УДК 004.05

Дедуктивная верификация и оптимизация предикатной программы конкатенации строк

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Для любой программы на языке Си можно построить эквивалентную предикатную программу, провести её дедуктивную верификацию, применить к ней набор оптимизирующих трансформаций и в результате получить исходную программу на языке Си.

Данный метод иллюстрируется для известной библиотечной программы конкатенации строк strcat. Описывается построение, дедуктивная верификация и оптимизирующая трансформация предикатной программы конкатенации строк как объектов алгебраического типа «список» в языке предикатного программирования. Разработан аппарат сканирования списков и новый метод кодирования списков через массивы.

Анализируется возможность применения технологии предикатного программирования для дедуктивной верификации программ на языке Си.

Ключевые слова*: дедуктивная верификация, трансформации программ, алгебраические типы данных, строки в языке Си.*

1. Введение

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Преимущество по времени верификации оценивалось примерно в 5 раз. Для программы быстрой сортировки с двумя опорными элементами зафиксировано преимущество в 10 раз [20]. Отметим чрезвычайно высокую сложность проведения формальной спецификации и дедуктивной верификации императивных программ.

В языке предикатного программирования Р [8] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением *оптимизирующих трансформаций*. Они определяют отличную от

классической оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу.

Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной [6, 7];
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) при помощи массивов и указателей.

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [1, 11, 15, 16, 22].

Программа принадлежит *классу программ-функций* [14], если она не взаимодействует с внешним окружением программы; точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод собран в конце программы. Программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов. Предикатная программа относится к классу *программ-функций*.

Предикатное программирование универсально. Для всякой императивной программы, принадлежащей классу программ-функций и решающей некоторую математическую задачу, можно построить эквивалентную предикатную программу. При этом императивная программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций.

Необходимость дедуктивной верификации библиотечных программ обычно требуется в случае, когда они вызываются в составе программ, для которых проводится дедуктивная верификация. В нашем случае, программа конкатенации строк strcat вызывается в модуле безопасности Linux Security Modules (LSM) ядра ОС Linux. Для программы LSM реализуется дедуктивная верификация на соответствие собственной спецификации, а также специально разработанной модели политики безопасности [3]. Дедуктивная верификация программы LSM необходима для сертификации ОС Astra Linux Special Edition [9] высоким уровнем доверия в соответствии со стандартом ГОСТ Р ИСО/МЭК 15408-3 [2].

Для программы strcat на языке Си построена эквивалентная предикатная программа. Процесс ее дедуктивной верификации описан в Приложении 1. Доступно доказательство формул корректности [5] в системе PVS. Применение набора оптимизирующих оптимизаций к предикатной программе дает императивную программу, эквивалентную исходной на языке Си.

Строки определены как частный случай алгебраического типа «список» в языке P [8]. Разработан новый метод кодирования списков через массивы с использованием двух указателей, а также аппарат сканирования списков.

Во втором разделе дается краткое описание языка предикатного программирования. Метод дедуктивной верификации описывается в третьем разделе. В четвертом разделе определено построение предикатной программы strcat. В следующем разделе описывается процесс оптимизирующей трансформации программы. Аппарат сканирования списков представлен в шестом разделе. В седьмом разделе анализируется причина хороших показателей дедуктивной верификации в предикатном программировании. В восьмом разделе рассматриваются особенности сертификации предикатных программ. В девятом разделе описывается опыт проектов по дедуктивной верификации библиотек на языке Си. В заключении суммируются результаты работы.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [8] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
measure <выражение>
{ <оператор> }
```

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [11, 15, 22]. Мера задается только для рекурсивных программ и используется для доказательства их завершения.

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

<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> **else** <оператор2>
<имя программы>(<список аргументов>**:** <список результатов>)
<тип> <пробел> <список имен переменных>

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **struct**(T_1 f_1, ..., T_n f_n) является *структура* из n значений, именуемых *полями* f_1, f_2,..., f_n и имеющих типы T_1, T_2,..., T_n, соответственно.

Значением *алгебраического типа* **union**(K_1, …, K_n) является один из конструкторов K_1, K_2, …, K_n. *Конструктор* K(T_1 f_1, ..., T_m f_m) определяется *именем конструктора* K и набором полей как у структуры; набор полей может быть пустым. Для объекта s алгебраического типа *распознаватель* K?(s) является истинным, если объект s определяется как конструктор вида K. Алгебраический тип может быть рекурсивно определяемым.

Типичными объектами алгебраических типов являются списки и деревья. *Список* определяет последовательность элементов некоторого произвольного типа T, являющегося параметром. Описание типа списка следующее:

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

Алгебраический тип list имеет два конструктора: nil, обозначающий пустой список, и cons, определяющий список, первый элемент которого представлен полем car, а остальная часть списка («хвост» – все элементы, кроме первого) определяется полем cdr. Тип list считается определенным в языке P и не требует описания в программе.

Пусть s – переменная типа список. Тогда s.car определяет первый элемент – «голову» списка s, s.cdr – список без первого элемента – «хвост» списка s. Распознаватель nil?(s) определяет принадлежность списка s конструктору nil, т.е. пустоту списка. Вместо nil?(s) обычно используется s = nil.

Для списков s и u определены операции: len(s) – длина списка, s + u – конкатенация списков s и u. В качестве операндов конкатенации допускаются также элементы списка.

При трансляции предикатной программы в императивный язык основным способом представления списка является массив. Другими возможными альтернативами кодирования списка являются: односвязный список, двунаправленный список, кольцевой список. В языке

P введены дополнительные конструкции, обеспечивающие эффективную реализацию списков через массивы [19].

3. Дедуктивная верификация

Предикатная программа относится к классу *программ-функций* [14]. Программа-функция должна всегда **нормально завершаться** с получением результата, поскольку бесконечно работающая и невзаимодействующая программа бесполезна.

Спецификацией предикатной программы H(x**:** y) являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Спецификация записывается в виде: [P(x), Q(x, y)].

Для языка **P0** построена формальная операционная семантика ℛ(H)(x, y) и доказано тождество ℛ(H) = H [17]. На базе языка **P0** последовательным расширением и сохранением тождества ℛ(H) = H построен язык предикатного программирования **P** [8].

Тотальная корректность программы относительно спецификации определяется формулой:

$$H(x\mathbf{:}\ y)\ \mathbf{corr}\ [P(x), Q(x, y)] \cong \forall x.\ P(x) \Rightarrow [\forall y.\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ H(x\mathbf{:}\ y)$$

Формулу тотальной корректности будем представлять в виде правила **COR**:

$$\mathbf{COR}:\ \frac{\forall x,y.\ P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y);\quad \forall x.\ P(x) \Rightarrow \exists y.\ H(x\mathbf{:}\ y)}{H(x\mathbf{:}\ y)\ \mathbf{corr}\ [P(x), Q(x, y)]}$$

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная система правил доказательства их корректности [10], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности. Корректность правил доказана [4] в системе PVS. В системе предикатного программирования реализован генератор формул корректности программы. Часть формул доказывается автоматически SMT-решателем CVC4. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [24]. Данный метод опробован для дедуктивной верификации более сотни программ [11, 15, 18, 22].

Предположим, что наборы переменных x, y и z не пересекаются, а x может быть пустым. Ниже приведены некоторые правила доказательства корректности операторов.

$$\textbf{QP:}\ \frac{B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)];\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x),\ R(x,\ z)];}{\{B(x\textbf{:}\ y)\ ||\ C(x\textbf{:}\ z)\}\ \textbf{corr}\ \ [P(x),\ Q(x,\ y)\ \&\ R(x,\ z)]}$$

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x),\ Q(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x,\ y)]\end{array}}{\{\textbf{if (}E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Далее следует правило для частного случая оператора суперпозиции, соответствующего *сведению к более общей задаче* $C(x,\ z\textbf{:}\ y)$.

$$\textbf{RB:}\ \frac{\begin{array}{l}\forall z\ C(x,\ z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x,\ z),\ Q_C(x,\ y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x,\ B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x),\ y) \Rightarrow Q(x,\ y)\ );\end{array}}{C(x,\ B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Запись вида $z = B(x)$ является эквивалентом $B(x\textbf{:}\ z)$. Истинность трех посылок правила **RB** гарантирует корректность следующей программы:

$$H(x\textbf{:}\ y)\ \textbf{pre}\ P(x)\ \textbf{post}\ Q(x,\ y)\ \{\ C(x,\ B(x)\textbf{:}\ y)\ \}$$

В случае рекурсивного вызова $C(x,\ B(x)\textbf{:}\ y)$ обозначение **corr*** означает, что первая посылка опускается, а $P^*_C(x,\ B(x))$ заменяется на $P_C(x,\ B(x))\ \&\ m(x) < m(v)$. Здесь m – натуральная функция *меры*, строго убывающая на аргументах рекурсивных вызовов, а v обозначает аргументы рекурсивной программы C.

4. Построение программы конкатенации строк

4.1. Постановка задачи

Имеется следующая программа конкатенации строк на языке Си:

```
char *strcat(char *dest, const char *src)
{
        char *tmp = dest;
        while (*dest)
                dest++;
        while ((*dest++ = *src++) != '\0');
        return tmp;
}
```

Строка по указателю src добавляется к строке по указателю dest. Предполагается, что указатель dest ссылается на участок памяти достаточного размера для результата strcat.

Требуется построить соответствующую предикатную программу, провести ее дедуктивную верификацию и трансформировать ее в эффективную императивную программу, эквивалентную приведенной выше.

4.2. Предикатная программа конкатенации строк

Построим предикатную программу конкатенации строк dest и src. Результат конкатенации – итоговое значение переменной dest. В языке P конкатенация строк определена операцией «+» [8, разд. 7.3]. Реализация строк в трансляторе должна быть поддержана специальной библиотекой. Поэтому программа проста:

```
strcat(string dest, src: string dest') post dest' = dest + src
{ dest' = dest + src };
```

Здесь штрих в имени dest' означает, что в итоговой императивной программе переменные dest и dest' должны быть склеены: dest' заменяется на dest.

Исходная программа на языке Си (разд. 4.1) соответствует библиотечной программе для реализации строковой операции «+». Нам предстоит построить соответствующую предикатную программу с реализацией строкового типа через тип списка list.

Строковый тип **string** является предопределенным в языке P. Его определение имеет вид:

```
type string = list(char);
```

Для произвольного объекта типа **string** используется традиционное представление в виде массива литер, завершающегося нулем, причем нуль не входит в значение строки. Дадим точное определение типа **string** для данного стандартного представления. Для строки s через sval обозначим *собственно значение* строки без завершающего нуля. Предикат nozero(sval) постулирует, что в значении строки недопустимо использование нуля. Далее операция «+» везде является операцией конкатенации для списков, реализуемая в библиотечной поддержке языка P.

```
char zero = \0;
type listchar = list(char);
formula isVal(listchar s, sval) = s = sval + zero;
formula nozero(listchar sval) = sval = nil or sval.car≠zero & nozero(sval.cdr);
formula Str(listchar s) = ∃ listchar sval. isVal(s, sval) & nozero(sval);
type string = subtype(listchar s: Str(s));
```

В приведенной последовательности описаний строковый тип определяется как множество последовательностей литер, не содержащих нуля и дополненных нулем в конце.

Проверка строки s на пустоту реализуется операцией s.car = \0, а не s = nil. В итоге, типы list и **string** несовместимы: со строковым объектом нельзя работать как со списком, в

частности, нельзя подставлять строковый объект параметром типа list. Библиотеки для списков неприменимы для строковых объектов.

В представленной ниже предикатной программе strcat, реализующей конкатенацию строк dest и src, сначала для первой строки dest выделяется ее собственно значение s, после чего реализуется конкатенация списков s и src.

```
strcat(string dest, src: string dest')
post ∃ listchar sval. isVal(s, sval) & dest' = sval + src
{   strval(dest: listchar s);
    listcat(s, src: dest')
};
```

Программа strval, представленная здесь как библиотечная, определяет собственно значение s для строки dest.

```
strval(string dest: listchar s) post dest = s + zero;
```

Реализация программы strval в виде предикатной программы описана ниже в разд. 6.

Программа listcat реализует оператор dest = s + src, где «+» – операция конкатенации списков. Учитывая особенности представления строк, здесь нужна предикатная программа, приведенная ниже.

