УДК 004.05

Верификация предикатной программы пирамидальной сортировки с применением обратных трансформаций

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

Проводится дедуктивная верификация алгоритма классической пирамидальной сортировки Дж. Вильямса, реализованного программой sort на языке Си в библиотеке ОС Linux. Сортировка реализуется для объектов произвольного типа. Чтобы упростить верификацию, применяются нетривиальные трансформации, заменяющие арифметические операции с указателями явными элементами сортируемого массива. Программа преобразуется на язык предикатного программирования. Конструируются спецификации предикатной программы. Дедуктивная верификация в системах Why3 и Coq оказалась сложной и трудоемкой.

Ключевые слова: *дедуктивная верификация, трансформации программ, функциональное программирование, предикатное программирование, неинтерпретированный тип.*

1. Введение

Исходной задачей является дедуктивная верификация программы sort на языке Си из библиотеки ядра ОС Linux. Используется алгоритм классической пирамидальной сортировки Дж. Вильямса [25]. И хотя это наиболее простой алгоритм в классе алгоритмов пирамидальной сортировки, его дедуктивная верификация оказывается нетривиальной.

Сортировка реализуется для объектов произвольного типа и произвольного размера. В языке Си такие объекты представляются указателями общего вида void*. Операции с объектами реализуются функциями, доступными через параметры программы сортировки. Имеются две таких операции: сравнение и обмен пары элементов. Массивы произвольного размера и программы, подставляемые параметрами, существенно затрудняют верификацию программы набором инструментов FramaC – Why3 [2, 16]. Есть еще одна особенность, принципиально затрудняющая верификацию: для вычисления указателей применяется оптимизация уменьшения силы операций с заменой в цикле умножения на сложение.

Дедуктивная верификация намного проще и быстрее для функциональных программ, чем для аналогичных императивных программ. Этот факт отмечается разными исследователями. Причина сложности императивных программ в том, что указатели, конструкции необходимые для оптимизации программ, существенно усложняют логику императивных программ. Для упрощения императивных программ применяются трансформации, устраняющие указатели в императивной программе [9]. Операции с указателями заменяются эквивалентными действиями без указателей. Далее к полученной программе применяются трансформации, превращающие ее в эквивалентную предикатную программу.

В настоящей работе применяется метод обратной трансформации [9] от исходной библиотечной программы sort.c к эквивалентной предикатной программе. Разрабатываются спецификации для полученной предикатной программы. Далее строятся формулы корректности программы относительно спецификации применением системы правил [11]. Совокупность формул корректности вместе с описаниями типов и переменных оформляется в виде набора теорий. Эти теории транслируются на язык спецификаций why3 [24]. Далее в системах дедуктивной верификации Why3[24] и Coq [17] реализуется процесс доказательства формул корректности.

Ранее в 2012г. дедуктивная верификация проводилась для трех алгоритмов пирамидальной сортировки [12]: классического алгоритма Дж. Вильямса [25], алгоритма Флойда [20] и улучшенного алгоритма, [23] бывшего тогда самым быстрым алгоритмом сортировки. В 2019г. дедуктивная верификация той же программы sort проводилась с применением прямых трансформаций, однако не была завершена.

Во втором разделе дается краткое описание языка предикатного программирования. Метод дедуктивной верификации описывается в третьем разделе. В четвертом разделе описывается обратная трансформация исходной программы sort с получением предикатной программы пирамидальной сортировки. В следующем разделе описывается процесс спецификации предикатной программы. Особенности процесса дедуктивной верификации предикатной программы в системах Why3[24] и Coq [17] описывается в шестом разделе. Далее обзор других работ по дедуктивной верификации программы heapsort. В заключении суммируются результаты работы. В Приложении 1 код исходной программы sort на языке Си из библиотеки ОС Linux. В Приложении 3 приведены три теории на языке Why3 для доказательства формул корректности на момент завершения работы по верификации. Доступна полная версия текста настоящей работы: https://persons.iis.nsk.su/files/persons/pages/sort9.pdf.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [4] 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 [7, 8, 12, 15, 22]. Мера задается только для рекурсивных программ и используется для верификации.

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.

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

В языке предикатного программирования P [4] 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* [3]. 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* [10]. 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 [13]. На базе языка $\mathbf{P_0}$ последовательным расширением и сохранением тождества $\mathcal{R}$(H) = H построен язык предикатного программирования **P** [4].

Тотальная корректность программы относительно спецификации определяется формулой:

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

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

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

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная система правил доказательства их корректности [6, 11], 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 [21]. Данный метод опробован для дедуктивной верификации более сотни программ [7, 8, 12, 15, 22].

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

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

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

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

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

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

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

В случае рекурсивного вызова C(x, B(x)**:** 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 пирамидальной сортировки

4.1. Постановка задачи

В библиотеке ядра ОС Linux имеется программа sort на языке Си, реализующая сортировку объектов произвольного типа и произвольного размера. Используется алгоритм классической пирамидальной сортировке Дж.Вильямса [25]. Программа sort приведена в Приложении 1. Представим заголовок программы:

```
void sort(void *base, size_t num, size_t size,
          int (*cmp_func)(const void *, const void *),
          void (*swap_func)(void *, void *, int size))
```

Здесь base – указатель сортируемого массива; num – число элементов массива; size - размер элемента в байтах; cmp_func – указатель функции сравнения двух элементов; swap_func – указатель функции обмена двух элементов или NULL. В случае swap_func = NULL обмен элементов реализуется одной из функций в составе программы sort.

