Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В.И. Шелехов

РАЗРАБОТКА ПРОГРАММЫ ПОСТРОЕНИЯ ДЕРЕВА СУФФИКСОВ В ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Препринт
115

Новосибирск 2004

Дерево суффиксов определяет компактное представление множества подстрок некоторой строки символов. Дерево суффиксов применяется в системах быстрого поиска, сжатия и других приложениях. С использованием технологии предикатного программирования описывается процесс разработки эффективной программы построения дерева суффиксов по алгоритму Маккрейта [11]. Для этого алгоритма строится предикатная программа в виде набора рекурсивных вычислимых определений предикатов. Каждый предикат имеет спецификацию на языке исчисления предикатов второго порядка. Проведено математическое доказательство каждого определения предиката, заключающееся в выводе спецификации из правой части определения. Доказательство базируется на фрагменте математической теории деревьев суффиксов, включающей систему понятий и восемь лемм. К предикатной программе применяются эквивалентные оптимизирующие преобразования с использованием специализации. Полученная предикатная программа преобразуется в эффективную императивную программу применением следующей системы трансформаций: склеивания переменных, замены хвостовой рекурсии циклом, подстановки тела определения на место вызова, кодирования последовательностей и множеств массивами. Итоговая программа не уступает по эффективности написанной вручную на императивном языке.

© Институт систем информатики им А. П. Ершова СО РАН, 2004

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Vladimir I. Shelekhov

DEVELOPMENT OF A SUFFIX TREE CONSTRUCTION PROGRAM IN THE PREDICATE PROGRAMMING TECHNOLOGY

Preprint
115

Novosibirsk 2004

A suffix tree provides the compact representation of the set of all substrings for some string of characters. The suffix trees are widely used for quick searching, data compression, etc. In the predicate programming framework, development of an efficient program of suffix tree construction based on the McCreight's algorithm [11] is described. The predicate program for this algorithm is a set of recursive computable definitions of predicates. For each predicate, the specification in the second-order predicate calculus language is included. Mathematical proof of conclusion the right part of predicate definition from the predicate specification is provided for each predicate. The proof is based on the fragment of mathematical suffix tree theory included the notion system and eight lemmas. Equivalent optimizing source-to-source transformations are applied to the predicate program. Result predicate program is transformed to efficient imperative one. The following transformations are applied: variable merging, replacement of tail-recursion by loop, substitution predicate definition instead of a predicate call, etc.

© A. P. Ershov Institute of Informatics Systems, 2004

ВВЕДЕНИЕ

Предикатная программа есть замкнутый набор рекурсивных вычислимых определений предикатов. Определение предиката есть вычислимая логическая формула, записанная на языке предикатного программирования P (**P**redicate programming language) [4]. Имеется также спецификация предиката на языке исчисления предикатов второго порядка. Определение предиката должно удовлетворять спецификации. Это значит, что спецификация должна быть доказуема из определения предиката. Проверка правильности предикатной программы посредством математического доказательства позволяет обеспечить высокую степень надежности программы.

Технология предикатного программирования предполагает написание программы на языке P, применение к ней набора трансформаций с получением эффективной программы на императивном расширении языка P и конвертацию на любой из императивных языков: C, C++ и др. Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей.

Технология применима для класса задач, спецификация которых представима в виде математического предиката. Язык P предоставляет достаточно полный набор языковых конструкций для адекватного представления алгоритмов. Предикат может быть определен в виде гиперфункции, представляющей гибрид распознавателя и преобразователя. Оператор расщепления на базе гиперфункции определяет иные формы ветвления по сравнению с условным оператором. Использование гиперфункций и операторов расщепления обеспечивает свободу и адекватность декомпозиции программы и, как правило, более эффективную реализацию по сравнению с традиционным стилем программирования.

Технология предикатного программирования демонстрируется на задаче построения дерева суффиксов, определяющего компактное представление множества подстрок некоторой строки символов. Существует много различных применений дерева суффиксов, особенно в системах быстрого по-

иска. Одним из важнейших является использование дерева суффиксов в наиболее быстром алгоритме выравнивания геномов [5].

Непосредственный алгоритм построения дерева суффиксов имеет квадратичную оценку по времени. Первый линейный по времени алгоритм предложен Вейнером [16]. Более простым и эффективным считается алгоритм Маккрейта [11], линейность которого обеспечивается использованием суффиксных ссылок. Эти же ссылки используются в алгоритме Укконена [15], который может строить дерево суффиксов одновременно с последовательной загрузкой символов исходной строки в память. Предложен двусторонний алгоритм построения дерева суффиксов [8]. Имеются также параллельные алгоритмы. Существуют разные подходы для компактного представления в памяти дерева суффиксов [12, 9, 13, 14].

В данной работе представлен процесс разработки эффективной программы построения дерева суффиксов по алгоритму Маккрейта [11]. Дается фрагмент математической теории деревьев суффиксов, включающей систему понятий и восемь лемм. На этой базе разработан набор определений предикатов. Приводится доказательство правильности каждого определения.

Конечная программа получается как результат последовательного развития ее версий. Сначала предлагается простая неэффективная версия алгоритма. Введение аппарата суффиксных ссылок с заменой дерева суффиксов графом дает алгоритм Маккрейта. В полученной программе обнаруживается неэффективность: повторный доступ к одной и той же дуге вершины дерева суффиксов. Третья версия получается применением эквивалентных оптимизирующих преобразований ко второй версии. Необычным преобразованием является превращение предиката-функции в гиперфункцию. Далее происходит втягивание в конец второй ветви гиперфункции, затем применяется специализация, что дает возможность других улучшений. К третьей версии применяется система трансформаций с получением эффективной программы.

В разд. 1 описывается подмножество языка P, используемое в программе построения дерева суффиксов, и система трансформаций. В разд. 2 вводятся основные понятия, связанные с деревьями суффиксов, и показана разработка простого неэффективного алгоритма построения суффиксов. В разд. 3 дается понятие суффиксной ссылки, формулируются и доказываются еще пять лемм. Систематически описывается предикатная программа алгоритма Маккрейта. Приводится доказательство каждого определения предиката. В разд. 4 отмечается неэффективность полученной программы. Для устранения неэффективности проводится система эквивалентных оп-

тимизирующих преобразований с получением другого набора определений предикатов. В разд. 5 реализуются первые две группы трансформаций итоговой предикатной программы: склеивание переменных и замена хвостовой рекурсии циклом. В разд. 6 определения предикатов подставляются в места их вызовов, формальные параметры заменяются глобальными переменными и реализуется кодирование последовательностей массивами. В разд. 7 даются замечания по смежным работам в функциональном программировании. В заключении суммируются основные положения работы и сопоставляются возможности функционального и предикатного программирования.

1. ЭЛЕМЕНТЫ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Предикатная программа есть замкнутый набор рекурсивных **определений предикатов**. Определение предиката имеет следующую структуру:

<имя предиката>(<описания аргументов>: <описания результатов>) ≡
<ограничения> ⇒ <собственно спецификация>
{ <оператор> }
Δ <математическое доказательство> □

<ограничения>, <собственно спецификация> и <оператор> являются формулами исчисления предикатов второго порядка, причем <оператор> является вычислимой формулой на языке предикатного программирования P. Параметрами предиката являются переменные — аргументы и результаты. Исполнение определения предиката заключается в исполнении <оператора>, вычисляющего значения результатов по значениям аргументов. <оператор> как часть определения предиката называется т**елом предиката**. **Спецификация предиката** состоит из <ограничений> и <собственно спецификации>. Ограничения определяют условия на аргументы предиката. Ограничения могут отсутствовать. <математическое доказательство> есть произвольный математический текст с доказательством правильности спецификации предиката из логической формулы, определяемой <оператором>.

Описание исходного или результирующего параметра предиката имеет следующий вид:

<тип><пробел><имя параметра>

Операторами являются: предикат равенства, вызов предиката, блок, параллельный оператор, условный оператор и другие. Предикатом равенст-

ва является конструкция <переменная> = <выражение>. Логическая композиция <оператор1> & <оператор2> является блоком и записывается в виде {<оператор1>; <оператор2>}, если результирующие переменные <оператора1> используются в <операторе2>. Если же результаты этих операторов в них самих не используются, то композиция является параллельным оператором и записывается в виде

<оператор1> || <оператор2>.

Исполнение каждого из двух операторов реализуется параллельно.

Среди операторов блока могут находиться описания локальных переменных: <тип> <пробел> <список имен переменных>. Возможно также описание переменной с инициализацией:

<тип> <пробел> <имя переменной> = <выражение>.

Пример 1. Определение предиката sign для знака вещественного числа x:

```
sign(real x: int s) ≡ s=sign(x)
 {     if x>0 then s = 1 elsif x = 0 then s = 0 else s = -1 end }
```

Приведенная форма тела предиката sign является **предикатной** (или операторной). Возможна **функциональная** форма определения:

```
sign(real x: int s) ≡ s=sign(x)
 {     if x>0 then 1 elsif x = 0 then 0 else -1 end   }
```

Допускается и гибридная форма определения:

```
sign(real x: int s) ≡ s=sign(x)
{      if x>0 then 1 elsif x = 0 then s = 0  else -1 end    }
```

Вызов предиката sign(r: a) эквивалентен предикату равенства a = sign(r), где sign(r) является вызовом функции.

Система типов языка P включает примитивные типы (**nat**, **int** **real**, **bool**, **char**), подмножество типа (в частности, диапазон целых чисел), предикатный тип и структурные типы (массив, кортеж, объединение, последовательность, множество).

Ниже даны примеры описаний переменных разных типов. Текст текущей строки после пары символов "//" является комментарием.

```
nat i = 0;                // описание натуральной переменной с инициализацией
seq int p;                // p — последовательность целых чисел
set char s;               // s — произвольное подмножество литер
```

Для обозначения типа, представленного изображением типа, некоторым именем используется описание типа. Например,

```
type SeqInt = seq int;
type Complex = struct real re, im end;
                                  // Complex — тип кортежа с полями re и im
type Nat10 = nat condition @>9; // Nat10 — тип натуральных ≥ 10
type Diapason(nat n) = 1..n+1;      // 1≤@≤ n+1; n — параметр типа Diapason
type Ar10(Nat10 n) = array 1..n of real;
                                  // Ar10 — тип массива с числом элементов ≥ 10
```

Описание переменой и описание типа может снабжаться условием вида **condition** <логическое выражение>, ограничивающим множество допустимых значений переменной (типа). Литера @ используется для обозначения произвольного значения типа (или переменной).

Переменные, используемые в описании типа, являются **параметрами типа**. Переменная n является формальным параметром типов Diapason и Ar10. В любом использовании типа Diapason или Ar10 вместо параметра типа подставляется некоторое выражение. Например, Diapason(15) будет обозначать тип 1..16.

Конструктор определяет объект (или значение) структурного типа по значениям компонентов объекта. В конструкторе <структурный тип>(<список выражений>) компоненты задаются <списком выражений>. Конструктор может быть записан в виде (<список выражений>), если тип однозначно определяется из позиции, в которой находится конструктор. Например, конструктор **seq int**(a, b, c) определяет последовательность целых чисел, где любой из аргументов a, b, c может быть последовательностью или целым. Допускается иная форма записи: a + b + c. Данная форма используется также для конструктора объекта типа множества. Например, конструктор s+e определяет новое множество добавлением элемента e к множеству s.

Массивом является объект типа **array** I **of** T, где T — тип элементов, а I — конечный тип индексов. Конструктором массива A является оператор:

```
forAll k in I do A[k] = <выражение> end
```

Вычисление тела конструктора для разных индексов k реализуется независимо, возможно параллельно. Другие виды конструкторов являются про-

изводными от конструктора “**forAll**”. Например, конструктор (2: 1.5, 3: 2.5) определяет массив типа **array** 2..3 **of real**, причем элемент с индексом 2 равен 1.5, а элемент с индексом 3 равен 2.5. Конструктор вида A + B определяет объединение массивов A и B с непересекающимися типами индексов. Конструктор вида A(i: x) определяет замещение в массиве A элемента с индексом i новым значением x. Через ind(A) обозначим тип индекса массива A. Замещение элемента в массиве определяется следующей спецификацией:

$$B=A(i: x) \equiv i\in ind(A) \Rightarrow (\forall j\in ind(A)\setminus\{i\}) (B[j]=A[j]) \ \& \ B[i]=x$$

Пример 2. Рассмотрим программу перестановки 5 и 6 элементов в массиве a типа Ar10:

```
Permutation(Nat10 m, Ar10(m) a: Ar10(m) b) ≡
                         b[5]=a[6] & b[6]=a[5] & (∀j=1..4, 7..m)(b[j]=a[j])
{     b = a(5: a[6], 6: a[5])     }
```

Описания аргументов ″Nat10 m, Ar10(m) a″ могут быть заменены на ″Ar10(Nat10 m) a″. Иначе говоря, параметр типа, снабженный описателем типа, считается аргументом определяемого предиката.

В языке P для непустой последовательности s определены функции head(s) и tail(s), вычисляющие соответственно начальный элемент последовательности s и оставшуюся часть s. Пустая последовательность обозначается константой **nil**.