```
listcat(listchar s, string src: string dest) post dest = s + src measure len(src)
{   listchar s1 = s + src.car;
    if (src.car = zero) dest = s1 else listcar(s1, src.cdr: dest)
};
```

Итак, полная предикатная программа состоит из программ strcat и listcat. Программа strval представлена в разд. 6.

5. Трансформации

Определим набор трансформаций, превращающих программу конкатенации строк, состоящую из программ strcat и listcat, в эффективную императивную программу. На первом этапе реализуется трансформация склеивания переменных. Например, операция склеивания dest ← dest', s реализует замену всех вхождений переменных dest' и s на переменную dest.

Склеивание: dest ← dest', s.

```
strcat(string dest, src: dest)
{   strval(dest: dest);
    listcat(dest, src: dest)
};
```

Склеивание: dest ← s, s1.

```
listcat(listchar dest, string src: dest)
{   dest = dest + src.car;
    if (src.car = zero) dest = dest else listcar(dest, src.cdr: dest)
};
```

На втором этапе проводится замена хвостовой рекурсии циклом в программе listcat:

```
listcat(listchar dest, string src: dest) post dest = s + src
{  loop {
    dest = dest + src.car;
    if (src.car = zero) {dest = dest; break} else |dest, src| = |dest, src.cdr|
  }
};
```

Раскрывается групповой оператор присваивания: в результате остается оператор src = src.cdr. Удаляется оператор dest = dest.

```
listcat(listchar dest, string src: dest) post dest = s + src
{  loop {
    dest = dest + src.car;
    if (src.car = zero) break;
    src = src.cdr
  }
};
```

На третьем этапе программа listcat подставляется на место вызова в strcat.

```
strcat(string dest, src: dest)
{   strval(dest: dest);
    loop {
       dest = dest + src.car;
       if (src.car = zero) break;
       src = src.cdr
    }
};
```

На четвертом этапе трансформаций списки кодируются через массивы. Стандартный метод кодирования списков через массивы описан в работе [13, разд.5.4]. Здесь применяется новый метод: кодирование списков через указатели, отличающийся от кодирования односвязными списками [12].

Список s кодируется массивом литер. Работа с массивом реализуется с помощью двух указателей sF и sL:

```
type STR = *char;
STR sF, sL;
```

Указатель sF «смотрит» на начало массива и используется для чтения. Указатель sL ссылается на следующий элемент за последним элементом списка s и предназначен для записи.

Операции со списками языка **P** необходимо представить эквивалентными операциями для массивов через пару указателей. Ниже приведено представление операций и операторов программы strcat.

```
src.car   →      *srcF;
dest = dest + src.car   →      *destL = *srcF; destL++
src = src.cdr     →      srcF++;
s = strval(dest)    →    sF = destF; for(sL = destF;*sL != zero; sL++);
```

Представленный способ кодирования вызова библиотечной программы strval определяется в разд. 6. Итоговая программа представлена ниже. В ней destF заменено на dest, srcF – на src.

```
strcat(char *dest, const char *src)
{   char *destL;
    for(destL = dest;*destL != zero; destL++); //strval(dest: dest);
    loop {
       *destL = *src; destL++;                    //dest = dest + src.car;
       if (*src = zero) break;
       src++                                      //src = src.cdr
    }
};
```

Сопоставляем с исходной программой на языке Си:

```
char *strcat(char *dest, const char *src)
{
        char *tmp = dest;
        while (*dest)
                dest++;
        while ((*dest++ = *src++) != '\0');
        return tmp;
}
```

Полученная в результате трансформаций программа почти эквивалентна исходной. Отличие в том, что в исходной программе указатель src продвинут дальше на одну позицию. Другая особенность: при завершении программы destL «смотрит» не на нуль, а на следующий за нулем. Это значит, что данный алгоритм не лучший для реализации серии конкатенаций.

6. Сканирование списков

Библиотечная программа strval, используемая при построении программы strcat для вычисления собственно значения строки, реализуется в виде предикатной программы применением *механизма сканирования списков*. *Сканирование* непустого списка a реализуется продвижением по нему другого списка b таким образом, что a = b + c для некоторого остатка – списка c. Сначала применяется операция **init**, в результате которой b = nil и c = a. Далее применяется операция **step**, которая начальный элемент списка c перемещает в конец списка b. Определение операций **init** и **step** представлено ниже в виде предикатных программ.

```
init(list(T) a: list(T) b, c) pre  a ≠ nil post b = nil & c = a
{ b = nil || c = a };

step(list(T) a, b, c: list(T) b′, c′)
pre a = b + c & c ≠ nil post b′ = b + c.car & c′ = c.cdr
{ b′ = b + c.car || c′ = c.cdr };
```

Программа strval реализуется сканированием вида dest = s + d, завершающимся при d = zero. Чтобы упростить программу, определим переменную dest глобальной. Результатом программы strval является переменная s – собственно значение строки dest.

```
string dest;
strval( : listchar s) post dest = s + zero
{ scan(init(dest): s) };
```

Программа scan является обобщением программы strval. Она реализует сканирование вида dest = s0 + d и завершается при d = zero.

```
scan(listchar s0, string d: listchar s)
pre dest = s0 + d post dest = s + zero measure len(d)
{ if (d.car = zero) s = s0 else scan(step(s0, d): s) };
```

Определим трансформации для программы strval. На первом этапе для программы scan проводится склеивание: s ← s0.

```
scan(listchar s, string d: s)
{ if (d.car = zero) s = s else scan(step(s, d): s) };
```

Далее хвостовая рекурсия в scan заменяется циклом. Удаляется оператор s = s. На месте рекурсивного вызова появляется групповой оператор присваивания |s, d| = step(s, d).

```
scan(listchar s, string d: s)
{ loop { if (d.car = zero) break; |s, d| = step(s, d) } };
```

На следующем этапе тело программы scan подставляется на место вызова в strval.

