УДК 004.05

Верификация предикатной программы бинарного поиска объекта произвольного типа

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

Описывается построение и дедуктивная верификация предикатной программы бинарного поиска, идентичной программе bsearch на языке Си из библиотеки ОС Linux. В языке предикатного программирования определяются новые конструкции для произвольных типов в качестве параметров программ. Для объектов произвольного типа вводятся трансформации кодирования через указатели.

Ключевые слова: *дедуктивная верификация, трансформации программ, бинарный поиск, язык программирования Си, функциональное программирование*.

1. Введение

Алгоритм бинарного поиска является классическим часто используемым алгоритмом. Программа бинарного поиска bsearch из библиотеки ядра ОС Linux максимально универсальна. Допускается произвольный тип элементов. В связи с этим размер элементов в байтах задается параметром. Операция сравнения элементов также поставляется как параметр программы bsearch. В дополнение к этому, из соображений эффективности используются битовые операции и адресная арифметика. Перечисленные особенности определяют повышенную сложность дедуктивной верификации программы bsearch. В частности, произвольный размер элементов и подстановка программы параметром существенно затрудняют применение набора инструментов FramaC – Why3 [1, 2], успешно используемых для дедуктивной верификации многих программ ядра ОС Linux.

В настоящей работе проводится верификации данной программы применением следующей технологии. В рамках предикатного программирования можно воссоздать любую императивную программу из класса программ-функций [11]. Строится предикатная программа, эквивалентная исходной императивной программе. Далее эта программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций. Данная технология построения предикатной программы, применения оптимизирующих трансформаций и дедуктивной верификации представлена в работе [10].

Дедуктивная верификация предикатной программы существенно проще и быстрее в сравнении с дедуктивной верификацией императивной программы [9].

Во втором разделе дается краткое описание языка предикатного программирования. Определяется синтаксис и семантика гиперфункций. Метод дедуктивной верификации описывается в третьем разделе. В четвертом разделе определено построение эквивалентной предикатной программы bsearch. В следующем разделе описывается процесс оптимизирующей трансформации программы. Процесс дедуктивной верификации предикатной программы в системах Why3 [25] и PVS [24] определяется в шестом разделе. Далее обзор работ. В заключении суммируются результаты работы. В расширенной версии данной статьи [26] включены приложения, где подробно документируется процесс дедуктивной верификации.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [5] 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 [8, 12, 23]. 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.

Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами.

Пусть E(x) – логическое выражение. Тип **subtype**(T x**:** E(x)) определяет *подтип* типа T при истинном предикате E(x), т.е. множество {x∈T | E(x)}. Определенный в языке P тип целых чисел **nat** представляется описанием:

type nat = **subtype**(**int** x: x ≥ 0) .

Допускаются подтипы, параметризуемые переменными. Примером является тип *диапазона* целых чисел:

type Diap(**nat** n) = **subtype**(**int** x: x ≥ 1 & x ≤ n) .

В языке P для изображения типа диапазона используется конструкция 1..n.

Тип может быть передан программе в качестве параметра. Для такого типа возможны лишь операции, передаваемые в виде функций другими параметрами программы. В целях верификации свойства такого типа задаются в виде теории. См. раздел 4.2.

Описания типов переменных являются частью спецификации программы. Описание переменной T x есть утверждение x∈T, которое становится частью предусловия, если x – аргумент предикатной программы, или частью постусловия, если x – результат программы. При этом утверждение x∈T обычно не пишется в составе предусловия или постусловия, хотя предполагается.

Гиперфункция – программа с несколькими *ветвями* результатов. Гиперфункция B(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Вызов гиперфункции записывается в виде B(x**:** y #M1**:** z #M2). Здесь M1 и M2 – метки программы, содержащей вызов. Операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2.

Вызов гиперфункции может комбинироваться с операторами обработки ветвей:

B(x**:** y #M1**:** z #M2) **case** M1: C(y**:** u) **case** M2: D(z**:** u) .

Конструкция вида B(x**:** y #M1**:** z #M2); M1**: ...** может быть представлена оператором B(x**:** y**:** z #M2).