Функция cmp_func(pa, pb), где pa и pb – указатели на элементы a и b, определяется следующим образом.

```
cmp_func(pa, pb) > 0 -  →  a > b
cmp_func(pa, pb) <= 0 -→  a <= b
cmp_func(pa, pb) = 0 -  →  a = b
```

Ниже представлен код программы sort. Данная программа sort ранее входила в состав библиотеки ядра ОС Linux до версии 5.1. В последующих версиях ОС Linux программа sort заменена другим более быстрым алгоритмом пирамидальной сортировки.

```
void sort(void *base, size_t num, size_t size,
         int (*cmp_func)(const void *, const void *),
         void (*swap_func)(void *, void *, int size))
{
        /* pre-scale counters for performance */
        int i = (num/2 - 1) * size, n = num * size, c, r;
………………………………………………………………………..
        /* heapify */
        for ( ; i >= 0; i -= size) {
                for (r = i; r * 2 + size < n; r  = c) {
                        c = r * 2 + size;
                        if (c < n - size &&
                                        cmp_func(base + c, base + c + size) < 0)
                                c += size;
                        if (cmp_func(base + r, base + c) >= 0)
                                break;
                        swap_func(base + r, base + c, size);
                }
        }
        /* sort */
        for (i = n - size; i > 0; i -= size) {
                swap_func(base, base + i, size);
                for (r = 0; r * 2 + size < i; r = c) {
                        c = r * 2 + size;
                        if (c < i - size &&
                                        cmp_func(base + c, base + c + size) < 0)
                                c += size;
                        if (cmp_func(base + r, base + c) >= 0)
                                break;
                        swap_func(base + r, base + c, size);
                }
        }
}
```

Отметим, что в коде программы опущена инициация значения функции swap_func в случае swap_func = NULL. Эту инициацию и используемые подпрограммы обмена элементов можно верифицировать независимо. Далее будем считать, что функция swap_func задана параметром программы sort.

Требуется трансформировать данную программу в эквивалентную предикатную программу и провести ее дедуктивную верификацию. Здесь применяются обратные трансформации по сравнению с обычными (прямыми) оптимизирующими трансформациями, используемыми в предикатном программировании.

4.2. Устранение указателей

Целью первой стадии обратных трансформаций является устранение указателей. Операции с указателями заменяются эквивалентными действиями без указателей. Например, адресное вычисление base + c заменяется явным элементом массива base[c/size].

Особенность программы sort.c в том, что для адресных вычислений в итоговой программе на языке Си проведена оптимизация уменьшения силы операций, в результате которой адресные выражения вида base + c * size заменены на base + c посредством изменения масштаба переменных, входящих в c. В такой ситуации для проведения трансформации программы sort необходимо сначала провести трансформацию, обратную уменьшению силы операций, которая заменила бы base + c на base + c * size. Данная трансформация реализуется крайне редко, лишь когда оптимизацию «уменьшение силы операций» применяет программист. Обычно такая оптимизация реализуется при оптимизирующей трансляции.

В случае, когда требуется провести открытую подстановку кода функции на место ее вызова, используется описатель **inline**, гарантирующий проведение данной оптимизации при трансляции. Однако по непонятным причинам, открытая подстановка в программе sort.c проведена явно программистом. Это вынуждает нас провести обратную трансформацию запроцедуривания. Тела внутренних циклов по r почти совпадают. Два цикла по r похожи и отличаются лишь в паре позиций. Введем параметры k и m для этих позиций и определим программу siftDown, телом которой является цикл по r.

Сначала реализуется трансформация запроцедуривания. Далее применяется трансформация, обратная оптимизации уменьшения силы операций, реализующей замену в циклах умножение на сложение. В программе переменные i, r, n и c пересчитываются на значение size в каждом из двух циклов. Проводятся замены переменных:

```
i → ii*size;
r → rr*size;
c → cc*size;
n → nn*size;
```

При этом в циклах параметр i заменяется на ii, в результате чего пересчет на size заменяется пересчетом на единицу. Внутренние циклы по r заменяются циклами по переменной rr. Применение двух трансформаций преобразует программу к следующему виду.

```
    int ii = (num/2 - 1), nn = num, cc, rr;
    for ( ; ii >= 0; ii -= 1) {
        siftDown(base, ii, nn)
    }
    for (ii = nn - 1; i > 0; ii -= 1) {
        swap_func(base, base + ii*size, size);
        siftDown(base, 0, ii)
    }

void siftDown(void *base, const int k, m) {
for (int rr = k; rr *2 + 1 < m; rr  = cc) {
        cc = rr * 2 + 1;
        if (cc < m - 1 &&
                cmp_func(base + cc*size, base + cc*size + size) < 0)
            cc += 1;
        if (cmp_func(base + rr*size, base + cc*size) >= 0)
            break;
        swap_func(base + rr*size, base + cc*size, size);
}
}
```

Будет удобным провести следующие обратные замены переменных:

ii → i; rr → r; cc → c; nn → num;

Доступ в программе к некоторому элементу m массива base реализуется указателем base + m*size. Заменим это указатель на base[m]. Далее применяются трансформации вида:

```
base + p*size                         → base[p]
swap_func(base+r*size, base+c*size, size)  →  swap(base, r, c)
cmp_func(base+r*size, base+c*size)   → cmp(base[r], base[c])
```

Получим следующую программу:

```
    int i = (num/2 - 1), c, r;
    for ( ; i >= 0; i -= 1) {
        siftDown(base, i, num);
    }
    for (i = num - 1; i > 0; i -= 1) {
        swap(base, 0, i);
        siftDown(base, 0, i);
    }
```

```
void siftDown(void *base, const int k, m) {
for (int r = k; r *2 + 1 < m; r  = c) {
            c = r * 2 + 1;
            if (c < m - 1 && cmp(base[c], base[c+1]) < 0)
                  c += 1;
            if (cmp(base[r], base[c]) >= 0)
                  break;
            swap(base, r, c);
}
}
```