Пример 3. Допустим, для последовательности s целых чисел требуется извлечь второй элемент и присвоить переменной e. Последовательность может содержать менее двух элементов. Логическая переменная exist = **true**, если второй элемент существует. Вычисление переменных e и exist можно представить следующим определением:

```
elemTwo(seq int s: int e, bool exist ) ≡
                    exist=(s≠nil & tail(s)≠nil) & (exist ⇒ e=head(head(s)))
{     if s=nil ∨ tail(s)=nil then        exst = false
      else        e=head(head(s)) || exist = true
      end
}
```

Предположим, имеется фрагмент программы с вызовом предиката elemTwo и последующими вызовами некоторых предикатов A и B, причем e используется среди аргументов A:

```
elemTwo(s: e, exist); if exist then A(e…) else B(…) end
```

Теперь подставим тело elemTwo на место вызова. Очевидно, что переменная exist и все действия с нею оказываются избыточными. Устраняя переменную exist, получим результат подстановки в более компактном виде:

if s=**nil** ∨ tail(s)= **nil** **then** B(…) **else** e=head(head(s)); A(e…) **end**

Отмеченная неэффективность определения elemTwo имеет место всегда, независимо от того, будет или не будет подставляться определение на место вызова. Другой недостаток: переменная e не определена для коротких последовательностей. Ниже будут определены новые языковые конструкции (гиперфункция и оператор расщепления), позволяющие адекватно и эффективно реализовать фрагмент из примера 2.

Допустим, имеются предикаты S(x: y), Q(x: z) и C(x), где x, y и z — непересекающиеся, возможно пустые, наборы переменных. Предикат H(x: y, z) определяется следующим тождеством:

$$H(x{:}\ y, z) \equiv (C(x) \Rightarrow S(x{:}\ y))\ \&\ (\neg C(x) \Rightarrow Q(x{:}\ z))$$

При выполнении условия C(x) предикат H(x: y, z) совпадает с функцией S(x: y), а при нарушении условия — с функцией Q(x: z). Таким образом, H(x: y, z) является результатом склеивания двух функций S(x: y) и Q(x: z). Предикат H(x: y, z) называется **гиперфункцией** с двумя **ветвями** и записывается в виде H(x: y | z). На первой ветви определен набор y, а на второй — z. В языке P спецификация гиперфункции H(x: y | z) записывается в следующем виде:

H(x: y, z) ≡ <ограничения> ⇒ <собственно спецификация>; #1: C(x)

Здесь <ограничения> определяют допустимое подмножество значений переменных набора x, которое далее делится на две части условием C(x); после “#1:” следует условие на первую ветвь, т.е. C(x). Очевидно, что <собственно спецификация> ≡ S(x: y) ∨ Q(x: z).

Исполнение гиперфункции завершается на одной из двух ветвей. При этом вычисляются значения результирующих переменных завершившейся ветви; переменные другой ветви не вычисляются. Для указания того, какой ветвью завершилось исполнение тела предиката-гиперфункции, в теле используется **указатель завершения**: #1 или #2.

Гиперфункция совмещает в себе преобразователь и распознаватель. Ветвь гиперфункции может быть **пустой**, т.е. не содержать переменных-результатов. По пустой ветви гиперфункция является только распознавателем.

Вернемся к примеру 3 и дадим другое определение предиката elemTwo в виде гиперфункции:

```
elemTwo(seq int s: int e | ) ≡
                    s=nil ∨ tail(s)=nil ∨ e=head(head(s)); #1: s≠nil & tail(s)≠nil
{     if s=nil ∨ tail(s)=nil then     #2
      else        e=head(head(s))   #1
      end
}
```

Оператор расщепления состоит из заголовка и альтернатив расщепления:

```
    split     <вызов предиката-гиперфункции>
    do        <оператор – первая альтернатива расщепления>
    do        <оператор – вторая альтернатива расщепления>
    end
```

Если вызов предиката-гиперфункции завершается первой ветвью, далее исполняется первая альтернатива расщепления, иначе — вторая.

Определение гиперфункции и оператора расщепления обобщается на произвольное число ветвей.

Фрагмент с вызовом предиката elemTwo из примера 3 может быть переписан следующим образом:

```
split elemTwo(s: e | ) do A(e…) do B(…) end
```

При исполнении данного оператора выход по указателю завершения #1 из тела elemTwo реализуется непосредственно на вызов A(e…), а выход по указателю завершения #2 — на B(…).

Можно сформулировать следующее утверждение: использование в программе переменной логического типа, за редкими исключениями, усложняет внутренний интерфейс программы и ухудшает эффективность.

Определим стандартный предикат Comp следующей спецификацией:
Comp(**seq int** s: | **int** d, **seq int** r) ≡ s=**nil** ∨ d=head(s) & r=tail(s); #1: s=**nil**

Пример 4. Построим определение предиката elemTwo из примера 3 через гиперфункцию Comp без использования операций head, tail и константы **nil**.

```
elemTwo(seq int s: int e | ) ≡
                    s=nil ∨ tail(s)=nil ∨ e=head(head(s)); #1: s≠nil & tail(s)≠nil
{     int e1; seq int s1, s2;
      split Comp(s: | e1, s1)
```

```
    do   #2
    do   split Comp(s1: | e, s2)
       do     #2
       do     #1
       end
    end
}
```

Альтернатива оператора расщепления является пустой, если в ней нет других действий, кроме возможного указателя завершения. Действует следующее правило: в пустой альтернативе указатель завершения переносится в конец соответствующей ветви вызова гиперфункции. Определение гиперфункции elemTwo переписывается следующим образом:

```
elemTwo(seq int s: int e | ) ≡
                  s=nil ∨ tail(s)=nil ∨ e=head(head(s)); #1: s≠nil & tail(s)≠nil
{     split Comp(s: #2 | int  e1, seq int s1)
      do
      do    Comp(s1: #2 | e, seq int s2 #1)
      end
}
```

В данном определении elemTwo кроме переноса указателей завершения реализован перенос описаний локальных переменных e1, s1 и s2 в позиции их **определения**, т.е. места в программе, где им присваиваются значения.

Оператор расщепления является вырожденным, если он содержит только одну непустую альтернативу. В этом случае оператор расщепления заменяется блоком. Последнее определение elemTwo должно быть заменено следующим:

```
elemTwo(seq int s: int e | ) ≡
                  s=nil ∨ tail(s)=nil ∨ e=head(head(s)); #1: s≠nil & tail(s)≠nil
{     Comp(s: #2 | _, seq int s1);
      Comp(s1: #2 | e, _ #1)
}
```

Дополнительно в этом определении проведена замена результирующих вхождений локальных переменных e1 и s2 стандартным именем "_", означающим, что соответствующий результат игнорируется при исполнении.

Использование "##" в вызове гиперфункции Comp(s: ## | d, r) означает невозможность выхода на первую ветвь при исполнении Comp. Иначе говоря, имеет место ограничение: условие первой ветви — ложно, т.е. s≠**nil**.

Константа **nil** и операции head и tail следующим образом определяются через гиперфункцию Comp:

d=head(s) & r=tail(s) ≡ Comp(s: ## | d, r)
s=**nil** ≡ Comp(s: | ##)

Технология предикатного программирования предполагает написание программы на языке P, применение к ней набора **трансформаций** с получением эффективной программы на императивном расширении языка P и конвертацию на любой из императивных языков: C, C++ и др. Базовыми трансформациями являются:

- **склеивание переменных**, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии (tail-рекурсии в языке Лисп) циклом;
- подстановка определения предиката на место его вызова;
- кодирование структурных объектов низкоуровневыми структурами с использованием массивов и указателей.

Результатом применения трансформаций является императивная программа, не уступающая по эффективности написанной вручную.

Трансформации применяются к программе на императивном расширении языка P, включающем операторы присваивания, циклы вида **loop**, **while** и **for**, операторы перехода и групповые операторы присваивания следующего вида:

| <список переменных> | := | <список выражений> |

При исполнении этой конструкции вычисленные значения списка выражений одновременно присваиваются соответствующим переменным левой части.

Если известно, что в определении предиката переменная-аргумент x склеивается с переменной-результатом y, то вместо y используется имя x′. Описание вида <тип> x* определяет описание двух параметров: аргумента x и результата x′. Например, предполагается склеивание параметров a и b в предикате Permutation(Ar10(Nat10 m) a: Ar10(m) b), и поэтому предикат записывается в виде Permutation(Ar10(Nat10 m) a: Ar10(m) a′) или Permutation(Ar10(Nat10 m) a*:).

Рассмотрим трансформацию предикатной программы Permutation(Ar10(Nat10 m) a*:). Поскольку a и a′ склеиваются, раскрытие конструктора в операторе a′ = a (5: a[6], 6: a[5]) дает следующий параллельный оператор: a′[5] := a[6] || a′[6] := a[5]. При замене a′ на a параллельный оператор превращается в групповой оператор присваивания:

| a[5], a[6] | := | a[6], a[5] |. Раскрытие последнего оператора, т.е. его замена на обычные операторы присваивания, возможна лишь при введении дополнительной переменной. В итоге получим следующую программу:

```
Permutation(Ar10(Nat10 m) a*: )
{       int t := a[5]; a[5] := a[6]; a[6] := t       }
```

Доказательство правильности предикатной программы реализуется доказательством истинности собственно спецификации предиката из предположения истинности ограничений данного предиката и истинности формулы, составляющей тело предиката. Любая исполняемая конструкция, операция или оператор, в соответствии с математической семантикой имеет ограничения и собственно спецификацию. Если декларирована истинность конструкции, то сначала необходимо проверить корректность конструкции, т.е. истинность ограничений конструкции. После этого становится истинной собственно спецификация конструкции, присоединяемая к списку истинных утверждений процесса доказательства. При наличии рекурсивного вызова применяется доказательство по индукции. Например, доказательство правильности определения предиката Permutation включает доказательство ограничений $5 \in ind(a)$ и $6 \in ind(a)$ для операции замещения и элементов массива a[5] и a[6].

2. ПРОСТОЙ АЛГОРИТМ ПОСТРОЕНИЯ ДЕРЕВА СУФФИКСОВ

Дерево суффиксов является эффективной структурой для многих алгоритмов обработки текстов. Типичная задача проверки наличия строки v внутри другой строки t решается за O(|v|) шагов (независимо от длины t), если построено дерево суффиксов для строки t. Непосредственный алгоритм построения дерева суффиксов имеет квадратичную оценку по времени. Первый линейный по времени алгоритм предложен Вейнером [16]. Более простым и эффективным считается алгоритм Маккрейта [11], линейность которого обеспечивается использованием суффиксных ссылок. Эти же ссылки используются в алгоритме Укконена [15], который может строить дерево суффиксов одновременно с последовательной загрузкой символов строки t в память. Появились другие алгоритмы, например [8, 12, 9, 14].

Введем базовые определения и обозначения. *Алфавитом* A называется конечное множество элементов, называемых *символами*. A^* обозначает множество строк в алфавите A. *Пустая строка* обозначается через **nil**. $A^+ = A^* \setminus \{$**nil**$\}$. Допустим, строка t является конкатенацией строк u, v и w, возможно пустых, т.е. t = u+v+w. Тогда строка u называется *префиксом*

строки t, w — *суффиксом*, а v — *словом* строки t. Будем также использовать предикаты $pref(u, t)$ и $suf(w, t)$. Префикс u называется *собственным*, если $u \neq t$. Длина строки z, т.е. число символов в ней, обозначается через $|z|$.

Определим понятие *строкового дерева* в алфавите A. Имеется единственная корневая вершина. Каждая ориентированная дуга, соединяющая две вершины, снабжается *меткой*, являющейся непустой строкой. Для любых двух различных дуг, исходящих из одной вершины, их метки должны начинаться с разных символов. Дуга, исходящая из вершины, называется *a-дугой*, если метка дуги начинается символом a.

Ограничимся рассмотрением компактных строковых деревьев, у которых любая вершина, отличная от листа и корня, имеет не менее двух исходящих дуг.

Через $path(k, T)$ обозначим результат конкатенации меток на пути от корня до вершины k в строковом дереве T. *Локусом* строки w, обозначаемым через $\overline{w}$, называется вершина (если такая существует), путь до которой есть w, т.е. $w = path(\overline{w}, T)$. В силу определений, в строковом дереве для любой строки w существует не более одного локуса.

Строка v *встречается* в строковом дереве T, если существует вершина k такая, что v является префиксом для $path(k, T)$. *Множество строк дерева* T, обозначаемое через $words(T)$, есть множество строк, встречающихся в дереве T. Рассмотрим множество $Paths(T) = \{w: \exists \text{ лист } k\ (\overline{w} = k)\}$, определяющее *множество путей* до всех листьев T. Любая строка, встречающаяся в T, является префиксом некоторой другой строки, представленной путем до некоторого листа дерева. Поэтому $Paths(T)$ может использоваться для представления множества строк дерева вместо $words(T)$.

Дерево суффиксов для строки t, обозначаемое через $sufTree(t)$, есть строковое дерево T, множество строк которого совпадает с множеством слов строки t, т.е. $words(T) = \{w \mid w \text{ есть слово } t\}$. Дерево суффиксов для строки agcgacgagd представлено на рис.1.

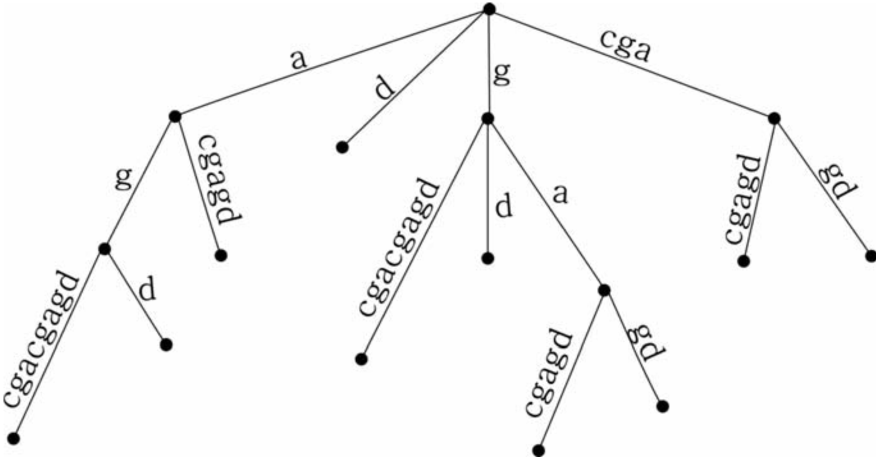

Рис. 1

Множество слов строки t может быть представлено множеством суффиксов t, поскольку любое слово является префиксом некоторого суффикса. Поэтому для построения дерева суффиксов достаточно включить в дерево все суффиксы строки t. Это подтверждается следующей леммой.

