентности в бесконечности. Далее мы рассматриваем алгоритм проверки выполнения отношения покрытия, после чего определяем вспомогательные объекты, используемые как для доказательства корректности этого алгоритма, так и для других задач теории формальных языков. В качестве одного из комментариев к алгоритму мы приводим соответствующую ему компьютерную программу, рассматриваем примеры её работы для конкретных входных языков, после чего формулируем определения связанных с ней объектов – в частности, определение бесконечных деревьев отношения покрытия. С их помощью мы доказываем корректность алгоритма проверки выполнения отношения покрытия, а также оцениваем сложность этого алгоритма.**

Ключевые слова—**формальные языки, итерации языков, бинарные отношения, бесконечные деревья, алгоритмы.**

В настоящей части II мы используем новую нумерацию ссылок на литературу и сносок, а остальная нумерация (разделов, рисунков и др.) продолжается.

VII. ДЕРЕВО ОТНОШЕНИЯ ПОКРЫТИЯ И НОВЫЕ КОММЕНТАРИИ К АЛГОРИТМУ

В этом разделе мы рассматриваем «развитие» дерева морфизма – другое специальное бесконечное дерево; оно, как мы уже сказали, не является рассмотренным ранее деревом морфизма – а является более сложным объектом [1]. Мы будем называть его *деревом отношения покрытия* (либо *бесконечным итерационным деревом*) – и оба названия отражают смысл такой конструкции [2]. Однако *строгих определений приводить не будем*:

- во-первых, основным предметом статьи мы считаем сам алгоритм, а деревья нужны, в первую очередь, для комментариев к нему;
- и во-вторых, на основе приведённых «нестрогих» определений несложно сформулировать необходимые «строгие».

Однако в следующих публикациях мы действительно предполагаем рассмотреть подробные определения итерационных деревьев и соответствующие возникающие задачи.

В разделе мы также приводим новые комментарии к основному алгоритму, рассматриваемом в статье. Комментарии начнём с уже рассмотренного нами в части I дерева морфизма; для удобства изложения мы повторяем рис. 4 части I – это рис. 13 ниже. При этом, как мы многократно отмечали в части I, нам необходимо рассматривать ситуации, когда у разных слов над алфавитом Δ совпадают их значения функции p. Таким образом, мы можем считать, что все вершины принадлежат нескольким классам эквивалентности (согласно значениям функции p) – причём таких классов конечное число [3].

Мы также уже отмечали в части I [1], что дерево морфизма в чём-то аналогично дереву конечного автомата, рассматривавшемуся в [2, стр. 177]. Однако есть и отличия – и основное заключается в том, что нам при постановке задачи автомат не задан [4], поэтому о полной аналогии говорить бессмысленно. Кроме того, в [2] сразу рассматривается конечное дерево.

Определяемое нами *бесконечное* дерево будет, как следует из предыдущего изложения, обладать следующими свойствами:

- каждая вершина дерева имеет сразу две пометки: во-первых, некоторое слово над алфавитом Δ, и, во-вторых, некоторое множество слов над алфавитом Σ (оно является значением функции p для первой пометки);
- каждая вершина принадлежит некоторому классу; при этом вторые пометки вершин, принадлежащих одному классу, обязательно совпадают, а вторые пометки вершин, принадлежащих разным классам, обязательно различны.

Статья получена 11 февраля 2021 г.

Борис Феликсович Мельников, Университет МГУ – ППИ в Шэньчжэне (bf-melnikov@yandex.ru).

Александра Александровна Мельникова, Димитровградский инженерно-технологический институт – филиал Национального исследовательского ядерного университета «МИФИ» (super-avahi@yandex.ru).

[1] По-видимому, не будет ошибкой сказать «классом-наследником».

[2] И понятно, что первое название «привязано» к конкретной задаче, рассматриваемой нами сейчас, – в то время как второе является его обобщением.

[3] Это очевидно. А при необходимости «строго доказать» этот факт можно в качестве вспомогательного применить приведённое ниже утверждение 2.

[4] Мы собираемся *построить* его (соответствующий *недерминированный* конечный автомат) в одной из последующих публикаций. «Входными параметрами» строимого автомата будут являться два языка – обозначаемые в этой статье A и B.

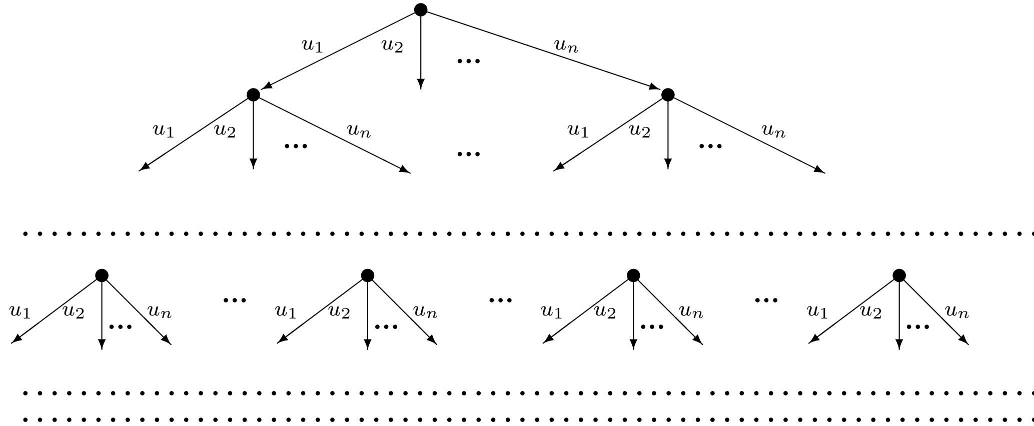

Рис. 13. Абстрактное бесконечное дерево морфизма для алфавита Σ.

