Верификация программы быстрой сортировки с двумя опорными элементами

В.И. Шелехов, М.С. Чушкин

Институт систем информатики СО РАН,
Новосибирский государственный университет, г. Новосибирск,
e-mail: vshel@iis.nsk.su

Аннотация. Быстрая сортировка с двумя опорными элементами является одной из наиболее эффективных в базисном наборе алгоритмов сортировки стандартной библиотеки Java Development Kit (JDK) для языка Java. Beckert B. и др. построили спецификацию этого алгоритма на языке JML и успешно провели дедуктивную верификацию в системе KeY.

Предикатное программирование обладает серьезными преимуществами перед императивным программированием. Дедуктивная верификация предикатной программы по нашим оценкам требует затрат примерно в 4 раза меньше в сравнении с верификацией аналогичной императивной программы. Применение оптимизирующих трансформаций к предикатной программе позволяет получить программу, по эффективности не уступающую написанной вручную императивной программе.

В настоящей работе описывается построение и дедуктивная верификация этой программы в технологии предикатного программирования. Дедуктивная верификация в системе PVS проведена одним из авторов за одну неделю, на порядок быстрее в сравнении с описанной в работе Beckert B. и др. Эффективная программа сортировки на языке Java получена применением набора оптимизирующих трансформаций к исходной предикатной программе. Планируется провести ее сравнение с эксплуатируемой в библиотеке JDK.

Ключевые слова: быстрая сортировка с двумя опорными элементами, дедуктивная верификация, SMT-решатель, система автоматического доказательства PVS

Verification of Dual Pivot Quicksort Program

V.I. Shelekhov, M.S. Chushkin

1 A.P. Ershov Institute of Informatics Systems,
Siberian Branch of the Russian Academy of Sciences,
Novosibirsk State University, Novosibirsk

Abstract. A dual pivot quicksort (DPQS) algorithm performs substantially better than other sorting algorithms in the Java Development Kit (JDK). Beckert B. et al. had constructed the specification of the DPQS algorithm in the JML language. It had been successfully proved correct using the deductive verification engine KeY.

Predicate programming has the essential advantages against imperative programming. Deductive verification of a predicate program is approximately 4 times faster than deductive verification of the analogous imperative program. Application of the special optimizing transformations to a predicate program results in the equivalent effective imperative program compared in efficiency with manually programmed.

In the current paper, the construction and deductive verification of the DPQS predicate program using the interactive prover PVS is described. Deductive verification of the DPQS predicate program has been made within a week. An effective Java program has been obtained by applying program transformations to the DPQS predicate program. It's comparison with the DPQS algorithm in the JDK is in our plans.

Keywords: dual pivot quicksort, deductive verification, SMT solver, PVS

В процессе дедуктивной верификации программы сортировки TimSort, входящей в стандартную библиотеку JDK для языка Java, обнаружена ошибка [2]. Этот факт привлек внимание общественности и вдохновил на проведение дедуктивной верификации программы быстрой сортировки с двумя опорными элементами, Dual Pivot Quicksort (DPQS) [3]. Это более эффективная версия классической быстрой сортировки – одна из эффективных программ сортировки в библиотеке JDK.

Дедуктивная верификация императивных программ высокой эффективности – непростая задача. Верификация алгоритма DPQS была успешно проведена [1]. Доступны детальные данные результатов верификации [4], которые свидетельствуют, что верификация была сложна, трудоемка (2.5 месяца) и потребовала высокой квалификации. Следует отметить, что работ по формальной верификации эффективных императивных программ сортировки крайне мало. Кроме указанных [1, 2] есть еще работа тех же авторов по верификации программ counting sort и radix sort [5]. Это при большом числе работ по формальной верификации и синтезу программ сортировки.

Наш интерес к данной работе обусловлен тем, что ранее нами была проведена дедуктивная верификации самой быстрой (на то время) программы

сортировки в технологии предикатного программирования [6]. Причем эта программа существенно сложнее программы DPQS. Предикатное программирование обладает серьезными преимуществами перед императивным программированием. Дедуктивная верификация предикатной программы по нашим оценкам требует затрат в 4 раза меньше в сравнении с верификацией аналогичной императивной программы. Теперь появилась возможность провести сравнение на конкретной программе. Отметим, что для предикатной программы применением системы оптимизирующих трансформаций можно получить императивную программу, по эффективности не уступающую написанной вручную в традиционной технологии и обычно короче.