Лемма 1. Допустим, T — строковое дерево, t — строка. Справедливо следующее утверждение: Paths(T) = {w: suf(w, t)} ⇒ T = sufTree(t).

Рассмотрим сначала простой алгоритм построения дерева суффиксов, соответствующий алгоритму Маккрейта [11]. Он заключается в последовательном включении в дерево всех суффиксов в порядке убывания их длины.

Определим типы данных. Алфавит A определен как произвольный конечный тип, и поэтому A — параметрический тип. Типы строки символов, строкового дерева и дуги вершины обозначим соответственно через Str, Tree и Arc. Определим тип дерева рекурсивным образом как вершину с набором дуг, присоединяющих поддеревья. Доступ к дуге, исходящей из вершины, реализуется по первому символу метки дуги. Поэтому вершина представляется массивом дуг; типом индексов является подмножество алфавита A. Типы данных представляются следующими описаниями:

```
type A; // алфавит
type Str = seq A;
type AA = set A;
type Arc = struct Str lab, Tree subtree end condition @.lab≠nil;
type Tree(AA s) = array s of Arc condition ∀a∈s (head(@[a].lab)=a);
```

Можно показать, что любой объект типа Tree является строковым деревом. Тип Tree параметризован подмножеством символов s. Если s — пусто, получим пустое дерево (соответствующее листу дерева), которое обозначим через NIL и определим следующим описанием:

Tree NIL = Tree(); // конструктор пустого массива

Исходная строка t, для которой строится дерево суффиксов, определим как глобальную переменную следующим описанием:

```
Str t condition guard(t);
A gu; // охранный символ
guard(Str v) ≡ ∃x∈Str (v=x+gu & ∀y,z∈Str ∀a∈A (x=y+a+z ⇒a≠gu));
```

Предикат guard(v) определяет наличие охранного символа в конце строки v и его отсутствие внутри v. Как следствие, v ≠ **nil**. Использование охранного символа существенно упрощает алгоритм.

Задачу построения дерева суффиксов определим предикатом suffixMake(: AA s, Tree(s) T) со спецификацией T = sufTree(t).

Допустим, v есть суффикс t. Обозначим через pTree(v) строковое дерево, включающее все суффиксы w строки t такие, что |w| > |v|.

Лемма 2. pTree(t) = NIL.

Лемма 3. pTree(**nil**) = sufTree(t).

Рассмотрим более общую задачу sufBuild(AA s, Tree(s) T, Str v: AA s′, Tree(s′) T′) с ограничениями suf(v,t) & T = pTree(v) и спецификацией T′ = sufTree(t). Справедливо следующее определение:

```
suffixMake( : AA s, Tree(s) T) ≡ suf(v,t) & T = pTree(v) ⇒ T = sufTree(t)
{     sufBuild(AA(), NIL, t: s, T) }
```

Запись AA() обозначает конструктор пустого множества символов.

Δ Ограничения sufBuild являются истинными для фактических параметров AA(), NIL и t в силу леммы 2. Тогда верна спецификация sufBuild, совпадающая со спецификацией suffixMake, что доказывает правильность данного определения. □

Для включения очередного суффикса v строки t в создаваемое строковое дерево используется предикат inclStr(AA s*, Tree(s) T, Str v: Tree(s′) T′) с ограничениями guard(v) & T = pTree(v) и спецификацией Paths(T′) = Paths(T) $\cup$ {v}.

```
sufBuild(AA s*, Tree(s) T, Str v: Tree(s′) T′) ≡
                                        suf(v, t) & T=pTree(v) ⇒ T′=sufTree(t)
{     if v = nil then s′ = s || T′ = T
      else        sufBuild(inclStr(T, v), tail(v): s′, T′)
      end
}
```

Δ Рассмотрим случай v = **nil**. Тогда s′ = s & T′ = T. Из ограничения T = pTree(v) и леммы 3 следует T = sufTree(t), и значит T′ = sufTree(t), т.е. спецификация истинна. Рассмотрим случай v$\neq$**nil**. Сначала необходимо установить истинность ограничений внутренних вызовов. Это очевидно для tail(v). Ограничение guard(v) следует из suf(v, t) и v $\neq$ **nil**. Ограничение T = pTree(v) есть ограничение sufBuild. Итак, выполняются ограничения для вызова inclStr(T, v). Используя обозначения s_0,T_0 для результатов, из спецификации inclStr получим: Paths(T_0) = Paths(T)$\cup${v}. Отсюда следует: T_0 = pTree(w), где w = tail(v). Таким образом, аргументы s_0,T_0, w вызова sufBuild удовлетворяют ограничениям sufBuild. Поскольку суффикс w короче v и для v = **nil** истинность sufBuild доказана, то можно применить индуктивное предположение о истинности спецификации для вызова sufBuild, т.е. имеет место T′ = sufTree(t), что доказывает правильность всего определения. □

Рассмотрим реализацию предиката

inclStr(AA s*, Tree(s) T, Str v: Tree(s′) T′).

Определяется разбиение v на две части v = z + w, где z встречается в дереве T, а строка z + head(w) не встречается. Если локус $\overline{z}$ отсутствует в дереве, то соответствующая вершина вставляется в дерево. К локусу $\overline{z}$ подсоединяется дуга с меткой w.

В определении inclStr используется гиперфункция getArc(AA s, Tree(s) T, A a: | Arc L) со спецификацией a$\in$s $\Rightarrow$ L = T[a]; #1: a$\notin$s. Гиперфункция для корневой вершины T находит a-дугу L, если таковая существует. Используется также гиперфункция diff(Str g, w: Str g′, w′ | Str w′) с ограничением $\neg$pref(w, g) и спецификацией $\exists$y$\in$Str (g = y+g′ & w = y + w′ & (g′ = **nil** $\vee$ head(g′) $\neq$ head(w′))); #1: $\neg$pref(g, w). Отметим, что $\neg$pref(g, w) $\equiv$ g′$\neq$**nil**. Гиперфункция diff определяет место различия при сравнении строк g и w,

фиксируя несовпавшие остатки в g′ и w′. Используется также функция z = headForTail(Str x, y) с ограничением suf(y, x) и спецификацией z + y = x.

```
inclStr(AA s*, Tree(s) T, Str v: Tree(s′) T′) ≡
                              guard(v) & T=pTree(v) ⇒ Paths(T′) = Paths(T)∪{v}
{     A a = head(v);
      split       getArc(s, T, a: | Arc L)
      do    s+a, T + (a: (v, NIL))
      do    split diff(tail(L.lab), tail(v): Str g, w | Str w)
            do    Tree U = (head(g): (g, L.subtree), head(w): (w, NIL));
                  s, T(a: (headForTail(L.lab, g), U))
            do    inclStr(L.subtree, w: _, Tree subT);
                  s, T(a: (L.lab, subT))
            end
      end
}
```

Δ Поскольку v ≠ **nil**, функции head(v) и tail(v) определены. Допустим, реализуется первая альтернатива внешнего оператора расщепления, т.е. имеют место a∉s, s′ = s + a и T′ = T+(a: (v, NIL)). При этом, поскольку a∉s, вторая операция "+" — корректна, а s′ и T — правильно согласованы, соблюдены также условия на тип Tree. Очевидно, что спецификация inclStr выполняется. Допустим, имеет место вторая альтернатива расщепления, т.е. верны a∈s и L = T[a]. Кроме того, из условия на тип Tree, имеем: L.lab ≠ **nil** и head(L.lab) = a. Следовательно, tail(L.lab) определено. Проверим ограничения для вызова diff. Допустим, что tail(v) = **nil**, из чего следует, что строка из единственного символа ga встретилась в T, что противоречит ограничению T = pTree(v). Следовательно, tail(v) ≠ **nil**, и тогда выполняется ограничение diff: guard(tail(v)). Допустим, что нарушается ограничение diff: ¬pref(tail(v), tail(L.lab)), из чего будет следовать pref(v, L.lab), и тогда v будет встречаться в T, что противоречит T = pTree(v). Итак, ограничение diff выполняется. Допустим, имеет место первая альтернатива внутреннего оператора расщепления, для которой g≠**nil**. Тогда для некоторой строки y выполняются: L.lab = y + g, v = y + w, head(g) ≠ head(w). Очевидно, что y = headForTail(L.lab, g). С учетом этого, первая альтернатива определяет соотношения s′ = s и T′ = T(a: (y, U)). Деревья T и T′ различаются лишь на поддеревьях для a-дуги, которые будем обозначать через T_a и T'_a. Оба эти дерева содержат поддерево L.subtree. Множество путей, определяемых L.subtree, совпадают для T_a и T'_a, поскольку L.lab = y + g. Наконец, для T'_a имеется дополнительная дуга (w, NIL), которая вместе с верхней частью — строкой y и в силу v = y + w определяет дополнительный путь v. Это дока-

зывает спецификацию inclStr. Отметим, что соблюдены условия на тип Tree во всех конструкторах деревьев.

Допустим, выполняется вторая альтернатива diff. Тогда L.lab + w = v. Проверим ограничения для параметров рекурсивного вызова inclStr. Невозможно w = **nil**, поскольку в этом случае v встречается в T и будет нарушено T = pTree(v). Тогда из guard(v) и w ≠ **nil** следует guard(w). Ограничение L.subtree = pTree(w) следует из T = pTree(v) и L.lab + w = v. Для рекурсивного вызова inclStr число вершин от корня до листьев на 1 меньше, а для дерева NIL из единственной вершины спецификация inclStr доказана, следовательно вызов inclStr(L.subtree, w: _, Tree subT) является истинным по индуктивному предположению. Тогда имеет место Paths(subT) = Paths(L.subtree) ∪ {w}. Наконец, поскольку T′ = T(a: (L.lab, subT), то спецификация inclStr верна. □

```
diff(Str g, w: Str g′, w′ | Str w′) ≡
    ¬pref(w, g) ⇒ ∃y∈Str (g=y+g′ & w=y+w′ & (g′=nil ∨ head(g′)≠head(w′)));
    #1: ¬pref(g, w). Условие на ветвь эквивалентно g′≠nil.
{     if g = nil then          w′ = w      #2    // x=nil
      elsif head(g)=head(w) then     diff(tail(g), tail(w): g′, w′ #1 | w′ #2)
      else        g′=g || w′ = w  #1
      end
}
```

Δ Из ограничения ¬pref(w, g) следует, что w ≠ **nil**. Допустим, истинна первая альтернатива условного оператора, т.е. g = **nil**. Тогда выполняется w′ = w. Для y = **nil** спецификация diff выполняется. Условие второй ветви гиперфункции pref(g, w) также выполняется. Допустим теперь, что первое условие ложно, т.е. g ≠ **nil**, и выполняется второе условие head(g) = head(w). Отметим, что операции head и tail определены для g и w. Проверим ограничения для параметров рекурсивного вызова diff. Если tail(w) = **nil**, то окажется нарушенным второе ограничение ¬pref(w, g). Следовательно, tail(w) ≠ **nil**. Если нарушается ¬pref(tail(w), tail(g)), то также нарушается исходное ограничение ¬pref(w, g). Поскольку tail(g) короче g и для g = **nil** спецификация diff доказана, то для рекурсивного вызова diff можно применить индуктивное предположение об истинности вызова, и тогда верна спецификация diff для аргументов вызова. Допустим, истинность спецификации реализуется при y_0. Легко видеть, что при $y_1 = y_0 + \text{head}(g)$ спецификация верна для аргументов g и w, что требовалось доказать. Далее, поскольку pref(tail(w), tail(g)) ≡ pref(w, g), то условия ветвей гиперфункции определены правильно. Наконец, предположим, что нарушаются оба условия условного оператора, т.е. реализуется последняя ветвь. Тогда g ≠ **nil**,

$head(g) \neq head(w)$, $g' = g$ и $w' = w$. При y = **nil** спецификация diff выполняется. Выполняется также условие первой ветви $\neg pref(g, w)$, ввиду $head(g) \neq head(w)$. □

Определения предикатов suffixMake, sufBuild, inclStr и diff вместе с описаниями типов и глобальных переменных составляют полную предикатную программу. Рекурсия в определениях sufBuild и diff является хвостовой и поэтому в реализации императивной программы может быть заменена циклом, а сами эти определения могут быть подставлены на место соответствующих вызовов. Рекурсия в определении inclStr не является хвостовой. Можно пытаться найти алгоритм для inclStr с хвостовой формой рекурсии. Однако более существенная неэффективность данной версии алгоритма заключается в том, что поиск очередного суффикса v в дереве T при его включении в дерево, проводится начиная с корня T, что определяет оценку $O(|t| \log |t|)$.

3. АЛГОРИТМ МАККРЕЙТА

Алгоритм Маккрейта [11] использует суффиксные ссылки и определяет позицию в дереве очередного включаемого суффикса за константное время, что обеспечивает общую оценку $O(|t|)$.

Допустим, w — непустой суффикс строки t. Префикс z суффикса w называется *вложенным*, если z является также префиксом другого суффикса w′ большей длинны: $|w'| > |w|$. Рассмотрим разбиение суффикса на две строки: $w = z + v$, где z — вложенный префикс максимальной длины. Строка z называется *головной частью* суффикса w, а v — *хвостовой частью*. Хвостовая часть не может быть пустой из-за наличия охранного символа в конце w. Разбиение суффикса $w = z + v$ на головную часть z и хвостовую часть v обозначается предикатом sufParts(z, v).

