УДК 004.05

# Дедуктивная верификация программы конкатенации строк с применением обратной трансформации

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

Дедуктивная верификация 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.

В данной работе на примере программы конкатенации строк исследуется метод дедуктивной верификации с применением обратной трансформации от программы на языке Си к эквивалентной предикатной программе. Обратная трансформация состоит из двух частей. В первой части реализуются трансформации упрощения с переводом программы на тот же язык Си без указателей. Верификация трансформированной программы конкатенации строк проведена в системах доказательства Why3 и Coq.

***Ключевые слова:*** *дедуктивная верификация, трансформации программ, предикатное программирование, автоматическое доказательство, формальная семантика*

## 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]. Указанное преимущество обусловлено близостью языка предикатного программирования Р [8] к языку математической логики.

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.

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.

Технология построения предикатных программ по исходным императивным программам описывается в работах [1-3]. Во всех случаях не было точного совпадения с исходной императивной программой после применения трансформаций. Хотя эквивалентность трансформированной и исходной программ была очевидна.

В настоящей работе исследуется метод обратной трансформации от программы на языке Си к эквивалентной предикатной программе. Трансформации, используемые в предикатном программировании, применяются в обратном направлении. Оказывается, это принципиально новая технология, требующая систематического исследования.

Метод обратной трансформации иллюстрируется на программе конкатенации строк strcat, входящей в стандартную библиотеку ядра ОС Linux. На этой программе впервые был опробован метод прямой трансформации [1].

На первом этапе обратной трансформации устраняются арифметические операции с указателями. Доступ через указатели заменяются явными обращениями к элементам массивов. Получается другая более простая программа на языке Си. Далее циклы заменяются рекурсивными программами. Получается эквивалентная предикатная программа. Для нее пишутся спецификации в виде набора предусловий и постусловий. Проводится дедуктивная верификация. Построение формул корректности описано в Приложении 1. Доказательство формул корректности проводилось в системах Why3 [26] и Coq [5].

# 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 **array**($T_e$, $T_i$) является *массив* с *элементами массива* типа $T_e$ и *индексами* конечного типа $T_i$. Присваивание элементу массива вида a[i] = x реализуется оператором a′ = a **with** (i**:** x) через операцию модификации **with**.

# 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 $\mathbf{P_0}$ построена формальная операционная семантика $\mathcal{R}(H)(x, y)$ и доказано тождество $\mathcal{R}(H) = H$ [17]. На базе языка $\mathbf{P_0}$ последовательным расширением и сохранением тождества $\mathcal{R}(H) = H$ 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**:

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

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 дедуктивной верификации опробован на многих программах [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.

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

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

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

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

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

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

В случае рекурсивного вызова C(x, B(x)**:** 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. Обратная трансформация программы конкатенации строк

## 4.1. Постановка задачи

Имеется следующая программа конкатенации строк на языке Си:

```
char *strcat(char *dest, const char *src)                    (1)
{
        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 трансформировать данную программу в эквивалентную предикатную программу и провести ее дедуктивную верификацию. Здесь рассматриваются обратные трансформации по сравнению с обычными (прямыми) оптимизирующими трансформациями, применяемыми в предикатном программировании.

## 4.2. Устранение адресной арифметики

Предварительно необходимо преобразовать программу к явному виду раскрытием умолчаний. В частности, заменим вхождения «++» эквивалентными действиями.

```
char *strcat(char *dest, const char *src)                    (2)
{
        char *tmp = dest;
        while (*dest != '\0') dest = dest + 1;
        const char *td, *ts;
        loop {  td = dest; dest = dest + 1;
                ts = src; src = src + 1;
                *td =*ts;
                if (*ts == '\0') break;
        };
        return tmp;
}
```

На первом этапе трансформаций реализуется преобразование структур данных программы в целях устранения адресной арифметики. Сначала проводится анализ внешних и внутренних структур данных.

Указатели dest, src и tmp определяют внешние структуры данных. Пусть Dest – массив, доступный по указателю dest. Пусть массив Src соответствует указателю src. Указатель tmp равен указателю dest, значение которого определено в начале программы. Поэтому tmp указывает на массив Dest, модифицированный программой strcat. Обозначим его через Dest′.

Введем переменную j – индекс элемента массива Dest, на который указывает dest перед исполнением оператора dest = dest + 1. После срабатывания оператора указатель dest будет соответствовать элементу с индексом j+1. Аналогично, k будет обозначать индекс очередного элемента в массиве Src.

В момент исполнения оператора *td =*ts указатель td равен dest после присваивания td = dest. Однако между этими операторами указатель dest модифицируется. Сохраним значения индекса j после оператора td = dest вставкой оператора j0 = j. Тогда указателю td будет соответствовать элемент с индексом j0.

Определим обратные трансформации:

*dest → dest[j];
*src → src[k];
*td → dest[j0];
*ts → src[k0];
dest = dest + 1 → j = j + 1;
src = src + 1 → k = k + 1.

Применение обратных трансформаций дает следующую программу:

```
char *strcat(char *dest, const char *src)                    (3)
{
        char*  tmp = dest;
        nat j=0, k =0;
        while (dest[j] != '\0') j = j+1;
        const char *td, *ts;
        nat j0, k0;
        loop {  td = dest; j0=j; j = j+1;
                ts = src; k0=k; k = k+1;
                dest[j0] =src[k0];
                if (src[k0] == '\0') break;
        };
        return tmp;
}
```

Указатели tmp, td и ts становятся ненужными. Все действия с ними можно удалить из программы. Чтобы упростить второй цикл, перенесем операторы j = j+1 и k = k+1 в конец тела цикла. Данный перенос не является эквивалентным преобразованием. Однако внутри цикла j и k остаются теми же. А вне цикла j и k не используются.