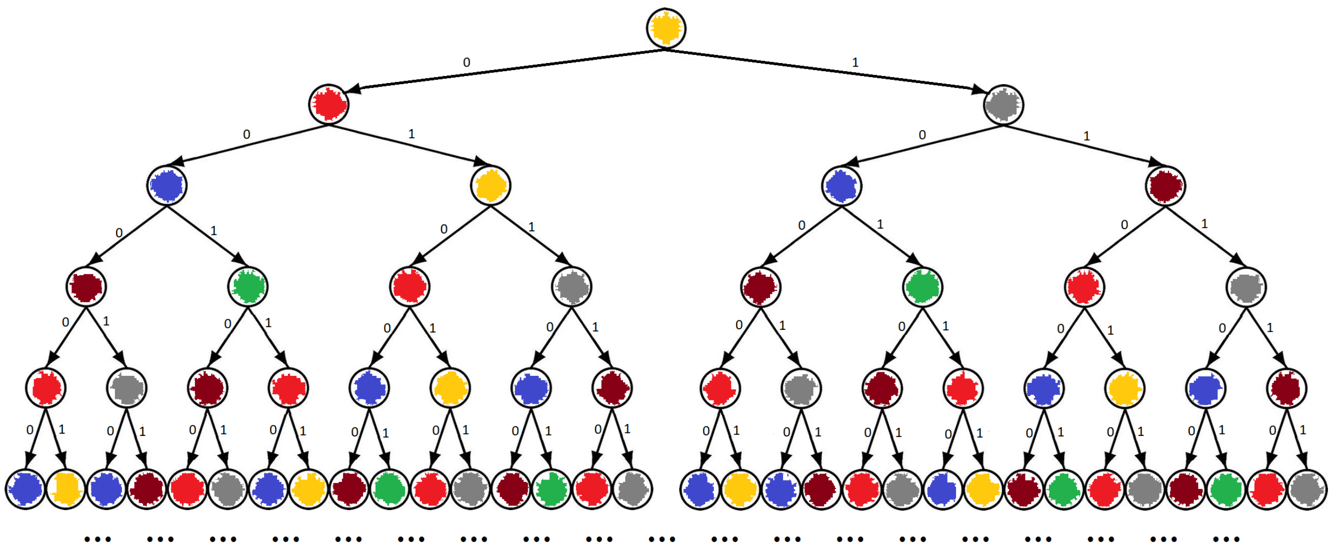

Рис. 14. Пример бесконечного итерационного дерева морфизма для алфавита $\Delta = \{0, 1\}$ (показаны уровни от 0-го до 5-го).

```
bool operator<=(Language& lA, Language& lB);
bool operator==(Language& lA, Language& lB);
```

Рис. 15. Заголовки дополнительных операторов для класса `Language` (их не было в части I).

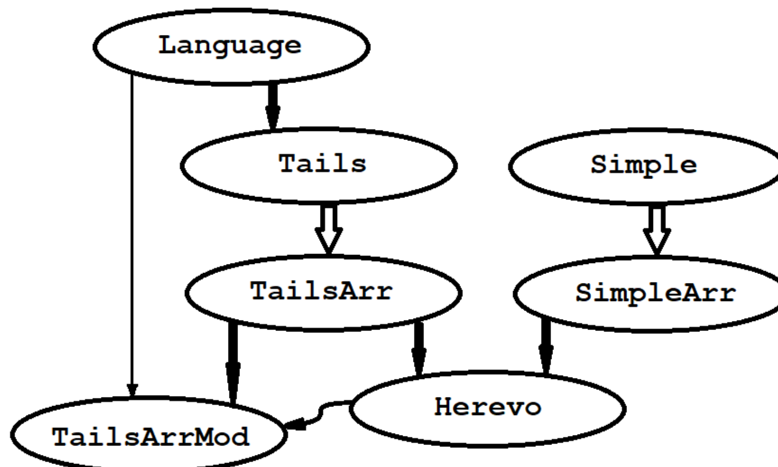

Рис. 16. Используемые классы в компьютерной программе на Си++ (подробные комментарии – в тексте статьи).

Также на основе приведённого алгоритма можно сформулировать следующее свойство дерева:

- поддеревья, соответствующие вершинам, принадлежащим одному и тому же классу, равны.

И дополнительно (что также можно вывести из описания алгоритма [5]):

- поддеревья, соответствующие вершинам, принадлежащим двум разным классам, не равны.

Возможный пример бесконечного итерационного дерева приведён на рис. 14; на нём: показаны уровни от 0-го до 5-го, алфавит $\Delta = \{0, 1\}$ (алфавит Σ и морфизм h_A не конкретнизированы), разные классы эквивалентности вершин показаны разными цветами [6].

VIII. Описание соответствующей компьютерной программы (продолжение)

Отметим ещё раз: основная цель того, что мы приводим текст программы, – в том, чтобы привести дополнительные комментарии к алгоритму, помочь его осознать; мы его несколько упростили по сравнению с [3] – но, по-видимому, даже приведённая в [1] версия сложна для быстрого восприятия. Также отметим, что все действия, описываемые в настоящей части II, могут быть несложно произведены «вручную» многократным вызовом функций, описанных в части I, – однако для языков большего размера это, конечно, очень неудобно; кроме того, мы предполагаем применить описываемые классы и в программах для будущих публикаций.

Описание программы продолжим небольшими дополнительными комментариями к тому тексту, который мы уже рассмотрели в части I – и некоторым добавкам к нему, приведённым здесь.

- Укажем, почему работа с языками выделена в специальный класс: мы много раз используем его свойства и методы в классах-наследниках.
- И вообще, применение объектно-ориентированного программирования, с нашей точки зрения, очень упрощает не только «обычные» области его применения – но и *алгоритмизацию* сложных задач (а может быть, в первую очередь именно её) [7].

А следующие три замечания связаны с модификацией классов (`Language` и др.), рассмотрение которых было начато в части I.

- В части II выдачу множества «множеств хвостов» `TailsArr` мы завершаем символом `#`.
- В связи с применяющемся в настоящей части II множественным наследованием, мы для всех классов заменили операторы вывода в поток (бывшие в части I) на функции `Print()`; по-видимому, так удобнее.
- Мы добавили операторы сравнения языков (рис. 15): первый означает включение языка `lA` в язык `lB`, а второй – равенство этих языков.

[5] Комментариев о возможности такого вывода приводить не будем. По-видимому, это *не столь очевидно* – однако для предмета статьи вряд ли представляет интерес.

[6] Также отметим, забегая вперёд, что и приведённую в конце статьи структуру, соответствующую результатам выполнения программы для конкретного примера пары языкав A и B, также можно назвать бесконечным итерационным деревом.

[7] «…летать быстро легче, чем летать медленно!» – Акка Кнебекайзе и Сельма Лагерлёф.

Теперь перейдём к принципиальным добавлениям. На рис. 16 приведена схема используемых нами классов на Си++ [8]; смысл нескольких из них уже был нами прокомментирован в части I. На этом рисунке:

- *«жирная сплошная»* стрелка означает обычное (в том числе множественное) *наследование*;
- *«прямая тонкая»* стрелка означает применение класса, от которого исходит стрелка, во втором классе в качестве *типа поля*;
- *«кривая тонкая»* стрелка означает применение класса, от которого исходит стрелка, во втором классе в качестве *типа параметра метода*;
- *«жирная пустая»* стрелка означает применение класса, от которого исходит стрелка, во втором классе в качестве *типа элемента массива* (при этом второй класс фактически предназначен именно для описания этого массива).

Немного изменённый (по сравнению с частью I) класс `TailsArr` показан на рис. 17. По-видимому, всё связанное с новыми методами этого класса понятно по названиям методов и тексту их реализации – см. также рис. 18.