Формально гиперфункция определяется через предикатную программу следующего вида:

```
B(x: y, z, e)
pre P(x) post e = E(x) & (E(x) ⇒ S(x, y)) & (¬E(x) ⇒ R(x, z))
{ ... };
```

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), E(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида e = **true** и e = **false** – последние исполняемые операторы в программе B. Программа B может быть заменена следующей программой в виде *гиперфункции*:

```
B(x: y #1:  z #2)
pre P(x)   pre 1: E(x) post 1: S(x, y) post 2: R(x, z)
{ … };
```

В теле гиперфункции каждое присваивание e = **true** заменено оператором перехода #1, а e = **false** – на #2. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции.

Спецификация гиперфункции состоит из двух частей. Утверждение после "**pre** 1" есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после "**post** 1" и "**post** 2" есть постусловия для первой и второй ветвей, соответственно.

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Традиционные подходы в реализации обработки аварийных ситуаций предполагают заведение дополнительных структур, усложняющих программу. Этого удается избежать при использовании гиперфункций. Использование гиперфункций делает программу короче, быстрее и проще для понимания [12, 13]. Подробнее в работе [6, разд. 4], подразделе «баланс информационных и управляющих связей».

В языке предикатного программирования P [5] 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* [4]. 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;
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) при помощи массивов и указателей.

3. Дедуктивная верификация

Предикатная программа относится к классу *программ-функций* [11]. 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 [14]. На базе языка **P0** последовательным расширением и сохранением тождества ℛ(H) = H построен язык предикатного программирования **P** [5].

Тотальная корректность программы относительно спецификации определяется формулой:

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

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

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

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная система правил доказательства их корректности [7, 16], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности. Корректность правил доказана [3] 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 [8, 12, 15].

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

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

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

Далее следует правило для частного случая оператора суперпозиции, соответствующего *сведению к более общей задаче* C(x, z**:** 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**:** z). Истинность трех посылок правила **RB** гарантирует корректность следующей программы:

H(x**:** y) **pre** P(x) **post** Q(x, y) { C(x, B(x)**:** 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. Построение программы бинарного поиска

4.1. Постановка задачи

Имеется следующая программа бинарного поиска на языке Си:

```
void *bsearch(const void *key, const void *base, size_t num, size_t size,
        int (*cmp)(const void *key, const void *elt))
{       const char *pivot;
        int result;
        while (num > 0) {
                pivot = base + (num >> 1) * size;
                result = cmp(key, pivot);
                if (result == 0)
                        return (void *)pivot;
                if (result > 0) {
                        base = pivot + size;
                        num--;
                }
                num >>= 1;
        }
        return NULL;
}
```

Имеется описание для пользователей: https://elixir.bootlin.com/linux/latest/source/lib/bsearch.c. Программа оперирует элементами произвольного размера size в байтах.

Элемент, доступный по указателю key размером size байтов, ищется в массиве по указателю base с элементами размера size байтов и числом элементов num. Массив упорядочен по возрастанию. Упорядоченность определяется функцией cmp(key, pivot), являющейся параметром программы bsearch. Параметры key и pivot являются указателями на элементы. Значения функции cmp интерпретируются следующим образом:

cmp(key, pivot) > 0 -→ key* > pivot*
cmp(key, pivot) < 0 -→ key* < pivot*
cmp(key, pivot) = 0 -→ key* = pivot*

Программа bsearch выдает результатом указатель на элемент key в исходном массиве при условии, что элемент со значение key* существует в массиве. В противном случае результат программы есть NULL.

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

4.2. Предикатная программа бинарного поиска

Определим тип элементов массива:

type E;

Тип E произвольный и ограничен лишь размером элементов size. Однако информация о размере size будет существенна лишь на стадии реализации соответствующей императивной программы. Принципиально, что для значений типа E недоступны операции, введенные в языке **P**, кроме операции “=” равенства элементов, определенной для всех типов. В предикатной программе мы будем использовать операторы присваивания, которые исчезают при склеивании переменных. Для значений типа E применимы лишь операции, специально для него определенные. В нашем случае это единственная операция сравнения cmp.

Определим тип функции cmp для сравнения двух элементов типа E:

type CMP= **predicate**(E, E**: int**) **pre true post true**;

Данное описание определяет предикатный тип с двумя аргументами типа E и результатом типа **int**. Никаких других ограничений на значения аргументов и результата нет.