Оформим полученную программу. Константные параметры программы sort определим глобальными переменными. Введем тип T для элементов сортируемого массива.

```
size_t num, size;
type T;
type Ar = array (T, int);
int cmp(const T, const T);
void swap (Ar, const int, const int);
void sort(Ar base) {
      int i = (num/2 - 1), c, r;
      for ( ; i >= 0; i -= 1) {
            siftDown(base, i, num);
      }
      for (i = num - 1; i > 0; i -= 1) {
            swap(base, 0, i);
            siftDown(base, 0, i);
      }
}
void siftDown(Ar base, const int k, m) {
for (int r = k; r*2 + 1 < m; r  = c) {
            c = r * 2 + 1;
            if (c < m - 1 && cmp(base[c], base[c+1]) < 0)
                  c += 1;
            if (cmp(base[r], base[c]) >= 0)
                  break;
            swap(base, r, c);

}
}
```

4.3. Трансформация в предикатную программу

Определим типы и глобальные переменные.

```
nat num, size;
type T;
nat Di = 0..num-1
type Ar = array (T, Di);
cmp(T, T: int);
swap(Ar, Di, Di: Ar);
```

Заменим циклы **for** на циклы вида **loop**.

```
void sort(Ar base: Ar base’) {
        int i = (num/2 - 1), c, r;
        loop {
                if (i < 0) break;
                siftDown(base, i, num: Ar base1);
                i -= 1;
        }
        i = num - 1;
        loop {
                if (i <= 0) break;
                swap(base1, 0, i: base2);
                siftDown(base2, 0, i: base’);
        i -= 1;
        }
}

void siftDown(Ar base, const int k, m: Ar base’) {
int r = k;
loop {
                if (r*2 + 1 >= m) break;
                int c = r *2 + 1;
                if (c < m - 1 && cmp(base[c], base[c+1]) < 0) c += 1;
                if (cmp(base[r], base[c]) >= 0)  break;
                swap(base, r, c: base);
                r  = c;
        }
}
```

Штрих в имени base’ предполагает склеивание переменных base и base’ в реализации.

Три цикла преобразуются в рекурсивные программы.

```
heapify(Ar base, int i: Ar base’) {
        if (i < 0) base’ = base
        else { SiftDown(base, i, num: Ar base1);  heapify(base1, i - 1: base’) }
}
sorting(Ar base, int i: Ar base’) {
        if (i <= 0) base’ = base
        else {  swap(base, 0, i: Ar base1);
                siftDown(base1, 0, i: Ar base2);
                sorting(base2 i - 1: base’)
        }
```

```
}
siftDown(Ar base, int r, m: Ar base') {
        int c = r *2 + 1;
        if (c >= m) base' = base
        else {   if (c < m - 1 && cmp(base[c], base[c+1]) < 0) c1 = c+1 else c1 = c;
                  if (cmp(base[r], base[c1]) >= 0) base' = base
                  else { swap(base, r, c1: Ar base1); siftDown(base1, c1, m: base')};
                }
}
```

В итоге получим:

```
sort(Ar base: Ar base') {
        heapify(base, num/2 - 1: Ar base1);
        sorting(base1, num - 1: base')
}
```

Полная предикатная программа, состоящая из программ sort, heapify, sorting и siftDown, в точности соответствует исходной библиотечной программе sort на языке Си.

5. Спецификация предикатной программы sort

Воспроизведем глобальные описания программы sort.

```
nat num;
type T;
nat Di = 0..num-1;
type Ar = array (T, Di);
cmp(T, T: int);
swap(Ar a, Di j, k: Ar a') post exchange(a, a', j, k);
Ar base;
```

Предикат exchange определен в библиотеке Array системы верификации Why3 [16]. Исходный сортируемый массив base определен здесь как глобальный.

Определение и свойства функции cmp для сравнения элементов задаются в виде теории:

```
theory Compare {
   type CMP= predicate(T, T: int)
   CMP cmp_func;
   axiom Refl :   ∀ x: T. cmp_func(x, x) = 0
   axiom Simm :   ∀ T x, y. cmp_func(x, y) = - cmp_func(y, x);
   axiom TotalLe : ∀ T x, y. cmp_func(x, y) <=0 or cmp_func(y, x) <=0;
   axiom TransLe : ∀ T x, y, z. cmp_func(x, y)<=0 & cmp_func(y, z)<=0 ⇒ cmp_func(x,z)<=0;
}
```

Здесь приведены именно те свойства функции сравнения, которые были использованы при доказательствах. Это значит, что в любом вызове программы sort параметр-функция, подставляемая на место cmp_func, должна удовлетворять перечисленным свойствам. Отметим, что аксиома антисимметричности не использовалась.

5.1. Спецификация главной программы

Определим спецификацию программы sort.

```
sort( : Ar base’) post sorted(base’) & perm(base, base’);
{  heapify(base, num/2 - 1: Ar base1);
   sorting(base1, num - 1: base’)
}
```

Спецификация определяет, что массив base′ должен быть сортирован и получен перестановкой исходного массива base. Свойство сортированности определяет упорядоченность элементов:

formula sorted(Ar a) = ∀ k, j= 0..num-1. k < j ⇒ cmp(a[k], a[j]) <= 0;
formula sortedP(Ar a, **nat** m) = ∀ k, j= m..num-1. k < j ⇒ cmp(a[k], a[j]) <= 0;

Вторая формула определяет упорядоченность части массива a от m до конца массива.

formula perm(Ar a, b) = permut_all(a, b);
formula permE(Ar a, b, **nat** m, n) = permut_sub(a, b, r, m);

Предикаты *перестановочности* заимствованы из Why3[16]. Предикат permut_sub определяет перестановочность на отрезке от m до n-1 и равенство массивов везде вне этого отрезка.