Лемма 4. Строка z является головной частью суффикса w тогда и только тогда, когда z встречается в дереве pTree(w), а $z + head(v)$ не встречается.

Для включения суффикса w в дерево pTree(w) достаточно найти позицию в дереве головной части z, поскольку строка z в составе суффикса w′ большей длины должна встречаться в pTree(w). Если локус $\overline{z}$ отсутствует в дереве, то соответствующая вершина должна быть создана. Наконец, к локусу $\overline{z}$ следует присоединить новую дугу с меткой, равной хвостовой части v.

Рассмотрим очередной суффикс w строки t. Допустим, w = a + s, где a — символ, s — непустая строка. Пусть s = g + v, причем a + g — головная часть суффикса w, а v — хвостовая часть w.

Лемма 5. Строка g является префиксом головной части суффикса s.

Δ Поскольку a + g — головная часть суффикса w, существует суффикс w′ большей длины, для которого a + g является префиксом. Пусть w′ = a + g + v′. Тогда строка g является префиксом суффикса g + v′ большей длины, чем s, и, следовательно, g является вложенным префиксом для s. Так как головная часть — максимальный вложенный префикс, то g является префиксом головной части суффикса s. □

При включении суффикса w в дерево pTree(w) получаем дерево pTree(s). Далее, требуется включить в pTree(s) следующий суффикс, т.е. s; результатом включения будет дерево pTree(tail(s)).

Лемма 6. Если нет локуса $\overline{g}$ в дереве pTree(s), то строка g является головной частью суффикса s.

Δ Поскольку дерево pTree(s) содержит локус $\overline{a+g}$, существует два разных символа c и d и строки x и y такие, что

$$|a + g + c + x| < |a + g + d + y| \leq |a + s|.$$

Тогда $|g + c + x| < |g + d + y| \leq |s|$. Поскольку $\overline{g}$ не является локусом в pTree(s), после строки g в этом дереве может следовать не более чем один символ. Из последнего неравенства следует, что таким символом может быть только c. А так как c ≠ d, то g + d + y = s. Отсюда следует, что g + d не является вложенным префиксом для s и тогда g является максимальным вложенным префиксом, т.е. головной частью для s. □

Для символа c и строки x дуга $\overline{c+x} \to \overline{x}$ называется *суффиксной ссылкой* локуса $\overline{c+x}$ на локус $\overline{x}$.

Лемма 7. Суффиксная ссылка определена на дереве pTree(tail(s)) для всех *внутренних* вершин pTree(s), т.е. всех вершин, кроме корня и листьев.

Δ Действительно, пусть внутренняя вершина в pTree(s) определяется некоторым локусом $\overline{c+x}$. Это значит, что существуют суффиксы c + x + y и c + x + y′ длиннее s, причем y и y′ начинаются с разных символов. Тогда x + y и x + y′ являются суффиксами, встречающимися в pTree(tail(s)), и

должен существовать локус $\overline{x}$. □ Отметим, что локус $\overline{a+g}$ определен в дереве pTree(s), а локус $\overline{g}$ — в дереве pTree(tail(s)). Поэтому суффиксная ссылка $\overline{a+g} \to \overline{g}$ определена в pTree(tail(s)), однако может быть не определена в pTree(s).

Дерево суффиксов с суффиксными ссылками для строки agcgacgagd представлено на рис. 2; суффиксные ссылки изображены пунктирными дугами.

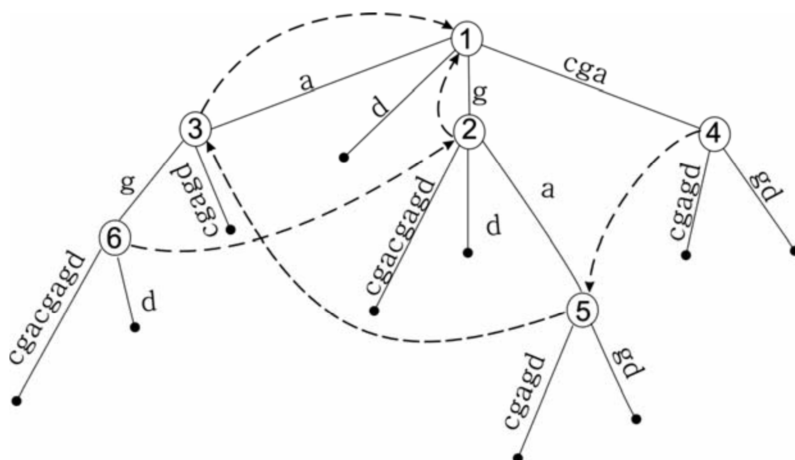

Рис. 2

Для произвольной строки z, встречающейся в строковом дереве T, используется представление в виде *древесной пары* (b, u), где b — вершина, u — строка и z = path(b, T) + u; кроме того, b является вершиной с максимально длинным путем path(b, T) среди всех вершин, удовлетворяющих указанным условиям. Из определения следует, что либо u — пуста, либо u является собственным префиксом метки одной из дуг, исходящих из b.

Пусть строка a + g , где a — символ, представляется в дереве T древесной парой (b, u). *Суффиксной ссылкой* древесной пары (b, u) является древесная пара, соответствующая строке g в дереве T.

Лемма 8. Суффиксная ссылка для головной части a + g суффикса w в дереве T определена.

Δ Следует из того, что строка g встречается в дереве T. □

Допустим, имеется суффикс w, включенный в дерево pTree(s). Головная часть суффикса w представляется в виде древесной пары (b, u) относительно дерева pTree(w), хвостовой частью является строка v. На очередном шаге алгоритма Маккрейта [11] находится головная часть следующего суффикса s и определяется его древесная пара относительно дерева pTree(s). Для этой цели сначала по древесной паре (b, u) для строки a + g находится древесная пара строки g с использованием суффиксной ссылки для вершины b. Далее, для получения головной части суффикса s реализуется сканирование дерева pTree(s), но не с корня, а с позиции, определяемой полученной древесной парой. Результатом сканирования является требуемая головная часть и ее позиция в дереве в виде древесной пары.

Строковое дерево с суффиксными ссылками нельзя представить в виде объекта типа Tree. Ниже определим другой способ представления строковых деревьев, в котором каждая вершина определяется независимой структурой.

Обозначим через Node некоторое генеральное множество, из которого берутся имена вершин. Будем считать, что Node — внешний тип, являющийся параметром предикатной программы и определяемый описанием:

```
type Node;
```

Для порождения имен используется класс genNames, определяемый следующей спецификацией:

```
class genNames(type NAME)
{     type NAMES = set NAME;
      NAME NIL;
      newName(NAMES j: NAME n, NAMES j′) ≡ n ≠ NIL & n∉j & j′ = j ∪ {n};
};
```

Имя NIL является специальным. Оно используется для вершин, являющихся листьями. Присоединение к предикатной программе некоторой реализации класса genNames задается директивой:

```
#include getNames(Node)
```

Возможны, например, две следующие реализации класса genNames. В первой — в качестве имен используется отрезок натурального ряда, а NIL = 0. Во второй — именами являются адреса вершин в памяти программы; предикат newName реализует также отведение памяти для очередной создаваемой вершины.

Для создаваемого дерева суффиксов в качестве корневой определим вершину, имя которой хранится в глобальной переменной root. Тип V для множества подмножеств имен вводится для сокращения записи. Для дуги, вершины, строкового дерева, графа суффиксиных ссылок и древесной пары определим соответственно типы Arc, NodeAtrs, ST, SUFL и Pair:

```
Node root;                 // имя корневой вершины дерева
type V = NAMES(Node);   // множество подмножеств имен вершин
type Arc = struct Str lab, Node to end condition @.lab≠nil;
type NodeAtrs(AA s) = array s of Arc condition ∀a∈s (head(@[a].lab)=a);
type ST(V j) = array j of NodeAtrs  condition rootedTree(@, root);
type SUFL(V i ) = array i of Node;
type Pair = struct Node locus, Str ext end;
```

Значение NIL в качестве поля to типа Arc определяет лист. Отметим, что для листа не создается вершины. Тип вершины NodeAtrs определяет набор дуг, исходящих из вершины. Объект типа ST определяет строковое дерево в виде массива вершин для некоторого подмножества имен j, причем каждая вершина независимо доступна по имени и определяет набор дуг. Условие rootedTree(@, root) определяет, что граф @ есть односвязная компонента, являющаяся правильным деревом с корнем root; в частности, root∈j и для всякой дуги r, принадлежащей дереву, r.to ∈ j, если только r.to ≠ NIL. С учетом условий, указанных при описании типов Arc и NodeAtrs, можно показать, что произвольный объект типа ST является строковым деревом. Отметим, что используются также типы A, Str, AA и глобальные переменные t и gu, определенные выше для более простой версии данной задачи.

Строковое дерево T и ассоциированный с ним граф суффиксных ссылок f должны удовлетворять следующему предикату:

```
sufLinks(ST(V j) T, SUFL(V i) f, Node last) ≡
      if last=NIL then i = j\{root} else i = j\{root, last} end &
      ∀k∈i (path(f[k], T) = tail(path(k, T)));
```

В соответствии с леммой 7 множество i должно содержать все вершины множества j, кроме корня и возможно еще одной вершины, обозначенной через last. Мы полагаем last = NIL при отсутствии вершины last.

Определим предикаты occ и pair и функцию str:

occ(Str z, (V j) T) ≡ строка z встречается в строковом дереве T;
pair(Pair e, (V j) T) ≡ e является древесной парой в строковом дереве T;
str(Pair e, (V j) T) = path(e.locus, T) + e.ext;

Отметим, что str(e, T) определено и при нарушении условия pair(e, T). Однако необходимо выполнение ограничения e.locus ∈ j.

Задачу построения дерева суффиксов по алгоритму Маккрейта определим предикатом

SufixMake_mcc(: V j, ST(j) T)

со спецификацией T = sufTree(t).

Попытаемся определить обобщающую задачу в виде предиката mcc(V j*, ST(j) T, SUFL(V i) f, Pair e, Str v: ST(j′) T′) со спецификацией T′ = sufTree(t), где T=pTree(s) для некоторого суффикса s строки t, при этом w — последний суффикс, включенный в T, т.е. s = tail(w); e — древесная пара головной части суффикса w, а v — его хвостовая часть, т.е. w = str(e, T) + v.

Обозначим через last вершину, созданную в дереве T при включении суффикса w. Тогда path(last, T) = str(e, T). Вершина, определяемая суффиксной ссылкой из вершины last, появится лишь в дереве pTree(tail(s)). Отметим, что при e.ext = **nil** не будет создано дополнительных вершин; в этом случае будем считать last = NIL. С учетом последнего соглашения выполняется ограничение sufLinks(T, f, last).

Вершина last должна быть параметром предиката, чтобы в определении предиката можно было сформировать суффиксную ссылку для last. Наиболее простая реализация — определить второй предикат mccL, содержащий те же параметры, что и mcc, а также параметр last. В этом случае определение mccL будет содержать вызовы mcc, а определение mcc — вызовы mccL. Однако рекурсию такого рода нельзя преобразовать в цикл. Чтобы избежать подобную ситуацию и получить определение с хвостовой формой рекурсии, необходимо интегрировать mcc и mccL в одном предикате mcc.

В вызове mcc внутри определения SufTree_mcc третий параметр last = NIL. Точная спецификация mcc дана ниже в определении mcc.

```
SufTree_mcc( : V j, ST(j) T) ≡ T=sufTree(t)
{     newName(V(): root, V j0); // j0={root}
      mcc(j0, ST(root: NodeAtrs(head(t): Arc(t, NIL))), NIL, V(), SUFL(),
                 Pair(root, nil), t: j, T)
}
```

Δ Из истинности newName(V(): root, V j0) следует j0 = {root}. Проверим ограничения вызова mcc. Параметры вызова mcc определяют следующие соотношения для аргументов mcc: j = {root}, дерево T состоит из единственного суффикса t, т.е. paths(P) = {t}, last = NIL, i = ∅, f — пустой

массив, e = (root, **nil**), v = t. Поскольку str(e, T) = **nil**, то w = t. Легко проверить, что все ограничения mcc истинны. Тогда истинна спецификация вызова T = sufTree(t), что совпадает со спецификацией SufTree_mcc. □

Определим спецификации предикатов, вызываемых в теле определения mcc. Сканирование дерева T для нахождения древесной пары d головной части очередного включаемого суффикса реализуется предикатом

scan(ST(V j) T, Node b, Str v: Pair d, Str v′)

с ограничениями b ∈ j & guard(v) & T = pTree(path(b, T)+v) и спецификацией

∃Str x (v = x + v′ & s = path(b, T) + x & sufParts(s, v′) &
str(d, T) = s) &
pair(d, T).