```
strval( : listchar s)
{  string d; |s, d| = init(dest);
   loop { if (d.car = zero) break;  |s, d| = step(s, d) }
};
```

Каждый список кодируется массивом литер и двумя указателями по схеме, описанной в разд. 5. Первый указатель «смотрит» на первый элемент списка, второй указатель указывает на элемент, непосредственно следующий за последним элементом списка. Определим описание переменных для указателей программы strval:

```
type STR = *char;
STR destF, sF, sL, dF;
```

Распределение указателей после очередного цикла работы программы strval показано на Рис.1. Список d размещается в памяти непосредственно после списка s. Для указателей реализуются следующие равенства: sF = destF и sL = dF. Учитывая это, далее вместо указателя dF будем использовать sL.

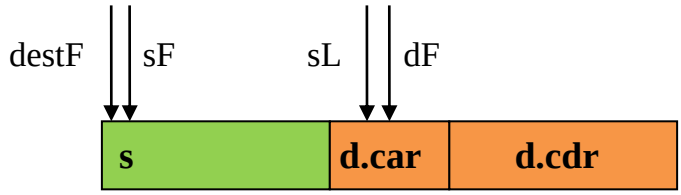

Рис. 1. Схема сканирования

Ниже приведено представление операций и операторов программы strval.

```
|s, d| = init(dest)  →   sF = destF; sL = destF;
|s, d| = step(s, d) →  sL++;
d.car       → *sL;
```

Результат кодирования строковых операций представлен ниже.

```
strval(string dest: listchar s)
{   sF = destF; sL = destF;
   loop { if (*sL = zero) break;  sL++ }
};
```

Преобразование к циклу **for** дает итоговую программу:

```
strval(string dest: listchar s)
{ sF = destF; for (sL = destF; *sL != zero;  sL++); }
```

В дальнейшем, при трансформации предикатных программах вызов strval будет кодироваться данным фрагментом кода; см. кодирование s = strval(dest) в разд. 5.

7. Кривая логика императивных программ

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Преимущество по времени верификации более чем в 5 раз. Какова причина этого? Кривая логика императивных программ.

Программа из класса программ-функций [14] имеет две основы: логику и вычислимость, причем логика является первичной, главной. Предикатная программа – предикат, вычислимая логическая формула. Набор операторов языка **P0** [17], на базе которого строится язык **P**, определяется Таблицей 1 [17, разд. 3.2]:

H(x**:** y) ≅ ∃z. B(x**:** z) & C(z**:** y)	H(x**:** y) { B(x**:** z); C(z**:** y) }
H(x**:** y, z) ≅ B(x**:** y) & C(x**:** z)	H(x**:** y, z) { B(x**:** y) || C(x**:** z) }
H(x**:** y) ≅ (e ⇒ B(x**:** y)) & (¬e ⇒ C(x**:** y))	H(x**:** y) { **if** (e) B(x**:** y) **else** C(x**:** y) }
H(x**:** y) ≅ B(x˜**:** y)	H(x**:** y) { B(x˜**:** y) }
H(A, x**:** y) ≅ A(x**:** y)	H(A, x**:** y) { A(x**:** y) }
H(x**:** D) ≅ ∀y,z. D(y**:** z) ≡ B(x, y**:** z)	H(x**:** D) { D(y**:** z) {B(x, y**:** z)} }
H(A, x**:** D) ≅ ∀y,z. D(y**:** z) ≡ A(x, y**:** z)	H(A, x**:** D) { D(y**:** z) {A(x, y**:** z)} }

Таблица 1. Вычислимые предикаты и их программная форма.

Для предикатной программы используется программная форма в правой колонке. При этом каждому оператору предикатной программы непосредственным очевидным образом сопоставляется соответствующая логическая формула. Соответствие между предикатной программой и ее логикой простое и естественное.

Если к программе применяется оптимизирующее преобразование, то ее логика не исчезает. Она усложняется, искривляется. Усложняется также соответствие между логикой и программой. Поскольку императивная программа получается из эквивалентной предикатной программы применением серии оптимизирующих трансформаций, то искривление логики императивной программы довольно существенное. Искривляется также и формальная спецификация императивной программы по сравнению со спецификацией соответствующей предикатной программы. В частности, инварианты циклов принципиально сложнее предусловий соответствующих рекурсивных программ, преобразуемых в циклы.

Сложность формальных спецификаций все чаще становится предметом обсуждения в последних работах. В качестве одного из способов решения проблемы предлагается приблизить язык спецификаций к языку программирования [23].

8. Особенности сертификации

Сертификация программ, разработанных с применением дедуктивной верификации, налагает дополнительные требования на технологию программирования и сопутствующие инструменты. Определим набор действий *оценщика* – эксперта, которому в соответствии с правилами стандарта ГОСТ Р ИСО/МЭК 15408-3 [2] надлежит сертифицировать программы, разработанные по технологии предикатного программирования.

Например, для сертификации программы strcat, представленной в данной статье, объектом оценки стал бы материал разделов 4–6 и Приложения 1. То есть описание построения предикатной программы, ее оптимизирующей трансформации и дедуктивной верификации. Для удобства анализа в Приложении 1 показывается каждое применяемое правило доказательства корректности вместе с его конкретизацией. Дополнительно требуется свидетельство того, что применяемое правило корректно, т.е. присутствует в списке правил, доказанных [4] в системе PVS.

После разработки производственной системы предикатного программирования, появятся требования к ее корректности и предоставления набора свидетельств, оценка которых позволит определить уровень доверия к ней. Во-первых, необходимо свидетельство того, что набор правил, применяемых генератором формул корректности в подсистеме дедуктивной верификации, покрывается списком правил, доказанных [4] в системе PVS, вместе со свидетельством корректности работы самого генератора. Необходимы также свидетельства корректности трансляции с языка **P**, а также корректности реализации оптимизирующих трансформаций. Обеспечить такие два свидетельства чрезвычайно трудно. Разработка моделей трансляции и оптимизирующей трансформации, их анализ и верификация должны повысить уровень доверия к системе. Дополнительным артефактом оценки может стать набор листингов программ после каждой автоматически применяемой трансформации.

9. Другие работы по верификации библиотечных программ

В работе [21] описывается дедуктивная верификация 26 библиотечных функций языка Си, в основном для работы со строковыми объектами, в рамках международного проекта по верификации ядра ОС Linux. Используется стек инструментов дедуктивной верификации AstraVer [27] с выходом на большое число SMT-решателей. Доказательство генерируемых условий корректности функций реализуется исключительно применением SMT-решателей с использованием простых лемм, которые возможно доказываются применением интерактивных систем доказательства типа PVS [24]. Формальные спецификации были