Спецификация вида [**true**, **true**] является наиболее общей и абсолютно бесполезной. А другой здесь быть не может, поскольку предусловие и постусловие можно было бы выразить лишь с помощью других операций на типе E, а их нет.

Свойства операции cmp задаются в виде теории в стиле алгебраических спецификаций.

```
theory Compare {
    type E;
    axiom EnotEmpty: ∃ E x. true; // тип E не пустой
    CMP cmp;
    axiom TotalCmp: ∀ E x, y. ∃ int z. cmp(x, y) = z; // тотальность функции cmp
    axiom GrCmp: ∀ E x, y. cmp(x, y) > 0 ⇒ cmp(y, x) < 0;   // симметрия
    axiom TrCmpE: ∀ E x, y, z. cmp(x, y) <= 0 & cmp(y, z) < 0 ⇒ cmp(x, z) < 0;
    axiom TrCmpF: ∀ E x, y, z. cmp(x, y) < 0 & cmp(y, z) <= 0 ⇒ cmp(x, z) < 0;
}
```

В точности этот набор свойств теории Compare был задействован при доказательстве формул корректности предикатной программы bsearch. Тип E не может быть пустым, поскольку содержит элемент, поставляемый аргументом key. Тотальность функции cmp необходима для доказательства тотальности программы bsearch. Остальные свойства используются при доказательстве двух последних формул корректности RB13 и RB23 (см. Разд. 6.2).

Определим тип массива с элементами типа E:

```
type di(nat p, n) = p..p+n-1;
type Ar(nat p, n) = array(E, di(p, n));
```

Здесь n – число элементов массива. Тип di определяет индексы массива: от p до p+n-1. Параметром типа Ar является также тип E. Он не указывается в списке параметров, хотя подразумевается.

Ниже перечислены аргументы программы bsearch, которые для удобства написания программы определены как глобальные переменные.

```
type E;
E key;
CMP cmp;
nat num;
```

Здесь key – элемент типа E, который ищется в исходном массиве base, num – число элементов в массиве base и cmp – функция сравнения.

Программу бинарного поиска bsearch определим в виде гиперфункции:

```
bsearch( Ar(0, num) base : nat p, n, Ar(p, n) b #1 : #2 )
```

Аргумент base – исходный массив длины num с индексами от 0 до num-1. Первая ветвь гиперфункции реализуется при наличии в массиве base элемента, равного key; вторая ветвь – при отсутствии элемента key. Результатом первой ветви является массив b – вырезка массива base от элемента с индексом p до конца массива base, причем b[p] = key. Другой результат n – длина массива b. По второй ветви гиперфункции результатов нет.

Отметим, что в библиотечной программе bsearch на языке Си нет результата n – длины оставшейся части массива. Хотя такой результат нужен – в программе на языке Си он вычисляется после вызова bsearch.

Вырезка b[p..p+n-1] используется в предикатной программе bsearch как аналог указателя, на который вырезка заменяется на этапе оптимизирующей трансформации. Возможен более простой аналог указателя в виде пары (base, p).

Ниже представлена программа bsearch. Спецификация определяется общим предусловием pBsearch, предусловием по первой ветви p1key и постусловием по первой ветви qBsearch. Постусловие по второй ветви отсутствует, так как во второй ветви нет результатов.

```
formula pBsearch(nat p, n, Ar(p, n) b) =
                    p+n=num & ∀ di(p, n) i, j. i < j ⇒ cmp(b[i], b[j])≤0;
formula p1key(nat p, n, Ar(p, n) b) = ∃ di(p, n) j. cmp(key, b[j]) = 0;
formula qBsearch(Ar(0, num) base, nat p, n, Ar(p, n) b) =
                              p+n = num & b = base[p..num-1] & b[p] = key;
```

Предусловия определены в более общем в виде для использования также в другой программе bse. Для программы bsearch параметр p в предусловиях равен нулю.

```
bsearch( Ar(0, num) base : nat p, n, Ar(p, n) b #1 : #2)
pre pBsearch(0, num, base)
pre 1: p1key(0, num, base)
post 1: qBsearch(base, p, n, b)
{   bse(0, num, base : p : #2);
   { n = num – p; b = base[p..p+n-1] }
   #1
};
```

Вызов гиперфункции bse по первой ветви вычисляет индекс p массива base, на котором base[p] = key. При завершении первой ветвью исполняются операторы после вызова. При завершении вызова bse второй ветвью реализуется переход #2, завершающий исполнение гиперфункции bsearch второй ветвью, что означает отсутствие key в массиве base.