Для предиката scan строка path(b, T) + v называется *сканируемым суффиксом*. Нахождение суффиксной ссылки для древесной пары a реализуется предикатом link(ST(V j) T, SUFL(V i) f, Pair a*:) с ограничениями b = a.locus & b∈j & b ≠ last & a.ext ≠ **nil** & sufLinks(T, f, last) & s = str(a, T) & occ(tail(s)) и спецификацией str(a′, T) = tail(s) & pair(a′, T). Включение в дерево T нового суффикса, определяемого древесной парой a для головной части и строкой v для хвостовой части, определяется предикатом

Add(V j*, ST(j) T, Pair a, Str v: ST(j′) T′, Node new)

с ограничениями pair(a, T) & s = str(a, T) & suf(s + v, t) & sufParts(s, v) и спецификацией

Paths(T′) = Paths(T) ∪ {s + v} & **if** a.ext = **nil** **then** j′ = j & new = NIL **else**
j′ = j ∪ {new} & path(new,T′) = s **end**.

```
mcc(V j*, ST(j) T, Node last, SUFL(V i) f, Pair e, Str v: ST(j') T') ≡
  w=str(e, T)+v & suf(w, t) & guard(w) & T=pTree(tail(w)) & sufLinks(T, f, last) &
  pair(e, pTree(w)) & if e.ext=nil then last=NIL else
                                  last∈j & path(last, T)=str(e, T) end
            ⇒    T'=sufTree(t)
{     e -> (Node b, Str u);
      if length(v)=1 & b=root & u=nil then
            j, T
      elsif u = nil then
            if b=root then
                  scan(T, b, tail(v): Pair d, Str z)
            else  scan(T, f[b], v: Pair d, Str z)
            end ;
            mcc(Add(j, T, d, z), i, f, d, z)
      else
            link(T, f, e: Pair a);
            if a.ext = nil then
                  scan(T, a.locus, v: Pair d, Str z);
                  mcc(Add(j, T, d, z), i+last, f+(last: a.locus), d, z)
            else  Add(j, T, a, v: V k, ST(k) U, Node new);
                  mcc(k, U, new, i+last, f+(last: new), a, v)
            end
      end
}
```

Δ Допустим, что истинно первое условие в условном операторе. Строка v состоит из одного символа, поскольку length(v) = 1. Из b = root & u = **nil** следует str(e, T) = **nil**. Ограничение w = str(e, T) + v переписывается в виде w = v. Следовательно, tail(w) = **nil**. В соответствии с леммой 3 из ограничения T = pTree(tail(w)) следует T = sufTree(t). Предложение j, T после **then** эквивалентно j′ = j & T′ = T, откуда следует T′ = sufTree(t), что совпадает со спецификацией mcc.

Допустим, что первое условие ложно и истинно второе условие u = **nil** в условном операторе. Рассмотрим вложенный условный оператор при истинном условии b = root. Имеем e = (root, **nil**) и str(e, T) = **nil**. Из ограничения w = str(e, T) + v следует w = v. Из ограничения guard(w) и w = v следует v ≠ **nil**. Из ложности length(v) = 1 & b = root & u = **nil** и истинности второго и третьего конъюнктов следует length(v) ≠ 1. Из v ≠ **nil** и length(v) ≠ 1 следует tail(v) ≠ **nil**. Поскольку tail(v) — суффикс t, реализуется guard(tail(v)) — второе ограничение вызова scan; первое ограничение root ∈ j также истинно. Отметим, что path(b) + tail(v) = tail(w) есть сканируемый суффикс в дан-

ном вызове scan, удовлетворяющий третьему ограничению scan. Допустим, что во вложенном условном операторе условие b = root ложно. Тогда вызов предиката scan(T, f[b], v: Pair d, Str z) является истинным. Поскольку last = NIL, $b \neq$ root и $b \in j$, то (из ограничения sufLinks(T, f, last)) $b \in i$, f[b] определено и path(f[b], T) = tail(path(b, T)). По лемме 7, $f[b] \in j$, и следовательно, первое ограничение вызова scan верно, второе также выполняется. Сканируемым суффиксом является строка

$$\mathrm{path}(f[b], T) + v = \mathrm{tail}(\mathrm{path}(b, T)) + v = \mathrm{tail}(w).$$

Итак, во вложенном условном операторе оба вызова scan имеют один и тот же сканируемый суффикс tail(w). Из спецификации scan следует: str(d, T) является головной частью суффикса tail(w), а z — хвостовой частью. Вызов mcc(Add(j, T, d, z), i, f, d, z) эквивалентен: Add(j, T, d, z: j_0, T_0, new_0); mcc(j_0, T_0, new_0, i, f, d, z: j′, T′). Очевидно, выполняются ограничения для вызова Add. Поэтому справедлива спецификация Add. Из первого конъюнкта спецификации Paths(T_0) = Paths(T) $\cup$ {tail(w)} следует T_0 = pTree(tail(tail(w))). Покажем, что выполняются ограничения вызова mcc. Обозначим w_0 = str(d, T) + z = tail(w). Очевидно, выполняется suf(w_0, t). Строка $z \neq$ **nil**, поскольку z — хвостовая часть. Поэтому $w_0 \neq$ **nil** и выполняется guard(w_0). Истинно T_0 = pTree(tail(w_0)). Ограничение pair(d, pTree(w_0)) следует из pair(d, T). Ограничение **if** d.ext $\neq$ **nil** **then** new_0 = NIL **else** $new_0 \in j_0$ & path(new_0, T_0) = str(d, T_0)) **end** следует из второго конъюнкта спецификации Add. Проверим первый конъюнкт ограничения sufLinks(T_0, f, new_0). Поскольку last = NIL, то i = j\{root}. При new_0 = **nil** из второго конъюнкта Add следует j_0 = j, и тогда i = j_0\{root}. При $new_0 \neq$ **nil** верно i = j_0\{root, new_0} ввиду $j_0 = j \cup \{new_0\}$. Второй конъюнкт sufLinks(T_0, f, new_0) совпадает со вторым конъюнктом sufLinks(T, f, last). Итак, ограничения вызова mcc верны. Поскольку суффикс w_0 короче w, а для суффикса w длины 1 спецификация mcc доказана, можно применить индуктивное предположение об истинности спецификации вызова, т.е. T′ = sufTree(t), что доказывает спецификацию исходного определения mcc.

Далее предположим, что нарушаются условия length(v) = 1 & b = root & u = **nil** и u = **nil** в условном операторе. Тогда истинна альтернатива после **else**. Поскольку e.ext $\neq$ **nil**, то из последнего ограничения mcc следует: $last \in j$, last $\neq$ NIL и path(last, T) = str(e, T). Из ограничения sufLinks(T, f, last) следует i = j\{root, last}. Справедливость ограничения $b \neq$ last вызова link(T, f, e: Pair a) следует из path(last, T) = str(e, T) и e.ext $\neq$ **nil**; остальные ограничения очевидны. Тогда реализуется спецификация link: str(a, T) = tail(str(e, T)) & pair(a, T). Предположим, выполняется

условие a.ext = **nil** вложенного условного оператора. Истинность ограничений вызова scan(T, a.locus, v: Pair d, Str z) очевидна. Поскольку tail(w) = str(a, T) + v = path(a.locus, T) + v, то tail(w) — сканируемый суффикс, для которого str(d, T) — головная часть, а z — хвостовая часть в соответствии со спецификацией scan. Результаты вызова Add обозначим j_0, T_0, new_0. Выполняются ограничения вызова Add и поэтому справедлива спецификация Add. Проверим ограничения вызова mcc. Для всех ограничений, кроме пятого и седьмого, проверка реализуется так же, как и для первого рассмотренного выше вызова mcc. Операции i + last и f + (last: a.locus) являются корректными, поскольку $last \notin i$. Обозначим $i_0 = i \cup \{last\}$, $f_0 = f + (last: a.locus)$. Отметим, что $f_0[last] = a.locus$. Проверим ограничение sufLinks(T_0, f_0, new_0). Допустим, d.ext = **nil**. Тогда из второго конъюнкта спецификации Add следует $j_0 = j$ и new_0 = NIL Соотношение $i_0 = j_0 \backslash \{root\}$, определяемое первым конъюнктом sufLinks(T_0, f_0, new_0), справедливо, поскольку оно эквивалентно $i \cup \{last\} = j \backslash \{root\}$. Второй конъюнкт sufLinks(T_0, f_0, new_0) достаточно проверить лишь для k = last. Соотношение path(f_0[last], T_0) = tail(path(last, T_0)) справедливо, поскольку f_0[last] = a.locus, path(last, T_0) = path(last, T) = str(e, T), str(a, T) = tail(str(e, T)), и str(a, T) = path(a.locus, T) = path(a.locus, T_0). Допустим теперь, что d.ext ≠ **nil**. Из второго конъюнкта спецификации Add следует $j_0 = j \cup \{new_0\}$, $new_0 \neq$ NIL и path(new_0, T_0) = str(a, T_0). Соотношение $i_0 = j_0 \backslash \{new_0, root\}$ в else-альтернативе первого конъюнкта sufLinks(T_0, f_0, new_0) эквивалентно $i \cup \{last\} = j \backslash \{root\}$ (или $i = j \backslash \{root, last\}$), что следует из равенств

$$j_0 \backslash \{new_0, root\} = j \cup \{new_0\} \backslash \{new_0, root\} = j \backslash \{root\} \text{ и } i_0 = i \cup \{last\}.$$

Истинность второго конъюнкта доказывается так же, как при d.ext = **nil**. Последнее (седьмое) ограничение вызова mcc очевидным образом вытекает из спецификации Add. Концовка доказательства проводится аналогично, как для первого вызова mcc.

Рассмотрим случай, когда нарушается условие a.ext = **nil**, т.е. a.ext ≠ **nil**. Поскольку для строки str(a, T) нет локуса в дереве T, то в соответствии с леммой 6 строка str(a, T) является головной частью суффикса tail(w), а строка v — хвостовой. И следовательно, не нужно проводить сканирование суффикса, как это было в предыдущих трех случаях. Истинность ограничений вызова Add очевидна. Поскольку a.ext ≠ **nil**, из Add спецификации следует $k = j \cup \{new\}$ и path(new, U) = str(a, T). Проверку ограничений вызова mcc рассмотрим лишь для пятого и седьмого ограничений. Обозначим $i_0 = i \cup \{last\}$, $f_0 = f + (last: new)$. Отметим, что $f_0[last] = new$. Поскольку new ≠ NIL, для доказательства первого конъюнкта спецификации

sufLinks(U, f_0, new) достаточно проверить, что i_0 = k\{root, new}. Это соотношение сводится к истинному i ∪ {last} = j\{root} ввиду k = j ∪ {new} и i_0 = i ∪ {last}. Для полного доказательства достаточно проверить path(f_0[last], U) = tail(path(last, U)). Данное соотношение вытекает из следующей цепочки равенств: path(f_0[last], U) = path(new, U) = str(a, T) = tail(sta(e, T)) = tail(path(last, T)) = tail(path(last, U)). Наконец, из истинности утверждения new∈k & path(new, U) = str(a, U) вытекает правильность седьмого ограничения вызова mcc. Остальная часть доказательства проводится аналогично второму вызову mcc. □

В определении scan используются гиперфункция diff и функция headForTail, введенные выше для простой версии алгоритма. Для доступа к дуге вершины по первому символу используется гиперфункция

getArc(NodeAtrs(AA s) v, A a: | Arc L)

со спецификацией a∈s ⇒ L = v[a]; #1: a∉s. Для определения места несовпадения строк g и z используется гиперфункция

diff(Str g, z: Str g′, z′ | Str z′)

с ограничением ¬pref(z, g) и спецификацией ∃y∈Str (g = y + g′ & z = y + z′ & (g′ = **nil** ∨ head(g′) ≠ head(z′))); #1: ¬pref(g, z). Отметим, что ¬pref(g, z) ≡ g′ ≠ **nil**. Функция z = headForTail(Str x, y) определяется ограничением suf(y, x) и спецификацией z + y = x.

```
scan(ST(V j) T, Node b, Str v: Pair d, Str v′) ≡
    b∈j & guard(v) & T=pTree(path(b, T)+v) ⇒
    ∃Str x (v=x+v′ & s=path(b, T)+x & sufParts(s, v′) & str(d, T)=s) & pair(d, T)
{      split getArc(T[b], head(v): | Arc L)
       do   (b, nil), v
       do   split diff(tail(L.lab), tail(v): Str g, z | Str z)
             do   d = (b, headForTail(L.lab, g)) || v′=z
             do   scan(j, T, L.to, z)
             end
       end
}
```

Δ Поскольку v ≠ **nil**, функции head(v) и tail(v) определены. Допустим, реализуется первая альтернатива внешнего оператора расщепления, и тогда head(v)∉ind(T[b]), d = (b, **nil**) и v′ = v. Положим x = **nil**. Тогда path(b, T) + x + head(v′) = path(b, T) + head(v). Утверждение occ(path(b, T) + head(v)) ложно, поскольку не существует дуги вершины b, начинающейся с head(v). В соответствии с леммой 4 истинно sufParts(path(b, T) + x, v′). Нетрудно проверить истинность других конъюнктов спецификации scan. Допустим,

имеет место вторая альтернатива расщепления, т.е. верны head(v)∈ind(T[b]) и L = T[b][head(v)]. Из условий на типы Arc и NodeAtrs имеем: L.lab ≠ **nil** и head(L.lab) = head(v). Следовательно, tail(L.lab) определено. Проверим ограничения для вызова diff. Допустим, что tail(v) = **nil**, из чего следует, что строка path(b, T) + v встретилась в T, что противоречит ограничению T = pTree(path(b, T) + v). Следовательно, tail(v) ≠ **nil**. Допустим, что нарушается ограничение diff: ¬pref(tail(v), tail(L.lab)), из чего будет следовать pref(v, L.lab), и тогда v будет встречаться в T, что противоречит T = pTree(path(b, T) + v). Итак, ограничения diff выполняются. Допустим, имеет место первая альтернатива внутреннего оператора расщепления. Тогда g ≠ **nil**, d = (b, headForTail(L.lab, g)) и v′ = z. Для некоторой строки y выполняются: L.lab = y + g, v = y + v′, head(g) ≠ head(v′). Очевидно, что y = headForTail(L.lab, g). Положим x = y и проверим истинность всех конъюнктов спецификации scan. Строка path(b, T) + x + head(v′) не может встречаться в T. Иначе должен существовать локус строки path(b, T) + x, а это невозможно ввиду L.lab = x + g при g ≠ **nil**. Истинность других конъюнктов очевидна.