написаны для исходного кода библиотечных функций, и лишь коды двух функций подвергнуты предварительной модификации. Результаты работы [21] существенно превосходят достижения других известных проектов, в частности, представленных работами [25, 26], по дедуктивной верификации библиотечных функций языка Си.

10. Заключение

В настоящей работе представлена технология предикатного программирования для построения, оптимизирующей трансформации и дедуктивной верификации библиотечной программы конкатенации строк strcat. Разработан новый способ кодирования списков через массивы с использованием двух указателей. Данный способ соответствует традиционному стилю работы со строками через указатели. Разработан аппарат сканирования списков, используемый для нахождения конца строки. Возможно, существуют другие решения по кодированию строковых объектов.

Дедуктивная верификация предикатных программ существенно проще и быстрее, чем дедуктивная верификация аналогичных императивных программ. В предикатном программировании затраты на формальную спецификацию и дедуктивную верификацию в 3-6 раз превышают затраты на разработку программы в традиционном стиле с применением тестирования. По сравнению с императивным программированием это принципиально расширяет возможности применения дедуктивной верификации в производственном программировании.

Чтобы обеспечить высокий уровень доверия к инструментам верификации, необходимо гарантировать корректность применяемой технологии и сопутствующих инструментов. В рамках третьего релиза транслятора с языка P разработана модель внутреннего представления транслируемой программы. В дальнейшем планируется формализовать эту модель и провести ее верификацию. На базе этой модели предполагается также построить модель оптимизирующих трансформаций.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ. 2008. Т. 4 (33). С. 79-94.
2. ГОСТ Р ИСО/МЭК 15408-3-2013. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 3. Компоненты доверия к безопасности.

3. Девянин П.Н., Ефремов Д.В., Кулямин В.В., Петренко А.К., Хорошилов А.В., Щепетков И.В. Моделирование и верификация политик безопасности управления доступом в операционных системах // Коллективная монография, версия 1.3. ИСП РАН, 2018. 181с. [Электронный ресурс]. URL: http://www.ispras.ru/publications/security_policy_modeling_and_verification.pdf (дата обращения 12.11.2018)
4. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL: http://www.iis.nsk.su/persons/vshel/files/rules.zip (дата обращения 12.11.2018)
5. Доказательство формул корректности программы strcat в системе PVS. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/strcat.zip (дата обращения 12.11.2018)
6. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. Новосибирск, 2017. С. 21-48. Электрон. журн. 2018. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения 12.11.2018)
7. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. Новосибирск, 2012. 6с. (Препр. / ИСИ СО РАН; N 167).
8. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с., [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf (дата обращения 12.11.2018).
9. Операционные системы Astra Linux. [Электронный ресурс]. URL: http://www.astralinux.ru (дата обращения 12.11.2018).
10. Чушкин М.С. Система дедуктивной верификации предикатных программ // «Программная инженерия». 2016. № 5. С. 202-210. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf (дата обращения 12.11.2018).
11. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
12. Шелехов В.И. Дедуктивная верификация и реализация предикатной программы инвертирования односвязных списков. ИСИ СО РАН, Новосибирск, 2018. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/listinvert.pdf (дата обращения 12.11.2018).
13. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017), Том 2. Институт Математики СО РАН, Новосибирск, 2017. С. 156-165. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf (дата обращения 12.11.2018).
14. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. С. 531–538. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения 12.11.2018).
15. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).

16. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
17. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения 12.11.2018).
18. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы ’2017», Ростов-на-Дону. 2017. С.258-262. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf (дата обращения 12.11.2018).
19. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, ИСИ СО РАН, Новосибирск. Электрон. журн. 2014, №3. С. 25-43. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/Listcharing.pdf (дата обращения 12.11.2018).
20. Шелехов В.И., Чушкин М.С. Верификация программы быстрой сортировки с двумя опорными элементами // Научный сервис в сети Интернет. М.: ИПМ им. М.В.Келдыша, 2018. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf (дата обращения 12.11.2018).
21. Ефремов Д.В, Мандрыкин М.У. Формальная верификация библиотечных функций ядра Linux. Труды ИСП РАН, том 29, вып. 6, 2017. С. 49-76. DOI: 10.15514/ISPRAS-2017-29(6)-3
22. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.
23. Utting M., Pearce D.J., Groves L. Making Whiley Boogie! // Integrated Formal Methods. LNCS 10510, 2017. p. 69-84.
24. PVS Specification and Verification System. SRI International. [Электронный ресурс]. URL: http://pvs.csl.sri.com/ (дата обращения 12.11.2018).
25. Carvalho N., Silva Sousa C., Pinto J.S., Tomb A. Formal verification of kLIBC with the WP frama-C plug-in // NFM 2014. LNCS 8430, 2014. P. 343–358.
26. Torlakcik M. Contracts in OpenBSD. MSc thesis. University College Dublin. April, 2010.
27. AstraVer Toolset: инструменты дедуктивной верификации моделей и механизмов защиты ОС. ИСП РАН. URL: http://linuxtesting.org/astraver, 15.10.2017 (дата обращения 30.11.2018)

Приложение 1.
Дедуктивная верификация программы strcat

Приведем полный набор определений, необходимых для верификации программы strcat. Сначала определим формулы для предусловий и постусловий.