На рис. 19 приведён класс `Simple` – для пары слов над алфавитом Δ. В объекты этого класса мы заносим такие пары слов, для которых уже доказали равенство значений функции P. При этом первый элемент пары обязан иметь соответствующий элемент в классе `TailsArr` (т. е. элемент с равным значением ключевого поля этого класса), а второй элемент пары, наоборот, обязан в классе `TailsArr` не иметь соответствующего элемента. Текст методов класса опущен (он очевиден).

Как и для «множества хвостов», из объектов класс `Simple` формируется массив (класс `SimpleArr`, рис. 20): он содержит уже обработанные нами слова множества Δ^*, причём, как следует из изложенного здесь, значения функции P для всех этих слов встречаются уже не первый раз. Текст методов класса также опущен.

Итак, до своего рассмотрения каждое слово над алфавитом Δ:

- либо не имело аналога ранее (т. е. ранее не было слова с таким же значением функции P) – в таком случае это слово включается в массив, описываемый классом `TailsArr`;
- либо имело такой аналог – в таком случае оно включается в массив, описываемый классом `SimpleArr`.

Простейшее множественное наследование объединяет их в один класс, см. рис. 21. Как следует из изложенного, этот класс может быть назван классом для *окончательного* варианта итерационного дерева морфизма – в то время как класс `TailsArrMod` (первая, упрощённая его версия была рассмотрена в части I) можно в такой терминологии назвать классом для *промежуточного* варианта итерационного дерева морфизма.

Приведём небольшие комментарии к описанию новой версии класса `TailsArrMod` (рис. 16 и 22). В части I для упрощения мы каждый раз передавали два постоянно рассматриваемых языка (A и B в формулах, `lA` и `lB` в программе) как параметры – поэтому в соответствующем

[8] Вряд ли её сто́ит называть «иерархией классов».

```
class TailsArr {
protected:
    int nTails;
    Tails tSets[maxTails];
public:
    TailsArr() { nTails = 0; }
    void InitSimple() { nTails = 1; tSets[0].AddWord(""); }
    bool Empty() { return nTails<=0; }
    string Exists(Tails &T);
    void AddTail(Tails &T);
    void Del(int N);
    void Print();
};
```

Рис. 17. Дополненное описание класса для множества «множеств хвостов».

```
string TailsArr::Exists(Tails &T) {
    for (int i=0; i<nTails; i++) if (tSets[i]==T) return tSets[i].GetDelta();
    return "###";
}

void TailsArr::AddTail(Tails &T) {
    if (Exists(T)!="###") return;
    tSets[nTails++] = T;
}
```

Рис. 18. Описание новых методов класса для множества «множеств хвостов».

```
class Simple {
private:
    // оба поля - слова над алфавитом Delta={0,1,...N-1}, по количеству слов в коде
    string sDelta; // рассматриваемое слово, к которому применён морфизм
    string sDeltaFirst; // ранее рассмотренное эквивалентное ему слово
public:
    void Init(string sDelta, string sDeltaFirst);
    void Print();
};
```

Рис. 19. Класс для пары слов над алфавитом Δ.

```
class SimpleArr {
private:
    int nSimple;
    Simple siSets[maxSimple];
public:
    SimpleArr() { nSimple = 0; }
    void AddSimple(string sDelta, string sDeltaFirst) {
        siSets[nSimple++].Init(sDelta,sDeltaFirst);
    }
    void Print();
};
```

Рис. 20. Класс – массив объектов предыдущего класса.

```
class Herevo: public TailsArr, public SimpleArr {
public:
    Herevo() : TailsArr(), SimpleArr() {}
    void Print() { TailsArr::Print(); cout << endl; SimpleArr::Print(); cout << endl; }
};
```

Рис. 21. Класс для *окончательного* варианта итерационного дерева морфизма.

```
class TailsArrMod : public TailsArr {
private:
    Language lA;
    Language lB;
public:
    TailsArrMod(Language& lA, Language& lB);
    void ConcatSimple(Herevo& hOtv);
    void Run(Herevo& hOtv);
};
```

Рис. 22. Класс для *промежуточного* варианта итерационного дерева морфизма.

```
TailsArrMod::TailsArrMod(Language& lA, Language& lB) : TailsArr() {
    this->lA = lA;
    this->lB = lB;
    TailsArr::InitSimple();
}

void TailsArrMod::ConcatSimple(Herevo& hOtv) {
    if (nTails<=0) return;
    Tails tExtract = tSets[0];
    for (int i=0; i<=nTails-2; i++) tSets[i] = tSets[i+1];
    int nTailsOld = --nTails;
    hOtv.AddTail(tExtract);
    if (tExtract.KolWords()<=0) return;
    for (int i=0; i<lA.KolWords(); i++) {
        tSets[nTails] = tExtract;
        tSets[nTails].ConcatLetterByInt(i);
        tSets[nTails].ConcatWord(lA[i]);
        tSets[nTails].EraseSleva(lB);
        nTails++;
    }
    for (int i=nTailsOld; i<nTails; i++) tSets[i].DeleteDaleko(lB);
    for (int i=nTails-1; i>=0; i--) {
        string s = hOtv.Exists(tSets[i]);
        if (s=="###") continue;
        hOtv.AddSimple(tSets[i].GetDelta(),s);
        Del(i);
    }
}

void TailsArrMod::Run(Herevo& hOtv) {
    for (;;) {
        ConcatSimple(hOtv);
        if (Empty()) return;
    }
}
```

Рис. 23. Методы класса, описанного на предыдущем рисунке.

```
aaa|aabba|abba|bb|
aaaa|abb|abba|bbb|
:|* 0:aaa|* 1:* 2:a||* 3:bb|* 00:aa|* 20:aaa||* 23:abb|bb||* 33:b|* 000:a|* 200:
aaa|aa|* 233:bb|b|* 0003:abb||* 2000:aa|a|* 2333:b||* 23333:bb||* #
22:2 21:1 03:1 02:1 01:2 32:1 31:1 30:1 003:1 002:1 001:1 203:3 202:2 201:2 232:
2 231:2 333: 332:1 331:1 330:1 0002:1 0001:1 0000: 2003:1 2002:1 2001:2 230:200
2332:1 2331:1 2330:1 00033:3 00032:2 00031:2 00030:200 20003:0003 20002:1 20001:
1 20000:2 23332:2 23331:1 23330:0 233333:233 233332:2 233331:1 233330:0 #
```

Рис. 24. Результаты работы программы:

- строки 1–2 — исходные языки: A и B в формулах, `lA` и `lB` в программе;
- строки 3–4 (в выдаче одна строка) — различные «множества хвостов»; формат выдачи совпадает с применявшимся в части I (при этом множество множеств заканчивается символом #);
- строки 5–8 (в выдаче одна строка) — множества пар слов над алфавитом Δ с равными в каждой паре значениями функции P.