Допустим, выполняется вторая альтернатива diff. Тогда v = L.lab + z. Проверим ограничения для параметров рекурсивного вызова scan. Невозможно v′ = **nil**, поскольку в этом случае path(b, T) + v встречается в T и будет нарушено T = pTree(path(b, T) + v). Тогда из guard(v) и v′ ≠ **nil** следует guard(v′). Ограничение L.to∈j следует из rootedTree(T, root). Ограничение T = pTree(path(L.to, T) + v′) следует из path(L.to, T) = path(b, T) + L.lab. Для рекурсивного вызова scan число дуг от вершины b до листьев на 1 меньше, а для числа дуг = 1 спецификация scan доказана. Поэтому применимо индуктивное предположение истинности спецификации вызова scan(j, T, L.to, z). Допустим, спецификация вызова истинна для некоторой строки x_0, т.е. верны $z = x_0 + v'$, sufParts(path(L.to, T) + x_0, v′), path(L.to, T) + x_0 = str(d, T)) и pair(d, T). Тогда v = L.lab + z = (L.lab + x_0) + v′. Положим x = L.lab + x_0. Поскольку path(L.to, T) = path(b, T) + L.lab, нетрудно убедиться в истинности всех конъюнктов спецификации исходного определения scan. □

Предикат link определяет суффиксную ссылку для произвольной древесной пары a = (b, u) в дереве T. При b ≠ root равенство

tail(path(b, T) + u) = path(f[b], T) + u

вытекает из условия sufLinks. Строка str((f[b], u), T) встречается в T, однако (f[b], u) не всегда является правильной древесной парой. Суффиксная ссылка древесной пары (root, u) соответствует строке tail(u), однако (root, tail(u))

не обязательно является правильной древесной парой. Например, суффиксная ссылка древесной пары (root, “cg”) в дереве суффиксов для строки agcgacgagd (рис. 2) есть древесная пара ($\overline{g}$, **nil**), а не (root, “g”).

Для приведения (f[b], u) и (root, tail(u)) к эквивалентной древесной паре в определении link используется предикат toPair(ST(V j) T, Pair a*:) с ограничениями a.locus ∈ j & a.ext ≠ **nil** & s = str(a, T) & occ(s, T) и спецификацией s = str(a’, T) & pair(a’, T).

```
link(ST(V j) T, SUFL(V i) f, Pair a*: ) ≡
    b=a.locus & b∈j & b≠last & a.ext≠nil & sufLinks(T, f, last) & s=str(a, T) &
    occ(tail(s))   ⇒     str(a′, T)=tail(s) & pair(a′, T)
{      a -> (Node b, Str u);
       if b=root then  Str u1=tail(u);
                       if u1=nil then (b, u1) #1 else Pair a1=(b, u1) end
       else Pair a1=(f[b], u)
       end;
       a′ = toPair(a1)
}
```

Δ tail(u) определено, поскольку u ≠ **nil**; f[b] определено ввиду b ≠ root и b ≠ last. При u1 = **nil** результат a′ = (b, u1) удовлетворяет спецификации link. Нетрудно проверить, что str(a1, T) = tail(str(a, T)). Для a1 выполняются ограничения toPair. Тогда истинна спецификация toPair, т.е. str(a1, T) = str(a′, T) и pair(a′, T). Очевидно, выполняется спецификация link. □

В определении предиката toPair используется функция z = tailForHead(Str x, y) с ограничением pref(y, x) и спецификацией y + z = x..

```
toPair(ST(V j) T, Pair a*: ) ≡
        a.locus∈j & a.ext≠nil & s=str(a, T) & occ(s, T) ⇒ s=str(a′, T) & pair(a′, T)
{      a -> (Node b, Str u);
       getArc(T[b], head(u): ## | Arc L);
       nat k = length(L.lab); nat p = length(u);
       if k > p then
             a
       elsif k = p then
             (L.to, nil)
       else        toPair((L.to, tailForHead(u, L.lab)))
       end
}
```

Δ Параметры T[b], head(u) вызова getArc — определены. Поэтому истинна спецификация getArc. Условие head(u)∈ind(T[b]) следует из истинности occ(str((b, u), T). Поэтому ложно условие на первую ветвь гиперфункции getArc; следовательно, использование ## корректно. Из спецификации

getArc следует L = T[b][head(u)], а из условия на тип NodeArcs — head(L.lab) = head(u).

Для строк L.lab и u существуют строки x, y и z такие, что L.lab = x + y и u = x + z. Если y ≠ **nil** и z ≠ **nil**, то для вершины b должна существовать дуга с меткой x, что невозможно. Поэтому либо u является частью L.lab, либо L.lab является частью u, либо u = L.lab. Распознать, какая из трех альтернатив реализуется, можно сравнивая длины L.lab и u соответственно k и p. Допустим, истинно первое условие k > p вложенного условного оператора. Тогда a′ = a. В этом случае строка u является частью L.lab, и поэтому истинно pair(a′, T). Следовательно, верна спецификация toPair. Допустим, что нарушается условие k > p и истинно второе условие k = p в условном операторе. Тогда a′ = (L.to, **nil**). Поскольку path(L.to, T) = path(b, T) + L.lab, то истинна спецификация toPair. Допустим теперь, что нарушаются условие k > p и k = p во вложенном условном операторе, и тогда k > p, L.lab является частью u и истинен вызов toPair(L.to, tailForHead(u, L.lab)). Выполняется ограничение pref(L.lab, u) для функции tailForHead. Обозначим b_0 = L.to и u_0 = tailForHead(u, L.lab). Тогда L.lab + u_0 = u, str((b_0, u_0), T) = path(L.to, T) + u_0 = path(b, T) + L.lab + u_0 = path(b, T) + u = str(a, T). Кроме того, u_0 ≠ **nil**. Итак, выполняются ограничения рекурсивного вызова toPair. Поскольку строка u_0 короче u, а для строки u длины 1 спецификация toPair доказана, допустимо индуктивное предположение истинности спецификаций вызова, т.е. str((b_0, u_0), T) = str(a′, T) и pair(a′, T), откуда следует спецификация исходного определения toPair. □

```
Add(V j*, ST(j) T, Pair a, Str v: ST(j') T', Node new) ≡
        pair(a, T) & s=str(a, T) & suf(s+v, t) &  sufParts(s, v) ⇒
        Paths(T') = Paths(T)∪{s+v} &
        if a.ext=nil then j'=j & new=NIL else j'=j∪{new} & path(new,T')=s end.
{   a -> (Node b, Str u);
    if u = nil then           j, T(b: T[b]+(head(v): (v, NIL))), NIL
    else    {       A c = head(u);
                    Arc L = T[b][c];
                    Str z = tailForHead(L.lab, u) ||
                    newName(j: new, j')
            };
            T' = T(b: T[b](c: (u, new))) +
                                    (new: (head(z): (z, L.to), head(v): (v, NIL)))
    end
}
```

Δ Допустим, истинно u = **nil** условие в условном операторе. Тогда j′ = j, T′ = T(b: T[b] + (head(v): (v, NIL))), new = NIL. Из pair(a, T) следует b∈j, и

поэтому T[b] и замещение T(b: ...) являются корректными. Далее, head(v)∉ind(T[b]), иначе строка str(a, T) + head(v) будет встречаться в T, и тогда str(a, T) не является головной частью. Следовательно, операция head(v): (v, NIL) корректна; она также удовлетворяет условиям на типы Arc и NodeArcs, поскольку хвостовая часть v ≠ **nil**. Итак, дерево T′ корректно и Paths(T′) = Paths(T) ∪ {path(b, T) + v}, что доказывает истинность первого конъюнкта спецификации Add. Истинность второго конъюнкта очевидна.

Допустим, ложно условие u = **nil** в условном операторе, и тогда истинны следующие соотношения: u ≠ **nil**, c = head(u), L = T[b][c], u + z = L.lab, new∉j, j′ = j ∪ {new}, new ≠ NIL. Отметим, что head(u) и T[b][c] определены. Операция замещения T(b: ...) корректна, поскольку b∈j. Выполняются условия на типы Arc и NodeArcs для c: (u, new). Операция замещения T[b](c: (u, new)) корректна, поскольку c = head(u)∈ind(T[b]). Операция "+" корректна ввиду new∉j. Нетрудно проверить ограничения на другие операции, используемые при построении вершины new.

Дуга L заменена в дереве T′ парой дуг c: (u, new) и head(z): (z, L.to), и поэтому любой путь в дереве T, проходящий через вершины b и L.to, входит в число путей дерева T′ и наоборот. Таким образом, дерево T′ содержит все пути дерева T и дополнительный путь {str(a, T) + v}, что доказывает истинность первого конъюнкта спецификации Add. Наконец, path(new, T′) = path(b, T) + u = str(a, T), что определяет истинность второго конъюнкта и спецификации Add в целом. □

4. ОПТИМИЗИРУЮЩИЕ ПРЕОБРАЗОВАНИЯ

Неочевидной является реализация доступа к дуге вершины по первому символу метки. Этот доступ осуществляется гиперфункцией getArc(v, a: | L) и операциями v + (a: **L**) и v(a: **L**). Непосредственная реализация типа NodeAtrs массивом **array** A **of** Arc, безусловно, обеспечит наиболее быстрый доступ к дуге. Однако такая реализация расточительна по памяти и поэтому может оказаться неприемлемой. Реализация типа NodeAtrs с разной степенью компактности является отдельной задачей [12, 14], которая здесь не рассматривается.

Недостаток предикатной программы заключается в том, что доступ к одной и той же дуге реализуется дважды: сначала через getArc в теле scan или toPair, а затем в Add в операциях T[b] + (head(v): (v, NIL)) и T[b](c: (u, new)). Быстрый повторный доступ к дуге можно реализовать с помощью хэш-функции. Например, можно использовать гиперфункцию getArc(NodeAtrs(AA s) v, A a: H h| Arc L, H h), где h — некоторый аналог ре-

зультата хэш-функции; операции v+(a: **L**) и v(a: **L**) заменяются соответственно на предикаты addArc(v, h, L : v′) и updateArc(v, h, L : v′).

Непосредственная модификация предикатной программы потребует введения дополнительного параметра h в предикаты scan, link, toPair и Add. Тем не менее, более результативным является другое решение — перенести каждый из вызовов предиката Add ближе к соответствующему вызову getArc. Далее мы покажем технику перемещения вызовов Add, пока без вставки параметра h. Фрагмент определения mcc:

```
scan(T, a.locus, v: Pair d, Str z);
mcc(Add(j, T, d, z), i+last, f+(last: a.locus), d, z)
```

заменим на

```
scan(T, a.locus, v: Pair d, Str z);
Add(j, T, d, z: V k, ST(k) U, Name new);
mcc(k, U, new, i+last, f+(last: a.locus), d, z)
```

Определим новый предикат scanAdd следующим образом:

```
scanAdd(V j, ST(j) T, Node b, Str v: Pair d, Str v′, V j′, ST(j′) T′, Node new) ≡
        scan(T, b, v: d, v′) & Add(j, T, d, v′: j′, T′, new);
```

Тогда указанный выше фрагмент mcc может быть заменен на

```
scanAdd(j, T, a.locus, v: Pair d, Str z, V k, ST(k) U, Name new);
mcc(k, U, new, i+last, f+(last: a.locus), d, z)
```

Построим определение предиката scanAdd. Оно имеет следующую структуру:

```
scanAdd(V j, ST(j) T, Node b, Str v: Pair d, Str v′, V j′, ST(j′) T′, Node new) ≡
        scan(T, b, v: d, v′) & Add(j, T, d, v′: j′, T′, new)
{  <тело определения scan>; Add(j, T, d, v′: j′, T′, new)  }
```

Внесем вызов Add в конец каждой ветви определения scan.

```
scanAdd(V j, ST(j) T, Node b, Str v: Pair d, Str v′, V j′, ST(j′) T′, Node new) ≡
        scan(T, b, v: d, v′) & Add(j, T, d, v′: j′, T′, new)
{    split getArc(T[b], head(v): | Arc L)
     do  {d=(b, nil) || v′=v}; Add(j, T, d, v: j′, T′, new)
     do  split diff(tail(L.lab), tail(v): Str g, z | Str z)
         do  {d = (b, headForTail(L.lab, g)) || v′=z}; Add(j, T, d, z: j′, T′, new)
         do  scan(j, T, L.to, z: d, v′); Add(j, T, d, v′: j′, T′, new)
         end
     end
}
```

Введем следующие две специализации предиката Add:

```
AddExtNil(V j*, ST(j) T, Pair a, Str v: ST(j’) T’, Node new) ≡
                Add(j, T, a, v: j’, T’, new) & a.ext=nil;
AddNode(V j*, ST(j) T, Pair a, Str v: ST(j’) T’, Node new) ≡
                Add(j, T, a, v: j’, T’, new) & a.ext≠nil;
```

Определения этих предикатов получаются специализацией определения предиката Add.

```
AddExtNil(V j*, ST(j) T, Pair a, Str v: ST(j’) T’, Node new) ≡
                Add(j, T, a, v: j’, T’, new) & a.ext=nil
{     Node b=a.locus;
      j, T(b: T[b]+(head(v): (v, NIL))), NIL
}
AddNode(V j*, ST(j) T, Pair a, Str v: ST(j’) T’, Node new) ≡
                                        Add(j, T, a, v: j’, T’, new) & a.ext≠nil
{     a -> (Node b, Str u);
    { A c = head(u);
    Arc L = T[b][c];
    Str z = tailForHead(L.lab, u) ||
    newName(j: new, j’)
    };
    T’ = T(b: T[b](c: (u, new))) + (new: (head(z): (z, L.to), head(v): (v, NIL)))
}
```