```
char zero = \0;
type listchar = list(char);
formula isVal(listchar s, sval) = s = sval + zero;
formula nozero(listchar sval) = sval = nil or sval.car≠zero & nozero(sval.cdr);
formula Str(listchar s) = ∃ listchar sval. isVal(s, sval) & nozero(sval);
type string = subtype(listchar s: Str(s));
formula Pstrcat(string dest, src) = Str(src) & Str(dest);
formula Qstrcat(string dest, src, dest') = ∃ listchar sval. isVal(dest, sval) & dest' = sval + src;
```

Программа strcat:

```
strcat(string dest, src: string dest') post Qstrcat(dest, src, dest')
{   strval(dest: listchar s);
    listcat(s, src: dest')
};

strval(string s: listchar s') post isVal(s, s');
formula Plistcat(listchar s, string src) = Str(src);
formula Qlistcat(listchar s, string src, dest) = dest = s + src & Str(dest);
listcat(listchar s, string src: string dest) post Qlistcat(s, src, dest);
```

Следует учитывать, что для аргументов типа **string** в предусловие необходимо добавить предикат Str от аргумента. Аналогично, для результата типа **string** в постусловие добавляется предикат Str от результата. Это учтено в приведенных формулах для предусловий и постусловий.

Сначала применяется правило **RS** для оператора суперпозиции strval(dest: listchar s); listcat(s, src: dest'), составляющего тело программы strcat:

$$\textbf{RS}:\ \frac{\begin{array}{l} B(x{:}\ z)\ \textbf{corr}^*\ [P_B(x), Q_B(x, z)];\\ \forall z.\ C(x, z{:}\ y)\ \textbf{corr}^*\ [P_C(x, z), Q_C(x, z, y)];\\ \forall z, y.\ (\ P(x)\ \&\ Q_B(x, z)\ \&\ Q_C(x, z, y) \rightarrow\ Q(x, y)\ )\\ \forall z.\ (\ P(x)\ \&\ Q_B(x, z) \rightarrow\ P^*_C(x, z)\ );\\ P(x) \rightarrow\ P^*_B(x); \end{array}}{B(x{:}\ z);\ C(x, z{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Здесь B(x: z) соответствует strval(dest: listchar s), а C(x, z: y) – listcat(s, src: dest'). Ниже конкретизация правила **RS**.

RS:
$$\frac{\begin{array}{l}\text{strval(dest: s)B(x: z) } \mathbf{corr}^{*} \text{ [Str(dest), isVal(dest, s)];}\\ \text{listcat(s, src: dest') } \mathbf{corr}^{*} \text{ [Plistcat(s, src), Qlistcat(s, src, dest')];}\\ \text{Pstrcat(dest, src) \& isVal(dest, s) \& Qlistcat(s, src, dest')} \rightarrow \text{Qstrcat(dest, src, dest')}\\ \text{Pstrcat(dest, src) \& isVal(dest, s)} \rightarrow \text{ Plistcat(s, src);}\\ \text{Pstrcat(dest, src)} \rightarrow \text{ Str(dest);}\end{array}}{\begin{array}{l}\text{strval(dest: s); listcat(s, src: dest') } \mathbf{corr} \text{ [Pstrcat(dest, src), Qstrcat(dest, src, dest')]}\end{array}}$$

Первая посылка определяет корректность программы strval, считающейся здесь предопределенной. Корректность программы listcat, определяемая второй посылкой, будет рассмотрена ниже. Посылки с третьей по пятую определяют следующие формулы корректности:

Pstrcat(dest, src) & isVal(dest, s) & Qlistcat(s, src, dest') ⇒ Qstrcat(dest, src, dest');
Pstrcat(dest, src) & isVal(dest, s) ⇒ Str(src);
Pstrcat(dest, src) ⇒ Str(dest);

Истинность двух последних формул очевидна. Первая формула будет доказана в PVS.

Рассмотрим программу listcat и построим для нее формулы корректности. Задание меры len(src) обязательно для рекурсивных программ.