Цель нашей работы: построить предикатную программу быстрой сортировки с двумя опорными элементами, провести ее дедуктивную верификацию в системе интерактивного автоматического доказательства PVS [7]. Сравнить данные дедуктивной верификации с работой [1]. Далее методом оптимизирующей трансформации получить императивную программу и сравнить ее с эксплуатируемой в библиотеке JDK.

В первом разделе настоящей работы дается краткое описание языка предикатного программирования P [8]. Используемый метод дедуктивной верификации изложен в разделе 2. В разделе 3 описывается построение предикатной программы быстрой сортировки с двумя опорными элементами. Дедуктивная верификация предикатной программы описывается в разделе 4. Построение эффективной Java-программы методом оптимизирующей трансформации представлено в разделе 5. В заключении представлены данные по сравнению дедуктивной верификации у нас и в работе [1].

1. Предикатное программирование

Предикатная программа относится к классу программ-функций [9] и является предикатом в форме вычислимого оператора H(x**:** y) с аргументами x и результатами y. Минимальный полный *базис предикатных программ* определен в виде языка $\mathbf{P_0}$. Предикатная программа определяется следующей конструкцией:

<имя предиката>(<аргументы>**:** <результаты>) { <оператор> }

Пусть x, y и z обозначают разные непересекающиеся наборы переменных. Набор x может быть пустым, наборы y и z не пусты. В составе набора переменных x может использоваться логическая переменная e со значениями **true** и **false**. Пусть B и C – имена предикатов, A и D – имена переменных предикатного типа. Операторами являются: *оператор суперпозиции* B(x**:** z); C(z**:** y), *параллельный оператор* B(x**:** y) || C(x**:** z), *условный оператор* **if (**e) B(x**:** y) **else** C(x**:** y), *вызов предиката* B(x**:** y). Программа вида H(x**:** D) { D(y**:** z) {B(x, y**:** z)} } содержит *оператор каррирования* D(y**:** z) {B(x, y**:** z)}. Его результат – новый предикат D, получаемый фиксацией значения набора x.

На базе языка $\mathbf{P_0}$ последовательным расширением [10] построен язык предикатного программирования **P** [8]: $\mathbf{P_0} \rightarrow \mathbf{P_1} \rightarrow \mathbf{P_2} \rightarrow \mathbf{P_3} \rightarrow \mathbf{P_4} = \mathbf{P}$. В языке **P** нет

циклов и указателей; вместо них используются рекурсивные программы и алгебраические типы данных.

Определим программу with модификации результата предиката A для значения аргумента x = i через оператор каррирования:

with(A, i, z: D) { D(x: y) { **if** (x = i) y = z **else** A(x: y)} } .

Вызов программы with(B, i, z: C) будем записывать в виде C = B **with** (i: z). Модификация вида B **with** (i: z, j: w) определяется как (B **with** (i: z)) **with** (j: w).

В языке P имеется развитая система типов: подтипы, структуры, множества, алгебраические типы, предикатные типы, массивы. Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами. Типы могут быть параметризованы переменными. Для диапазона целых значений от m до n используется обозначение m..n. Для *вырезки массива* a на диапазоне m..n используется запись a[m..n].

Гиперфункция – программа с несколькими *ветвями* результатов. Гиперфункция A(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 A(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.

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций [10], переводящих программу на императивное расширение языка **P**:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей.

2. Дедуктивная верификация

Спецификацией программы H(x: y) являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Спецификация записывается в виде: [P(x), Q(x, y)]. *Тотальная корректность* программы относительно спецификации определяется формулой:

$$H(x: y)\ \textbf{corr}\ [P(x), Q(x, y)] \cong \forall x.\ P(x) \Rightarrow [\forall y.\ \mathcal{R}(H)(x, y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ \mathcal{R}(H)(x, y) \quad (1)$$

Здесь $\mathcal{R}(H)$ – формальная операционная семантика [11], определяющая предикат, истинный при завершении программы H(x: y).

Для основных операторов (параллельного, условного и суперпозиции) разработана система правил доказательства их корректности [12], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности (1). Корректность правил доказана [13] в системе PVS [7]. В системе предикатного программирования реализован генератор формул корректности программы. Значительная часть формул доказывается автоматически SMT-решателем CVC4 [14]. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [7].