В получившемся тексте определения scanAdd вызов Add на первой ветви можно заменить на AddExtNil, вызов Add на второй ветви — на AddNode, композицию на третьей ветви scan(j, T, L.to, z: d, v’); Add(j, T, d, v’: j’, T’, new) в соответствии с определением scanAdd можно заменить на scanAdd(j, T, L.to, z: d, v’, j’, T’, new). Кроме того, подставим определение AddExtNil на место вызова. Получим:

```
scanAdd(V j, ST(j) T, Node b, Str v: Pair d, Str v’, V j’, ST(j’) T’, Node new) ≡
    scan(T, b, v: d, v’) & Add(j, T, d, v’: j’, T’, new)
{  split getArc(T[b], head(v): | Arc L)
   do d=(b, nil) || v’=v || j’=j || T’=T(b: T[b]+(head(v): (v, NIL))) || new=NIL
   do      split diff(tail(L.lab), tail(v): Str g, z | Str z)
           do    {d = (b, headForTail(L.lab, g)) || v’=z};
                 AddNode(j, T, d, z: j’, T’, new)
           do    scanAdd(j, T, L.to, z: d, v’, j’, T’, new)
           end
   end
}
```

Рассмотрим задачу перемещения вызова

Add(j, T, a, v: V k, ST(k) U, Node new)

внутрь определений предикатов link и toPair для следующего фрагмента определения mcc:

```
link(T, f, e: Pair a);
if a.ext = nil then
        scan(T, a.locus, v: Pair d, Str z);
        mcc(Add(j, T, d, z), i+last, f+(last: a.locus), d, z)
else    Add(j, T, a, v: V k, ST(k) U, Node new);
        mcc(k, U, new, i+last, f+(last: new), a, v)
end
```

Отметим, что рассматриваемый вызов можно заменить вызовом AddNode ввиду a.ext ≠ **nil**. Поскольку вызов перемещается только с ветви **else**, необходимо сначала заменить link эквивалентным представлением в виде гиперфункции. Определим:

```
linkH(ST(V j) T, SUFL(V i) f, Pair a: Pair a' | Pair a') ≡
        link(T, f, a: a'); #1: a'.ext = nil;
```

Определение linkH может быть получено из определения link включением в конце его оператора: **if** a'.ext=**nil then** a #1 **else** a #2 **end**. Заменим приведенный фрагмент mcc эквивалентным оператором расщепления:

```
split linkH(T, f, e: Pair a | Pair a)
do      scan(T, a.locus, v: Pair d, Str z);
        mcc(Add(j, T, d, z), i+last, f+(last: a.locus), d, z)
do      AddNode(j, T, a, v: V k, ST(k) U, Node new);
        mcc(k, U, new, i+last, f+(last: new), a, v)
end
```

Теперь можно присоединить вызов

AddNode(j, T, a, v: V k, ST(k) U, Node new)

ко второй ветви вызова linkH. Композицию такого вида в форме оператора расщепления обозначим следующей гиперфункцией:

```
linkAdd(ST(V j) T, SUFL(V i) f, Pair a, Str v:
                        Pair a' | Pair a', V j', ST(j') T', Node new) ≡
{   linkH(T, f, a: a' #1 | a'); AddNode (j, T, a', v: j', T', new) #2   }
```

Внесем вызов AddNode на вторую ветвь определения linkH. Получим:

```
linkAdd(ST(V j) T, SUFL(V i) f, Pair a, Str v:
                        Pair a' | Pair a', V j', ST(j') T', Node new)
```

```
{      a -> (Node b, Str u);
       if b=root then  Str u1=tail(u);
                       if u1=nil then (b, u1) #1 else Pair a1=(b, u1) end
       else        Pair a1=(f[b], u)
       end;
       a′ = toPair(a1)
       if a′.ext=nil then #1 else AddNode(j, T, a′, v: j′, T′, new) #2 end
}
```

Перенос вызова AddNode внутрь определения toPair проводится аналогичным образом. Определим:

```
toPairH(ST(V j) T, Pair a: Pair a′ |  Pair a′) ≡
                                   toPair(T, f, a: a′); #1: a′.ext = nil;
toPairAdd(ST(V j) T,  Pair a, Str v: Pair a′ | Pair a′, V j′, ST(j′) T′, Node new) ≡
{      toPairH(T, a: a′ #1| a′); AddNode(j, T, a′, v: j′, T′, new) #2        }
```

В соответствии с определением toPairAdd определение linkAdd переписывается следующим образом:

```
linkAdd(ST(V j) T, SUFL(V i) f, Pair a, Str v:
                                        Pair a′ | Pair a′, V j′, ST(j′) T′, Node new)
{   a -> (Node b, Str u);
    if b=root then     Str u1=tail(u);
                       if u1=nil then (b, u1) #1 else Pair a1=(b, u1) end
    else    Pair a1=(f[b], u)
    end;
    toPairAdd(T, a1, v: a′ #1 | a′ , j′, T′, new #2)
}
```

Определение toPairH получается из определения toPair вставкой указателя ветви #1 в конце ветвей для условия k=p, а также указателя ветви #2 в конце ветви по условию k>p. Далее, определение toPairAdd получается из определения toPairH вставкой вызова AddNode(j, T, a′, v: **j′**, T′, new) в конце ветвей с указателем #2:

```
toPairAdd(ST(V j) T,  Pair a, Str v: Pair a′ | Pair a′, V j′, ST(j′) T′, Node new)
{      a -> (Node b, Str u);
       getArc(T[b], head(u): ## | Arc L);
       nat k = length(L.lab); nat p = length(u);
       if k > p then
              a′=a || AddNode(j, T, a, v: j′, T′, new) #2
       elsif k = p then
              (L.to, nil) #1
       else toPairH((L.to, tailForHead(u, L.lab)): a′ #1|a′);
              AddNode(j, T, a′, v: j′, T′, new)
              #2
       end
}
```

В соответствии с определением toPairAdd композиция на ветви **else** может быть заменена оператором:

```
toPairAdd(T, (L.to, tailForHead(u, L.lab)), v: a′ #1 | a′ , j′, T′, new #2)
```

Возможно еще одно улучшение. Параметр — древесную пару a′ на первой ветви предикатов linkAdd и toPairAdd — можно заменить вершиной r, соответствующей a′.locus, поскольку вторая компонента a′.ext не используется в теле mcc.

Наконец, приведем изменившееся определение mcc:

```
mcc(V j*, ST(j) T, Node last, SUFL(V i) f, Pair e, Str v: ST(j') T') ≡

  w=str(e, T)+v & suf(w, t) & guard(w) & T=pTree(tail(w)) & sufLinks(T, f, last) &
  pair(e, pTree(w)) & if e.ext=nil then last=NIL else
                                              last∈j & path(last, T)=str(e, T) end
                    ⇒             T'=sufTree(t)
{     e -> (Node b, Str u);
      if u = nil then
            if b=root then
               if length(v)=1 then
                     j, T #1
               else  scanAdd(j, T, b, tail(v): Pair d, Str z, V k, ST(k) U, Node new)
               end
            else   scanAdd(j, T, f[b], v: Pair d, Str z, V k, ST(k) U, Node new)
            end ;
            mcc(k, U, new, i, f, d, z)
      else
            split linkAdd(j, T, f, e, v: Node r | Pair a, V k, ST(k) U, Node new)
            do    SUFL(i+last) f1 = f+(last: r);
                  scanAdd(j, T, r, v: Pair d, Str z, V k, ST(k) U, Node new1);
                  mcc(k, U, new1, i+last, f1, d, z)
            do    mcc(k, U, new, i+last, f+(last: new), a, v)
            end
      end
}
```

Кроме изменений, обусловленных перемещением вызовов Add, проведено также эквивалентное преобразование структуры условных операторов. Появившийся в результате этого указатель ветви #1 эквивалентен оператору **return**. Вынесение вычисления SUFL(i+last) f1 = f+(last: a.node) из вызова mcc будет объяснено позже.

В получившейся программе рассмотрим определения предикатов AddNode, scanAdd и toPairAdd. Обнаруживается, что дуга L вычисляется дважды: в getArc и AddNode. Далее, легко проверить, что строка tailForHead(L.lab, u), вычисляемая в AddNode, есть в точности строка g, получаемая как результат diff в теле scanAdd. Введение строки g в качестве дополнительного параметра предиката AddNode ускорит программу. Кроме того, вторым дополнительным параметром целесообразно передавать вершину L.to, а не дугу L, поскольку других вхождение L в AddNode нет. Приведем модифицированные версии определений предикатов AddNode, scanAdd и toPairAdd:

```
AddNode(V j*, ST(j) T, Pair a, Str v, g, Node lto: ST(j') T', Node new)
{     {     a -> (Node b, Str u) || newName(j: new, j')      };
      T' = T(b: T[b](head(u): (u, new))) +
                                     (new: (head(g): (g, lto), head(v): (v, NIL)))
}
scanAdd(V j, ST(j) T, Node b, Str v: Pair d, Str v', V j', ST(j') T', Node new)
{   split getArc(T[b], head(v): | Arc L)
    do  d=(b, nil) || v'=v || j'=j || T'=T(b: T[b]+(head(v): (v, NIL))) || new=NIL
    do  split diff(tail(L.lab), tail(v): Str g, z | Str z)
          do     {d = (b, headForTail(L.lab, g)) || v'=z};
                 AddNode(j, T, d, z, g, L.to: j', T', new)
          do     scanAdd(j, T, L.to, z: d, v', j', T', new)
          end
    end
}
toPairAdd(ST(V j) T,  Pair a, Str v: Node r | Pair a', V j', ST(j') T', Node new)
{     a -> (b, u);
      getArc(T[b], head(u): ## | Arc L);
      nat k = length(L.lab); nat p = length(u);
      if k > p then
               a'=a || AddNode(j, T, a, v, tailForHead(L.lab, u), L.to: j', T', new)
               #2
      elsif k = p then
               r=L.to #1
      else  toPairAdd(T, (L.to, tailForHead(u, L.lab)), v: r #1 | a' , j', T', new #2)
      end
}
```

5. СКЛЕИВАНИЕ ПЕРЕМЕННЫХ И УСТРАНЕНИЕ РЕКУРСИИ

Описания типов, описания глобальных переменных и определения предикатов SufTree_mcc, mcc, scanAdd, linkAdd, toPairAdd, AddNode и diff представляют полную предикатную программу построения дерева суффиксов по алгоритму Маккрейта. Проведем склеивание переменных и замену хвостовой рекурсии циклом. Эти два вида преобразований являются начальными при трансформации предикатной программы в эффективную императивную программу. Приведем группы склеиваемых переменных по всем определениям предикатов. Переменные в каждой группе заменяются первой.

```
SufTree_mcc:        j<-j0
mcc:      j<- j', k; T<-T', U; e <- d, a; last <- new, new1; v <- z; f <- f1; b <- r;
scanAdd:            j<-j'; T<-T'; v<-v', z;
linkAdd:            j<-j'; T<-T'; a<-a1, a'; b <- r; u<-u1;
toPairAdd:          j<-j'; T<-T'; a<-a'; b <- r;
AddNode:            j<-j'; T<-T';
diff:               g<-g'; w <- w';
```

Склеивание переменных структурного типа (для древесных пар) реализуется в два этапа: сначала склеиваются структурные переменные, затем переменные для их компонентов. После склеивания переменных e, a и d в mcc, а также a и a' в linkAdd и toPairAdd заменим оставшиеся переменные (e в mcc, d в scanAdd и a в linkAdd и toPairAdd) компонентными переменными: вершиной b и строкой u. Далее, во всех определениях переменная r должна быть склеена с b.

В определении mcc склеивание переменных e, a и d было бы невозможным без предварительного вынесения вычисления SUFL(i + last) f1 = f + (last: a.node) из вызова mcc. Аналогичным образом, чтобы склеить last и new1, необходимо вынести V i1 = i + last из вызова mcc перед вызовом scanAdd, причем далее переменные i и i1 должны быть склеены. Отметим, что нельзя склеивать переменную new в вызове linkAdd с переменной last из-за вхождений переменной last в вызове mcc на второй ветви оператора расщепления. При замене рекурсии циклом для последнего вызова mcc возникает групповой оператор присваивания:

| last, i, f |:= | new, i + last, f + (last: new) |

Для его раскрытия присваивание last := new необходимо делать последним.

```
mcc(V j*, ST(j) T, Node last, SUFL(V i) f, Node b, Str u, v: ST(j) T)
{     loop
          if u = nil then
               if b=root then
                    if length(v)=1 then   exit
                    else  scanAdd(j, T, b, tail(v): b, u, v, j, T, last)
                    end
               else    scanAdd(j, T, f[b], v: b, u, v, j, T, last)
               end
          else
               split linkAdd(j, T, f, b, u, v: b | b, u, j, T, Node new)
               do   i := i+last; f[last]:=b;
                    scanAdd(j, T, b, v: b, u, v, j, T, last)
               do   i := i+last, f[last]:=new; last := new
```

```
                end
            end
        end
}
```

В определении scanAdd дополнительно заменим параметр new на last.

```
scanAdd(V j, ST(j) T, Node b, Str v: b, Str u, v, j, T, Node last)
{  loop
       split getArc(T[b], head(v): | Arc L)
       do  u := nil || T[b][head(v)]:=(v, NIL) || last=NIL
            exit
       do  split diff(tail(L.lab), tail(v): Str g, v | v)
           do   u := headForTail(L.lab, g); AddNode(j, T, b, u, v, g, L.to: j, T, last);
                exit
           do   b:=L.to
           end
       end
   end
}

toPairAdd(ST(V j) T,  Node b, Str u, v: b | b, u, j, T, Node new)
{      loop
           getArc(T[b], head(u): ## | Arc L);
           nat k = length(L.lab); nat p = length(u);
           if k > p then
                AddNode(j, T, b, u, v, tailForHead(L.lab, u), L.to: j, T, new)
                #2
           elsif k = p then       b := L.to           #1
           else b := L.to; u := tailForHead(u, L.lab)
           end
     end
}

linkAdd(ST(V j) T, SUFL(V i) f, Node b, Str u, v: b | b, u, j, T, Node new)
{      if b=root then u:=tail(u); if u=nil then #1 end else b:=f[b] end;
       toPairAdd(T, b, u, v: b #1 | b, u, j, T, new #2)
}
AddNode(V j, ST(j) T, Node b, Str u, v, g, Node lto: j, T, Node new)
{      newName(j: new, j);
       T[b][head(u)]:=(u, new);
       T[new][head(g)]:=(g, lto);
       T[new][head(v)]:=(v, NIL)
}
```

```
diff(Str g, v: g, v | v)
{     loop      if g = nil then        #2
                elsif head(g)=head(v) then g := tail(g); v := tail(v)
                else        #1
                end
      end
}
```

6. РЕАЛИЗАЦИЯ ИМПЕРАТИВНОЙ ПРОГРАММЫ

Во второй части трансформации предикатной программы реализуется подстановка определения предиката на место его вызова и кодирование строковых объектов массивами. В результате получается эффективная императивная программа.