```
formula Plistcat(listchar s, string src) = Str(src);
formula Qlistcat(listchar s, string src, dest) = dest = s + src & Str(dest);
listcat(listchar s, string src: string dest) post Qlistcat(s, src, dest) measure len(src)
{   listchar s1 = s + src.car;
    if (src.car = zero) dest = s1 else listcar(s1, src.cdr: dest)
};
```

Тело программы listcat является оператором суперпозиции. В данном случае удобнее применить следующую модификацию правила **QS**:

QS:
$$\frac{\begin{array}{l}\text{P(x)} \rightarrow \exists \text{z. B(x: z);}\\ \forall \text{z. C(x, z: y) } \mathbf{corr} \text{ [P(x) \& B(x: z), Q(x, y)];}\end{array}}{\begin{array}{l}\text{B(x: z); C(x, z: y) } \mathbf{corr} \text{ [P(x), Q(x,y]}\end{array}}$$

Приведем конкретизацию данного правила **QS**.

QS:
$$\frac{\begin{array}{l}\text{Plistcat(s, src)} \rightarrow \exists \text{ s1. s1 = s + src.car;}\\ \mathbf{if} \text{ ... } \mathbf{corr} \text{ [Plistcat(s, src) \& s1 = s + src.car, Qlistcat(s, src, dest)];}\end{array}}{\begin{array}{l}\text{s1 = s + src.car; } \mathbf{if} \text{ ... } \mathbf{corr} \text{ [Plistcat(s, src), Qlistcat(s, src, dest)]}\end{array}}$$

Посылки правила **QS** определяют следующие формулы для listcat.

Str(src) ⇒ ∃s1. s1 = s + src.car;
if (src.car = zero) dest = s1 **else** listcar(s1, src.cdr: dest) **corr**
[Str(src) & s1 = s + src.car, Qlistcat(s, src, dest)];

Истинность первой формулы очевидна. Вторая формула определяет новую цель. Здесь применяется правило **QC** для условного оператора.

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x),\ Q(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x,\ y)]\end{array}}{\{\textbf{if}\ (E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Конкретизация правила **QC**:

$$\textbf{QC:}\ \frac{\begin{array}{l}dest = s1\ \textbf{corr}\ [Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero,\ Qlistcat(s,\ src\textbf{,}\ dest)];\\ listcar(s1,\ src.cdr\textbf{:}\ dest)\ \textbf{corr}\\ [Str(src)\ \&\ s1 = s + src.car\ \&\ \neg\ src.car = zero,\ Qlistcat(s,\ src\textbf{,}\ dest)]\end{array}}{\begin{array}{l}\textbf{if}\ (src.car = zero)\ dest = s1\ \textbf{else}\ listcar(s1,\ src.cdr\textbf{:}\ dest)\ \textbf{corr}\\ [Str(src)\ \&\ s1 = s + src.car,\ Qlistcat(s,\ src\textbf{,}\ dest)]\end{array}}$$

Посылки правила **QC** определяют следующие две цели:

$$\begin{array}{l}dest = s1\ \textbf{corr}\ [Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero,\ Qlistcat(s,\ src\textbf{,}\ dest)\ ]\\ listcar(s1,\ src.cdr\textbf{:}\ dest)\ \textbf{corr}\\ \qquad\qquad [Str(src)\ \&\ s1 = s + src.car\ \&\ src.car \neq zero,\ Qlistcat(s,\ src\textbf{,}\ dest)\ ]\end{array}$$

Для первой цели применяется правило **COR**, соответствующее определению «**corr**».

$$\textbf{COR:}\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,\ y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Конкретизация правила **COR**:

$$\textbf{COR:}\ \frac{\begin{array}{l}Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero\ \ \&\ dest = s1 \Rightarrow Qlistcat(s,\ src\textbf{,}\ dest);\\ Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero\ \Rightarrow \exists s1.\ dest = s1\end{array}}{dest = s1\ \textbf{corr}\ [Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero\ P(x),\ Qlistcat(s,\ src\textbf{,}\ dest)]}$$

Посылки правила определяет следующие формулы корректности:

$$\begin{array}{l}Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero\ \&\ dest = s1 \Rightarrow Qlistcat(s,\ src\textbf{,}\ dest);\\ Str(src)\ \&\ s1 = s + src.car\ \&\ src.car = zero \Rightarrow \exists dest.\ dest = s1;\end{array}$$

Истинность второй формулы очевидна. Первая формула будет доказана в PVS.

Для второй цели правила **QC** для рекурсивного вызова listcar(s1, src.cdr**:** dest) применим правило **RB**.

$$\textbf{RB:}\ \frac{\begin{array}{l}\forall z\ C(x,\ z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x,\ z),\ Q_C(x,\ y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x,\ B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x),\ y) \Rightarrow Q(x,\ y)\ );\end{array}}{C(x,\ B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Здесь B(s, src) = (s1, src.cdr), предикат P_B = src ≠ nil. Первая посылка правила **RB** для программы listcar отсутствует ввиду рекурсивности listcar. Далее, предикат $P^*_C(x, B(x))$ определяется следующим образом: $P^*_C(B(s, src)) = P_C(B(s, src))$ & len(src.cdr)<len(src).

Конкретизация правила **RB**:

listcat(s, src**:** dest) **corr*** [Str(src), Qlistcat(s, src**,** dest)];
Str(src) & s1 = s + src.car & src.car ≠ zero ⇒ cons?(src) & Str(src.cdr) & len(src.cdr)<len(src);
Str(src) & s1 = s + src.car & src.car ≠ zero & Qlistcat(s1, src.cdr, dest) ⇒ Qlistcat(s, src**,** dest);
RB: listcar(s1, src.cdr**:** dest) **corr**
[Str(src) & s1 = s + src.car & src.car ≠ zero, Qlistcat(s, src**,** dest)]

Ввиду рекурсивности listcar первая посылка опускается. Вторая и третья посылки правила **RB** определяют следующие формулы корректности:

Str(src) & s1 = s + src.car & src.car ≠ zero ⇒ cons?(src) & Str(src.cdr) & len(src.cdr)<len(src);
Str(src) & s1 = s + src.car & src.car ≠ zero & Qlistcat(s1, src.cdr**,** dest) ⇒
Qlistcat(s, src**,** dest);

Рассмотрим верификацию программы strval. Предварительно проведем открытую подстановку для вызовов стандартных операций **init** и **step**.

formula Pscan(listchar s, **string** d, dest) = Str(d) & Str(dest) & dest = s + d;
formula Qstrcat(**string** dest, src**,** dest') = ∃ listchar sval. isVal(dest, sval) & dest' = sval + src;
strval(**string** dest**:** listchar s) **post** isVal(dest, s)
{ scan(nil, dest, dest**:** s) };

$$\textbf{RB:}\ \frac{\begin{array}{l}\forall z\ C(x, z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x, z), Q_C(x, y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x, B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x), y) \Rightarrow Q(x, y)\ );\end{array}}{C(x, B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

B(s0, d, dest) = |nil, dest, dest|
scan(s0, dest**:** s) **corr*** [Pscan(s0, d, dest), isVal(dest, s)];
Str(dest) ⇒ Pscan(nil, dest, dest);
RB: ∀y (Str(dest) & isVal(dest, s) ⇒ isVal(dest, s));
scan(nil, d, dest**:** s) **corr** [Str(dest), isVal(dest, s)]

Цель: scan(s0, dest**:** s) **corr*** [Pscan(s0, d, dest), isVal(dest, s)]

Применяется обобщение задачи

scan(listchar s0, **string** d, dest**:** listchar s)
pre Pscan(s0, d, dest) **post** isVal(dest, s) **measure** len(d)
{ **if** (d.car = zero) s = s0 **else** scan(s0 + d.car, d.cdr**:** s) };

$$\textbf{QC}: \frac{\begin{array}{l}\text{B(x: y) } \textbf{corr} \text{ [P(x) \& E(x), Q(x, y)];}\\ \text{C(x: z) } \textbf{corr} \text{ [P(x) \& } \neg\text{E(x), Q(x, y)]}\end{array}}{\{\textbf{if (}\text{E(x)) B(x}\textbf{:}\text{ y) } \textbf{else} \text{ C(x}\textbf{:}\text{ y)}\} \textbf{ corr} \text{ [P(x), Q(x, y)]}}$$

Конкретизация **QC**.

$$\textbf{QC}: \frac{\begin{array}{l}\text{s = s0 } \textbf{corr} \text{ [Pscan(s0, d, dest) \& d.car = zero, isVal(dest, s)];}\\ \text{scan(s0 + d.car, d.cdr}\textbf{:}\text{ s) } \textbf{corr} \text{ [Pscan(s0, d, dest) \& } \neg \text{ d.car = zero, isVal(dest, s)]}\end{array}}{\begin{array}{l}\{ \textbf{if} \text{ (d.car = zero) s = s0 } \textbf{else} \text{ scan(s0 + d.car, d.cdr}\textbf{:}\text{ s)} \textbf{ corr}\\ \text{[Pscan(s0, d, dest), isVal(dest, s)]}\end{array}}$$

Для первой посылки применяется определение **corr**.

$$\textbf{COR}: \frac{\begin{array}{l}\forall x,y.\ P(x) \ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x\textbf{:}\ y)\end{array}}{H(x\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

lemma COR1: Pscan(s0, d, dest) & d.car = zero & s = s0 $\Rightarrow$ isVal(dest, s)
scan(s0 + d.car, d.cdr**:** s) **corr** [Pscan(s0, d, dest) & $\neg$ d.car = zero, isVal(dest, s)]

formula PSC(listchar s0, **string** d, dest) = Pscan(s0, d, dest) & $\neg$ d.car = zero;

$$\textbf{RB}: \frac{\begin{array}{l}\forall z\ C(x, z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x, z), Q_C(x, y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x, B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x), y) \Rightarrow Q(x, y)\ );\end{array}}{C(x, B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Здесь B(s0,d,dest) = | s0 + d.car, d.cdr, dest |, P_B(s0,d,dest) = cons?(d).

Конкретизация **RB**.

$$\textbf{RB}: \frac{\begin{array}{l}\forall z\ C(x, z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x, z), Q_C(x, y)];\\ PSC(s0, d, dest) \Rightarrow cons?(d)\ \&\ Pscan(s0 + d.car, d.cdr, dest)\ \&\ len(d.cdr)<len(d)\\ \forall y\ (PSC(s0, d, dest)\ \&\ isVal(dest, s) \Rightarrow isVal(dest, s)\ );\end{array}}{scan(s0 + d.car, d.cdr\textbf{:}\ s)\ \textbf{corr}\ [PSC(s0, d, dest), isVal(dest, s)]}$$

lemma RB22: PSC(s0, d, dest) $\Rightarrow$
cons?(d) & Pscan(s0 + d.car, d.cdr, dest) & len(d.cdr)<len(d)

В итоге получаем следующую теорию для доказательства в системе интерактивного доказательства PVS.