Метод дедуктивной верификации предикатных программ отличается от классического метода Хоара [15] для доказательства частичной корректности императивных программ. Генерируемый набор формул корректности проще и короче [12], в частности, учитываются различия между параллельным оператором и операторов суперпозиции.

3. Программа быстрой сортировки с двумя опорными элементами

Разработка программы на языке **P** начинается с определения типов данных исходной задачи. Объектами сортировки являются одномерные массивы элементов некоторого произвольного типа T. Для типа T должно быть определено отношение “≤” линейного (или тотального) порядка. Для алгоритма быстрой сортировки необходимо определить массив с произвольными границами. Пусть индексы массива находятся в диапазоне от le до ri. Тип T и границы le и ri являются внешними параметрами программы:

```
type T(“≤”, “<”, “≥”, “>”);
int le, ri;
```

Для упрощения описания программы применяется конструкция:

```
context int le, ri;
```

Переменные le и ri указываются в качестве внешних параметров при определении типов, программ и формул. Внешние параметры могут быть опущены при использовании типов, а также в вызовах программ и формул. Это способствует улучшению восприятия программы.

Определим тип диапазона от le до ri и тип сортируемого массива:

```
type LR(int le, ri) = le..ri;
type Arl(int le, ri) = array (T, LR);
```

Спецификация программы сортировки: итоговый массив должен быть отсортирован и получен из исходного массива перестановкой элементов. Свойство перестановочности определяется формулой perm. Сначала определим тип F биективных (взаимно-однозначных) функций.

```
type F(int le, ri) = subtype (predicate (LR: LR) f: bijective(f));
```

Здесь **predicate** (LR**:** LR) – тип функций, заданных на отрезке [le, ri] со значениями на этом же отрезке. Предикат bijective(f) истинен для биективной функции f.

```
formula perm(int le, ri) (Arl a, b) = ∃ F f. ∀ LR j. b[j] = a[f(j)];
```

Предикат perm определяет перестановочность массивов a и b в случае существования биективной функции f, отображающей элементы массива b в элементы массива a.

Свойство сортированности определено предикатом sorted: элемент с большим номером не может быть меньше элемента с меньшим номером.

```
formula sorted(int le, ri)(Arl a) = ∀ LR i, j. i < j ⇒ a[i] ≤ a[j];
```

Формальная спецификация программы сортировки самого верхнего уровня:

```
sort(int le, ri)(Arl a: Arl a') post perm(a, a') & sorted(a')
```

Для имени a' подразумевается, что в реализации переменная a' должна быть склеена с a. Иначе говоря, отсортированный массив должен быть получен в том же массиве a.

Алгоритм быстрой сортировки реализует *переупорядочивание* сортируемого массива с разбиением на две *секции* относительно некоторого *опорного элемента* piv. В первой секции перед элементом piv все элементы должны быть не больше piv. Во второй секции после piv все элементы должны быть не меньше piv. Далее независимо сортируются каждая из двух секций массива.

Более эффективным является алгоритм быстрой сортировки с двумя опорными элементами piv1 и piv2 (piv1<piv2) и тремя секциями [3]. После этапа переупорядочивания сортируемого массива реализуются следующие соотношения:

$$\begin{aligned} &a[x]=piv1,\ a[y]=piv2 \text{ для некоторых } x \text{ и } y, \text{ где } x<y; \\ &a[j]\le piv1 \quad \text{для } j<x; \\ &piv1\le a[j]\le piv2 \quad \text{для } x<j<y; \\ &piv2\le a[j] \quad \text{для } j>y. \end{aligned} \tag{2}$$

Есть дополнительное условие: все секции должны быть непустыми.

Алгоритм применяется для массива с числом элементов больше 46. На начальном этапе работы алгоритма случайным образом выбирается пять различных элементов массива. Они сортируются между собой. Если все элементы разные, второй и четвертый элементы назначаются в качестве piv1 и piv2. Если указанные условия не выполняются, запускаются другие алгоритмы сортировки.