5.2. Двоичная куча

Существует простой, эффективный и компактный способ представления двоичного дерева внутри массива a. Вершинами дерева являются элементы a[0], a[1], …, a[m-1], где m ≤ num. На рис.1 дается пример представления дерева для m = 8.

```
              a[0]
            /      \
       a[1]         a[2]
       /   \        /   \
    a[3]  a[4]  a[5]  a[6]
    /
a[7]
```

Рис.1. Представление двоичного дерева внутри массива

Родитель вершины с индексом j имеет индекс (j-1) / 2, а левая и правая дочерние вершины имеют индексы 2 j + 1 и 2 j + 2, соответственно. Определения функций для правого и левого потомка и родителя вершины a[j] даются ниже.

formula left(**nat** j **:** **nat**) = j * 2 + 1;
formula right(**nat** j **:** **nat**) = j * 2 + 2;
formula father(**nat** j **:** **int**) = (j-1) / 2;

Здесь «/» – операция целочисленного деления.

Двоичная максимальная куча (или пирамида) есть двоичное дерево, представленное внутри массива, в котором значение каждой вершины не меньше значений ее потомков.

formula heap(Ar a) = ∀ **nat** j = 0 .. num-1. heapJ(a, num, j)
formula heapJ(Ar a, **nat** j, m) = (left(j) < m ⇒ cmp(a[j], a[left(j)])>=0)) &
(right(j) < m ⇒ cmp(a[j], a[right(j)]>=0)) .

Предикат heapJ определяет, что вершина j обладает *свойством кучи*.

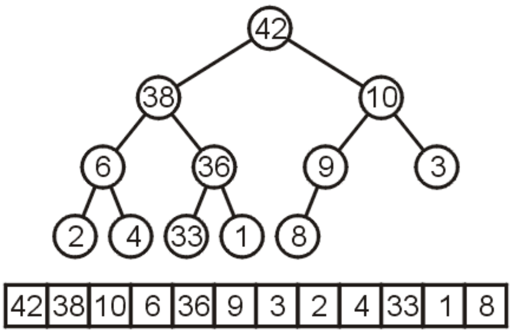

Рис.2. Пример двоичной максимальной кучи при num = 12

Используется также *обобщенная двоичная куча*, построенная на части массива a для элементов в диапазоне индексов от i до m-1 и определяемая формулой:

formula heap(Ar a, **nat** i, m) = ∀ **nat** j = i .. m-1. heapJ(a, j, m);

Дочерние вершины определяются функциями left и right как для полного дерева с корнем в вершине a[0]. В обобщенной двоичной куче верхушка дерева срезана. Поэтому в нем допускается более одной корневой вершины.

5.3. Верификация программ с неоднозначной спецификацией

Спецификация подпрограмм heapify и siftDown неоднозначна, поскольку по исходному массиву можно построить много разных куч, перестановочных с исходным массивом. Проблема состоит в том, что спецификация должна быть достаточно сильной, чтобы доказать корректность программы. Для однозначных спецификаций такой проблемы нет, поскольку спецификация тождественна программе.

Неоднозначность спецификации принципиально осложняет спецификацию и верификацию. Определить требуемое усиление спецификации априори проблематично. Неудачное усиление может сильно осложнить верификацию. Так, в предыдущем релизе верификации программы sort в 2019г. были использованы усиления, сделанные в более ранней верификации [12]. В текущем релизе было решено ограничиться лишь очевидными условиями, а необходимые усиления аккуратно определить в процессе доказательства и попытаться сделать их минимальными. Далее отмечается, какие части спецификации были внесены позже. В частности, в конце процесса верификации обнаружилось, что для постусловия программы siftDown необходимо использовать предикат permE вместо perm.

Несмотря на принятую стратегию аккуратного усиления спецификации, верификация оказалась сложной и трудоемкой.

5.4. Спецификация программы heapify

В программе heapify из соображений удобства заменим имена переменных. Определим спецификацию программы heapify.

```
heapify(Ar a, int i: Ar b)
   pre -1<=i<num & heap(a, i + 1, num)
   post perm(a, b) & heap(b)
   measure i
{       if (i < 0) b = a
        else { SiftDown(a, i, num: Ar c);  heapify(c, i - 1: b) }
}
```

Программа heapify по массиву a строит двоичную кучу b. Создание кучи начинается с конца массива a. В соответствии с начальным вызовом heapify(base, num/2 - 1: Ar base1) в программе sort куча считается построенной для диапазона num/2 - 2, поскольку состоит из одиночных вершин. На очередном шаге работы программы heapify(a, i: b) имеется обобщенная куча в диапазоне от i+1 до num-1. Необходимо построить обобщенную кучу от i до num-1. Для этого следует некоторым способом переместить элемент a[i] внутрь дерева с корнем в вершине с индексом i. Эта операция называется *просеиванием вниз* и реализуется программой SiftDown.

Ограничение -1<=i<num в предусловии появляется в результате дедуктивной верификации.

Отметим, что вместо условия i < 0 было бы лучше использовать i <= 0. Но это будет другая программа.

5.5. Спецификация программы sorting

Определим программу sorting вместе со спецификацией. Предварительно изменим имена переменных.

```
sorting(Ar b, int i: Ar b')
   pre  -1<=i<num & heap(b, 0, i+1) & sorted(b, i+1) & twoPart(b, i+1)
    post sorted(b') & perm(b, b')
    measure i
{       if (i <= 0) b' = b
        else {  swap(b, 0, i: Ar b1);
                 siftDown(b1, 0, i: Ar c);
                 sorting(c, i - 1: b')
        }
}
```

Программа sorting реализует сортировку массива b, состоящего из двух частей. Правая часть состоит из элементов b[i+1], b[i+2], …, b[num-1] и уже отсортирована. Левая часть в диапазоне от 0 до i является двоичной кучей. Для начального вызова sorting(base1, num - 1: base') в программе sort левая часть является пустой.