```
char *strcat(char *dest, const char *src)                    (4)
{
        nat j=0, k =0;
        while (dest[j] != '\0') j = j+1;
        loop {  dest[j] = src[k];
                if (src[k] == '\0') break;
                j = j+1; k = k+1;
        };
        return dest;
}
```

Полученная программа остается в рамках языка Си. И она намного проще для дедуктивной верификации.

Трансформация переноса операторов j = j+1 и k = k+1 не является обязательной. В принципе можно было бы проводить верификацию предыдущей версии программы. Однако данная трансформация удобна для преобразования цикла в рекурсивную программу в дальнейшей трансформации.

## 4.3. Трансформация императивной программы в предикатную

Определим структуры данных в терминах языка P [8]. Для массивов dest и src нет ограничений на длину. Будем использовать тип неограниченного массива литер, индексами которого являются все натуральный числа:

```
type arCh = array (char, nat);
```

Параметры программы strcat, массивы dest и src, удобнее определить глобальными переменными.

```
arCh dest, src;
```

Циклы будут преобразованы в рекурсивные программы. Поэтому их необходимо вынести из программы strcat. Получаем следующую программу.

```
strcat( : arCh dest')
{       scan( : nat j);
        cat(j: dest')
}
```

Программа scan вычисляет индекс j, для которого dest[j]= '\0'. Программа cat реализует конкатенацию массивов. Результатом является массив dest'.

Далее циклы преобразуются в рекурсивные программы. Параметр цикла становится аргументом рекурсивной программы. Параметр цикла – дополнительный параметр для каждой из определенных выше программ scan и cat. Начальное значение каждого параметра – нулевое. Получим декомпозицию вида:

```
scan( : nat j) { scan(0:  j) };
cat(nat j: arCh dest'){ cat(dest, j, 0: dest') };
```

Чтобы сократить число уровней декомпозиции программы, проведем открытую подстановку данных программ в программу strcat.

```
arCh dest, src;
strcat( : arCh dest')
{       scan(0: nat j);
        cat(dest, j, 0: dest')
}
```

Представим оставшуюся часть программы с заменой цикла **while** на **loop**.

```
char zero = '\0';
scan(nat j: nat j')
{       loop { if (sest[j] = zero) break; j = j+1}; }

cat(arCh dest, nat j, k: arCh dest')
        loop {  dest[j] = src[k];
                if (src[k] = zero) break;
                j = j+1; k = k+1;
```

```
        };
}
```

Заменим циклы рекурсивными программами. Присваивание элементу массива оформим через операцию модификации.

```
scan(nat j: nat j')
{    if (dest[j] = zero) j' = j else scan(j+1: j')   }

cat(arCh dest, nat j, k: arCh dest')
        arCh dest1 = dest with (j: src[k]);
        if (src[k] = zero) dest' = dest1 else cat(dest1, j+1, k+1: dest')
}
```

# 5. Спецификация предикатной программы strcat

Для проведения дедуктивной верификации предикатной программы необходимо построить спецификации, в виде предусловий и постусловий, для программ strcat, scan и cat.

Для произвольного массива a предикат str(a) определяет условие того, что a есть строка: строка завершается нулем, причем все предыдущие элементы отличны от нуля.

```
char zero = '\0';
type arCh = array (char, nat);
formula str(arCh a) = ∃ nat m. strm(a, m);
formula strm(arCh a, nat m) = a[m] = zero & ∀ k=0..m-1. a[k] != zero;
```

Программа strcat реализует конкатенацию строк dest и src, которые определены глобальными переменными. Константы l_dest и l_src обозначают длины этих строк, используемые при построении функций меры.

```
arCh dest, src;
nat l_dest;
axiom strm(dest, l_dest);
nat l_src;
axiom strm(src, l_src);
```

Постусловие qstrcat есть утверждение того, что строка dest' является конкатенацией строк dest и src.

```
formula qstrcat(arCh dest, src, dest') =
    ∃ nat m, n. strm(dest, m) & strm(src, n) &
                (∀ nat j. 0<=j<m ⇒ dest'[j]=dest[j]) &
                (∀ nat k. 0<=k<=n  ⇒ dest'[m+k]=src[k]) ;
```

В предусловии программы strcat утверждается, что массивы dest и src являются строками.

```
strcat( : arCh dest')
pre str(dest) & str(src) post qstrcat(dest, src, dest')
{       scan(0: nat j);
        cat(dest, j, 0: dest')
}
```

Предикат nozero(a, j) определяет условие того, что все элементы массива a с индексами, меньшими j, отличны от нуля.