Гиперфункция bse для массива b[p: p+n-1] определяет индекс p′, на котором b[p′] = key. Ветви гиперфункции определяются аналогично гиперфункции bsearch.

```
bse(nat p, n, Ar(p, n) b: nat p′ #1 : #2)
pre pBsearch(p, n, b)
pre 1: p1key(p, n, b)
post 1: ∃ nat m. cmp(key, b[p+m]) = 0 & p′ = p+m
measure n
{  if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p′ = p + m  #1 }
   if (result > 0)
         bse(p+m+1, n - m - 1, b[p+m+1..p+n-1] #1: p′ : #2)
   else  bse(p, m, b[p..p + m - 1] #1: p′ : #2)
};
```

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

Алгоритм реализуется по следующей схеме. Представим диапазон p..p + n – 1 в виде:

[p, … , p+m-1] p+m [p+m+1, … , p+n-1]

Если cmp(key, b[p+m]) = 0, то решение: p′ = p+m. Иначе в зависимости от знака cmp(key, b[p+m]) требуемый элемент key ищется в одном из диапазонов: p..p+m-1 или p+m+1..p+n-1.

Итак, полная предикатная программа состоит из программ bsearch и bse.

5. Оптимизирующие трансформации

Определим набор трансформаций, превращающих программу бинарного поиска, состоящую из программ bsearch и bse, в эффективную императивную программу. На первом этапе реализуется трансформация склеивания переменных. Например, операция склеивания num ← n реализует замену всех вхождений переменных n на переменную num.

Склеивание: num ← n.

```
bsearch( Ar(0, num) base : num, nat p, Ar(p, num) b #1 : #2)
{   bse(0, num, base: p : #2);
   {num = num – p;  b = base[p..p+num-1]}
   #1
};
```

Склеивание: p ← p′.

```
bse(nat p, n, Ar(p, n) b: p #1 : #2)
{  if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p = p + m  #1 }
   if (result > 0)
         bse(p+m+1, n - m - 1, b[p+m+1..p+n-1] #1: p : #2)
   else  bse(p, m, b[p..p + m - 1] #1: p : #2)
};
```

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

```
bse(nat p, n, Ar(p, n) b: p #1 : #2)
{loop {
   if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p = p + m  #1 }
   if (result > 0)
         |p, n, b| =|p+m+1, n - m - 1, b[p+m+1..p+n-1]|
   else  |p, n, b| =|p, m, b[p..p + m - 1]|
 }
};
```

Раскрываются групповые операторы присваивания. При раскрытии первого оператора необходимо поменять порядок присваивания.

```
bse(nat p, n, Ar(p, n) b: p #1 : #2)
{loop {
   if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p = p + m  #1 }
   if (result > 0)
         { b = b[p+m+1..p+n-1];  p = p+m+1; n = n - m - 1 }
   else  { n = m; b = b[p..p + m - 1] }
 }
};
```

На третьем этапе тело программы bse подставляется на место вызова в bsearch. При этом оператор #1 заменяется на **break**.

```
bsearch( Ar(0, num) base : num, nat p, Ar(p, num) b #1 : #2)
{   //bse(0, num, base: p : #2);
    | p, n, b | = | 0, num, base |;
  loop {
    if (n=0) #2
    nat m = n / 2;
    E pivot = b[p+m];
    int result = cmp(key, pivot);
    if (result = 0) { p = p + m; break }
    if (result > 0)
            { b = b[p+m+1..p+n-1];  p = p+m+1; n = n - m - 1 }
    else   { n = m; b = b[p..p + m - 1] }
  };
    {num = num – p;  b = base[p..p+num-1]}
     #1
};
```