Элемент b[0] является максимальным элементом массива. В отсортированном массиве b' он должен находиться в позиции i. Обменяем местами нулевой элемент и элемент i. После обмена элементов массив b перестал быть кучей. Свойство кучи можно восстановить, если запустить программу siftDown для нулевого элемента.

Условие -1<=i<num в предусловии было вставлено в процессе верификации. Условие twoPart(b, i + 1) было вставлено при верификации ранее еще в работе [12].

formula twoPart(Ar b, **nat** m) = m < num ⇒ b[0] <= b[m];

Данное условие необходимо для доказательства сортированности и отражает тот факт, что левая часть массива b должна быть не больше правой части.

5.6. Спецификация программы siftDown

Представим программу siftDown.

```
siftDown(Ar a, int k, r, m: Ar b)
  pre psiftD(a, k, r, m)
  post qsiftD(a, b, k, r, m)
  measure (r>= m)? 0 : m - r
{     int c = r *2 + 1;
      if (c >= m) b = a
      else { if (c < m - 1 && cmp(a[c], a[c+1]) < 0) c1 = c+1 else c1 = c;
             if (cmp(a[r], a[c1]) >= 0) b = a
             else { swap(a, r, c1: Ar a1); siftDown(a1, c1, m: b)};
      }
}
```

Предусловие psiftD и постусловие qsiftD будут определены ниже.

В предположении, что обобщенная куча построена от r+1 до m-1 (m ≤ num), программа siftDown строит обобщенную кучу от r до m-1. Элемент a[r] сравнивается с наибольшим из потомков и обменивается с ним операцией swap, если потомок оказался большим. После обмена элементов продолжается просеивание вниз исходного элемента a[r], находящегося в позиции потомка.

Спецификация siftDown очевидно должна содержать heap(a, r+1, m) в составе предусловия и perm(a, b) & heap(b, r, m) в постусловии. После нескольких этапов уточнений спецификации были получены следующие достаточно сложные предусловие и постусловие. Рассмотрим предусловие:

formula psiftD(Ar a, **nat** k, r, m) = k<=r<m<=num & heapH(a, k, r, m);

Условия heap(a, r+1, m) оказалось недостаточно, когда позиция r находится в глубине дерева и не является корневой. Предикат heapH определяет условие, что массив a на отрезке от k до m-1 обладает свойством кучи за возможным исключением позиции «дыры» r, где реализуется особое условие. Здесь также пришлось ввести дополнительный параметр k.

formula heapH(Ar a, **nat** k, r, m) =
(k <= father(r) ⇒ heapR(a, r, m)) &
(∀ **nat** j. k <= j < m & j ≠ r ⇒ heapJ(a, j, m))

В позиции «дыры» r свойство кучи реализуется для отца вершины r и любого из потомков вершины r.

formula heapR(Ar a, **nat** r, m) = ∀**nat** j=k..m-1. father(j) = r ⇒ cmp(a[father r], a[j]) >=0;

Рассмотрим постусловие:

formula qsiftD(Ar a, b, **nat** k, r, m) = permE(a, b, r, m) &
(∀**nat** j. j<num ⇒ **if** inTree(r, m, j) **then** heapJ(b, j, m) **else** b[j] = a[j]) &
(k<=father(r) ⇒ cmp(a[father(r)], b[r]) >=0)

Вместо perm(a, b) потребовалось более точное условие permE(a, b, r, m).

Недостаточно точным оказалось исходное установленное условие heap(b, r, m). Во втором конъюнкте формулы qsiftD уточняется, что свойство кучи реализуется только в дереве с корнем r. А за пределами этого дерева итоговый массив b должен совпадать с исходным массивом a.

Потребовалось дополнительное условие, указанное в третьем конъюнкте формулы qsiftD. Свойство кучи должно выполняться для отца вершины r и самой вершиной r.

Предикат inTree определяет принадлежность вершины j дереву с корнем r:

formula inTree(**nat** r, m, j) = r<=j<m & path(r, j);

Дерево с корнем r должно находиться в пределах обобщенной двоичной кучи от r до m-1.

Предикат path(r, j) определяет существование пути от вершины r к вершине j в дереве с корнем r. Очевидное рекурсивное определение предиката path оказалось недопустимым в языке спецификаций why3. Поэтому было использовано следующее индуктивное определение:

inductive path(**int** n, p) =
  | Ref**:** ∀ **int** n. n>=0 ⇒ path(n, n)
  | Lep**:** ∀ **int** n, p. path(n, p) ⇒ path(n, left (p))
  | Rip**:** ∀ **int** n, p. path(n, p) ⇒ path(n, right(p))

Отметим, что вместо cmp(a[c], a[c+1]) < 0 можно было бы написать cmp(a[c], a[c+1])<=0, однако тогда после трансформаций программа не совпадет с исходной программой на языке Си в Приложении 1.

6. Процесс дедуктивной верификации программы sort

Для предикатной программы пирамидальной сортировки, состоящей из программ sort, heapify, sorting и siftDown, построены формулы корректности по правилам, описанным в [11]. Процесс построения формул корректности детально документирован в Приложении 2. В Приложения 3 определена теория на языке P с набором формул корректности. Далее эта теория была закодирована на языке спецификаций why3 системы Why3 [24].

Использовалась система Why3 версии 1.1.1 с SMT-решателями CVC3 версии 2.4.1, CVC4 версии 1.6, Z3 версии 4.7.1 и Gappa версии 1.3.2. Впрочем, SMT-решатель Gappa оказался бесполезным. Стандартное время запуска SMT-решателей: в начале процесса доказательства – 22 сек, в конце – 47 сек. SMT-решатели запускались в параллельном режиме на трех 64-разрядных процессорах 3.3 GHz.