классе не было необходимости, мы вместо него ещё раз употребляли класс `TailsArr`. В связи с этим класс содержал две группы методов, фактически предназначенных для совершенно разных целей – что, по-видимому, не совсем красиво; в новой версии мы включили эти языки в поля класса.

Текст реализации методов этого класса (рис. 23) – либо несложен, либо уже рассматривался в части I. Единственное исключение – последний цикл метода `ConcatSimple()`: он включает в ответ (параметр `hOtv`) информацию об уже встречавшемся в процессе работы «множестве хвостов» – т. е. фактически в его поле класса `SimpleArr`.

Результаты работы программы показаны на рис. 24; как и в части I, в подписи к рисунку приведены подробные комментарии. Исходные языки A и B – те же самые. В отличие от части I, построение итерационного дерева осуществляется не «вручную» (т. е. путём многократных вызовов требуемых функций в `main()`) – а программой, методом `Run()`. А «вручную» результаты можно проверить (причём вообще без компьютера) – это займёт около 15–20 минут; однако, конечно, при бóльших количествах слов в языках A и B время проверки будет существенно увеличиваться.

Те же самые результаты «в формате дерева» приведены в конце статьи на рис. 25. Как и на рис. 4, приведённом в части I:

- каждая вершина имеет две пометки: слово над алфавитом Δ (чёрным цветом) и соответствующее ему «множество хвостов» (значение функции P, синим цветом);
- вершины, в которых алгоритм (в его простейшей интерпретации) может закончить работу, показаны без пометок: каждой из них может соответствовать «синее» множество $\varnothing$; мы рисуем эти вершины только на уровнях с 1-го по 3-й, и, конечно, каждая из них является листом.

Кроме того:

- красным цветом показаны слова, эквивалентные рассматриваемым (т. е. имеющие такое же значание функции P), но уже рассмоторенные ранее.

IX. О КОРРЕКТНОСТИ АЛГОРИТМА, ВСПОМОГАТЕЛЬНЫЕ УТВЕРЖДЕНИЯ

Перед доказательством теоремы о корректности алгоритма 1 приведём несколько вспомогательных утверждений. Простые утверждения приводим без доказательств (либо только со схемами доказательств). Также заранее отметим, что все приведённые в этом разделе утверждения можно проиллюстрировать рисунками 2 и 3 части I.

Утверждение 1: При каждом выполнении шага 1 выбор требуемого $u_{\vartriangle}$ осуществим, причём любое выбираемое слово отлично от всех выбиравшихся ранее. □

Утверждение 2: Во время работы и в момент окончания алгоритма 1 переменная-функция p всегда является сюръекцией, причём при выполнении подшага 2.2 значение $p^{-1}(C)$ всегда существует. □

Последние два утверждения, справедливость которых следует из указанных шагом 1 требований к $u_{\vartriangle}$ и способа построения p, показывают, что требуемые алгоритмом действия всегда возможны.

Утверждение 3: Для любых двух конечных языков $A, B \subset \Sigma^*$ алгоритм 1 конечен.

Доказательство. Допустим, что выполнение подшага 2.3 никогда не приводит к выходу из алгоритма. Тогда для

$$M = \|B\|_{\max} - 1$$

обозначим

$$C = \bigcup_{0 \leqslant i \leqslant M} \Sigma^i .$$

По определению функции F,

$$(\forall C' \subset \Sigma^*)\,(F(C') \subseteq C),$$

поэтому после каждого выполнения подшага 2.1 имеем

$$p_i \subseteq C\,,$$

следовательно, по построению

$$H \subseteq \mathcal{P}(C)\,.$$

В связи с тем, что $\mathcal{P}(C)$ – конечное множество, в какой-то момент работы алгоритма возникнет такая ситуация:

$$(\forall C'' \subseteq C)\,(\exists C' \in H)\,(C' \subseteq C'')\,.$$

С этого момента для всех обрабатываемых шагом 2 слов $u_{\vartriangle}$ для каждой буквы $d \in \Delta$ подшаг 2.2 выполняет присваивание (добавление элемента в множество)

$$L :+ \{\, u_{\vartriangle} d \,\}\ .$$

Поэтому после конечного числа таких увеличений количества элементов множества L и выполнения шага 6 получим $L \ni e$, после чего на шаге 5 осуществим выход из алгоритма. □

Утверждение 4: $F(A \cdot B) = F\bigl(F(A) \cdot B\bigr)$. □

Утверждение 5: Если алгоритм 1 закончил работу с ответом 1, то при любом $v_{\vartriangle} \in \Delta^*$ определены значения $S(v_{\vartriangle})$ и $P(v_{\vartriangle})$, причём $P(v_{\vartriangle}) \neq \varnothing$. □

Справедливость последних двух утверждений непосредственно следует из определений функций F, S и P.

Утверждение 6: Если значение $p(w_{\vartriangle})$ определено, то

$$p(w_{\vartriangle}) = F\bigl(h_A(w_{\vartriangle})\bigr)\,. \qquad (1)$$

Доказательство проведём индукцией по $|w_{\vartriangle}|$. По определению,

$$p(e) = F\bigl(h_A(e)\bigr)\,,$$

поэтому база индукции выполняется; покажем выполнение шага.

Пусть равенство (1) выполняется для некоторого (известного) слова

$$w_{\vartriangle} \in \Delta^*\,,$$

и для некоторой (произвольной) буквы $d \in \Delta$ имеем

$$v_{\vartriangle} = w_{\vartriangle} d \quad (\text{т. е. } |v_{\vartriangle}| = |w_{\vartriangle}| + 1\,);$$

пусть, кроме того, уже определено $p(v_{\vartriangle})$. Достаточно показать, что доказываемое утверждение выполнено для

данного v_Δ – поскольку, по определению функции p, если существует $p(v_\Delta)$, то при любом

$$u_\Delta \in \mathrm{pref}(v_\Delta)$$

определено и значение $p(u_\Delta)$.

Применяя выражение из описания подшага 2.1 [9], а также предположение индукции (1) и утверждение 4, получаем следующую цепочку равенств:

$$p(v_\Delta) = F\big(p(w_\Delta)\cdot h_A(d)\big) =$$
$$F\big(F(h_A(w_\Delta))\cdot h_A(d)\big) =$$
$$F\big(h_A(w_\Delta)\cdot h_A(d)\big) = F\big(h_A(v_\Delta)\big)\,,$$

т.е. для указанного слова v_Δ условие, аналогичное (1) [10], выполняется. □

Утверждение 7: Если алгоритм 1 закончил работу с ответом 1, то для произвольного $w_\Delta \in \Delta^*$ выполнено условие

$$P(w_\Delta) \subseteq F\big(h_A(w_\Delta)\big)\,. \qquad (2)$$

Доказательство также проведём индукцией по $|w_\Delta|$. Имеем

$$P(e) = F\big(h_A(e)\big)\,,$$

т.е. для $w_\Delta = e$ условие (2) выполнено, база индукции доказана. Докажем шаг индукции.