В теле цикла заменим n на num, а b на base. Далее операторы за циклом втянем внутрь тела цикла, изменив их порядок.

```
bsearch( Ar(0, num) base : num, nat p, Ar(p, num) b #1 : #2)
{ p = 0; nat num0 = num;
  loop {
    if (num=0) #2
    nat m = num / 2;
    E pivot = base[p+m];
    int result = cmp(key, pivot);
    if (result = 0) {
      p = p + m;
      num = num0 – p ;
      b  = base[p..p+num-1]
      #1
    }
    if (result > 0)
            { base = base[p+m+1..p+ num -1];  p = p+m+1; num = num - m – 1 }
    else   { num = m; base = base[p..p + m - 1] }
  };
};
```

Здесь num0 обозначает значение num в начале исполнения bsearch.

На четвертом этапе трансформаций реализуем кодирование типов и операций с ними через типы и операции языка Си.

Все переменные типа E кодируются указателем на переменную. Причем используется указатель типа **void***. Массив base получает тип **void***. Элемент массива base[p+m] идентифицируется указателем base + m*size. Ниже приведено представление операций и операторов программы bsearch.

```
E pivot = base[p+m]                → void*pivot = base + m*size;
base = base[p..p + m - 1]          → base = base
base = base[p+m+1..p+ num -1]      → base = base + (m+1)*size;
num / 2                            → num >> 1
```

Результатом кодирования объектов является следующая программа.

```
bsearch(void *base, nat num : num, nat p, void *b #1 : #2)
{ p = 0; nat num0 = num;
  loop {
    if (num=0) #2
    nat m = num >> 1;
    void*pivot = base + m*size;
    int result = cmp(key, pivot);
    if (result = 0) {
      b  = base + m*size;
      p = p + m;
      num = num0 – p
      #1
    }
    if (result > 0)
            { base = base + (m+1)*size;  p = p+m+1; num = num - m – 1 }
    else   { num = m; base = base }
  };
};
```

Вычисления указателей не зависят от p. Поэтому все операторы, вычисляющие p, можно удалить. Кроме того, ненужными становятся операторы p = p + m и num = num0 – p. После удаления неиспользуемых вычислений программа преобразуется к виду:

```
bsearch(void *base, nat num :  void *b #1 : #2)
{ loop {
    if (num=0) #2
    nat m = num >> 1;
    void*pivot = base + m*size;
    int result = cmp(key, pivot);
    if (result = 0) {
       b  = base + m*size
      #1
    }
    if (result > 0)
            { base = base + (m+1)*size;  num = num - m – 1 }
    else   num = m;
  };
};
```

Далее превратим гиперфункцию в обычную функцию, совместив два выхода в один. С этой целью выход #2 закодируем оператором b = NULL с учетом того, что нулевой

указатель не появится на первой ветви. Проведем также очевидные упрощения при вычислении указателей. Получим итоговую программу.

```
bsearch(void *base, nat num :  void *)
{ loop {
        if (num=0) return NULL;
        nat m = num >> 1;
        void*pivot = base + m*size;
        int result = cmp(key, pivot);
        if (result = 0) return pivot;
        if (result > 0) { base = pivot + size;  num = num - m - 1 }
        else  num = m;
  };
};
```

Сравним ее с исходной программой на языке Си:

```
void *bsearch(const void *key, const void *base, size_t num, size_t size,
              int (*cmp)(const void *key, const void *elt))
{       const char *pivot; int result;
        while (num > 0) {
                pivot = base + (num >> 1) * size;
                result = cmp(key, pivot);
                if (result == 0) return (void *)pivot;
                if (result > 0) { base = pivot + size;  num--; }
                num >>= 1;
        }
        return NULL;
    }
```

Чтобы определить эквивалентность программ, достаточно проверить, что значение переменной num в конце цикла является одинаковым для двух программ при условии, что значения num в начале цикла совпадают для двух программ. Рассмотрим случаи: num = 2*k и num = 2*k + 1. Проведя несложные вычисления для num в начале цикла легко проверить, что значения num в конце цикла для каждого из случаев будут совпадать.

Таким образом, программы эквивалентны. Разумеется, можно было бы построить предикатную программу, которая после трансформации совпадает с исходной программой.