Исходное задание содержало 16 формул корректности. В процессе доказательства введено 95 лемм. Большинство формул корректности и лемм доказано с помощью SMT-решателей при использовании трансформаций. В системе Coq [17] проведено 35 различных доказательств формул корректности, лемм и их частей. Для сложных формул корректности выстраивалась цепочка конкретизирующих лемм. Это помогло лишь частично. Процесс доказательства в целом оказался заметно сложнее, чем в предыдущей версии в 2019г..

В процессе доказательства обнаружена недоказуемость формул корректности. Проведено уточнение предусловий в программах heapify и sorting. Исправлены мелкие ошибки. Ошибочная перестановка параметров формулы heapJ была обнаружена достаточно поздно.

Учитывая неудачный опыт предыдущих попыток [12], почти все предыдущие уточнения спецификаций были отвергнуты. В текущем релизе сначала используются лишь очевидные условия, а необходимые усиления аккуратно определяются в процессе доказательства с целью ограничиться минимальными усилениями. С этой целью пришлось достаточно глубоко пройти в процессе доказательства для определения нужных уточнений.

Проведено три этапа уточнения спецификаций. Сначала определена необходимость предиката heapH, определяющего кучу с «дырой». Далее установлено, что необходимо условие того, что изменения реализуются лишь в пределах дерева (второй конъюнкт qsiftD). Определен предикат inTree , а затем – предикат path. В конце введено свойство кучи для отца текущей вершины r. Это уточнение намного проще введенного в релизе 2019г. предиката root.

Несмотря на все принятые меры, процесс доказательства оказался сложным т трудоемким. SMT-решатели достаточно часто не справлялись с вроде бы простыми формулами. Приходилось переводить доказательство в систему Coq, где доказательство редко было простым. В конце процесса верификации SMT-решатели перестали доказывать пару формул, которые ранее быстро доказывали.

Итоги верификации. Все формулы корректности и все дополнительные леммы полностью доказаны. Верификация проводилась в течение более двух недель параллельно с изучением и освоением системы интерактивного доказательства Coq [17].

7. Обзор работ

В нашей работе [12] проводилась дедуктивная верификация трех алгоритмов пирамидальной сортировки: классического алгоритма Дж. Вильямса [25], алгоритма Флойда [20] и улучшенного алгоритма [23] – самого быстрого алгоритма сортировки в то время. Для классического алгоритма верификация не была доведена до конца. Основной целью была верификация самого быстрого алгоритма. Материал работы [12] был заимствован в релизе 2019г., однако это оказало скорее негативное влияние. Алгоритм и спецификации пришлось сводить от массивов a[1..n] к массивам вида a[0..n-1], алгоритм был существенно модифицирован под исходную программу в Приложении 1. Описание программы было полностью заменено. Заимствованные леммы оказались бесполезными.

Наиболее значимой является работа [19] по дедуктивной верификации трех алгоритмов сортировки: insertion sort, quicksort и heapsort в системе Coq [17]. Файлы, иллюстрирующие процесс доказательства корректности heapsort приведены на странице:

http://why.lri.fr/examples/index.en.html.

Программа и спецификации писались на языке WhyML [24], а доказательство проводилось в Coq. Спецификация программы downheap существенно проще аналогичной siftDown. Не используется свойство кучи с дырой. Нет требования сохранения равенства элементов вне сортируемого поддерева. Вместо этого используется предикат inftree, декларирующий сохранение после работы downheap следующего свойства: для произвольного v все элементы поддерева меньше данного v. Сопоставление по объему и сложности доказательства провести трудно.

В работе [18] на примере программы heapsort демонстрируется технология программирования на базе инвариантов с использованием системы PVS [21]. В спецификации siftdown используется свойство кучи с дырой, но нет предиката, аналогичного inTree или inftree. Поэтому есть серьезные сомнения, что предложенные спецификации достаточны для доказательства корректности heapsort.

Дедуктивная верификация программы heapsort проводилась в работе [5] с использованием среды верификации Isabelle/HOL. В спецификации siftDown свойство кучи с дырой не используется. В постусловии используется предикат равенства элементов до и после на диапазоне. Как показал наш опыт, этого недостаточно. Есть сомнения, что дедуктивная верификация была доведена до конца.

8. Заключение

Проведена обратная трансформация программы пирамидальной сортировки sort.c из библиотеки ядра ОС Linux. Использовались трансформация запроцедуривания и трансформация, обратная уменьшению силы операций. Эти трансформации ранее не использовались. Для полученной предикатной программы написаны спецификации. По программе и спецификациям построены формулы корректности, оформленных в виде теории, перенесенной на язык спецификаций why3. Доказательство проводилось в системах Why3[24] и Coq [17]. В процессе верификации уточнялись спецификации. Доказательство формул корректности и всех лемм проведено полностью. Процесс дедуктивной верификации оказался сложным и трудоемким.

Итоги верификации. Библиотечная программа sort корректна относительно спецификации. Для функции cmp_func определена спецификация в виде следующей теории.

```
theory Compare {
type CMP= predicate(T, T: int)
    CMP cmp;
    axiom Refl :   ∀ x: T. cmp(x, x) = 0
    axiom Simm :   ∀ T x, y. cmp(x, y) = - cmp(y, x);
    axiom TotalLe : ∀ T x, y. cmp(x, y) <=0 or cmp(y, x) <=0;
    axiom TransLe : ∀ T x, y, z. cmp(x, y)<=0 & cmp(y, z)<=0 ⇒ cmp(x,z)<=0;
}
```

Аксиома антисимметричности: x <= y & y <= x ⇒ x = y – не использовалась.

Следует отметить, что спецификация и верификации исходной программы sort.c существующими инструментами была бы на порядок сложнее проделанной и описанной в настоящей работе.

Доступна полная версия текста настоящей работы: https://persons.iis.nsk.su/files/persons/pages/sort9.pdf.