Спецификация начального этапа представлена ниже в виде гиперфункции sortG.

```
formula Pivots(int le, ri)(Arl a, T piv1, piv2) = piv1 < piv2 &
                                                  (∃ int j. le<j<ri & a[j]<piv1) &
                                                  (∃ int j. le<j<ri & a[j]>piv2);
sortG(int le, ri)(Arl a: Arl a' #1 : Arl a', T piv1, piv2 #2)
pre 2: ri - le > 46
post 1: perm(a, a') & sorted(a')
post 2: perm(a, a'with(le: piv1, ri: piv2)) & Pivots(a', piv1, piv2);
```

Первая ветвь гиперфункции соответствует случаю, когда запускаются другие алгоритмы сортировки. Постусловие по первой ветви определяет сортированность итогового массива и его перестановочность с исходным, т.е. выполнение спецификации верхнего уровня.

Если все указанные выше условия выполняются, реализуется вторая ветвь гиперфункции. Опорные элементы помещаются в переменные piv1 и piv2. Кроме того, элемент piv1 обменивается с a[le], а piv2 – с a[ri]. В действительности, элементы piv1 и piv2 не записываются в позиции le и ri, поскольку в этом нет необходимости. С учетом этого, постусловие по второй ветви определяет, что модифицированный массив a' перестановочен с исходным. Формула Pivots фиксирует наличие элемента, меньшего piv1, и элемента, большего piv2.

Полная программа сортировки представлена ниже.

```
sort(int le, ri)(Arl a:  Arl a')
post perm(a, a') & sorted(a') measure (ri<le)? 0 : ri – le + 1 {
    sortG(le, ri)(a:  a' #return :   Arl a1, T piv1, piv2);
    partition(le, ri)(a1, piv1, piv2:   Arl a2, LR L, R);
    {   sort(le, L-2)(a2[le..L-2]:   Arl(le, L-2) a'[le..L-2]) ||
        sort(R+2, ri)(a2[R+2..ri]:   Arl(R+2, ri) a'[R+2..ri]) ||
        sort(L, R)(a2[L..R]:   Arl(L, R) a'[L..R])
    }
}
```

Сначала вызывается гиперфункция sortG. Выход #**return** первой ветви гиперфункции реализует завершение программы sort. Вызов программы partition реализует переупорядочивание массива a1 с разбиением на три секции. Далее в параллельном режиме запускаются рекурсивные вызовы программы sort для каждой из трех секций.

Границы между секциями в программе partition определяются переменными L и R. Средняя секция – часть (вырезка) массива с индексами от L до R. В позицию L-1 записывается элемент piv1, а в позицию R+1 – элемент piv2. Первая секция – вырезка массива от le до L-2. Третья секция – вырезка от R+2 до ri.

Для упрощения описания оставшейся части расширим набор переменных, определяемых внешними параметрами:

```
context int le, ri, T piv1, piv2;
```

Перечисленные переменные являются параметрами-аргументами программ и формул. Эти переменные опускаются в вызовах программ и формул.

Постусловие программы partition определяется формулой:

```
formula Qpart(int le, ri, T piv1, piv2)(Arl a, Arl a', LR L, R) =
    perm(a with(le: piv1, ri: piv2), a') &
    L > le & R < ri &
    a'[L-1]=piv1 & a'[R+1]=piv2 &
    (∀int j. le<=j<L-1 ⇒ a'[j]<piv1) &
    (∀int j. R+1<j<=ri ⇒ piv2<a'[j]) &
    (∀int j. L<=j<=R ⇒ piv1<=a'[j]<=piv2);
```

Здесь итоговый массив a' перестановочен с исходным и удовлетворяет соотношениям (2).

Общая схема разбиения массива a на каждом очередном шаге работы программы partition показана на Рис.1.

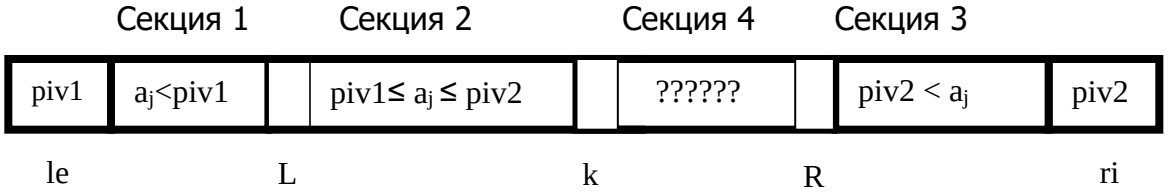

Рис. 1. Секции программы быстрой сортировки с двумя опорными элементами

Позиции le и ri зарезервированы под элементы piv1 и piv2. Однако эти элементы там не хранятся. Переменные L, R и k определяют *границы* между секциями. Секция 1 размещается в диапазоне от le+1 до L-1. Секция 3 – вырезка от R+1 до ri-1. Секция 2 – вырезка от L до k-1. Секция 4 – вырезка от k до R. В ней находятся необработанные элементы массива a.