Пусть условие (2) выполняется для некоторого (известного) слова $w_\Delta \in \Delta^*$, а для некоторой (произвольной) буквы $d \in \Delta$ имеем

$$v_\Delta = w_\Delta d \quad (\text{т.е.} \quad |v_\Delta| = |w_\Delta| + 1\ ).$$

Тогда:

$$P(v_\Delta) = p(S(w_\Delta d)) = p(s(S(w_\Delta)d)). \qquad (3)$$

Возможны следующие два случая.

- $s(S(w_\Delta)d) = S(w_\Delta)d$, тогда согласно выполняемому в подшаге 2.1 алгоритма, выполнено такое равенство:

  $$p(v_\Delta) = p(S(w_\Delta)d) = F\big(p(S(w_\Delta)\cdot h_A(d))\big)\,.$$

  Применив утверждение 4 и заменив равенство на несобственное включение, получим следующее:

  $$P(v_\Delta) \subseteq F\big(F(h_A(S(w_\Delta)))\cdot h_A(d)\big)\,. \qquad (4)$$

- $s(S(w_\Delta)d) = z_\Delta \neq S(w_\Delta)d$, тогда, учитывая вытекающие из описания подшага 2.2 алгоритма включение $C \subseteq p_i$ и равенство

  $$s(u_\Delta d_i) = p^{-1}(C)\,,$$

  получим при

  $$u_\Delta = s(w_\Delta) \quad \text{и} \quad d_i = d$$

  следующее:

  $$p(s(S(w_\Delta)d)) \subseteq F\big(p(S(w_\Delta))\cdot h_A(d)\big)\,,$$

  что, учитывая (3) и утверждение 6, совпадает с (4).

Итак, в обоих возможных случаях выполнено (4).

Снова применим утверждение 6.

$$F\big(h_A(S(w_\Delta))\big) = p(S(w_\Delta)) = P(w_\Delta)\,,$$

поэтому (4) можно переписать в таком виде:

$$P(v_\Delta) \subseteq F\big(P(w_\Delta)\cdot h_A(d)\big)\,.$$

Учитывая предположение индукции (2) и утверждение 4, получаем из последнего утверждения следующее:

$$P(v_\Delta) \subseteq F\big(F(h_A(w_\Delta)\cdot h_A(d)\big) = F\big(h_A(v_\Delta)\big)\,,$$

что для рассматриваемого слова v_Δ и требовалось доказать.

А в силу произвольности выбора w_Δ и d можно утверждать, что условие (2) выполнено для любого слова $w_\Delta \in \Delta^*$. □

X. ДОКАЗАТЕЛЬСТВО КОРРЕКТНОСТИ, ОСНОВНЫЕ РЕЗУЛЬТАТЫ

Применяемая ниже терминология, связанная с ω-словами и ω-языками, согласована с [4], [5].

Теорема 1: Следующие три утверждения равносильны:

$$A^\omega \subseteq B^\omega; \qquad (5)$$

$$A^* \overset{\subseteq}{\infty} B^*; \qquad (6)$$

алгоритм 1 заканчивает работу с ответом 1. (7)

Доказательство. **1.** Пусть выполнено условие (7). Возьмём любое $\alpha \in A^\omega$, пусть

$$\alpha = \prod_{i\in\mathbb{N}} w_i\,,$$

где все $w_i \in A$. Для каждого $n \in \mathbb{N}$ обозначим

$$u_{\Delta,n} = \prod_{1\leqslant i\leqslant n} h_A^{-1}(w_i)\,.$$

Вследствие утверждений 5 и 7 имеем:

$$(\forall n \in \mathbb{N})\,\big(F\big(h_A(u_{\Delta,n})\big) \neq \varnothing\big)\,,$$

из чего по определениям F и h_A следует:

$$(\forall n \in \mathbb{N})\,(\exists u_1 \in B^*,\ u_2 \in \mathrm{opref}(B))$$
$$\Big(\prod_{1\leqslant i\leqslant n} w_i = u_1u_2\Big)\,.$$

Поэтому существует некоторая функция

$$f_\alpha : \mathbb{N} \to B^*\,,$$

такая что

$$(\forall n \in \mathbb{N}) \quad \Big(\ \big(\prod_{1\leqslant i\leqslant n} f_\alpha(i) \in \mathrm{pref}\big(\prod_{1\leqslant i\leqslant n} w_i\big)\big)$$
$$\&\ \big(\prod_{1\leqslant i\leqslant n} w_i \in \mathrm{pref}\big(\prod_{1\leqslant i\leqslant n+1} f_\alpha(i)\big)\big)\ \Big)\,. \qquad (8)$$

Рассмотрим ω-слово

$$\beta = \prod_{i\in\mathbb{N}} f_\alpha(i)\,.$$

[9] С одновременной заменой:
- p_i на $p(v_\Delta)$;
- u_Δ на w_Δ;
- u_i на $h_A(d)$.

Возможность такой замены следует из смысла введённых в настоящем утверждении обозначений.

[10] Полученное из (1) заменой w_Δ на v_Δ.

По построению, $\beta \in B^{\omega}$. Предположим, что $\beta \neq \alpha$, тогда для некоторого $n \in \mathbb{N}$ имеем

$$\mathrm{pref}_n(\alpha) \neq \mathrm{pref}_n(\beta)\,,$$

откуда выводится противоречие с (8).

Таким образом, построенное нами ω-слово β совпадает с выбранным ранее α. Поскольку выбиралось любое $\alpha \in A^{\omega}$, заключаем, что $A^{\omega} \subseteq B^{\omega}$, таким образом, из (7) следует (5).

2. (6) выводится из (5) согласно доказанному в [5].

3. Теперь докажем, что из (6) следует (7). Предположим противное: (6) выполнено, а (7) – нет. Тогда в силу утверждения 3 алгоритм 1 заканчивает работу с ответом 0. Для зафиксированного при выходе из алгоритма элемента (пусть это – $v_{\vartriangle}$) выполнено условие $p(v_{\vartriangle}) = \varnothing$ (это следует из описания алгоритма 1. Кроме того, согласно утверждению 6,

$$F\big(h_A(v_{\vartriangle})\big) = \varnothing\,.$$

По определению функции F,

$$(\forall u \in B^*)\,\big(h_A(v_{\vartriangle}) \notin \mathrm{pref}(u)\big)\,,$$

а это противоречит условию (6). □

Понятно, что применяя дважды алгоритм 1, мы получаем алгоритм проверки выполнения отношения $\overset{\sim}{\infty}$.