Список литературы

1. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL:http://www.iis.nsk.su/persons/vshel/files/rules.zip (дата обращения 02.09.2020)
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. 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
4. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P // Новосибирск, 2018. 42с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения 02.09.2020)
5. Ковалев М.С., Далингер Я.М., Мяготин А.В. Формальная верификация программной реализации алгоритма пирамидальной сортировки на языке Си-0 // Научно-технические ведомости СПбГПУ. 2010. № 4.С.83-92.
6. 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 02.09.2020)
7. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21. [Электронный ресурс]. URL: https://www.iis.nsk.su/files/preprints/154.pdf (дата обращения 02.09.2020)
8. 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

9. Шелехов В.И. Дедуктивная верификация программы конкатенации строк с применением обратной трансформации // Знания-Онтологии-Теории (ЗОНТ-19). Новосибирск, 2019. 19с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/logcflc1.pdf (дата обращения 02.09.2020)
10. 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 02.09.2020)
11. 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 02.09.2020)
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 02.09.2020)
13. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с.http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf . (дата обращения 02.09.2020)
14. 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).
15. 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).
16. 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).
17. The Coq Proof Assistant. [Электронный ресурс]. URL: http://coq.inria.fr (дата обращения 02.09.2020)
18. Eriksson J. and Back R.-J. Applying PVS Background Theories and Proof Strategies in Invariant Based Programming // LNCS 6447, 2010. P. 24–39.
19. Filliâtre J.-C., Magaud N. Certification of sorting algorithms in the system Coq // Theorem Proving in Higher Order Logics: Emerging Trends, 1999.
20. Floyd R.W. Algorithm 245 – Treesort **3** // Commun. ACM. 1964. Vol. 7 (12). P. 701.
21. PVS Specification and Verification System. SRI International. [Электронный ресурс]. URL: http://pvs.csl.sri.com/ . (дата обращения 02.09.2020)
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. Wang X.D., Wu Y. J. An improved HEAPSORT algorithm with n logn – 0.788928n comparisons in the worst case // J. Computer Science and Technology. 2007. Vol. 22 (6). P. 898–903.
24. Why 3. Where Programs Meet Provers. [Электронный ресурс]. URL: http://why3.lri.fr (дата обращения 02.09.2020)
25. Williams J.W.J., Algorithm 232 // Commun. ACM. 1964. P. 347–348.

Приложение 1

Исходная программа sort на языке Си

```
#define KERNRELEASE "TEST"
#define CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS 1
#define __take_second_arg(__ignored,val,...) val
#define ____is_defined(arg1_or_junk) __take_second_arg(arg1_or_junk 1, 0)
#define ____or(arg1_or_junk,y) __take_second_arg(arg1_or_junk 1, y)
#define ___is_defined(val) ____is_defined(__ARG_PLACEHOLDER_ ##val)
#define ___or(x,y) ____or(__ARG_PLACEHOLDER_ ##x, y)
#define __is_defined(x) ___is_defined(x)
#define __or(x,y) ___or(x, y)
#define IS_BUILTIN(option) __is_defined(option)
#define IS_MODULE(option) __is_defined(option ##_MODULE)
#define IS_ENABLED(option) __or(IS_BUILTIN(option), IS_MODULE(option))
typedef unsigned long __kernel_ulong_t;
typedef unsigned int u32;
typedef unsigned long long u64;
typedef __kernel_ulong_t __kernel_size_t;
typedef __kernel_size_t         size_t;

//----------------------------------------------------------------------------

static void generic_swap(void *a, void *b, int size)
{
    char t;

    do {
            t = *(char *)a;
            *(char *)a++ = *(char *)b;
            *(char *)b++ = t;
    } while (--size > 0);
}

static void u32_swap(void *a, void *b, int size)
{
    u32 t = *(u32 *)a;
    *(u32 *)a = *(u32 *)b;
    *(u32 *)b = t;
}

static void u64_swap(void *a, void *b, int size)
{
    u64 t = *(u64 *)a;
    *(u64 *)a = *(u64 *)b;
    *(u64 *)b = t;
}

static int alignment_ok(const void *base, int align)
```

```
{
   return IS_ENABLED(CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS) ||
          ((unsigned long)base & (align - 1)) == 0;
}

void sort(void *base, size_t num, size_t size,
     int (*cmp_func)(const void *, const void *),
     void (*swap_func)(void *, void *, int size))
{
   /* pre-scale counters for performance */
   int i = (num/2 - 1) * size, n = num * size, c, r;

   if (!swap_func) {
          if (size == 4 && alignment_ok(base, 4))
                 swap_func = u32_swap;
          else if (size == 8 && alignment_ok(base, 8))
                 swap_func = u64_swap;
          else
                 swap_func = generic_swap;
   }

   /* heapify */
   for ( ; i >= 0; i -= size) {
          for (r = i; r * 2 + size < n; r  = c) {
                 c = r * 2 + size;
                 if (c < n - size &&
                               cmp_func(base + c, base + c + size) < 0)
                        c += size;
                 if (cmp_func(base + r, base + c) >= 0)
                        break;
                 swap_func(base + r, base + c, size);
          }
   }

   /* sort */
   for (i = n - size; i > 0; i -= size) {
          swap_func(base, base + i, size);
          for (r = 0; r * 2 + size < i; r = c) {
                 c = r * 2 + size;
                 if (c < i - size &&
                               cmp_func(base + c, base + c + size) < 0)
                        c += size;
                 if (cmp_func(base + r, base + c) >= 0)
                        break;
                 swap_func(base + r, base + c, size);
          }
   }
}
```

