УДК 004.415.52

Верификация программы быстрой сортировки с двумя опорными элементами

В.И. Шелехов, М.С. Чушкин

Институт Систем Информатики СО РАН, г. Новосибирск,
e-mail: vshel@iis.nsk.su

Быстрая сортировка с двумя опорными элементами является одной из наиболее эффективных в базисном наборе алгоритмов сортировки библиотеки JDK для языка Java. В настоящей работе описывается построение и дедуктивная верификация этой программы в технологии предикатного программирования. Дедуктивная верификация проведена за одну неделю, на порядок быстрее в сравнении с описанной в работе [1].

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

Введение

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]. Используемые методы дедуктивной верификации изложены 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]. Приведены результаты сравнения Java-программы нашей версии и эксплуатируемой в библиотеке JDK.

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) и *оператор каррирования*. В таблице 1 представлен полный базис вычислимых предикатов и соответствующих им операторов.

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

Таблица 1. Вычислимые предикаты и их программная форма

Набор x˜ составлен из набора переменных x с возможным добавлением имен предикатных программ.

Имеется два вида оператора каррирования: D(y**:** z) {B(x, y**:** z)} и D(y**:** z) {A(x, y**:** z)}. Результатом исполнения оператора каррирования является 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

Операционную семантику программы H(x**:** y) определим в виде предиката:

$\mathcal{R}(H)(x, y)$ ≅ для значения набора x исполнение программы H всегда завершается и существует исполнение программы, при котором результатом вычисления является значение набора y.

Для языка $\mathbf{P_0}$ построена формальная операционная семантика и доказано тождество $\mathcal{R}(H) = H$ [11]. На базе языка $\mathbf{P_0}$ последовательным расширением и сохранением тождества $\mathcal{R}(H) = H$ построен язык предикатного программирования **P** [8].

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:

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

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

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

Для 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].

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

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

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

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

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

Запись вида z = B(x) является эквивалентом B(x: z). Истинность двух посылок правила **RBE** 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 $P{*}_C(x)$ 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.

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 ти 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:

a[x]=piv1, a[y]=piv2 для некоторых x и y, где x<y; (2)

a[j]≤piv1 для j<x;

piv1≤a[j]≤piv2 для x<j<y;

piv2≤a[j] для j>y.

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, запускается другой более простой алгоритм 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 запускается другой алгоритм 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 фиксируются в переменных piv1 и piv2. 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 программы 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.

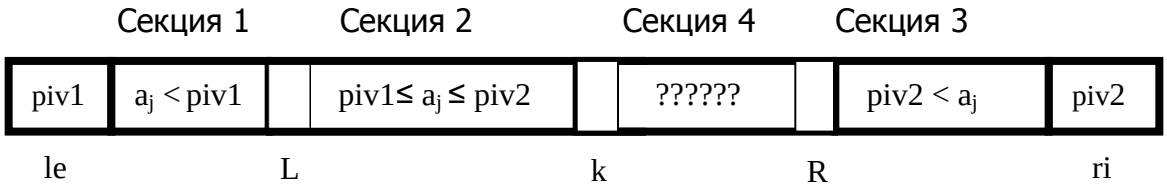

Рис.1

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, T piv1, piv2: 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

4.1. Генерация формул корректности

Полная программа sort начинается вызовом гиперфункции sortG, представленной спецификацией:

```
sortG(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);
```

Генерация формул корректности для вызова гиперфункции реализуется аналогично условному оператору. Доказательство для первой ветви тривиально. Нас интересует далее только вторая ветвь.

```
sort(le, ri)(Arl a:  Arl a') post perm(a, a') & sorted(a') {
    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])
    }
}
```

Для полной программы sort, приведенной выше, применяется правило **QSB**:

$$\textbf{QSB}:\ \frac{B(x: z)\ \textbf{corr}^*\ [P_B(x), Q_B(x, z)];\ P(x) \Rightarrow P^*_B(x);\quad \forall z.\ C(x, z: y)\ \textbf{corr}\ [P(x)\ \&\ Q_B(x, z), Q(x, y)];}{Corr(B(x: z); C(x, z: y)\ \textbf{corr}\ [P(x), Q(x,y]}$$

Здесь в качестве B(x: z) – вторая ветвь вызова гиперфункции sortG, а C(x, z: y) соответствует последовательности последующих операторов программы. При этом истинность второй посылки правила доказывать не надо. Третья посылка определяет цель:

```
formula QsortG(Arl a, Arl a', T piv1, piv2) =
            perm(a, a'with(le: piv1, ri: piv2)) & Pivots(a', piv1, piv2);
{...} corr [ri - le > 46 & QsortG(a, a1, piv1, piv2), perm(a, a') & sorted(a')];
```

Оператор {...} обозначает оператор суперпозиции вызова partition с остальной частью программы.

Далее снова применяется правило **QSB**, где в качестве B(x: z) – вызов программы partition. Получаем следующие три цели.

```
partition(le, ri)(a1, piv1, piv2: Arl a2, LR L, R) corr
                                        [Pivots(a1, piv1, piv2), Qpart(a1, a2, L, R)];
ri - le > 46 & QsortG(a, a1, piv1, piv2) ⇒ Pivots(a1, piv1, piv2);
{...} corr [ri - le > 46 & QsortG(a, a1, piv1, piv2) & Qpart(a1, a2, L, R),
                perm(a, a') & sorted(a')];
```

Оператор {...} соответствует блоку с тремя рекурсивными вызовами программы sort.

Первая цель – корректность программы partition. Будет доказана позже. Истинность второй цели очевидна. Оператор {...} третьей цели сначала преобразуется на язык $\mathbf{P}_2$.

```
{Arl(le, L-2) b = a2[le..L-2] || Arl(R+2, ri) c = a2[R+2..ri] || Arl(L, R) d = a2[L..R]};
{   sort(le, L-2)(b:   Arl(le, L-2) b') ||
    sort(R+2, ri)(c:   Arl(R+2, ri) c') ||
    sort(L, R)(d:   Arl(L, R) d')
};
a' = a2 with (le..L-2: b', L..R: d', R+2..ri: c');
```

Далее для данного фрагмента применяется правило **QS**.

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

Вторая посылка правила дает новую цель:

formula Qbcd(Arl a2, Arl(le, L-2) b, Arl(R+2, ri) c, Arl(L, R) d) =
b = a2[le..L-2] & c = a2[R+2..ri] & d = a2[L..R];
formula Pfr(Arl a, a1, a2, T piv1, piv2, LR L, R, Arl(le, L-2) b, Arl(R+2, ri) c, Arl(L, R) d) =
ri - le > 46 & QsortG(a, a1, piv1, piv2) & Qpart(a1, a2, L, R) & Qbcd(a2, b, c, d);
{ Z }**corr**[Pfr(a, a1, a2, piv1, piv2, L, R, b, c, d), perm(a, a') & sorted(a')];
Здесь { Z } – обозначает оператор, образованный суперпозицией второго и третьего операторов представленного выше фрагмента. К оставшейся части применим правило **QSB**.

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

Здесь B(x**:** z) соответствует второму оператору, а C(x, z**:** y) – третьему. Определим постусловие второго оператора:
formula Q3sort(Arl(le, L-2) b, b', Arl(R+2, ri) c, c', Arl(L, R) d, d') =
perm(b, b') & sorted(b') & perm(c, c') & sorted(c') & perm(d, d') & sorted(d');
Посылки правила **QSB** дают следующие три цели:
formula Pfin(Arl a, a1, a2, T piv1, piv2, LR L, R,
Arl(le, L-2) b, b', Arl(R+2, ri) c, c', Arl(L, R) d, d') =
Pfr(a, a1, a2, piv1, piv2, L, R, b, c, d) & Q3sort(b, b', c, c', d, d');
{ Z2 } **corr** [le < L & R < ri, Q3sort(b, b', c, c', d, d')];
Pfr(a, a1, a2, piv1, piv2, L, R, b, c, d) ⇒ le < L & R < ri;
{ Z3 } **corr** [Pfin(a, a1, a2, piv1, piv2, L, R, b, b', c, c', d, d'), perm(a, a') & sorted(a')];

Для первой цели дважды применяется правило **RP** для параллельного оператора.

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

Подоператорами параллельного оператора являются рекурсивные вызовы программы sort, для которых срабатывает индукционное предположение. Последние две посылки правила **RP** дают следующие цели:
perm(b, b') & sorted(b') & perm(c, c') & sorted(c') & perm(d, d') & sorted(d')
⇒ Q3sort(b, b', c, c', d, d');
formula m(LR le, ri**:** **nat**) = (ri<le)? 0 **:** ri – le + 1;
le < L & R < ri ⇒ m(le, L-2)< m(le, ri) & m(R+2, ri)< m(le, ri) & m(L, R) < m(le, ri);
Для третьей цели, доказательства корректности Z3, используется правило **COR**, т.е. формула корректности (1):
Pfin(a, a1, a2, piv1, piv2, L, R, b, b', c, c', d, d') &
a' = a2 **with** (le..L-2: b', L..R: d', R+2..ri: c')
⇒ perm(a, a') & sorted(a');
Доказательство данной формулы использует следующие две леммы:

pe3**:** **lemma** Qbcd(a, b, c, d) & perm(b, b') & perm(c, c') & perm(d, d') &
a' = a **with** (le..L-2**:** b', L..R**:** d', R+2..ri**:** c') ⇒ perm(a, a');
so1**:** **lemma** Qbcd(a, b, c, d) & sorted(b') & sorted(c') & sorted(d') &
a' = a **with** (le..L-2**:** b', L..R**:** d', R+2..ri**:** c') ⇒ sorted(a');

Доказательство лемм проводится в системе PVS.

Далее рассмотрим доказательство программы partition. Для тела программы применим правило **RS**.

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

В качестве B и C здесь используются программы scanL и split. Корректность этих программ, определяемая первыми двумя посылками, рассмотрим позже. Представим формулы для следующих трех посылок.

formula Psplit(Arl a, LR L0, R0, k) =
Pivots(a, piv1, piv2) & partG(a, L0, R0) & (∀**int** j. L0<=j<k ⇒ piv1<=a[j]<=piv2);
Pivots(a, piv1, piv2) & QscanL(a, le+1, L0) & Qpart(a, a', R, L) ⇒ Qpart(a, a', R, L);
Pivots(a, piv1, piv2) & QscanL(a, le+1, L0) ⇒ Psplit(Arl a, L0, ri-1, L0);
Pivots(a, piv1, piv2) ⇒ PscanL(a, le+1);

Рассмотрим доказательство корректности программы scanL. Применим правило **QC**. Получим цели.

scanL(a, L+1**:** L0) **corr** [PscanL(a, L) & a[L]<piv1, QscanL(Arl a, LR L, L0)];
L0 = L **corr** [PscanL(a, L) & **not** a[L]<piv1, QscanL(Arl a, LR L, L0)];

Для второй цели используем определение формулы корректности **COR** (1).

PscanL(a, L) & **not** a[L]<piv1 & L0 = L ⇒ QscanL(Arl a, LR L, L0);

Для первой цели используем правило **RBE**.

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

Первая посылка опускается для рекурсивного вызова программы scanL. Вторая посылка дает формулу:

formula mL(**int** L**:** **nat**) = ri - L;
PscanL(a, L) & a[L]<piv1 ⇒ PscanL(a, L+1) & mL(L+1) < mL(L);

Рассмотрим доказательство корректности программы split. Применим правило **QC**. Получим цели.

Fin(L0, R0, a**:** a', L, R) **corr** [Psplit(a, L0, R0, k) & k > R0, Qpart(a, a', R, L)];
{...} **corr** [Psplit(a, L0, R0, k) & **not** k > R0, Qpart(a, a', R, L)];

К первой цели применим правило **COR** , т.е. определение (1).

Psplit(a, L0, R0, k) & k > R0 & L = L0 & R = R0 &
a' = a **with**(le**:** a[L0-1], ri**:** a[R0+1]) **with** (L0-1**:** piv1, R0+1**:** piv2) ⇒
Qpart(a, a', R, L)

Ко второй цели применим правило **QS**.

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

Вторая посылка правила дает новую цель:

{…} **corr** [Psplit(a, L0, R0, k) & **not** k > R0 & ak = a[k], Qpart(a, a', R, L)];

К оставшейся части программы применим правило **QC**. Получим цели.

split(a **with**(L0**:** ak, k**:** a[L0]), L0+1, R0, k+1**:** a', L, R) **corr**
[Psplit(a, L0, R0, k) & **not** k > R0 & ak = a[k] & ak<piv1, Qpart(a, a', R, L)];
{…} **corr** [Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1, Qpart(a, a', R, L)];

Для первой цели используем правило **RBE**. Первая посылка опускается для рекурсивного вызова программы split. Вторая посылка дает формулу:

formula ms(LR L0, R0, k**: nat**) = R0 - L0 + ri - k;
Psplit(a, L0, R0, k) & k <= R0 & ak = a[k] & ak<piv1 ⇒
Psplit(a **with**(L0**:** ak, k**:** a[L0]), L0+1, R0, k+1) &
ms(L0+1, R0, k+1) < ms(L0, R0, k);

Для оставшейся части программы во второй цели применим правило **QC**. Получим цели.

{…} **corr** [Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2, Qpart(a, a', R, L)];
split(a, L0, R0, k+1**:** a', L, R) **corr**
[Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak<=piv2,
Qpart(a, a', R, L)];

Для второй цели используем правило **RBE**. Первая посылка опускается для рекурсивного вызова программы split. Вторая посылка дает формулу:

Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak<=piv2 ⇒
Psplit(a, L0, R0, k+1) & ms(L0, R0, k+1) < ms(L0, R0, k);

Для оставшегося фрагмента первой цели применим правило **QSB**.

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

Здесь B(x**:** z) соответствует вызову scanR(a, R0 **:** LR R1), а C(x, z**:** y) – оставшейся части фрагмента. Получаем цели.

scanR(a, R0 **:** R1) **corr** [PscanR(a, R0), QscanR(a, R0, R1)];
Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 ⇒ PscanR(a, R0);
{…} **corr** [Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 &
QscanR(a, R0, R1), Qpart(a, a', R, L)];

Для третьей цели применяем правило **QC**. Получим цели.

Fin(L0, R1, a**:** a', L, R) **corr** [Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 &
ak>piv2 & QscanR(a, R0, R1) & k > R1, Qpart(a, a', R, L)];
{…} **corr** [Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 &
QscanR(a, R0, R1) & k <= R1, Qpart(a, a', R, L)];

К первой цели применим правило **COR** , т.е. определение (1).

Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 & QscanR(a, R0, R1) &
k > R1 & L = L0 & R = R1 &
a′ = a **with**(le**:** a[L0-1], ri**:** a[R1+1]) **with** (L0-1**:** piv1, R1+1**:** piv2) ⇒ Qpart(a, a′, R, L);

Для оставшегося фрагмента второй цели применим правило **QS**.

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

Вторая посылка правила дает новую цель:

{…} **corr** [Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 &
QscanR(a, R0, R1) & k <= R1 & aR = a[R1], Qpart(a, a′, R, L)];

Применим правило **QC**. Получим цели.

split(a **with**(k**:** aR, R1**:** ak), L0, R1-1, k+1**:** a′, L, R) **corr** [Psplit(a, L0, R0, k) & k<=R0 &
ak = a[k] & ak >= piv1 & ak>piv2 & QscanR(a, R0, R1) & k <= R1 &
aR = a[R1] & aR>=piv1, Qpart(a, a′, R, L)];

split(a **with**(L0**:** aR, k**:** a[L0], R1**:** ak), L0+1, R1-1, k+1**:** a′, L, R) **corr**
[Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 &
QscanR(a, R0, R1) & k <= R1 & aR = a[R1] & aR<piv1, Qpart(a, a′, R, L)];

Для каждой цели используем правило **RBE**.

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

Получим итоговые формулы корректности.

Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 & QscanR(a, R0, R1) &
k <= R1 & aR = a[R1] & aR>=piv1 ⇒
Psplit(a **with**(k**:** aR, R1**:** ak), L0, R1-1, k+1) & ms(L0, R1-1, k+1) < ms(L0, R0, k);

Psplit(a, L0, R0, k) & k<=R0 & ak = a[k] & ak >= piv1 & ak>piv2 & QscanR(a, R0, R1) &
k <= R1 & aR = a[R1] & aR<piv1 ⇒
Psplit(a **with**(L0**:** aR, k**:** a[L0], R1**:** ak), L0+1, R1-1, k+1) &
ms(L0+1, R1-1, k+1) < ms(L0, R0, k);

Рассмотрим доказательство корректности программы scanR. Применим правило **QC**. Получим цели.

scanR(a, R0 − 1**:** R) **corr** [PscanR(a, R0) & a[R0]>piv2, QscanR(a, R0, R)];

R = R0 **corr** [PscanR(a, R0) & a[R0]<=piv2, QscanR(a, R0, R)];

Ко второй цели применим правило **COR**, т.е. определение (1).

PscanR(a, R0) & a[R0]<=piv2 & R = R0 ⇒ QscanR(a, R0, R);

Для первой цели применяется правило **RBE**. Получим:

formula mR(LR R0**: nat**) = R0;

PscanR(a, R0) & a[R0]>piv2 ⇒ PscanR(a, R0 − 1) & mR(R0 − 1) < mR(R0)

4.2. Доказательство формул корректности в системе PVS

Формулы корректности, представленные в разд. 4.2, 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.

Формулы корректности с обилием параметрических типов были диагностированы как ошибочные в системе PVS. Это ограничение удалось преодолеть введением двух групп параметров в стиле каррирования.

При доказательстве в системе PVS потребовались дополнительные леммы. Их всего три.

```
pe2: LEMMA   perm(a, b) & perm(b, c) IMPLIES perm(a, c)
pe9: LEMMA   FORALL (b, b9: Arl[le, L-2], c, c9: Arl[R+2, ri], d, d9: Arl[L, R]):
   Qbcd(L, R)(a2, b, c, d) & perm[le, L-2](b, b9) & perm[R+2, ri](c, c9) & perm[L, R](d, d9) &
         (FORALL (j: LR): le <= j AND j < L - 1 IMPLIES a2(j) < piv1) &
         (FORALL (j: LR): 1 + R < j AND j <= ri IMPLIES piv2 < a2(j)) &
         (FORALL (j: LR): L <= j AND j <= R IMPLIES piv1 <= a2(j) AND a2(j) <= piv2)
      IMPLIES    (FORALL (j: LR): le <= j AND j < L - 1 IMPLIES b9(j) < piv1) &
            (FORALL (j: LR): 1 + R < j AND j <= ri IMPLIES piv2 < c9(j)) &
            (FORALL (j: LR): L <= j AND j <= R IMPLIES piv1 <= d9(j) AND d9(j) <= piv2)
sl1: LEMMA   L-1 < 1+R IMPLIES perm(a WITH [le := piv1, ri := piv2],
                                    a WITH [le := a(L - 1), (L - 1) := piv1, ri := a(1 + R), (1 + R) := piv2])
```

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 применяется склеивание переменных. Далее для каждой программы в составе полной программы сортировки указывается набор команд склеивания и результат склеивания. Команда склеивания задает замену набора переменных справа, на переменную, указанную слева.

Склеивание: a ← a′, a1, a2;

```
sort(le, ri)(Arl a:  Arl a) {
    sortG(le, ri)(a:  a #return :   Arl a, T piv1, piv2);
    partition(le, ri)(a, piv1, piv2:   Arl a, LR L, R);
    {   sort(le, L-2)(a[le..L-2]:   Arl(le, L-2) a[le..L-2]) ||
         sort(R+2, ri)(a[R+2..ri]:   Arl(R+2, ri) a[R+2..ri]) ||
         sort(L, R)(a[L..R]:   Arl(L, R) a[L..R])
    }
}
```

Склеивание: a ← a'; L ← L0;

```
partition(Arl a: Arl a, LR L, R) {
    scanL(Arl a, le+1:  LR L);
    split(a, L, ri-1, L:  a, L, R)
};
```

Склеивание: L ← L0;

```
scanL(Arl a, LR L:  LR L) {
    if (a[L]<piv1) scanL(a, L+1: L)
    else L = L
}
```

Склеивание: R ← R0;

```
scanR(Arl a, LR R:  LR R) {
    if (a[R]>piv2) scanR(a, R – 1: R)
    else R = R
}
```

Склеивание: a ← a'; L ← L0; R ← R0;

```
Fin(LR L, R, Arl a: Arl a, LR L, R)
{      L = L || R = R ||
       a = a with(le: a[L-1], ri: a[R+1]) with (L-1: piv1, R+1: piv2)
}
```

Склеивание: a ← a'; L ← L0; R ← R0, R1;

```
split(Arl a, LR L, R, k:   Arl a, LR L, R) {
    if (k > R) Fin(L, R, a: a, L, R)
    else {
       T ak = a[k];
       if (ak<piv1) split(a with(k: a[L], L: ak), L+1, R, k+1:  a, L, R)
       else if (ak>piv2) {
                scanR(a, R : LR R);
                if (k > R) Fin(L, R, a: a, L, R)
                else {
                    T aR = a[R];
                    if (aR>=piv1) split(a with(k: aR, R: ak), L, R-1, k+1:  a, L, R)
                    else split(a with(k: a[L], L: aR, R: ak), L+1, R-1, k+1:  a, L, R)
                }
       } else split(a, L, R, k+1:  a, L, R)
    }
};
```

На втором этапе проводится замена хвостовой рекурсии циклом **for**. Также реализуются упрощения.

```
scanL(Arl a, LR L:  LR L) {
    for(;a[L]<piv1; L = L+1)
}
scanR(Arl a, LR R:  LR R) {
    for(;a[R]>piv2; R = R – 1)
}
Fin(LR L, R, Arl a: Arl a, LR L, R)
{      a = a with(le: a[L-1], L-1: piv1, ri: a[R+1], R+1: piv2) }

split(Arl a, LR L, R, k:   Arl a, LR L, R) {
 for( ; ; k = k+1) {
    if (k > R) { Fin(L, R, a: a, L, R); break}
    else {
       T ak = a[k];
       if (ak<piv1) (a = a with(k: a[L], L: ak); L = L+1;
       else if (ak>piv2) {
                scanR(a, R : LR R);
                if (k > R) { Fin(L, R, a: a, L, R); break}
                else {
                    T aR = a[R];
                    if (aR>=piv1) a = a with(k: aR, R: ak); R=R-1
                    else a = a with(k: a[L], L: aR, R: ak); L=L+1; R=R-1
                }
       } else {}
    }
 }
};
```

На следующем этапе раскрываются модификаторы **with** с заменой на присваивания элементам массива. Реализуются упрощения.

```
Fin(LR L, R, Arl a: Arl a, LR L, R)
{  a[le]= a[L-1]; a[L-1] = piv1; a[ri] = a[R+1]; a[R+1] = piv2 }

split(Arl a, LR L, R, k:   Arl a, LR L, R) {
  for( ; ; k = k+1) {
     if (k > R) { Fin(L, R, a: a, L, R); break}
     else {
        T ak = a[k];
        if (ak<piv1) { a[k] = a[L]; a[L] = ak; L = L+1 }
        else if (ak>piv2) {
                  scanR(a, R : LR R);
                  if (k > R) { Fin(L, R, a: a, L, R); break}
                  else {
                      T 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}
                  }
        } else {}
     }
  }
};
```

Далее вызов Fin выносится за цикл. Проводится открытая подстановка тел программ на место их вызовов.

```
partition(Arl a: Arl a, LR L, R) {
     for(L = le+1; a[L]<piv1; L = L+1);
     R = ri-1;
     // split(a, L, ri-1, L:  a, L, R);
     for(LR k = L ; ; k = k+1) {
        if (k > R) break;
        T 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;
                  T 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;
};
```

Реализуется подстановка программы partition на место вызова в программу sort. Сортируемый массив представляется вырезкой глобального массива. Получаем итоговую программу на императивном расширении языка **P**.

```
nat n;
type Arn = array (T, 0..n);
Arn a;
sort(int le, ri) {
    sortG(le, ri)( :  #return :   T piv1, piv2);
    LR L; LR R = ri-1;
    for(L = le+1; a[L]<piv1; L = L+1);
    for(LR k = L ; ; k = k+1) {
       if (k > R) break;
       T 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;
                T 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(le, L-2) || sort(R+2, ri) || sort(L, R)  }
};
```

Наконец, программа конвертируется 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; int R = ri-1;
    for(L = le+1; a[L]<piv1; L = L+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 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.

Если снабжать формулы корректности необходимыми леммами, то число формул, которые могут быть доказанными SMT-решателями, возрастает с 20% до 90%. Это и другие свойства специализированного решателя способны примерно вдвое ускорить процесс дедуктивной верификации предикатных программ.

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

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. http://codeblab.com/wpcontent/ uploads/2009/09/DualPivotQuicksort.pdf.
4. Proving JDK's dual pivot quicksort correct. Blog post, companion website. https:// 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). http://www.iis.nsk.su/files/preprints/164.pdf
7. PVS Specification and Verification System. SRI International. 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). 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. http://persons.iis.nsk.su/files/persons/pages/predbase.pdf
11. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. — Новосибирск, 2015. — 13с. 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. http://persons.iis.nsk.su/files/persons/pages/paper.pdf
13. Доказательство правил корректности операторов предикатной программы. 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. Теории и доказательства в системе PVS предикатной программы быстрой сортировки с двумя опорными элементами. 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. http://persons.iis.nsk.su/files/persons/pages/sintr.pdf

Приложение

Теории для программы сортировки

```
QuickDecl[le, ri: int] : THEORY
BEGIN
  T: TYPE = int
  LR: TYPE = subrange(le, ri)
  Arl: TYPE = [LR -> T]
END QuickDecl

Perm[le, ri: int] : THEORY
BEGIN
  IMPORTING QuickDecl[le, ri]
  a, b, c: VAR Arl
  i, j: VAR LR
  perm(a, b): bool = EXISTS(f: [LR -> LR]): bijective?(f) & FORALL j: b(j) = a(f(j))
  pe2: LEMMA perm(a, b) & perm(b, c) IMPLIES perm(a, c)
END Perm

Sort[le, ri: int] : THEORY
BEGIN
  IMPORTING QuickDecl[le, ri]
  a: VAR Arl
  i, j: VAR LR
  sorted(a): bool =  FORALL i, j: i < j IMPLIES a(i) <= a(j)
END Sort

SortGe[le: int, ri: {x:int| x>=le}] : THEORY
BEGIN
  IMPORTING Sort[le, ri], Perm[le, ri]
  a, a1, a2, a9: VAR Arl
  j, i: VAR int
  piv1, piv2: VAR T
  L, L0, R, R0, k: VAR LR
  Pivots(a, piv1, piv2): bool = piv1 < piv2 &
        (EXISTS j: le<j & j<ri & a(j)<piv1) & (EXISTS j: le<j & j<ri & a(j)>piv2)
  Qpart(a, a9, L, R, piv1, piv2): bool =
     perm(a WITH [le:= piv1, ri:= piv2], a9) &
     L > le & R < ri & L <= R+1 &
     a9(L-1)=piv1 & a9(R+1)=piv2 &
    (FORALL j: le<=j & j<L-1 IMPLIES a9(j)<piv1) &
    (FORALL j: R+1<j & j<=ri IMPLIES piv2<a9(j)) &
    (FORALL j: L<=j & j<=R IMPLIES piv1<=a9(j) & a9(j)<=piv2);
  partG(a, L0, R0, k, piv1, piv2):bool =
```

```
      L0 > le & R0 < ri & L0 <= R0+1 & L0 <= k &
     (FORALL j: le<j & j<L0 IMPLIES a(j)<piv1) &
     (FORALL j: R0<j & j<ri IMPLIES piv2<a(j));
  Psplit(a, L0, R0, k, piv1, piv2): bool =
    Pivots(a, piv1, piv2) & partG(a, L0, R0, k, piv1, piv2) &
      (FORALL j: L0<=j & j<k IMPLIES piv1<=a(j) & a(j)<=piv2);
  PscanL(a, L, piv1): bool =
      L > le & (FORALL j: le<j & j<L IMPLIES a(j)<piv1) & (EXISTS i:  L<=i & i<ri & a(i)>=piv1);
  QscanL(a, L, L0, piv1): bool =
       L0 > le & L<=L0 & piv1<=a(L0) & (FORALL j: le<j & j<L0 IMPLIES a(j)<piv1)
  PscanR(a, R0, piv2): bool =
      R0 < ri & (FORALL j: R0<j & j<ri IMPLIES a(j)>piv2) & (EXISTS i: le<i & i<=R0
a(i)<=piv2);
  QscanR(a, R0, R, piv2): bool =
      R < ri & R<=R0 & a(R)<=piv2 & (FORALL j: R<j & j<=R0 IMPLIES a(j)>piv2);
  QsortG(a, a9, piv1, piv2): bool =
      perm(a, a9 WITH [le:=piv1, ri:=piv2]) & Pivots(a9, piv1, piv2);

  So1: LEMMA ri - le > 46 & QsortG(a, a1, piv1, piv2) IMPLIES Pivots(a1, piv1, piv2);

END SortGe

Slice[le: int, ri: {x:int| x>=le}, x, y: subrange(le, ri)]: THEORY
BEGIN
  IMPORTING SortGe[le, ri]
  ArlS: TYPE = [subrange(x, y) -> T]
  z: VAR subrange(x, y)
  a: VAR Arl
  slice(a): ArlS = (LAMBDA z: a(z))
END Slice

SortMi[le: int, ri: {x:int| x>=le}] : THEORY %, L, R: subrange(le, ri)] : THEORY
BEGIN
  IMPORTING SortGe[le, ri], Slice
  L, R: VAR subrange(le+2,ri-2)
  R0, k: VAR LR
  x, y: VAR LR
  piv1, piv2: VAR T
  a, a1, a2, a9: VAR Arl
  ArlSu(x, y): TYPE = [subrange(x, y) -> T]

  Qbcd(L, R)(a2: Arl, b: ArlSu(le, L-2), c: ArlSu(R+2, ri), d: ArlSu(L, R)): bool =
    b = slice[le,ri, le, L-2](a2) & c = slice[le,ri, R+2, ri](a2) & d = slice[le,ri, L, R](a2);
  Pfr(L, R)(a, a1, a2: Arl, piv1, piv2: T, b: ArlSu(le, L-2), c: ArlSu(R+2, ri), d: ArlSu(L, R)): bool =
```

```
      ri - le > 46 & QsortG(a, a1, piv1, piv2) & Qpart(a1, a2, L, R, piv1, piv2) & Qbcd(L, R)(a2, b, c,

   Q3sort(L, R)(b, b9: ArlSu(le, L-2), c, c9: ArlSu(R+2, ri), d, d9: ArlSu(L, R)): bool =
      perm[le, L-2](b, b9) & sorted[le, L-2](b9) &
      perm[R+2, ri](c, c9) & sorted[R+2, ri](c9) &
      perm[L, R](d, d9) & sorted[L, R](d9);
   Pfin(L, R)(a, a1, a2: Arl, piv1, piv2: T, b, b9: ArlSu(le, L-2), c, c9: ArlSu(R+2, ri), d, d9:
ArlSu(L, R)): bool =
      Pfr(L, R)(a, a1, a2, piv1, piv2, b, c, d) & Q3sort(L,R)(b, b9, c, c9, d, d9);

   So2: LEMMA FORALL (b: Arl[le, L-2], c: Arl[R+2, ri], d: Arl[L, R]):
                Pfr(L, R)(a, a1, a2, piv1, piv2, b, c, d) IMPLIES le < L & R < ri;

   So3: LEMMA FORALL (b, b9: Arl[le, L-2], c, c9: Arl[R+2, ri], d, d9: Arl[L, R]):
                perm[le, L-2](b, b9) & sorted[le, L-2](b9) &
                perm[R+2, ri](c, c9) & sorted[R+2, ri](c9) &
                perm(d, d9) & sorted(d9)        IMPLIES  Q3sort(L, R)(b, b9, c, c9, d, d9);
   m(le, ri: LR): nat = IF ri<le THEN 0 ELSE ri - le + 1 ENDIF
   So4: LEMMA le<L & R<ri IMPLIES
                   m(le, L-2) < m(le, ri) & m(R+2, ri) < m(le, ri) & m(L, R) < m(le, ri);
   upd(L, R)(a2: Arl, b9: ArlSu(le, L-2), d9: ArlSu(L, R), c9: ArlSu(R+2, ri)): Arl =
          LAMBDA (j: LR): IF le<=j & j<= L-2 THEN b9(j)
                          ELSIF j = L-1 OR j = R+1 THEN a2(j)
                          ELSIF L <=j & j<= R THEN d9(j)
                          ELSE c9(j)  ENDIF

   pe9: LEMMA FORALL (b, b9: Arl[le, L-2], c, c9: Arl[R+2, ri], d, d9: Arl[L, R]):
         Qbcd(L, R)(a2, b, c, d) & perm[le, L-2](b, b9) & perm[R+2, ri](c, c9) & perm[L, R](d, d9)
            (FORALL (j: LR): le <= j AND j < L - 1 IMPLIES a2(j) < piv1) &
            (FORALL (j: LR): 1 + R < j AND j <= ri IMPLIES piv2 < a2(j)) &
            (FORALL (j: LR): L <= j AND j <= R IMPLIES piv1 <= a2(j) AND a2(j) <= piv2)
         IMPLIES     (FORALL (j: LR): le <= j AND j < L - 1 IMPLIES b9(j) < piv1) &
                     (FORALL (j: LR): 1 + R < j AND j <= ri IMPLIES piv2 < c9(j)) &
                     (FORALL (j: LR): L <= j AND j <= R IMPLIES piv1 <= d9(j) AND d9(j) <= piv2)

   So5: LEMMA FORALL (b, b9: Arl[le, L-2], c, c9: Arl[R+2, ri], d, d9: Arl[L, R]):
        Pfin(L, R)(a, a1, a2, piv1, piv2, b, b9, c, c9, d, d9) &  a9 = upd(L, R)(a2, b9, d9, c9)
           IMPLIES perm(a, a9) & sorted(a9);
END SortMi

Partition[le: int, ri: {x:int| x>=le}] : THEORY
BEGIN
   IMPORTING SortGe[le, ri]
   a, a9: VAR Arl
```

```
  piv1, piv2: VAR T
  R, L, L0: VAR LR

  Pa1: LEMMA Pivots(a, piv1, piv2) & QscanL(a, le+1, L0, piv1) & Qpart(a, a9, L, R, piv1, piv2)
                                                    IMPLIES   Qpart(a, a9, L, R, piv1, piv2);
  Pa2: LEMMA Pivots(a, piv1, piv2) & QscanL(a, le+1, L0, piv1)  IMPLIES  Psplit(a, L0, ri-1,
L0, piv1, piv2);
  Pa3: LEMMA Pivots(a, piv1, piv2)  IMPLIES  PscanL(a, le+1, piv1);

  Pa4: LEMMA PscanL(a, L,piv1) & NOT a(L)<piv1 & L0 = L  IMPLIES  QscanL(a, L, L0, piv1);
END Partition

ScanL[le: int, ri: {x:int| x>=le}] : THEORY
BEGIN
  IMPORTING SortGe[le, ri]
  mL(L: int): nat = IF ri - L < 0 THEN 0 ELSE ri - L ENDIF
  a: VAR Arl
  L: VAR LR
  piv1: VAR T
  Sca1: LEMMA PscanL(a, L, piv1) & a(L)<piv1 IMPLIES
                              PscanL(a, L+1, piv1) & mL(L+1) < mL(L);
END ScanL

SplitLems[le: int, ri: {x:int| x>=le}] : THEORY
BEGIN
  IMPORTING SortGe[le, ri]

  a, b, a8, a9: VAR Arl
  piv1, piv2: VAR T
  R, R0, R1, L, L0, k: VAR subrange(le+1, ri-1)

  sl1: LEMMA L-1 < 1+R IMPLIES perm(a WITH [le := piv1, ri := piv2],
                      a WITH [le := a(L - 1), (L - 1) := piv1, ri := a(1 + R), (1 + R) := piv2])

END SplitLems

Split[le: int, ri: {x:int| x>=le}] : THEORY
BEGIN
  IMPORTING SplitLems[le, ri]

  a, a9: VAR Arl
  piv1, piv2: VAR T
  R, R0, R1, L, L0, k: VAR subrange(le+1, ri-1)
  Sp1: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k > R0 & L = L0 & R = R0 &
```

```
          a9 = a WITH [le:=a(L0-1),(L0-1) := piv1, ri:=a(R0+1), (R0+1) := piv2]
                                                IMPLIES  Qpart(a, a9, L, R, piv1, piv2)
  ak, aR: VAR T
  ms(L0, R0, k: int): nat = IF R0 - L0 + ri - k < 0 THEN 0 ELSE  R0 - L0 + ri - k ENDIF
  Sp2: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k <= R0 & ak = a(k) & ak<piv1 IMPLIES
      Psplit(a WITH[k := a(L0), L0 := ak], L0+1, R0, k+1, piv1, piv2) & ms(L0+1, R0, k+1) <
ms(L0, R0, k);
  Sp3: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k<=R0 & ak = a(k) & ak >= piv1 & ak<=piv2
         IMPLIES  Psplit(a, L0, R0, k+1, piv1, piv2) & ms(L0, R0, k+1) < ms(L0, R0, k);
  Sp4: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k<=R0 & ak = a(k) & ak >= piv1 & ak>piv2
                  IMPLIES PscanR(a, R0, piv2);
  Sp5: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k<=R0 & ak = a(k) & ak >= piv1 & ak>piv2 &
       QscanR(a, R0, R1, piv2) & k > R1 & L = L0 & R = R1 &
            a9 = a WITH [le:=a(L0-1),(L0-1) := piv1, ri:=a(R1+1), (R1+1) := piv2]
                 IMPLIES Qpart(a, a9, L, R, piv1, piv2);
  Sp6: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k<=R0 & ak = a(k) & ak >= piv1 & ak>piv2 &
        QscanR(a, R0, R1, piv2) & k <= R1 & aR = a(R1) & aR>=piv1 IMPLIES
        Psplit(a WITH[k:= aR, R1:= ak], L0, R1-1, k+1, piv1, piv2) & ms(L0, R1-1, k+1) <
ms(L0, R0, k);
  Sp7: LEMMA Psplit(a, L0, R0, k, piv1, piv2) & k<=R0 & ak = a(k) & ak >= piv1 & ak>piv2 &
        QscanR(a, R0, R1, piv2) & k <= R1 & aR = a(R1) & aR<piv1 IMPLIES
            Psplit(a WITH[k:= a(L0), L0:= aR, R1:= ak], L0+1, R1-1, k+1, piv1, piv2) &
            ms(L0+1, R1-1, k+1) < ms(L0, R0, k)
END Split

ScanR[le: int, ri: {x:int| x>=le}] : THEORY
BEGIN
  IMPORTING SortGe[le, ri]
  a: VAR Arl
  R, R0: VAR LR
  piv2: VAR T
  Scr1: LEMMA PscanR(a, R0, piv2) & a(R0)<=piv2 & R = R0 IMPLIES QscanR(a, R0, R, piv2);
  mR(R0): nat = R0 - le;
  Scr2: LEMMA PscanR(a, R0, piv2) & a(R0)>piv2 IMPLIES
                    PscanR(a, R0 - 1, piv2) & mR(R0 - 1)< mR(R0)
END ScanR
```