Определение toPairAdd может быть подставлено в linkAdd на место вызова, определение linkAdd — в mcc, а mcc — в SufTree_mcc. Кроме того, определение diff может быть подставлено в scanAdd. Объявим переменные: j, T, b, u, v, g, new, last, lto — глобальными во всей программе. Это позволяет заменить предикаты scanAdd и AddNode процедурами без параметров.

Исходную строку t будем кодировать массивом длины n, определяя n как глобальный параметр программы:

```
nat n;
type Ar(nat m) = array 1..m of A;
Str t -> Ar(n) t;
```

Произвольная переменная x типа Str в общем случае кодируется вырезкой xa[xj..xk], где xa — некоторый массив типа Ar. В нашей программе любая строка является подстрокой исходной строки t. Поэтому переменная x кодируется парой xj и xk: Str x -> **nat** xj, xk. Кроме того, строку v, являющуюся суффиксом строки t, можно представить одним значением vj, поскольку vk = n, т.е. Str v -> **nat** vj. Операции над строками кодируются следующим образом:

```
head(x) -> t[xj];   x:=tail(x) -> xj:=xj+1;     u = nil –> uj>uk;
z=headlForTail(x, y) -> { zj:=xj; zk:=yj-1 };
z=tailForHead(x, y) -> { zj:=yk+1 или zj:=xj+length(y); zk:=xk };
```

Тип Arc кодируется следующим образом:

type Arc = **struct** Str lab, Node to **end** ->
type Arc = **struct** **nat** labj, labk , Node to **end**;

Граф суффиксных ссылок f и дерево T объединяются в одну структуру, при этом суффиксная ссылка всякой вершины b хранится как атрибут sufLink вершины b:

type NodeAtrs(AA s) = **struct** Node sufLink; **array** s **of** Arc end;

Таким образом, суффиксная ссылка f[b] кодируется в виде T[b].sufLink. Однако по-прежнему будем использовать обозначение T[b][c] для доступа к символу метки.

Приведем программу, полученную в результате подстановки определений на место вызовов и кодирования строк массивами:

V j; ST(j) T; Node b; **nat** uj, uk, vj, gj. gk; Node new, last, lto;

Перечисленные описания переменных соответствуют формальным параметрам предикатов, преобразованных в глобальные переменные. В результате подстановки тела diff на место вызова в определении scanAdd получим:

```
scanAdd( )
{     loop
          split getArc(T[b], t[vj]: | Arc L)
          do    uk:=uj-1 || T[b][t[vj]]:=(vj, n, NIL) || last=NIL
                exit
          do    gj:=L.labj+1; gk:=L.labk; vj:=vj+1;
                loop if gj>gk then            goto M2
                      elsif t[gj]=t[vj] then gj:= gj+1; vj:=vj+1
                      else         goto M1
                      end
                end;
                M1:     uj:=L.labj; uk:=gj-1; lto:=L.to; AddNode(); last:=new;
                        exit
                M2:     b:=L.to
          end
      end
}
```

Ветви тела scanAdd, определяемые метками M1 и M2, можно втянуть внутрь цикла. Кроме того, в вызове getArc заменим дугу L ее компонентами: Arc L -> **nat** Lj, Lk, lto.

```
scanAdd( )
{     loop
          split   getArc(T[b], t[vj]: | nat Lj, Lk, lto)
          do      uk:=uj-1 || T[b][t[vj]]:=(vj, n, NIL) || last=NIL
                  exit
          do      gj:=Lj+1; gk:=Lk; vj:=vj+1;
                  loop if gj>gk then        b:=lto; exit
                       elsif t[gj]=t[vj] then gj:= gj+1; vj:=vj+1
                       else uj:=Lj; uk:=gj-1; AddNode(); last:=new;
                            return
                       end
                  end;
          end
      end
}
```

Реализация предиката AddNode примет следующий вид:

```
AddNode( )
{     newName(j: new, j);
      T[b][t[uj]]:=(uj, uk, new);
      T[new][t[gj]]:=(gj, gk, lto);
      T[new][t[vj]]:=(vj, n, NIL)
}
```

Приведем итоговую реализацию описания предиката SufTree_mcc:

```
SufTree_mcc( : V j, ST(j) T) ≡ T=sufTree(t)
{   newName(V(): root, j);
    T[root][t[1]]:=(1, n, NIL); last:=NIL; i:={}; b:=root; uj:=1; uk:=0; vj:=1;
    loop  if uj>uk then  if b=root then  if vj=n then    exit
                                          else  vj:=vj+1; scanAdd()
                                          end
                         else b:=T[b].sufLink; scanAdd()
                         end
          else if b=root  then uj:=uj+1;
                               if uj>uk then goto M1 end
               else  b:=T[b].sufLink
               end;
               loop getArc(T[b], t[uj]: ## | nat Lj, Lk, lto);
                    nat k:=Lk-Lj+1; nat p:=uk-uj+1;
                    if k > p then   gj:=uk+1; gk:=Lk;
                                    AddNode();          goto M2
                    elsif k = p then   b:=lto;          goto M1
                    else b := lto; uj:=uj+k
                    end
               end;
    M1:        i := i+last; T[last].sufLink:=b; scanAdd(); goto MM;
    M2:        i := i+last, T[last].sufLink:=new; last := new
    MM:
          end
    end
}
```

Для завершения реализации императивной программы необходимо выбрать конкретное представление типа ST для строкового дерева (массива вершин) и типа NodeAtrs для множества дуг произвольной вершины. Здесь возможны разные решения, и их эффективность зависит от конкретного приложения.

В программе могут быть удалены неиспользуемые вычисления

i := i + last, i := {} и last := NIL.

ЗАКЛЮЧЕНИЕ

В настоящей работе описывается разработка предикатной программы построения дерева суффиксов по алгоритму Маккрейта [11] с проведением

полного математического доказательства ее правильности. Реализуется оптимизация программы на языке P и трансформация предикатной программы в эффективную императивную программу, которая может быть переписана на любой из действующих языков программирования и использована в различных приложениях. Отметим, что за счет применения гиперфункций и операторов расщепления эффективность конечной императивной программы выше по сравнению с обычно получаемой в традиционном стиле императивного программирования.

Язык предикатного программирования P является чистым функциональным языком, а предикатная программа — правильным математическим объектом. Язык P построен на базе исчисления вычислимых предикатов [3]: каждой исполняемой конструкции языка P соответствует некоторая формула исчисления предикатов. Поэтому доказательство правильности предикатной программы реализуется как обычное математическое доказательство.

Систему трансформаций предикатной программы невозможно эффективно реализовать в стиле традиционной автоматической оптимизирующей трансляции. Наиболее сложной для реализации является эффективное кодирование типов предикатной программы. На примере кодирования строковых объектов и дерева суффиксов видно, что эффективная реализация возможна лишь при “подсказке” программиста в рамках некоторой подсистемы задания представлений. Эффективная реализация замены хвостовой рекурсии циклом и подстановки определения предиката на место вызова не представляет существенных трудностей. Задача эффективного автоматического склеивания переменных решена при условии, что задано склеивание для параметров-аргументов и параметров-результатов [2]. Однако автоматическое склеивание не всегда оптимально. Поэтому необходимо иметь средства ручной коррекции полученных списков склеиваемых переменных. С учетом всех особенностей следует рассматривать трансформационную машину [1], где трансформационная программа описывает набор совершаемых трансформаций для конкретной предикатной программы, со средствами визуализации и контроля процесса трансформации.

Развитие языков и систем функционального программирования направлено, в частности, на повышение эффективности функциональных программ, в том числе за счет все более изощренной оптимизации в процессе автоматической трансляции. При этом соображения эффективности вынуждают ограничивать возможности языка. Например, в языке функционального программирования Haskell [6] не разрешаются произвольные неупорядоченные множества в качестве типа индексов массива, а также немонолит-

ные конструкторы массива вида A+B для объединения двух массивов A и B. Напомним, что эффективная реализация A+B подразумевает склеивание A или B с результатом. Между тем, именно такие конструкции используются в предикатной программе построения дерева суффиксов. Поэтому было бы трудно адекватно записать алгоритм Маккрейта на языке Haslel. Показательно, что при желании представить алгоритмы Маккрейта [11] и Укконена [15] в виде функциональных программ на языке Haslell авторы работы [7] ограничились простыми версиями этих алгоритмов наподобие простого алгоритма в разд. 2.

Имеется ряд других преимуществ языка P в сравнении с языками функционального программирования. Операторный стиль записи предикатной программы предоставляется не вместо функционального, а наряду с ним, что дает дополнительные удобства для программиста. Язык P ближе к языкам C, Паскаль и Модула-2, нежели языки функционального программирования. Оператор расщепления на базе гиперфункции обеспечивает гибкую форму декомпозиции алгоритма — подобного нет ни в одном из известных языков программирования.

Итоговую предикатную программу построения дерева суффиксов, полученную в результате оптимизации, можно было бы запрограммировать непосредственно с проведением математического доказательства ее правильности. Очевидно, что оптимизация обходится намного дешевле, чем перепрограммирование с повторным доказательством.

СПИСОК ЛИТЕРАТУРЫ

1. Ершов А.П.. Трансформационная машина. Тема и вариации // Проблемы теоретического и системного программирования. — Новосибирск: ВЦ СО АН СССР, 1982. — С. 5–24.
2. Петров Э.Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. – Новосибирск, ИСИ СО РАН. — 2003. — С.48–61.
3. Шелехов В.И. Введение в предикатное программирование. — Новосибирск, 2002. — 82с. — (Препр. / ИСИ СО РАН; N 100).
4. Шелехов В.И. Язык предикатного программирования P. — Новосибирск, 2002. — 40с. — (Препр. / ИСИ СО РАН; N 101).
5. Delcher A.L., Karif S., Fleischmann R.D, Peterson J., White O., Salzberg S.L Alignment of whole genomes // Nucleic Acids Research. — 1999. — Vol. 27, No.11. — P. 2369–2376.
6. Report of the Programming Language Haskell. A Non-strict, Purely Functional Language, Version 1.2 // ASM SIGPLAN Notices. — 1992. — Vol. 27, No. 5.

7. Giegerich R., Kurtz S. A Comparison of Imperative and Purely Functional Suffix Tree Constructions // Science of Computer Programming. — 1995. — Vol. 25 — P. 187–218.
8. Inenaga S. Bidirectional construction of suffix trees // Nordic Journal of Computing. — 2003. — Vol. 10 (1). — P.52–67.
9. Kurtz S. Reducing the Space Requirement of Suffix Trees // Software—Practice & Experience. — 1999. — Vol. 29, No. 13. — P. 1149–1171.
10. Kim D.K., Sim J.S., Park K. Suffix Trees for Integer Alphabets Revisited. Seoul National University. `http://theory.snu.ac.kr/paper/dkkim_waac99.ps`
11. McCreight E.M. A space-economical suffix tree construction algorithms // J. ACM — 1976. — Vol. 23.— P. 262–272.
12. Kunihiko Sadakane Compressed Suffix Trees with Full Functionality. — Kyushu University. — 2003. `http://tcslab.csce.kyushu-u.ac.jp/~sada/papers/cst.ps`
13. Schürmann K.-B. and Stoye B. Suffix Tree Construction for Large Strings. / Workshop of Fundamentals of Databases. Rostock, Germany. — 2002. — `citeseer.nj.nec.com/540745.html`
14. Schürmann K.-B., Stoye J. Suffix Tree Construction and Storage with Limited Main Memory. — Universität Bielefeld, Forschungsbericht der Technischen Fakultät, Abteilung Informationstechnik. — Report 2003-06, 2003. `http://www.techfak.uni-bielefeld.de/~stoye/rpublications/report2003-06.pdf`
15. Ukkonen E. On-line construction of suffix trees // Algorithmica. — 1995. — Vol. 14. — P. 249–260.
16. Weiner P. Linear pattern matching algorithms. / Proc. 14th IEEE Symp. Switching and Automata Theory. — 1973. — P. 1–11.

В. И. Шелехов

РАЗРАБОТКА ПРОГРАММЫ ПОСТРОЕНИЯ ДЕРЕВА СУФФИКСОВ В ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Препринт
115

Рукопись поступила в редакцию 30.04.04
Рецензент В. А. Евстигнеев
Редактор З. В. Скок

Подписано в печать 19.07.04
Формат бумаги 60 × 84 1/16 Объем 3.0 уч.-изд.л., 3.3 п.л.
Тираж 80 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 34-22-02