6. Процесс дедуктивной верификации

Теория с формулами корректности (Разд. 6.2) была закодирована на языке спецификаций why3 системы Why3 [25]. При этом массивы вида b[p: p+n-1] всюду были сведены к массивам b[0: 0+n-1]. Как следствие, во многих конструкциях появилось вычитание значения p. Эти изменения нетривиальны. Несмотря на то, что вырезки полноценно

присутствуют в языке программирования whyML, в языке спецификаций why3 и системе доказательства Why3 вырезки никак не поддержаны. Лямбда-функций нет в why3. В итоге вырезки были определены через предикаты sliceL и sliceR.

В рамках системы Why3 в течение полутора дней были доказаны все формулы корректности, кроме RB13 и RB23, наиболее сложных. Причем для доказательства формулы корректности Rp2 были введены промежуточные леммы Rp2_key, Rp2_key0 и KeyEx. Примечательно, что все формулы и леммы были доказаны с помощью одного решателя AltErgo. Попытки доказать RB13 и RB23 за пару дней оказались безуспешными.

Было решено перевести теорию из why3 на язык спецификаций PVS в проекции на RB13 и RB23. Из-за сложной системы ограничений на диапазоны два дня не удавалось пройти семантический контроль (type checking) в PVS. Пришлось изменить спецификацию постусловия программы bse. Доказательство в PVS для формул RB13 и RB23 удалось построить за пару часов.

Сопутствующей целью верификации было определение точных ограничений на функцию cmp. Эти ограничения определены и представлены в виде теории Compare (Разд. 4.2).

7. Обзор

Программа бинарного поиска является весьма популярной. Довольно часто она используется для иллюстрации новых методов верификации.

В работе [18] методом дедуктивной верификации оценивается объем используемой памяти, в частности, размер стека, для программ на языке Си. В качестве тестовой использовалась преобразованная в рекурсивную программа быстрого поиска bsearch с логарифмическими оценками.

Программа bsearch использована для демонстрации аннотаций, вставляемых для оценки наихудшего времени исполнения, в руководстве к пользованию сертифицированным оптимизирующим компилятором CompSert [21], корректность которого доказана в системе верификации Coq.

В инструменте верификации Frama C [1] применяется язык спецификаций ACSL [17] программ на языке Си. На языке ACSL записываются спецификации программ в виде контрактов. Контракт вида behavior определяет вариант спецификации. Полная спецификация слагается из набора вариантов. Такой способ спецификации аналогичен спецификации предусловия и постусловия для одной ветви гиперфункции. Метод спецификации иллюстрируется на программе bsearch.

С применением технологии программирования в ограничениях в работе [22] предлагается новый метод автоматического построения инвариантов с проведением доказательства их корректности. Программа bsearch использована в качестве тестовой.

В работе [19] сообщается об обнаружении ошибки балансировки в программе поиска в бинарном дереве. При этом программа оставалась корректной, но более медленной. Например, если бы в нашей программе мы использовали бы вызов

```
bse(p+1, n - 1, b[p+1..p+n-1] #1: p' : #2)
```

вместо вызова

```
bse(p+m+1, n - m - 1, b[p+m+1..p+n-1] #1: p' : #2).
```

Тогда вместо логарифмической оценки сложности на определенных входных данных была бы линейная оценка. В работе [20] разработан метод дедуктивной верификации, гарантирующий не только функциональную корректность, но и определяющий наихудшую оценку сложности. Метод иллюстрируется на программе bsearch.

8. Заключение

В настоящей работе представлена дедуктивная верификация программы бинарного поиска bsearch из библиотеки ядра ОС Linux. Программа максимально универсальна. Размер элемента и операция сравнения элементов задаются параметрами программы bsearch. Обработку объектов произвольного типа можно запрограммировать в языке Си только через указатели общего вида **void** *. Из соображений эффективности используются битовые операции и адресная арифметика. Перечисленные особенности определяют повышенную сложность дедуктивной верификации программы bsearch.

Библиотечная программа bsearch получена из предикатной программы (разд. 4) применением набора оптимизирующих трансформаций. В действительности, в результате трансформаций получена другая программа, близкая к исходной библиотечной. Их эквивалентность легко проверяется. Следует отметить, что полного совпадения исходной и трансформированной программы не было также и для программы конкатенации строк strcat [10].