```
formula nozero(arCh a, nat j) = ∀ nat k<j. a[k]≠zero;
```

Предусловие программы scan определяет, что dest – строка, элементы которой с индексами, меньшими j, отличны от нуля. Постусловие strm(dest, j′) есть утверждение: строка dest завершается нулевым элементом с индексом j′. Хотя здесь было бы достаточно условия dest[j′] = zero. Мера l_dest + 1 - j используется при построении формул корректности рекурсивной программы scan.

```
scan(nat j: nat j′)
pre str(dest) & nozero(dest, j) post strm(dest, j′) measure l_dest + 1 - j
{    if (dest[j] = zero) j′ = j else scan(j+1: j′)   }
```

Постусловие qcat есть утверждение того, что строка dest′ совпадает с dest для элементов с индексами меньше j. Оставшаяся часть строки dest′, начиная с j–го элемента, совпадает с вырезкой строки src от k–го элемента.

```
formula qcat(arCh dest, src, dest′, nat j, k) =
    ∃ nat n. strm(src, n) &
                (∀ nat i. 0<=i<j ⇒ dest′[i]=dest[i]) &
                (∀ nat p. k<=p<=n  ⇒ dest′[j+p-k]=src[p]) ;
```

Предусловие программы cat определяет, что src – строка, элементы которой с индексами, меньшими k, отличны от нуля. В постусловии утверждается, что строка dest′ есть дополнение dest вырезкой src[k..]. Используется мера l_src + 1 - k.

```
cat(arCh dest, nat j, k: arCh dest′)
pre str(src) & nozero(src, k) post qcat(dest, src, dest′, j, k) measure l_src + 1 - k
       arCh dest1 = dest with (j: src[k]);
       if (src[k] = zero) dest′ = dest1 else cat(dest1, j+1, k+1: dest′)
}
```

# 6. Процесс дедуктивной верификации

Построение формул корректности для предикатной программы конкатенации строк подробно документируется в Приложении 1. Формулы корректности представлены теорией:

```
theory Strcat {
 char zero = '\0';
 type arCh = array (char, nat);
 formula strm(arCh a, nat m) = a[m] = zero & ∀ k=0..m-1. a[k] != zero;
 arCh dest, src;
 nat l_dest, l_src;
 axiom strm(dest, l_dest);
 axiom strm(src, l_src);
 formula str(arCh a) = ∃ nat m. strm(a, m);
 formula qstrcat(arCh dest, src, dest′) =
    ∃ nat m, n. strm(dest, m) & strm(src, n) &
                (∀ nat j. 0<=j<m ⇒ dest′[j]=dest[j]) &
                (∀ nat k. 0<=k<=n  ⇒ dest′[m+k]=src[k]) ;
```

**formula** nozero(arCh a, **nat** j) = ∀ **nat** k<j. a[k]≠zero;

**formula** qcat(arCh dest, src, dest', **nat** j, k) =
∃ **nat** n. strm(src, n) &
(∀ **nat** i. 0<=i<j ⇒ dest'[i]=dest[i]) &
(∀ **nat** p. k<=p<=n ⇒ dest'[j+p-k]=src[p]) ;
**formula** pstrcat () = str(dest) & str(src);
**formula** pscan (j) = str(dest) & nozero(dest, j);
**formula** pcat (k) = str(src) & nozero(src, k);
**lemma RS1:** pstrcat () & strm(dest, j) & qcat(dest, src, dest', j, 0) ⇒ qstrcat(dest, src, dest');
**lemma RS2:** pstrcat () & strm(dest, j) & qcat(dest, src, dest', j, 0) ⇒ qstrcat(dest, src, dest');
**lemma RS3:** pstrcat ()⇒ pscan (0)
**lemma COR1**: pscan (j) & dest[j] = zero & j' = j ⇒ strm(dest, j');
**lemma RB1:** pscan (j) & dest[j] != zero ⇒
pscan (j) & l_dest - (j+1) < l_dest - j & l_dest - (j+1)>=0;
**formula** wd(dest1, dest, j, k) = dest1[j]=src[k] & ∀nat p. p != j ⇒ dest1[p]=dest[p];
**lemma COR2:** pcat(k) & wd(dest1, dest, j, k) & src[k] = zero & dest' = dest1 ⇒
qcat(dest, src, dest', j, k);
**lemma RB2:** pcat(k) & wd(dest1, dest, j, k) & src[k] != zero ⇒
pcat (k+1) & l_src - (k+1) < l_src - k & l_src - (k+1)>=0;
**lemma RB3:** pcat(k) & wd(dest1, dest, j, k) & src[k] != zero &
qcat(dest1, src, dest', j+1, k+1) ⇒ qcat(dest, src, dest', j, k);
}

Данная теория была переведена на язык спецификаций Why3 [26]. Доказательство формул корректности, представленных в виде 8 целей теории Strcat, проводилось в системах Why3 [26] и Coq [5].

Обнаружено две ошибки в спецификациях. Одна в формуле nozero. Другая – неточная функции меры. Введены дополнительные леммы:

**lemma** LeZero: **forall** j m: **int**, a: arCh. strm a m /\ nozero a j -> j <= m

**lemma** NoZn: **forall** k: **int**, a: arCh. nozero a k /\ a[k] <> zero -> nozero a (k+1)

Трижды доказательство переводилось в систему Coq, где ошибки были быстро локализованы.

После исправления ошибок формулы корректности были полностью доказаны в системе Why3 без использования системы Coq.

Построение формул корректности (Приложение 1), перевод на язык спецификаций Why3, доказательство в системах Why3 и Coq проводилось в течение двух дней.

# 7. Обзор работ

Обратные трансформации обычно являются частью обратной трансляции на язык более высокого уровня. Для таких языков более типична прямая трансляция на язык Си, в частности, для переносимости на другие платформы. Трансляция с языка Си на другие языки реализуется в целях миграции кода обширных библиотек и программ на языке Си.

Возможно, первым успешным транслятором с языка Си был транслятор на язык Turbo Pascal. В это же время разрабатывался транслятор на язык Ada. Обширный обзор работ по обратной трансляции представлен в [9]. Обратная трансляция применялась также в проектах по миграции кода со старых языков типа COBOL. Мы должны исключить из рассмотрения трансляцию на язык Verilog [25], поскольку по отношению к нему язык Си является языком более высокого уровня.

В результате раскрытия инкрементов и декрементов (операций ++ и --), находящихся внутри выражений, в программе появляются громоздкие участки кода, затрудняющие понимание программы и вызывающие раздражение пользователей. Например, в нашем случае это результат раскрытия ++ в программах (2) и (3) в Разд. 4.2. В подобных ситуациях в работе [23] применяются трансформации, улучшающие структуру программы. В других работах предлагается использовать прагмы, подсказывающие транслятору шаблоны адекватных преобразований программ.

Полностью автоматическая трансляция для полного языка Си, включая расширения GNU C, реализована только в трансляторе C2Eif [12] на язык Eifell. Использование промежуточного языка CIL [13] (C intermediate language) существенно упростило реализацию транслятора. Остальные трансляторы с языка Си не автоматические и требуют модификации кода, полученного при трансляции.

Адресная арифметика языка Си воспринимается лишь трансляторами C2J [21] на язык Java и C2Eif [12]. Чтобы обеспечить это, в языке Eifell была введена адресная арифметика, а трансляторе C2J все данные программы помещаются в один массив. Замена адресной арифметики элементами массива заявлена в работе [16], описывающей транслятор Ephedra на язык Java, однако обзор работы [12] этого не подтверждает.

# 8. Предположения об архитектуре обратных трансформаций

Обратная трансформация, реализуемая для программы на языке Си в целях проведения дедуктивной верификации трансформированной программы, складывается из двух частей. Первая часть состоит из трансформаций упрощения. Во второй части реализуется перевод на язык предикатного программирования. В реализации соответствующего инструмента удобно будет использовать промежуточный язык CIL [13]. Результат первой части – упрощенная программа на языке Си. Потребуется некоторое расширение языка Си. Например, при обратной трансформации программ бинарного поиска fsearch.c [2] и пирамидальной сортировки sort.c [3] необходимо будет введение произвольных типов, операции для которых задаются внешними функциями. После первой части обратной трансформации в программе нет указателей. Указатели в качестве параметров программ должны быть заменены объектами, на которые ссылаются указатели. Очевидно, такая программа станет неэффективной. Однако она предназначена только для использования в дедуктивной верификации.

Трансформация устранения арифметики указателей является наиболее сложно реализуемой из всех трансформаций первой части. Внутри каждой программы (функции) на языке Си требуется предварительный анализ информационных связей и значений указателей. На базе этого анализа вводятся дополнительные индексные переменные для замены арифметических действий с указателями. Доступ через указатели заменяются явными обращениями к элементам массивов.

Раскрытие инкрементов и декрементов внутри выражений может привести к введению дополнительных промежуточных переменных. Как следствие, усложняется программа. Плохие участки кода упрощаются применением эквивалентных трансформаций. Отметим, что применение данных трансформаций не обязательно. Можно было бы не делать переноса

операторов j = j+1 и k = k+1 в программе (3), ограничиться лишь удалением действий с указателями и получить программу:

```
char *strcat(char *dest, const char *src)                    (3')
{
        nat j=0, k =0;
        while (dest[j] != '\0') j = j+1;
        loop {  j0=j; j = j+1; k0=k; k = k+1;
                dest[j0] =src[k0];
                if (src[k0] = '\0') break;
        };
        return dest;
}
```

И далее работать с этой программой. Программу (3') можно было бы трансформировать в предикатную и проводить ее дедуктивную верификацию. Разумеется, дедуктивная верификация будет сложнее.

Трансформацию, обратную уменьшению силы операций с заменой в циклах умножения на сложение, можно проводить до устранения арифметики указателей. А можно и после с одновременным преобразованием байтового массива в массив элементов заданного размера size.

Полезность обратных трансформаций можно оценить по сложности дедуктивной верификации, которая складывается из сложности спецификации и сложности сгенерированных формул корректности. Такая оценка была бы показательной для трех программ: исходной программы (1), программы (4) после первой части обратных трансформаций и конечной предикатной программы.

## 9. Заключение

Трансформации, используемые в предикатном программировании, могут применяться в обратном направлении от программы на языке Си к эквивалентной предикатной программе. Однако алгоритмы и методы применения обратных трансформаций совсем другие и требуют систематического исследования.

Для практического применения дедуктивной верификации трансформированной программы требуется разработка сертифицированного обратного компилятора с языка Си. Причем сначала разработка его первой части на тот же самый язык Си без указателей с типами данных предположительно как в языке спецификаций Why3 [26].

Идея обратной трансформации сформулирована по опыту прямой трансформации всего трех программ [1-3]. Данного материала недостаточно для определения полного набора обратных трансформаций и точной спецификации обратного компилятора. Алгоритмы прямой трансформации [6] здесь вряд ли пригодятся.

## Список литературы

1. Шелехов В.И. Дедуктивная верификация и оптимизация предикатной программы конкатенации строк. *Системная информатика*, № 12. Новосибирск, 2018. С. 61-84. http://persons.iis.nsk.su/files/persons/pages/strcat.pdf

2. Шелехов В.И. Верификация предикатной программы бинарного поиска объекта произвольного типа. Новосибирск, 2019. 26с. http://persons.iis.nsk.su/files/persons/pages/fsearch2.pdf

3. Шелехов В.И. Верификация предикатной программы пирамидальной сортировки — Новосибирск, 2019. 36с.
4. Доказательство правил корректности операторов предикатной программы. http://www.iis.nsk.su/persons/vshel/files/rules.zip
5. The Coq Proof Assistant. https://coq.inria.fr/
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. http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf
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 Pr. Версия 0.12. Новосибирск, 2013. 28с. http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
9. Plaisted D. Source-to-Source Translation and Software Engineering. *J. of Software Engineering and Applications*, Vol. 6 No. 4A, 2013, P. 30-40.
10. Чушкин М.С. Система дедуктивной верификации предикатных программ. *Программная инженерия*. 2016. № 5. С. 202-210. http://persons.iis.nsk.su/files/persons/pages/paper.pdf.
11. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования. *Программная инженерия*, 2011, № 2. С. 14-21.
12. Trudel M., Furia C. A., Nordio M., Meyer B., Oriol M. C to O-O Translation: Beyond the Easy Stuff. *19th Working Conference on Reverse Engineering*, 2012, P. 19-28.
13. Necula G. C., McPeak S., Rahul S., Weimer W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Compilier Construction*, 2002, P. 213–228.
14. Шелехов В.И. Классификация программ, ориентированная на технологию программирования. *Программная инженерия*, Том 7, № 12, 2016. С. 531–538. http://persons.iis.nsk.su/files/persons/pages/prog.pdf.
15. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
16. Martin J., M¨uller H. A. Strategies for migration from C to Java. *CSMR*. IEEE Computer Society, 2001, P. 200–210.
17. Шелехов В.И. Семантика языка предикатного программирования. *ЗОНТ-15*. Новосибирск, 2015. 13с. http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf.
18. Шелехов В.И. Синтез операторов предикатной программы. *Труды конф. «Языки программирования и компиляторы ’2017»*, Ростов-на-Дону. 2017. С.258-262. http://persons.iis.nsk.su/files/persons/pages/sintr.pdf.
19. Шелехов В.И. Списки и строки в предикатном программировании. *Системная информатика*, ИСИ СО РАН, Новосибирск. Электрон. журн. 2014, №3. С. 25-43. http://persons.iis.nsk.su/files/persons/pages/Listcharing.pdf.

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. http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf.

21. Novosoft C2J: a C to Java translator. http://www.novosoft-us.com/ solutions/product c2j.shtml, 2001.

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. Kennedy T.R. Using program transformations to improve program translation. *Massachusetts Institute of Technology*. AI Memo No.962, 1897. 38p.

24. PVS Specification and Verification System. *SRI International*. http://pvs.csl.sri.com/ .

25. Leung A., Bounov D., Lerner S. C-to-Verilog translation validation. *ACM/IEEE International Conference on Formal Methods and Models for Codesign*. 2015. P.42-47

26. Why 3. Where Programs Meet Provers. URL: http://why3.lri.fr

.

# Приложение 1.
# Построение формул корректности программы strcat

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