Программа partition последовательно просматривает секцию 4. После сравнения с piv1 и piv2 очередной элемент перемещается в одну из первых трех секций с соответствующим изменением границ. В начальный момент работы программы секция 4 – весь массив a без граничных элементов.

```
partition(int le, ri)(Arl a: Arl a', LR L, R)
pre Pivots(a, piv1, piv2) post Qpart(a, a', L, R) {
    scanL(Arl a, le+1: LR L0);
    split(a, L0, ri-1, L0: a', L, R)
};
```

Программа scanL сканирует элементы от позиции le+1, пропуская относящиеся к первой секции, и останавливается на первом элементе из других секций. Соответственно продвигаются границы L и k, которые подаются аргументами вызываемой далее программы split, реализующей обработку секции 4.

Границы L, R и k являются аргументами программы split, реализующей последовательную обработку секции 4. Формула partG является частью предусловия.

```
formula partG(int le, ri, T piv1, piv2)(Arl a, LR L0, R0, k) =
    L0 > le & R0 < ri & L0 <= R0 +1 & L0 <= k &
    (∀int j. le<j<L0 ⇒ a[j]<piv1) &
    (∀int j. R0<j<ri ⇒ piv2<a[j]);
```

Здесь фиксируются условия для первой и третьей секции для текущих границ. Определяются соотношения для разных границ. Они необходимы для успешного проведения доказательств. Полное предусловие определяется формулой Psplit.

```
formula Psplit(int le, ri, T piv1, piv2)(Arl a, LR L0, R0, k) =
    Pivots(a, piv1, piv2) & partG(a, L0, R0, k) & (∀int j. L0<=j<k ⇒ piv1<=a[j]<=piv2);
```

Здесь дополнительно включено условие на вторую секцию.

Программа split приведена ниже.

```
split(int le, ri, T piv1, piv2)(Arl a, LR L0, R0, k:   Arl a', LR L, R)
pre Psplit(a, L0, R0, k) post Qpart(a, a', R, L) measure R0 - L0 + ri - k {
    if (k > R0) Fin(L0, R0, a: a', L, R)
    else {
      T ak = a[k];
      if (ak<piv1) split(a with(k: a[L0], L0: ak), L0+1, R0, k+1:  a', L, R)
      else if (ak>piv2) {
              scanR(a, R0 : LR R1);
              if (k > R1) Fin(L0, R1, a: a', L, R)
              else {
                  T aR = a[R1];
                  if (aR>=piv1) split(a with(k: aR, R1: ak), L0, R1-1, k+1:  a', L, R)
                  else split(a with(k: a[L0], L0: aR, R1: ak), L0+1, R1-1, k+1:  a', L, R)
              }
      } else split(a, L0, R0, k+1:  a', L, R)
    }
};
```

Аргументы L0 и R0 определяют начальные значения границ, результаты L и R – конечные. Постусловие то же самое, что и для программы partition.

Окончанию просмотра секции 4 соответствует условие k > R0. В этом случае вызывается программа Fin, которая вставляет элементы piv1 и piv2 у границ секции 2. А элементы, которые ранее там находились, перемещаются в позиции le и ri.

Вызов программы scanR реализует сканирование в обратном порядке элементов, прилегающих к секции 3, на предмет вхождения туда этих элементов. Соответственно сдвигается граница R. При таком сканировании секция 4 может оказаться исчерпанной. В этом случае вызывается программа Fin.

В остальных случаях очередной элемент ak перемещается в нужную секцию. При этом возможен обмен с другим элементом и продвижение границ. Если ak>piv2, то сначала ak обменивается с последним элементом секции 4 (после scanR), который затем перемещается в первую или вторую секцию.