Замечание. В случае бесконечных языков A и B доказательство эквивалентности условий (5) и (6) может быть построено по приведённому выше для конечного случая, связанные результаты были нами рассмотрены в [6]. При этом важно отметить, что рассмотрение бесконечных итерируемых языков нужны «не только для теории»: их применение на практике, для описания грамматических конструкций языков программирования, было нами показано, например, в [7].

XI. О СЛОЖНОСТИ АЛГОРИТМА ПРОВЕРКИ ВЫПОЛНЕНИЯ ПОКРЫТИЯ В БЕСКОНЕЧНОСТИ И СВЯЗАННЫХ ПРОБЛЕМАХ

Для рассмотренного нами алгоритма несложно получить верхнюю оценку сложности. Однако мы не будем делать это строго – ограничившись общими соображениями – причём можно указать сразу несколько (причём совершенно разных) причин того, что строгие оценки и соответствующие доказательства нас сейчас не интересуют.

- Во-первых, строгие оценки не являются предметом настоящей статьи.
- Во-вторых, по мнению авторов (неоднократно высказывавшемуся в наших предыдущих публикациях), для труднорешаемых задач (к которым относится и рассматриваемая нами) описание *эвристических* алгоритмов их решения существенно важнее построения точных оценок. Последние, по-видимому, в подавляющем большинстве подобных задач нужны лишь с теоретической точки зрения.
- При том получить такие оценки довольно сложно – как «вообще», так и в нашей конкретной задаче. По поводу неё сразу отметим следующее (некоторые подробности далее). «Сложность получения сложности» возникает из-за того, что необходимо получить оценки, зависящие *от размеров задачи* (упрощая ситуацию – от суммы длин всех слов языков A и B), а мы в алгоритме обычно знаем не это число, а общее количество слов рассматриваемых языков.
- По-видимому, можно получить и *достижимую* оценку сложности алгоритма (подробности также см. ниже) – поэтому верхняя оценка вряд ли интересна.
- Самая важная причина: для рассматриваемой нами задачи могут существовать и другие алгоритмы – и подходы к реализации одного из них мы предполагаем рассмотреть в последующих публикациях.

Однако, конечно, мы приведём краткую схему получения верхней оценки сложности рассмотренного алгоритма. Пусть исходный язык L состоит из n слов; в этом случае «минимальное» множество слов A, итерация которого находится в отношении $\overset{\sim}{\infty}$ с итерацией L, может содержать любое подмножество, состоящее из этих n слов. Поэтому верхней оценкой является 2^n (т. е. алгоритм экспоненциальный) – и такую же оценку мы получим при «входных данных, измеряемых в общей длине задачи» (т. е. суммарной длине рассматриваемых слов язык L). И *косвенным подтверждением* достижимости нужной нам оценки является рассмотренный нами пример (см. рис. 24 и комментарии к нему, а также рис. 25): на рисунке имеются 15 множеств (мы не считаем $\varnothing$, т. е. фактически на этом рисунке считаем неповторяющиеся «синие» множества) – что, кстати, совпадает с количеством непустых подмножеств множества из 4 элементов.

В качестве иллюстрации, а именно – примера для существенно более простой задачи, но при этом задачи, связанной с рассматриваемой нами, рассмотрим извлечение корня из языка; мы этим занимались, например, в [8], и приведённый здесь пример является обобщением примера, рассмотренного в той статье [11]. Конкретно, пусть

$$A = \{\, a,\, aba,\, ababa,\, abababa \,\} \cup \{\, babab,\, bababab,\, bababab \,\}\,,$$

и при этом $L = A^2$. Тогда множество потенциальных корней можно видеть на основе приведённого в следующей

[11] Аналогий между этими двумя задачами несколько – приведём такую. В процитированной статье, среди прочего, рассматриваются множества *потенциальных корней* n-й степени, обычно обозначаемые нами

$$\overline{\sqrt[n]{A}}\,;$$

именно из этого множества в обеих задачах – причём в разных их постановках – формируются языки-ответы.

Один из возможных алгоритмов для рассматриваемой нами задачи (общей задачи – т. е. проверки выполнения условия $A^* \overset{\sim}{\infty} B^*$) также состоит в построении множества потенциальных корней; причём степень n в этом случае – не заданная заранее, а произвольная. Мы собираемся вернуться к этой задаче в одной из следующих публикаций.

При этом разные постановки обеих задач состоят в том, что надо найти:

- либо одно любое решение;
- либо всё множество решений (множество множеств слов);
- либо решение, минимальное по некоторой «естественной» норме (или любое из таких решений, или все такие решения);
- и др.

Очевидно, что в обеих ситуациях имеется тривиальный (но малоинтересный с практической точки зрения) алгоритм решения – переборный (“brute force method”), откуда и возникает экспоненциальная сложность.

таблице:

Таб. 1. Потенциальные корни в примере к задаче извлечения корня 2-й степени из заданного языка.

	babab	bababab	babababab
a	—	(ab)^2	–
aba	(ab)^2	–	(ab)^3
ababa	–	(ab)^3	–
abababa	(ab)^3	–	(ab)^4

	a	aba	ababa	abababa
babab	–	(ba)^2	–	(ba)^3
bababab	(ba)^2	–	(ba)^3	–
babababab	–	(ba)^3	–	(ba)^4

В обеих частях таблицы по строкам – первый элемент конкатенации для получения A^2, по столбцам – второй элемент, а в клетках таблицы – квадратные корни из слов-конкатенаций. Несложно убедиться, что в дополнение ко всем элементам языка A это множество включает ещё и такие слова:

$$(ab)^2, \ (ab)^3, \ (ab)^4, \ (ba)^2, \ (ba)^3, \ (ba)^4.$$

Отметим также, что заменяя минимальную и максимальную степени в фактически применяемых здесь выражениях $a(ba)^k$ и $b(ab)^l$ (в нашем примере k изменяется от 0 до 3, а l – от 2 до 4), мы получаем новые примеры.

Итак, в задаче на входе – только язык L (а также требуемая степень 2). В нашем примере общая длина тех корней, которые войдут в ответ [12], равна

$$1+3+5+7+5+7+9=37\,;$$

при этом общая длина всех «плохих» корней (которые не войдут в ответ) равна

$$4+6+8+4+6+8=36\,,$$

т. е. примерно равна предыдущему значению. Несложно убедиться, что при бо́льших максимальных значениях k и l будет аналогичная ситуация, т. е. отношение общей длины «хороших» корней к общей длины «плохих» примерно равно 1 – что не свидетельствует о *практической* целесообразности применения рассмотренного нами алгоритма в задачах больших размерностей.