```
char zero = '\0';
type arCh = array (char, nat);
arCh dest, src;

formula strm(arCh a, nat m) = a[m] = zero & ∀ k=0..m-1. a[k] != zero;
nat l_dest;
axiom strm(dest, l_dest);
nat l_src;
axiom strm(src, l_src);

formula str(arCh a) = ∃ nat m. strm(a, m);
formula qstrcat(arCh dest, src, dest') =
    ∃ nat m, n. strm(dest, m) & strm(src, n) &
                (∀ nat j. 0<=j<m ⇒ dest'[j]=dest[j]) &
                (∀ nat k. 0<=k<=n  ⇒ dest'[m+k]=src[k]) ;

strcat( : arCh dest')
pre str(dest) & str(src) post qstrcat(dest, src, dest')
{       scan(0: nat j);
        cat(dest, j, 0: dest')
}

function len(arCh a: nat) = {nat m | strm(a, m)};
formula nozero(arCh a, nat j) = ∀ nat k<j. a[k]≠zero;

scan(nat j: nat j')
pre str(dest) & nozero(dest, j) post strm(dest, j') measure len(dest) - j
{    if (dest[j] = zero) j' = j else scan(j+1: j')   }

formula qcat(arCh dest, src, dest', nat j, k) =
    ∃ nat n. strm(src, n) &
                (∀ nat i. 0<=i<j ⇒ dest'[i]=dest[i])) &
                (∀ nat p. k<=p<=n  ⇒ dest'[j+p-k]=src[p])) ;

cat(arCh dest, nat j, k: arCh dest')
pre str(src) & nozero(src, k) post qcat(dest, src, dest', j, k) measure len(src) - k
        arCh dest1 = dest with (j: src[k]);
        if (src[k] = zero) dest' = dest1 else cat(dest1, j+1, k+1: dest')
}
```

---

```
strcat(arCh dest : arCh dest')
pre str(dest) & str(src) post qstrcat(dest, src, dest')
{       scan(0: nat j);
        cat(dest, j, 0: dest')
}

formula pstrcat () = str(dest) & str(src);
formula pscan (j) = str(dest) & nozero(dest, j);
formula pcat (k) = str(src) & nozero(src, k);
```

Сначала применяется правило **RS** для оператора суперпозиции scan(0**:** **nat** j); cat(dest, j, 0**:** dest'), составляющего тело программы strcat:

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

Здесь B(x: z) соответствует scan(0**:** j), а C(x, z**:** y) – cat(dest, j, 0**:** dest'). Ниже конкретизация правила **RS**.

$$\textbf{RS}:\ \frac{\begin{array}{l} \text{scan}(0\textbf{:}\ j)\ \textbf{corr}^*\ [\text{pscan}\ (0), \text{strm}(\text{dest}, j)];\\ \text{cat}(\text{dest}, j, 0\textbf{:}\ \text{dest}')\ \textbf{corr}^*\ [\text{pcat}\ (0), \text{qcat}(\text{dest}, \text{src}, \text{dest}', j, 0)];\\ \text{pstrcat}\ () \ \&\ \text{strm}(\text{dest}, j)\ \&\ \text{qcat}(\text{dest}, \text{src}, \text{dest}', j, 0) \rightarrow \text{qstrcat}(\text{dest}, \text{src}, \text{dest}')\\ \text{pstrcat}\ () \ \&\ \text{strm}(\text{dest}, j) \rightarrow\ \text{pcat}\ (0);\\ \text{pstrcat}\ () \rightarrow\ \text{pscan}\ (0); \end{array}}{\text{scan}(0\textbf{:}\ j);\ \text{cat}(\text{dest}, j, 0\textbf{:}\ \text{dest}')\ \textbf{corr}\ [\text{pstrcat}\ (), \text{qstrcat}(\text{dest}, \text{src}, \text{dest}')]}$$

Первая и вторая посылки определяют корректность программ scan и cat. Посылки с третьей по пятую определяют следующие формулы корректности:

**RS1:** pstrcat () & strm(dest, j) & qcat(dest, src, dest', j, 0) ⇒ qstrcat(dest, src, dest');
**RS2:** pstrcat () & strm(dest, j) & qcat(dest, src, dest', j, 0) ⇒ qstrcat(dest, src, dest');
**RS3:** pstrcat ()⇒ pscan (0)

Рассмотрим программу scan и построим для нее формулы корректности. Задание меры len(dest) - j обязательно для рекурсивных программ.