Программа scanL реализует начальное построение первой секции. Массив a сканируется от позиции L с пропуском элементов, относящихся к первой секции. В предусловии PscanL указывается, что элементы перед границей L принадлежат секции 1. Гарантируется наличие элемента, не принадлежащего секции 1. Это необходимо для доказательства завершения программы scanL. В постусловии QscanL определяется: все элементы перед позицией L0 принадлежат секции 1, а элемент a[L0] – не принадлежит.

```
formula PscanL(int le, ri, T piv1)(Arl a, LR L) =
            L>le & (∀int j. le<j<L ⇒ a[j]<piv1) & (∃ int i. L<=i<ri & not a[i]<piv1);
formula QscanL(int le, T piv1)(Arl a, LR L, L0) =
            L0>le & L<=L0 & piv1<=a[L0] & (∀int j. le<j<L0 ⇒ a[j]<piv1);
scanL(int le, ri, T piv1)(Arl a, LR L: LR L0)
pre PscanL(a, L) post QscanL(a, L, L0) measure ri-L {
    if (a[L]<piv1) scanL(a, L+1: L0)
    else L0 = L
}
```

Программа scanR сканирует в обратном порядке элементы от границы R, представленной переменной R0. Элементы, принадлежащие секции 3, пропускаются с продвижением границы R. Программа останавливается на первом элементе, не принадлежащем секции 3. В предусловии PscanR определяется, что все элементы после границы R принадлежат секции 3. В целях доказательства завершения программы scanR имеется условие существования элемента, не принадлежащего секции 3. В постусловии QscanR определяется, что элементы после границы R принадлежат секции 3, а элемент a[R] – не принадлежит.

```
formula PscanR(int le, ri, T piv2)(Arl a, LR R0) =
          R0<ri & (∀int j. R0<j<ri ⇒ a[j]>piv2) & (∃ int i. le<i<=R0 & a[i]<=piv2);
formula QscanR(int ri, T piv2)(Arl a, LR R0, R) =
          R<ri & R<=R0 & a[R]<=piv2 & (∀int j. R<j<=R0 ⇒ a[j]>piv2);
scanR(int le, ri, T piv2)(Arl a, LR R0: LR R)
pre PscanR(a, R0) post QscanR(a, R0, R) measure R0 {
    if (a[R0]>piv2) scanR(a, R0 – 1: R)
    else R = R0
}
```

Программа Fin вставляет элементы piv1 и piv2 у границ секции 2. А элементы, которые ранее там находились, перемещаются в позиции le и ri. В роли спецификации выступает сама программа.

```
Fin(int le, ri, T piv1, piv2)(LR L0, R0, Arl a: Arl a', LR L, R)
{       L = L0 || R = R0 ||
        a' = a with(le: a[L0-1], L0-1: piv1, ri: a[R0+1], R0+1: piv2)
}
```

4. Дедуктивная верификация

Формулы корректности предикатной программы быстрой сортировки с двумя опорными элементами генерируются автоматически в системе предикатного программирования. Для каждой формулы корректности сначала делается попытка доказать ее с помощью SMT-решателя CVC4. Если доказать не удается, формула транслируется для доказательства в системе PVS.

Последняя версия языка **P** разрабатывалась с ориентацией на систему PVS. Тем не менее, ряд конструкций, например, оператор модификации, не имеют прямого эквивалента в PVS. Это оператор a' = a2 **with** (le..L-2: b', L..R: d',

R+2..ri: c') и вырезка вида a2[le..L-2]. Конструкции такого вида вручную закодированы для PVS.

В процессе доказательства приходилось несколько раз уточнять предусловия программ, чтобы обеспечить доказуемость формул корректности. Так, например, набор условий:

L0 > le & R0 < ri & L0 <= R0 +1 & L0 <= k

изначально отсутствовал в составе формулы partG. Сначала потребовалось вставить условие L0>le & R0<ri. Для доказательства одной из формул потребовалось вставить L0<=R0+1, а еще позже – L0<=k.

Доказательство всех формул корректности проведено одним из авторов в течение недели. Доступны теории и доказательства [16] в системе PVS.