Из изложенного в этом разделе следует также верхняя оценка числа *классов эквивалентности* (числа неповторяющихся множеств, включаемых в объект класса `TailsArr`) – мы о таких классах говорили выше; для данных общей длиной n возможно порядка $\sqrt{n}$ различных слов (подсчёт коэффициентов опускаем) – и, следовательно, порядка $2^{n/2}$ таких классов. И, конечно же, при генерации всех классов рассмотренным нами алгоритмом зацикливания не будет [13] – поскольку все возможные «множества хвостов» заведомо образуют конечное множество.

[12] Точнее – в ответ на любую из задач извлечения корня, сформулированных в предыдущей сноске. Мы не будем доказывать (неочевидный) факт, что *в приведённом примере* все эти задачи эквивалентны: этот факт очень далёк от вопросов, рассматривающихся в настоящей статье.

[13] Это особенно важно для приведённой выше программы на Си++, поскольку, как было отмечено выше, она несколько отличается от рассматриваемого алгоритма 1.

XII. Заключение

Кратко перечислим полученные результаты и сформулируем направления дальнейшей работы, связанные с рассмотренными в настоящей статье.

Мы рассмотрели алгоритм проверки выполнения отношения эквивалентности в бесконечности двух конечных языков; этот алгоритм вряд ли необходим для практического применения – однако:

- во-первых, он показывает принципиальную возможность алгоритмизации проверки выполнения этого отношения (ранее рассмотренные нами алгоритмы – см. [5], [9], [10] и др. – были неконструктивными);
- во-вторых, на его основе мы рассмотрели бесконечные итерационные деревья, которые предполагаем применять и в других задачах теории формальных языков;
- и в-третьих, при его рассмотрении возникают различные связанные проблемы – как «вспомогательные», так и являющиеся, по-видимому, довольно важными.

Одной из подобных проблем является сведе́ние бесконечных итерационных деревьев к недетерминированным конечным автоматам: такое сведе́ние несложно получается путём отождествления всех вершин дерева, принадлежащих одному классу эквивалентности (т. е. вершин с одинаковыми значениями функции P). Понятно, что каждой паре языков при этом соответствует некоторый недетерминированный конечный автомат над рассматриваемым нами алфавитом [14]; понятно, что возникает вопрос про решению «обратной задачи»: всякому ли недетерминированному конечному автомату соответствует некоторая пара языков, определяющее, во-первых, бесконечное итерационное дерево, и, во-вторых, автомат, строящийся по этому дереву и *совпадающий с заданным*?

Список литературы

[1] Мельников Б., Мельникова А. *Бесконечные деревья в алгоритме проверки условия эквивалентности итераций конечных языков. Часть I* // International Journal of Open Information Technologies. – 2021. – Vol. 9. No. 4. – P. 1–11.

[2] Лаллеман Ж. *Полугруппы и комбинаторные приложения*. – М., Мир. – 1985. – 440 с.

[3] Мельников Б. *Алгоритм проверки равенства бесконечных итераций конечных языков* // Вестник Московского университета, серия 15 («Вычислительная математика и кибернетика»). – 1996. – № 4. – С. 49–54.

[4] Саломаа А. *Жемчужины теории формальных языков*. – М., Мир. – 1986. – 159 с.

[5] Melnikov B. *The equality condition for infinite catenations of two sets of finite words* // International Journal of Foundation of Computer Science. – 1993. – Vol. 4. No. 3. – P. 267–274.

[6] Brosalina A., Melnikov B. *Commutation in global supermonoid of free monoids* // Informatica (Lithuanian Academy of Sciences). – 2000. – Vol. 11. No. 4. – P. 353–370.

[7] Дубасова О., Мельников Б. *Об одном расширении класса контекстно-свободных языков* // Программирование (РАН). – 1995. – № 6. – С. 46–58.

[8] Melnikov B., Korabelshchikova S., Dolgov V. *On the task of extracting the root from the language* // International Journal of Open Information Technologies. – 2019. – Vol. 7. No. 3. – P. 1–6.

[9] Мельников Б. *Описание специальных подмоноидов глобального надмоноида свободного моноида* // Известия высших учебных заведений. Математика. – 2004. – № 3. – С. 46–56.

[10] Алексеева А., Мельников Б. *Итерации конечных и бесконечных языков и недетерминированные конечные автоматы* // Вектор науки Тольяттинского государственного университета. – 2011. – № 3 (17). – С. 30–33.

[14] В наших обозначениях – над Δ; заметим при этом, что сами языки A и B задаются над алфавитом Σ.