```
theory strcat {
char zero = \0;
type listchar = list(char);
formula isVal(listchar s, sval) = s = sval + zero;
formula nozero(listchar sval) = sval = nil or nozero(sval.cdr);
formula Str(listchar s) = ∃ listchar sval. isVal(s, sval) & nozero(sval);
type string = subtype(listchar s: Str(s));
formula Pstrcat(string dest, src) = Str(src) & Str(dest);
formula Qstrcat(string dest, src, dest') = ∃ listchar sval. isVal(dest, sval) & dest' = sval + src;
formula Plistcat(listchar s, string src) = Str(src);
formula Qlistcat(listchar s, string src, dest) = dest = s + src & Str(dest);

lemma QS3: Pstrcat(dest, src) & isVal(dest, s) & Qlistcat(s, src, dest') ⇒ Qstrcat(dest, src, dest');
lemma COR: Str(src) & s1 = s + src.car & src.car = zero & dest = s1 ⇒ Qlistcat(s, src, dest);
lemma RB2: Str(src) & s1 = s + src.car & src.car ≠ zero ⇒ cons?(src) & Str(src.cdr);
lemma RB3: Str(src) & s1 = s + src.car & src.car ≠ zero & Qlistcat(s1, src.cdr, dest) ⇒
                                                                    Qlistcat(s, src, dest);

formula Pscan(listchar s, string d, dest) = Str(d) & Str(dest) & dest = s + d;
formula PSC(listchar s0, string d, dest) = Pscan(s0, d, dest) & ¬ d.car = zero;
lemma COR1: Pscan(s0, d, dest) & d.car = zero & s = s0 ⇒ isVal(dest, s)
lemma RB22: PSC(s0, d, dest) ⇒ cons?(d) & Pscan(s0 + d.car, d.cdr, dest) & len(d.cdr)<len(d)
}
```

Последние четыре строки получены при верификации программы strval.

В этой теории необходимо доказать четыре леммы, соответствующие полученным выше формулам корректности. Спецификации этой теории пришлось существенно модифицировать, чтобы они смогли пройти семантический контроль при трансляции в системе PVS. Ниже приведена итоговая теория на языке PVS.

```
strcat: THEORY BEGIN
  char: TYPE = below[256];
  zero: char = 0;
  listchar: TYPE = list[char];
  s0, s, s1, sval: VAR listchar

  val(s, sval):bool = s = append(sval, cons(zero,null));
  nozero(sval): RECURSIVE bool = CASES sval OF
        null: TRUE, cons(x,y): NOT (x = zero) & nozero(y) ENDCASES MEASURE length(sval);
  Str(s):bool = EXISTS sval: val(s, sval) & nozero(sval);
  string: TYPE = {s:listchar | Str(s)};

  d, dest, dest9, src: VAR string

  Pstrcat(dest, src):bool = Str(src) & Str(dest);
  Qstrcat(dest, src, dest9):bool = EXISTS sval: val(dest, sval) & dest9 = append(sval, src);
  Plistcat(s, src):bool = Str(src);
  Qlistcat(s, src, dest):bool = dest = append(s,src) & Str(dest);
  Pscan(s0, d, dest):bool = Str(d) & Str(dest) & dest = append(s0, d);
  PSC(s0, d, dest):bool = Pscan(s0, d, dest) & NOT (car(d) = zero);

  Notnil:    LEMMA Str(src) IMPLIES cons?(src);
  Empstr:    LEMMA car(src) = zero IMPLIES src = cons(zero, null)
  Notemp:  LEMMA src = append(s, cons(zero, null)) & NOT car(src) = zero IMPLIES cons?(s)
  appCons: LEMMA cons?(src) IMPLIES append(cons(car(src), null), cdr(src)) = src

  QS3: LEMMA  Pstrcat(dest, src) & val(dest, s) & Qlistcat(s, src, dest9)
                                                    IMPLIES  Qstrcat(dest, src, dest9);
  COR: LEMMA  Str(src) & s1 = append(s,cons(car(src),null)) & car(src) = zero & dest = s1
                                                        IMPLIES Qlistcat(s, src, dest);
  RB2: LEMMA  Str(src) & s1 = append(s,cons(car(src),null)) & NOT car(src) = zero
                          IMPLIES cons?(src) & Str(cdr(src)) & length(cdr(src)) < length(src);
  RB3: LEMMA  Str(src) & s1 = append(s,cons(car(src),null)) & NOT car(src) = zero &
                                    Qlistcat(s1, cdr(src), dest)     IMPLIES Qlistcat(s, src, dest);
  COR1: LEMMA  Pscan(s0, d, dest) & car(d) = zero & s = s0 IMPLIES val(dest, s)
  RB22: LEMMA PSC(s0, d, dest) IMPLIES cons?(d) &
                      Pscan(append(s0, cons(car(d), null)), cdr(d), dest) & length(cdr(d))<length(d)
END strcat
```

В этой теории введены дополнительные простые леммы Notnil, Empstr, Notemp и appCons для упрощения доказательства основных формул корректности. Доступно доказательство [5] всех лемм данной теории, включая TCC, в системе PVS:

http://persons.iis.nsk.su/files/persons/pages/strcat.zip