Дедуктивная верификация предикатной программы bsearch должна бы быть существенно проще и быстрее в сравнении с дедуктивной верификацией исходной императивной программы. Однако верификация предикатной программы bsearch осложнилась из-за неучтенных особенностей системы автоматического доказательства Why3 [25]. Завершение дедуктивной верификации проводилось в системе PVS [24]. Система Why3 предполагает

иной стиль спецификации программ по сравнению со спецификацией для системы PVS. Этот стиль предстоит освоить для успешной верификации других программ.

Дедуктивная корректность подтвердила корректность программы bsearch. Сопутствующей целью верификации было определение точных ограничений на операцию сравнения элементов. Эти ограничения представлены в виде теории Compare (Разд. 4.2), определяющей симметричность, транзитивность и тотальность операции сравнения. Тотальность – это возможность применения операции к любым аргументам. В пользовательской документации к программе bsearch данной информации нет.

Чтобы обеспечить высокий уровень доверия к инструментам верификации, необходимо гарантировать корректность применяемой технологии и сопутствующих инструментов. С этой целью на базе модели внутреннего представления транслируемой программы с языка P предполагается построить модель оптимизирующих трансформаций и провести ее верификацию.

Имеется расширенная версия данной статьи [26] с приложениями, где подробно документируется процесс дедуктивной верификации.

Список литературы

1. AstraVer Toolset: инструменты дедуктивной верификации моделей и механизмов защиты ОС. ИСП РАН. [Электронный ресурс]. URL: http://linuxtesting.org/astraver, 15.10.2017 (дата обращения 30.11.2018).
2. 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
3. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL: http://www.iis.nsk.su/persons/vshel/files/rules.zip (дата обращения 12.11.2018)
4. 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)
5. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.14. Новосибирск, 2018. 45с., [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения 12.03.2019).

6. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. http://persons.iis.nsk.su/files/persons/pages/lift1.pdf
7. 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).
8. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21. https://www.iis.nsk.su/files/preprints/154.pdf
9. 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).
10. Шелехов В.И. Дедуктивная верификация и оптимизация предикатной программы конкатенации строк // Системная информатика, № 12. Новосибирск, 2018. С. 61-84. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/strcat.pdf (дата обращения 12.11.2018).
11. 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).
12. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164). [Электронный ресурс]. URL: https://www.iis.nsk.su/files/preprints/164.pdf (дата обращения 12.11.2018)
13. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
14. 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).
15. 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).

16. Шелехов В.И. Правила доказательства корректности предикатных программ. — Новосибирск, ИСИ СО РАН, 2019. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prrules.pdf (дата обращения 12.06.2019).
17. Baudin P., Filliatre J. C., Cuoq P., March C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language, 2015. [Электронный ресурс]. URL: http://frama-c.com/download.html. (дата обращения 15.05.2019)
18. Carbonneaux Q., Hoffmann J., Ramananandro T. , Shao Z. End-to-end verification of stack-space bounds for C programs // PLDI '14, 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014. P. 270-281.
19. Filliatre J.C., Letouzey P. Functors for proofs and programs. // European Symposium on Programming (ESOP). LNCS 2986, 2004, P. 370–384.
20. Guéneau A., Charguéraud A., Pottier F. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification // Programming Languages and Systems. ESOP. LNCS 10801. 2018. P. 533-560.
21. Leroy X. The CompCert C verified compiler: Documentation and user's manual. 2014. 60p.
22. Ponsini O., Collavizza H., Fedele C., Michel C., Rueher M. Automatic Verification of Loop Invariants // 2010 IEEE International Conference on Software Maintenance, 2010, P.2-11.
23. 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.
24. PVS Specification and Verification System. SRI International. [Электронный ресурс]. URL: http://pvs.csl.sri.com/ (дата обращения 12.11.2018).
25. Why 3. Where Programs Meet Provers. [Электронный ресурс]. URL: http://why3.lri.fr, (дата обращения 15.05.2019)
26. Шелехов В.И. Верификация предикатной программы бинарного поиска объекта произвольного типа. Новосибирск, 2019. 26с. http://persons.iis.nsk.su/files/persons/pages/fsearch2.pdf