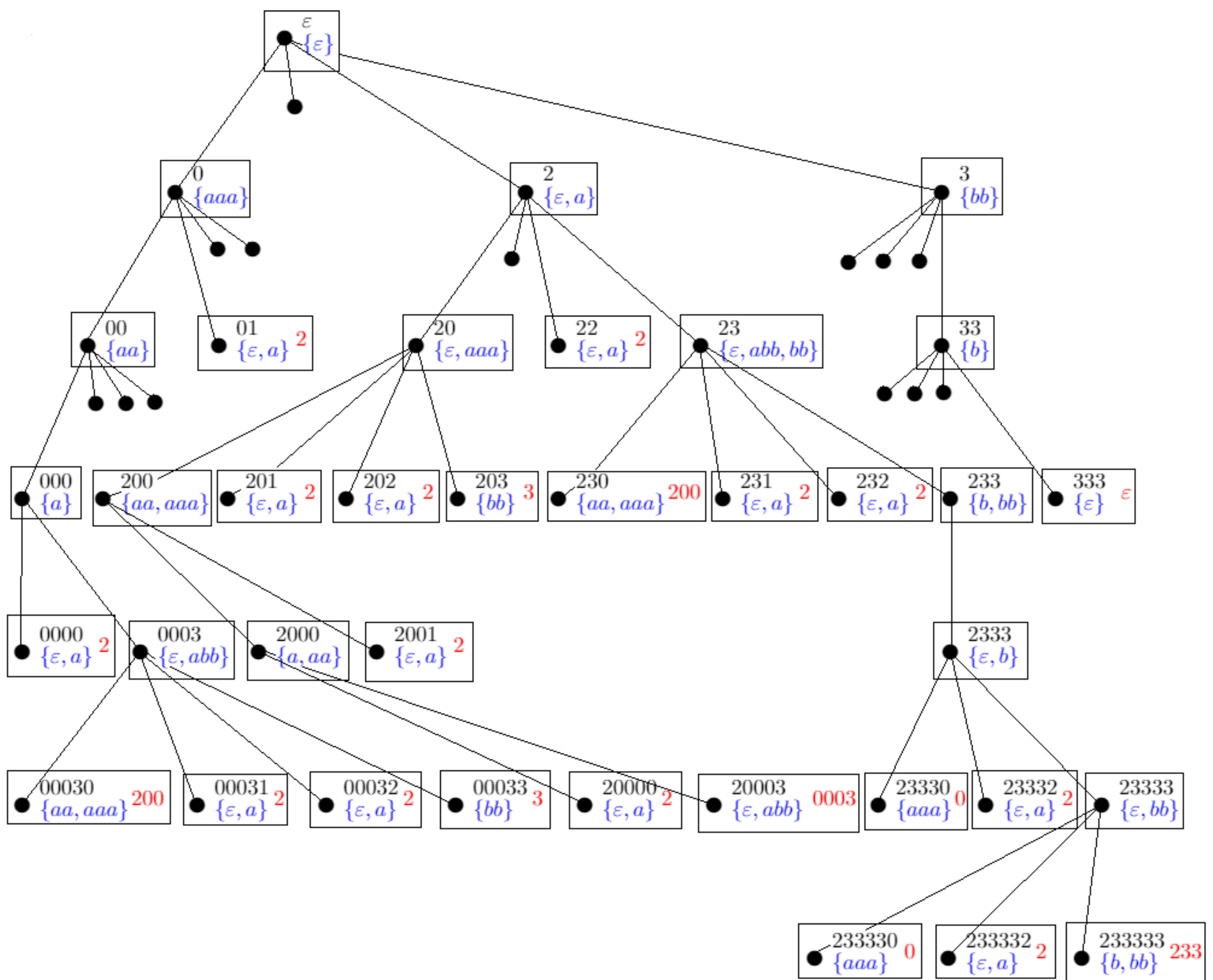

Рис. 25. Итерационное дерево морфизма для примера из статьи (без повторения вершин с одинаковыми значениями p).

Борис Феликсович МЕЛЬНИКОВ,
профессор Университета МГУ – ППИ в Шэньчжэне (http://szmsubit.ru/),
email: bf-melnikov@yandex.ru,
mathnet.ru: personid=27967,
elibrary.ru: authorid=15715,
scopus.com: authorId=55954040300,
ORCID: orcidID=0000-0002-6765-6800.

Александра Александровна МЕЛЬНИКОВА,
доцент Димитровградского инженерно-технологического института –
филиала Национального исследовательского ядерного университета «МИФИ» (https://diti-mephi.ru/),
email: super-avahi@yandex.ru,
mathnet.ru: personid=148963,
elibrary.ru: authorid=143351,
scopus.com: authorId=6603567251,
ORCID: orcidID=0000-0002-1658-6857.

Infinite trees in the algorithm for checking the equivalence condition of iterations of finite languages. Part II

Boris Melnikov, Aleksandra Melnikova

***Abstract*—In this paper, we return to the topic related to one important binary relation on the set of formal languages (considered primarily on the set of iterations of nonempty finite languages), i.e. the equivalence relation at infinity. First of all, we consider examples of the application of this relation (both examples of the need for its implementation, and examples of its use) in various fields of the theory of formal languages, discrete mathematics, and abstract algebra. To simplify the consideration of equivalence at infinity we formulate a simpler binary relation over the set of languages, i.e. the covering relation, the double application of which is equivalent to the application of the equivalence relation at infinity. Next, we consider an algorithm for verifying the fulfillment of the coverage relation, and then we define auxiliary objects used both for proving the correctness of this algorithm and for other problems in the theory of formal languages. As one of the comments on the algorithm, we give the corresponding computer program, consider examples of its operation for specific input languages, after that, we formulate the definitions of the objects associated with them, in particular, the definition of infinite trees of the coverage relation. With their help we prove the correctness of the algorithm for checking the fulfillment of the coverage relation, and also estimate the complexity of this algorithm.**

***Keywords*—formal languages, iterations of languages, binary relations, infinite trees, algorithms.**

References

[1] Melnikov B., Melnikova A. *Infinite trees in the algorithm for checking the equivalence condition of iterations of finite languages. Part I* // International Journal of Open Information Technologies. – 2021. – Vol. 9. 2021. – Vol. 9. No. 4. – P. 1–11 (in Russian).

[2] Lallement G. *Semigroups and Combinatorial Applications*. – NJ, Wiley & Sons, Inc. – 1979. – 376 p.

[3] Melnikov B. *Algorithm for checking the equality of infinite iterations of finite languages* // Bulletin of the Moscow University, Series 15 (“Computational Mathematics and Cybernetics”). – 1996. – No. 4. – P. 49–54 (in Russian).

[4] Salomaa A. *Jewels of Formal Language Theory*. – Rockville, Maryland, Computer Science Press. – 1981. – 144 p.

[5] Melnikov B. *The equality condition for infinite catenations of two sets of finite words* // International Journal of Foundation of Computer Science. – 1993. – Vol. 4. No. 3. – P. 267–274.

[6] Brosalina A., Melnikov B. *Commutation in global supermonoid of free monoids* // Informatica (Lithuanian Academy of Sciences). – 2000. – Vol. 11. No. 4. – P. 353–370.

[7] Dubasova O., Melnikov B. *On an extension of the context-free language class* // Programming (Russian Academy of Sciences). – 1995. – No. 6. – P. 46–58 (in Russian).

[8] Melnikov B., Korabelshchikova S., Dolgov V. *On the task of extracting the root from the language* // International Journal of Open Information Technologies. – 2019. – Vol. 7. No. 3. – P. 1–6.

[9] Melnikov B. *The description of special submonoids of the global supermonoid of the free monoid* // News of Higher Educational Institutions. Mathematics. – 2004. – No. 3. – P. 46–56 (in Russian).

[10] Alekseeva A., Melnikov B. *Iterations of finite and infinite languages and nondeterministic finite automata* // Vector of Science of Togliatti State University. – 2011. – No. 3 (17). – P. 30–33 (in Russian).

Boris MELNIKOV,
Professor of Shenzhen MSU – BIT University, China
(`http://szmsubit.ru/`),
email: `bf-melnikov@yandex.ru`,
mathnet.ru: `personid=27967`,
elibrary.ru: `authorid=15715`,
scopus.com: `authorId=55954040300`,
ORCID: `orcidID=0000-0002-6765-6800`.

Aleksandra MELNIKOVA,
Associated Professor of
Dimitrovgrad Engineering and Technology Institute –
Branch of National Research Nuclear University “MEPhI”
(`https://diti-mephi.ru/`),
email: `super-avahi@yandex.ru`,
mathnet.ru: `personid=148963`,
elibrary.ru: `authorid=143351`,
scopus.com: `authorId=6603567251`,
ORCID: `orcidID=0000-0002-1658-6857`.