```
scan(nat j: nat j')
pre str(dest) & nozero(dest, j) post strm(dest, j') measure l_dest - j
{    if (dest[j] = zero) j' = j else scan(j+1: j')   }
```

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**:

$$\textbf{QC:}\ \frac{\begin{array}{l} j' = j\ \textbf{corr}\ [pscan\ (j)\ \&\ dest[j] = zero,\ strm(dest,\ j')];\\ scan(j+1\textbf{:}\ j')\ \textbf{corr}\ [pscan\ (j)\ \&\ dest[j] \mathrel{!}= zero,\ strm(dest,\ j')] \end{array}}{\textbf{if}\ (dest[j] = zero)\ j' = j\ \textbf{else}\ scan(j+1\textbf{:}\ j')\ \textbf{corr}\ [pscan\ (j),\ strm(dest,\ j')]}$$

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

$j' = j\ \textbf{corr}\ [pscan\ (j)\ \&\ dest[j] = zero,\ strm(dest,\ j')];$
$scan(j+1\textbf{:}\ j')\ \textbf{corr}\ [pscan\ (j)\ \&\ dest[j] \mathrel{!}= zero,\ strm(dest,\ j')]$

Для первой цели применяется правило **COR**.

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

Конкретизация правила **COR**:

$$\textbf{COR:}\ \frac{\begin{array}{l} pscan\ (j)\ \&\ dest[j] = zero\ \&\ j' = j \Rightarrow strm(dest,\ j');\\ pscan\ (j)\ \&\ dest[j] = zero\ \Rightarrow \exists j'.\ j' = j \end{array}}{j' = j\ \textbf{corr}\ [pscan\ (j)\ \&\ dest[j] = zero,\ strm(dest,\ j')]}$$

Первая посылка правила определяет следующую формулу корректности:

**COR1**: $pscan\ (j)\ \&\ dest[j] = zero\ \&\ j' = j \Rightarrow strm(dest,\ j');$

Для второй посылки правила **QC** применяется 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 $B(j) = j+1$, предикат $P_B$ = **true**. Первая посылка правила **RB** для программы scan отсутствует ввиду рекурсивности scan. Предикат $P^*_C(x,\ B(x))$ определяется следующим образом: $P^*_C(B(j)) = P_C(B(j))\ \&\ l\_dest - (j+1) < l\_dest - j\ \&\ l\_dest - (j+1) >= 0$.

Конкретизация правила **RB**:

$$\textbf{RB:}\ \frac{\begin{array}{l} scan(\textbf{j:}\ j')\ \textbf{corr}^*\ [pscan\ (j),\ strm(dest,\ j')];\\ pscan\ (j)\ \&\ dest[j] \mathrel{!}= zero \Rightarrow pscan\ (j)\ \&\ l\_dest - (j+1) < l\_dest - j\ \&\ l\_dest - (j+1) >= 0;\\ pscan\ (j)\ \&\ dest[j] \mathrel{!}= zero\ \&\ strm(dest,\ j') \Rightarrow strm(dest,\ j'); \end{array}}{scan(j+1\textbf{:}\ j')\ \textbf{corr}\ [pscan\ (j)\ \&\ dest[j] \mathrel{!}= zero,\ strm(dest,\ j')]}$$

Вторая посылка правила **RB** определяет следующую формулу корректности:

**RB1:** $pscan\ (j)\ \&\ dest[j] \mathrel{!}= zero \Rightarrow pscan\ (j)\ \&\ l\_dest - (j+1) < l\_dest - j\ \&\ l\_dest - (j+1) >= 0;$

Рассмотрим программу cat и построим для нее формулы корректности.

cat(arCh dest, **nat** j, k**:** arCh dest')
**pre** str(src) & nozero(src, k) **post** qcat(dest, src, dest', j, k) **measure** len(src) - k
arCh dest1 = dest **with** (j**:** src[k]);

**if** (src[k] = zero) dest′ = dest1 **else** cat(dest1, j+1, k+1**:** dest′)
}

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

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

Приведем конкретизацию данного правила **QS**.

$$\textbf{QS}:\ \frac{\begin{array}{l}pcat(k) \to \exists\ dest1.\ dest1 = dest\ \textbf{with}\ (j\textbf{:}\ src[k]);\\ \textbf{if}\ \ldots\ \textbf{corr}\ [pcat(k)\ \&\ dest1 = dest\ \textbf{with}\ (j\textbf{:}\ src[k]), qcat(dest, src, dest', j, k)];\end{array}}{\begin{array}{l}dest1 = dest\ \textbf{with}\ [j\textbf{:}\ src[k]];\ \textbf{if}\ \ldots\ \textbf{corr}\ [pcat(k), qcat(dest, src, dest', j, k)]\end{array}}$$

Вторая посылка правила **QS** определяют следующую цель.

**if** (src[k] = zero) dest′ = dest1 **else** cat(dest1, j+1, k+1**:** dest′) **corr**
[pcat(k) & dest1 = dest **with** (j**:** src[k]), qcat(dest, src, dest′, j, k)];

Применим правило **QC** для условного оператора.

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

Конкретизация правила **QC**:

$$\textbf{QC}:\ \frac{\begin{array}{l}dest' = dest1\ \textbf{corr}\ [pcat(k)\ \&\ dest1 = dest\ \textbf{with}\ (j\textbf{:}\ src[k])\ \&\ src[k] = zero, qcat(dest, src, dest', j, k)];\\ cat(dest1, j+1, k+1\textbf{:}\ dest')\ \textbf{corr}\\ [pcat(k)\ \&\ dest1 = dest\ \textbf{with}\ (j\textbf{:}\ src[k])\ \&\ src[k]\ !=\ zero, qcat(dest, src, dest', j, k)]\end{array}}{\begin{array}{l}\textbf{if}\ (src[k] = zero)\ dest' = dest1\ \textbf{else}\ cat(dest1, j+1, k+1\textbf{:}\ dest')\ \textbf{corr}\\ [pcat(k)\ \&\ dest1 = dest\ \textbf{with}\ (j\textbf{:}\ src[k]), qcat(dest, src, dest', j, k)]\end{array}}$$

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

dest′ = dest1 **corr** [pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] = zero, qcat(dest, src, dest′, j, k)];
cat(dest1, j+1, k+1**:** dest′) **corr**
[pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] != zero, qcat(dest, src, dest′, j, k)]

Для первой цели применяется правило **COR**.

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

Конкретизация правила **COR**:

pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] = zero & dest′ = dest1 ⇒
qcat(dest, src, dest′, j, k);

**COR:** pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] = zero ⇒ ∃ dest′. dest′ = dest1

dest′ = dest1 **corr**
[pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] = zero, qcat(dest, src, dest′, j, k)]

Первая посылка правила определяет формулу корректности:

**COR2:** pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] = zero & dest′ = dest1 ⇒
qcat(dest, src, dest′, j, k);

Для второй посылки правила **QC** 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(dest, j, k) = (dest1, j+1, k+1), предикат $P_B$ = **true**. Первая посылка правила **RB** для программы car отсутствует ввиду рекурсивности listcar. Предикат $P^*_C(x, B(x))$ определяется следующим образом:

$P^*_C(B(x)) = P_C(B(\text{dest1}, j+1, k+1))$ & l_src - (k+1) < l_src - k & l_src - (k+1)>=0.

Конкретизация правила **RB**:

cat(dest, j, k**:** dest′) **corr**$^*$ [pcat (k), qcat(dest, src, dest′, j, k)];
pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] != zero ⇒
pcat (k+1) & l_src - (k+1) < l_src - k & l_src - (k+1)>=0;
pcat(k) & dest1 = dest **with** [j**:** src[k]] & src[k] != zero & qcat(dest1, src, dest′, j+1, k+1) ⇒

**RB:** qcat(dest, src, dest′, j, k);

cat(dest1, j+1, k+1**:** dest′) **corr**
[pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] != zero, qcat(dest, src, dest′, j, k)]

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

**RB2:** pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] != zero ⇒
pcat (k+1) & l_src - (k+1) < l_src - k & l_src - (k+1)>=0;
**RB3:** pcat(k) & dest1 = dest **with** (j**:** src[k]) & src[k] != zero &
qcat(dest1, src, dest′, j+1, k+1) ⇒ qcat(dest, src, dest′, j, k);

Оператор dest1 = dest **with** (j**:** src[k]) определим в виде формулы:

**formula** wd(dest1, dest, j, k) = dest1[j]=src[k] & ∀nat p. p != j ⇒ dest1[p]=dest[p];
==========

В итоге получаем следующую теорию на языке P для доказательства в системе верификации Why3 и в системе интерактивного доказательства Coq.