Приложение 3

Теории для доказательства формул корректности

Соберем все сгенерированные формулы корректности в теорию.

```
   theory Sort {
nat num;
type T;
nat Di = 0..num-1;
type Ar = array (T, Di);
cmp(T, T: int);
swap(Ar a, Di j, k: Ar a’) post exchange(a, a’, j, k);
Ar base;
      axiom Refl :   ∀ x: T. cmp(x, x) = 0
      axiom Simm :   ∀ T x, y. cmp(x, y) = - cmp(y, x);
      axiom TotalLe : ∀ T x, y. cmp(x, y) <=0 or cmp(y, x) <=0;
      axiom TransLe : ∀ T x, y, z. cmp(x, y)<=0 & cmp(y, z)<=0 ⇒ cmp(x,z)<=0;
formula sorted(Ar a, nat m) =  ∀ k, j= m..num-1. k < j ⇒  cmp(a[k], a[j]) <= 0;
formula perm(Ar a, b) = permut_all(a, b);
formula permE(Ar a, b, nat m, n) = permut_sub(a, b, r, m);

formula left(nat j : nat) = j * 2 + 1;
formula right(nat j : nat) = j * 2 + 2;
formula father(nat j : int) = (j-1) / 2;
   formula heap(Ar a) = ∀ nat j = 0 .. num-1. heapJ(a, num, j)
   formula heapJ(Ar a, nat j, m) =    (left(j) < m ⇒ cmp(a[j], a[left(j)])>=0)) &
                                       (right(j) < m ⇒ cmp(a[j], a[right(j)])) .
   formula heap(Ar a, nat i, m) = ∀ nat j = i .. m-1. heapJ(a, j, m);
formula twoPart(Ar b, nat m) = m < num ⇒ b[0] <= b[m];
   formula heapR(Ar a, nat r, m) = ∀nat j=k..m-1. father(j) = r ⇒ cmp(a[father r], a[j])
   >=0;
   formula heapH(Ar a, nat k, r, m) =
               (k <= father(r) ⇒ heapR(a, r, m)) &
               (∀ nat j.  k <= j < m & j ≠ r  ⇒ heapJ(a, j, m))
   formula psiftD(Ar a, nat k, r, m) = k<=r<m<=num & heapH(a, k, r, m);
   formula inTree(nat r, m, j) = r<=j<m & path(r, j);
   inductive path(int n, p) =
         | Ref:   ∀ int n. n>=0 ⇒ path(n, n)
         | Lep:   ∀ int n, p. path(n, p) ⇒ path(n, left (p))
         | Rip:    ∀ int n, p. path(n, p) ⇒ path(n, right(p))
   formula qsiftD(Ar a, b, nat k, r, m) = permE(a, b, r, m) &
          (∀nat j. j<num ⇒ if inTree(r, m, j)  then  heapJ(b, j, m)  else  b[j] = a[j]) &
          ( k<=father(r) ⇒ cmp(a[father(r)], b[r]) >=0 )
formula qsort(Ar base, base9) = sorted(base9, 0) & perm(base, base9);
formula pheapify(Ar a, int i) = -1<=i<num & heap(a, i+1, num);
formula qheapify(Ar a, b) = perm(a, b) & heap(b, 0, num);
```

```
formula psorting(Ar b, int i) = -1<=i<num & heap(b, 0, i+1) & sorted(b, i+1) & twoPart(b,
   i+1);
formula qsorting(Ar b, b9) = sorted(b9) & perm(b, b9);

   RS1: pheapify(base, (num div 2) – 1);
   RS2: qheapify(base, b) ⇒ psorting(b, num-1);
   RS3: qheapify(base, b) & qsorting(b, base9) ⇒ qsort(base, base9);

   COR1: pheapify(a, i) & i < 0 & b = a ⇒ qheapify(a, b);
   RS4: pheapify(a, i) & i >= 0 ⇒ psiftD(a, i, num);
   RS5: pheapify(a, i) & i >= 0 & qsiftD(a, c, i, num) ⇒ pheapify(c, i - 1) & i-1<i;
   RS6: pheapify(a, i) & i >= 0 & qsiftD(a, c, i, num) & qheapify(c, b) ⇒ qheapify(a, b);
   COR2: psorting(b, i) & i <= 0 & b9 = b ⇒ qsorting(b, b9);
   QS1: psorting(b, i) & i > 0 ⇒ 0<num & i<num;
   RS7: psorting(b, i) & i > 0 & exchange(b, b1, 0, i) ⇒ psiftD(b1, 0, i);
   RS8: psorting(b, i) & i > 0 & exchange(b, b1, 0, i) & qsiftD(b1, c, 0, i) ⇒
                                                  psorting(c, i - 1) & i-1<i;
   RS9: psorting(b, i) & i > 0 & exchange(b, b1, 0, i) & qsiftD(b1, c, 0, i) &
                                            qsorting(c, b9) ⇒ qsorting(b, b9)
   } Sort

   theory SiftDown {
   import Sort;
   COR3: psiftD(a, r, m) & c = r *2 + 1 & c >= m & b = a ⇒ qsiftD(a, b, r, m);
   formula cc2(Ar a, nat m, c, c1) =
               if (c < m - 1 && cmp(a[c], a[c+1]) < 0) c1 = c+1 else c1 = c;
   formula fcc(Ar a, nat r, m, c, c1) =
               psiftD(a, r, m) & c = r *2 + 1 &  c < m & cc2(a, m, c, c1);
   COR4: fcc( a, r, m, c, c1) & (cmp(a[r], a[c1]) >= 0) & b = a ⇒ qsiftD(a, b, r, m);
   formula h(nat r, m: nat) = (r>= m)? 0 : m - r
   RB1: fcc( a, r, m, c, c1) & cmp(a[r], a[c1]) < 0 & exchange(a, a1, r, c1) ⇒
                             0<= c1 < num & psiftD(a1, c1, m) & h(c1, m)<h(r, m);
   RB2: fcc( a, r, m, c, c1) & cmp(a[r], a[c1]) < 0 & exchange(a, a1, r, c1) & qsiftD(a1, b, c1,
   m) ⇒
                                                   qsiftD(a, b, r, m);
   } SiftDown
```