5. Оптимизирующие трансформации

Эффективная императивная программа получается из предикатной программы применением системы оптимизирующих трансформаций. На первом этапе проводится склеивание переменных. Далее реализуется замена хвостовой рекурсии циклом. На следующем этапе раскрываются модификаторы **with** с заменой на присваивания элементам массива. Реализуются упрощения. Вызовы Fin выносятся за цикл. Тела программ открыто подставляются на место их вызовов. Сортируемый массив представляется вырезкой глобального массива. Получаем программу на императивном расширении языка **P**. Наконец, реализуется конвертация на язык Java с заменой типов LR и T на **int**.

```
sort(int[] a, int le, ri) {
    <  Сортировка малых массивов. Вычисление piv1 и piv2  >
    int L;
    for(L = le+1; a[L]<piv1; L = L+1);
    int R = ri-1;
    for(int k = L ; ; k = k+1) {
       if (k > R) break;
       int ak = a[k];
       if (ak<piv1) { a[k] = a[L]; a[L] = ak; L=L+1 }
       else if (ak>piv2) {
                for(;a[R]>piv2; R=R–1);
                if (k > R)  break;
                int aR = a[R];
                if (aR >= piv1) { a[k] = aR; a[R] = ak; R=R-1 }
                else  { a[k] = a[L]; a[L] = aR; a[R] = ak; L=L+1; R=R-1 }
       }
    };
    a[le] = a[L-1]; a[L-1] = piv1; a[ri] = a[R+1]; a[R+1] = piv2;
    {  sort(a, le, L-2); sort(a, R+2, ri); sort(a, L, R)  }
};
```

Заключение

В настоящей работе описывается построение предикатной программы быстрой сортировки с двумя опорными элементами. Дедуктивная верификация предикатной программы в системе PVS проведена одним из авторов в течение недели [16]. Для сравнения, дедуктивная верификация аналогичной программы из библиотеки JDK потребовала два с половиной месяца [1, 4], т.е. на порядок больше. Основная причина этого: предикатная программа намного «ближе» к языку спецификаций, чем Java-программа к языку JML. Спецификация на языке JML, приведенная в материалах [4], в несколько раз длиннее и принципиально сложнее. При этом исходная Java-программа была существенно модифицирована для того, чтобы верификация стала возможной.

Эффективная программа сортировки на языке Java получена применением набора оптимизирующих трансформаций к исходной предикатной программе. Планируется провести ее сравнение с эксплуатируемой в библиотеке JDK.

Дальнейшая задача – разработка инструментов, поддерживающих разработку и верификацию предикатных программ и способствующих снижению затрат на проведение дедуктивной верификации. Архитектура подсистемы верификации предложена в исследованиях по программному синтезу предикатных программ [17]. Необходим собственный специализированный решатель, работающий интерактивно в контексте создаваемой программы и набора теорий. Решатель проводит преобразования и удобную визуализацию генерируемых формул корректности. Преобразования решателя: унификация термов, перебор термов, подстановки, обеспечивающие истинность формул с использованием лемм, и другие.

Доступна полная версия настоящей статьи: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 16-01-00498-а.

Литература

1. Beckert B., Schiffl J., Schmitt P.H., Ulbrich M. Proving JDK’s Dual Pivot Quicksort Correct // VSTTE 2017: Verified Software. Theories, Tools, and Experiments. – 2017. – P. 35-48.
2. de Gouw C.P.T, Rot J.C, de Boer F.S, Bubel R, Haehnle R. OpenJDK’s Java.utils.Collection.sort() is broken: The good, the bad and the worst case // Computer Aided Verification (CAV). LNCS 9206. – 2015. – P. 273-289.
3. Yaroslavskiy V. Dual-pivot quicksort algorithm. 2009. URL: http://codeblab.com/wpcontent/ uploads/2009/09/DualPivotQuicksort.pdf
4. Beckert B. et al. Proving JDK’s dual pivot quicksort correct. Blog post, companion website. URL: www.key-project.org/2017/08/17/dual-pivot/