```
theory strcat {
  char zero = '\0';
  type arCh = array (char, nat);
  formula strm(arCh a, nat m) = a[m] = zero & ∀ k=0..m-1. a[k] != zero;
  arCh dest, src;
  nat l_dest, l_src;
  axiom strm(dest, l_dest);
  axiom strm(src, l_src);
  formula str(arCh a) = ∃ nat m. strm(a, m);
  formula qstrcat(arCh dest, src, dest') =
      ∃ nat m, n. strm(dest, m) & strm(src, n) &
                  (∀ nat j. 0<=j<m ⇒ dest'[j]=dest[j]) &
                  (∀ nat k. 0<=k<=n  ⇒ dest'[m+k]=src[k]) ;
  formula nozero(arCh a, nat j) = ∀ nat k<j. a[k]≠zero;
  formula qcat(arCh dest, src, dest', nat j, k) =
      ∃ nat n. strm(src, n) &
                  (∀ nat i. 0<=i<j ⇒ dest'[i]=dest[i]) &
                  (∀ nat p. k<=p<=n  ⇒ dest'[j+p-k]=src[p]) ;
  formula pstrcat () = str(dest) & str(src);
  formula pscan (j) = str(dest) & nozero(dest, j);
  formula pcat (k) = str(src) & nozero(src, k);
  lemma RS1: pstrcat () & strm(dest, j) & qcat(dest, src, dest', j, 0) ⇒ qstrcat(dest, src, dest');
  lemma RS2: pstrcat () & strm(dest, j) & qcat(dest, src, dest', j, 0) ⇒ qstrcat(dest, src, dest');
  lemma RS3: pstrcat ()⇒ pscan (0)
  lemma COR1: pscan (j) & dest[j] = zero & j' = j ⇒ strm(dest, j');
  lemma RB1: pscan (j) & dest[j] != zero ⇒
        pscan (j) & l_dest - (j+1) < l_dest - j  & l_dest - (j+1)>=0;
  formula wd(dest1, dest, j, k) = dest1[j]=src[k] & ∀nat p. p != j ⇒ dest1[p]=dest[p];
  lemma COR2: pcat(k) & wd(dest1, dest, j, k) & src[k] = zero  & dest' = dest1 ⇒
              qcat(dest, src, dest', j, k);
  lemma RB2: pcat(k) & wd(dest1, dest, j, k) & src[k] != zero ⇒
        pcat (k+1) & l_src - (k+1) < l_src - k  & l_src - (k+1)>=0;
  lemma RB3: pcat(k) & wd(dest1, dest, j, k) & src[k] != zero &
              qcat(dest1, src, dest', j+1, k+1) ⇒ qcat(dest, src, dest', j, k);

}
```