5. de Gouw S., de Boer F.S., Rot J. Verification of counting sort and radix sort // Deductive Software Verification – The KeY Book, LNCS 10001. – 2016. – P. 609–618.
6. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164).
URL: http://www.iis.nsk.su/files/preprints/164.pdf
7. PVS Specification and Verification System. SRI International. URL: http://pvs.csl.sri.com/
8. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2010. 42с. (Препр. / ИСИ СО РАН; N 153). URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
9. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538.
10. Шелехов В.И. Основы предикатного программирования. — ИСИ СО РАН, Новосибирск, 2016. — 25с.
URL: http://persons.iis.nsk.su/files/persons/pages/predbase.pdf
11. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. — Новосибирск, 2015. — 13с.
URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
12. Чушкин М.С. Система дедуктивной верификации предикатных программ // «Программная инженерия», № 5, 2016. – С. 202-210. URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf
13. Доказательство правил корректности операторов предикатной программы. URL: http://www.iis.nsk.su/persons/vshel/files/rules.zip
14. CVC4 – the SMT solver. URL: http://cvc4.cs.stanford.edu/web/
15. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.
16. Шелехов В.И. Теории и доказательства в системе PVS предикатной программы быстрой сортировки с двумя опорными элементами.
URL: http://persons.iis.nsk.su/files/persons/pages/dqsortpvs.zip
17. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы – 2017», Ростов-на-Дону — 2017 — С.258-262. URL: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf

References

1. Beckert B., Schiffl J., Schmitt P.H., Ulbrich M. Proving JDK’s Dual Pivot Quicksort Correct // VSTTE 2017: Verified Software. Theories, Tools, and Experiments. – 2017. – P. 35-48.
2. de Gouw C.P.T, Rot J.C, de Boer F.S, Bubel R, Haehnle R. OpenJDK’s Java.utils.Collection.sort() is broken: The good, the bad and the worst case // Computer Aided Verification (CAV). LNCS 9206. – 2015. – P. 273-289.

3. Yaroslavskiy V. Dual-pivot quicksort algorithm. 2009.
URL: http://codeblab.com/wpcontent/ uploads/2009/09/DualPivotQuicksort.pdf
4. Beckert B. et al. Proving JDK's dual pivot quicksort correct. Blog post, companion website. URL: www.key-project.org/2017/08/17/dual-pivot/
5. de Gouw S., de Boer F.S., Rot J. Verification of counting sort and radix sort // Deductive Software Verification – The KeY Book, LNCS 10001. – 2016. – P. 609–618.
6. Shelekhov V.I. Razrabotka i verifikatsiia algoritmov piramidalnoi sortirovki v tekhnologii predikatnogo programmirovaniia. – Novosibirsk, 2012. – 30s. – (Prepr. / ISI SO RAN. N 164).
URL: http://www.iis.nsk.su/files/preprints/164.pdf
7. PVS Specification and Verification System. SRI International.
URL: http://pvs.csl.sri.com/
8. Karnaukhov N.S., Pershin D.Iu., Shelekhov V.I. Iazyk predikatnogo programmirovaniia P. Novosibirsk, 2010. 42s. (Prepr. / ISI SO RAN; N 153).
URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
9. Shelekhov V.I. Klassifikatsiia programm, orientirovannaia na tekhnologiiu programmirovaniia // «Programmnaia inzheneriia», Tom 7, № 12, 2016. — S. 531–538.
10. Shelekhov V.I. Osnovy predikatnogo programmirovaniia. — ISI SO RAN, Novosibirsk, 2016. — 25s.
URL: http://persons.iis.nsk.su/files/persons/pages/predbase.pdf
11. Shelekhov V.I. Semantika iazyka predikatnogo programmirovaniia // ZONT-15. — Novosibirsk, 2015. — 13s.
URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
12. Chushkin M.S. Sistema deduktivnoi verifikatsii predikatnykh programm // «Programmnaia inzheneriia», № 5, 2016. – S. 202-210.
URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf
13. Dokazatelstvo pravil korrektnosti operatorov predikatnoi programmy.
URL: http://www.iis.nsk.su/persons/vshel/files/rules.zip
14. CVC4 – the SMT solver. URL: http://cvc4.cs.stanford.edu/web/
15. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.
16. Shelekhov V.I. Teorii i dokazatelstva v sisteme PVS predikatnoi programmy bystroi sortirovki s dvumia opornymi elementami.
URL: http://persons.iis.nsk.su/files/persons/pages/dqsortpvs.zip
17. Shelekhov V.I. Sintez operatorov predikatnoi programmy // Trudy konf. «Iazyki programmirovaniia i kompiliatory – 2017», Rostov-na-Donu — 2017 — S.258-262. URL: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf