J Autom Reasoning (2014) 53:129–139
DOI 10.1007/s10817-013-9300-y

Proof Pearl: The KeY to Correct and Stable Sorting

Stijn de Gouw · Frank de Boer · Jurriaan Rot

Received: 26 July 2013 / Accepted: 19 December 2013 / Published online: 15 January 2014

Abstract We discuss a proof of the correctness of two sorting algorithms: Counting sort and Radix sort. The semi-automated proof is formalized in the state-of-the-art theorem prover KeY.

Keywords Sorting · Correctness · Theorem prover · KeY · Counting sort · Radix sort

1 Introduction

Sorting is an important algorithmic task used in many applications. Two main aspects of sorting algorithms which have been studied extensively are complexity and correctness. In 1971, Foley and Hoare [8] published the first formal correctness proof of a sorting algorithm (Quicksort). While this is a handwritten proof, the development and application of (semi)-automated theorem provers has since taken a huge flight. The major sorting algorithms Insertion sort, Heapsort and Quicksort were proven correct by Filliâtre and Magaud [7] using the proof assistant Coq. Recently, Sternagel [11] formalized a proof of Mergesort within the interactive theorem prover Isabelle/HOL.

In this paper we discuss the formalization of the correctness of *Counting sort* and *Radix sort*. Counting sort is a sorting algorithm based on arithmetic rather than comparisons. Radix sort is tailored to sorting arrays of large numbers. It uses an auxiliary sorting algorithm to sort the digits of the large numbers one-by-one. The correctness of Radix sort assumes the

S. de Gouw · F. de Boer · J. Rot (✉)
LIACS – Leiden University, Leiden, Netherlands
e-mail: j.c.rot@liacs.leidenuniv.nl

S. de Gouw
e-mail: cdegouw@gmail.com

F. de Boer
e-mail: frb@cwi.nl

S. de Gouw · F. de Boer · J. Rot
CWI, Amsterdam, Netherlands

stability of the auxiliary sorting algorithm. Stability here means that the order between different occurrences of the same number is preserved. To the best of our knowledge, stability has only been formalized in higher-order logic [11].

We provide the first formal proof of correctness of Counting sort and Radix sort, using the state-of-the-art interactive theorem prover KeY [5]. KeY is specifically tailored for proving the correctness of real-world Java programs with a high degree of automation. For the specification, KeY supports the widely used Java Modeling Language (JML) [6]. Several industrial case studies have already been carried out in KeY [1, 9, 10]. In contrast to most industrial code, which is relatively straightforward, Counting sort and Radix sort are ingenious and non-standard algorithms.

The present study contributes to the body of practical experience by considering these two relatively small algorithms which however have inherently complex correctness proofs. Our specifications and proofs are in *first-order* logic, extended with permutation predicates. These predicates have been added to KeY only recently [4], and we provide the first practical case study using them.

Plan of the paper. In the following section we introduce the Java implementation of the two sorting algorithms. In Section 3 we give a high-level correctness proof of Counting sort and Radix sort. The fourth and last section consists of an experience report of the use of KeY.

2 Counting Sort and Radix Sort

Counting sort is a sorting algorithm based on addition and subtraction rather than comparisons. As input it takes an array a of (non-negative) integers, and a bound k on the values occurring in a. The worst case time complexity is in $\mathcal{O}(n + k)$, where n is the number of elements in a.[1] The following Java implementation sorts the first N elements of a, given the bound k:

Listing 1 Counting sort

```
1  public int[] countSort(int[] a, int N, int k) {
2    int[] c = new int[k]; //initializes to zero
3    int[] res = new int[N];
4    for(int j=0; j<N; j++)
5      c[a[j]] = c[a[j]] + 1;
6    for(int j=1; j<k; j++)
7      c[j] = c[j] + c[j-1];
8    for(int j=N-1; j>=0; j--) {
9      c[a[j]] = c[a[j]] - 1;
10     res[c[a[j]]] = a[j];
11   }
12   return res;
13 }
```

Intuitively, the algorithm works as follows. After the first loop, for arbitrary i, $c[i]$ contains the number of times that i occurs in a. During the second loop, the partial sums of c are computed (i.e., $c[i] = c[0] + \ldots + c[i]$), so that $c[i]$ contains the number of elements in

[1] Note that this does not conflict with the well-known lower bound of $n\lg(n)$, since that bound concerns sorting algorithms based on *comparing* array elements.

a that are less than or equal to i. At this moment, for every value i occurring in a, $c[i]$ can thus be interpreted as being the index in the sorted array before which the value i should occur — if there are multiple occurrences of i, then these should be placed to the left. Indeed in the final loop, c is used to place the elements of a in the resulting sorted array *res* by, for each element $a[i]$, first decreasing $c[a[i]]$ by one and then placing $a[i]$ at position $c[a[i]]$. Notice that equal elements are thus inserted from right to left — so by starting at the last element of a and counting down, the algorithm becomes *stable*.

Radix sort sorts an array of *large numbers* (each large number represented by an array of digits) digit-by-digit, using another stable sorting algorithm to sort on each individual digit. This is reminiscent of the typical way one would sort a list of words: letter by letter. An implementation of Radix sort is given below.

Listing 2 Radix sort

```
1 public int[][] radixSort(int[][] a, int N, int M, int k) {
2   for(int i=0; i<M; i++) {
3     a = stableSort(a, N, i, k);
4   }
5   return a;
6 }
```

The above algorithm sorts a 2-dimensional array of integers a up until the N-th row, where each row represents a large base-k number as an array of M digits, with the least significant digit first. Clearly, the time complexity is linear in the complexity of `stableSort`. The algorithm proceeds by sorting the rows with respect to each column: the call to `stableSort(a, N, i, k)` sorts the array by the i-th column. The columns are processed from left (least significant digit) to right (most significant digit). Notice that it is essential for Radix sort that this auxiliary algorithm is stable, so that the order induced by the earlier iterations is preserved on equal elements in the i-th column.

A suitable candidate for `stableSort` is (a slight adaptation of) the above implementation of Counting sort. The adaptation is necessary to take as input a 2-dimensional array and a column i, so that it sorts by the i-th column.

3 Correctness and Stability

As we have seen in the previous section, the correctness of Radix sort depends on the *stability* of the underlying sorting algorithm. In this section, we formalize the property of being "stable" and give a high-level proof of the correctness and the stability of Counting sort. Subsequently we prove the correctness of Radix sort. All contracts and invariants are formalized in JML to ensure compatibility with KeY.

For a clear presentation we will focus on the version of Counting sort which operates on arrays of integers, instead of the adapted one which works on arrays of large numbers.[2] Moreover we assume that arrays are unbounded, they are always allocated (i.e., not `null`), and different array variables point to different arrays (no aliasing between different array variables). In Section 4 we drop these assumptions and show how this affects the specifications and corresponding correctness proofs.

[2]The actual formal proof in KeY uses the real Java implementation, which operates on arrays of large numbers.

The following JML contract specifies a generic sorting algorithm:

```
/*@ public normal_behavior
  @   requires
  @        N >= 0 && k > 0
  @     && (\forall int i; 0 <= i && i < N; a[i] < k);
  @   ensures
  @        \seqPerm(\old(a), \result)
  @     && (\forall int i; 0 <= i && i < N;
  @ \result[i]  <=  \result[i+1]);
  @*/
public int[] sort(int[] a, int N, int k);
```

The precondition, specified by the JML `requires` clause, states that all integers in the input array `a` are bounded by `k`. In the postcondition (`ensures`), `\seqPerm(\old(a), \result)` guarantees that the returned array `\result` is a permutation of a. Here `\seqPerm` is an auxiliary predicate not directly part of JML (though in principle it can be defined, see Section 4). The second conjunct of the postcondition states that `\result` is sorted up to index N.

The above contract specifies correctness, but not stability. While stability has a clear intuitive meaning, it is not possible to formalize it directly since equal array elements cannot be distinguished. We formalize stability with the help of the auxiliary array variable *idx*.[3] This variable should store for each index in the sorted array, the index of that element in the original input. For example, in Listing 1, directly after line 9, we add the assignment `idx[c[a[j]]]=j;`. The contract below specifies a *stable* sorting algorithm:

Listing 3 Contract of a generic stable sorting algorithm

```
/*@ public normal_behavior
  @   requires
  @        N >= 0 && k > 0
  @     && (\forall int i; 0 <= i && i < N; a[i] < k);
  @   ensures
  @        \seqPerm(\old(a), \result)
  @     && (\forall int i; 0 <= i; 0 <= i && i < N;
  @            \result[i] <= \result[i+1])
  @     && \seqNPerm(idx)
  @     && (\forall int i; 0 <= i && i < N;
  @            \result[i] == \old(a)[idx[i]])
  @     && (\forall int i; 0 <= i && i < N;
  @          \result[i] == \result[i+1] ==> idx[i] < idx[i+1]);
  @*/
public int[] stableSort(int[] a, int N, int k);
```

Here `\seqNPerm(a)` is a predicate which is true if and only if the array segment $a[0 \ldots a.length - 1]$ contains a permutation of the integers $0, \ldots, a.length - 1$. This contract expresses that idx is a bijection from the input a to the returned array, which keeps (indices of) equal input values in their original order (by the last conjunct).

[3] Auxiliary variables are variables that affect neither the values of the other variables or fields, nor the termination behavior.

This contract relies on adding an auxiliary variable *idx* to the implementation of `stableSort`. But since actual implementations used in practice do not contain this variable, the question arises how we can justify using those implementations. In particular, in Radix sort, can we use an implementation of `stableSort` that does not contain the auxiliary variable *idx*? The next theorem (which can be safely skipped by the reader) says we can avoid auxiliary variables, provided they do not occur in the postcondition of the contract we prove. This implies that in Radix sort we can use an implementation of `stableSort` not containing *idx*.

Let $D = \{P_i :: S_i \mid 1 \leq i \leq n\}$ be a set of procedure declarations and S be a statement which can call (only) these procedures. If z is an auxiliary variable for S with respect to D, then S' is the statement obtained from S by deleting assignments to z, and $D' = \{P_i :: S'_i \mid 1 \leq i \leq n\}$ is the corresponding set of procedure declarations (i.e., delete z from all procedure bodies). Furthermore if $A = \{\{p_i\}P_i\{q_i\} \mid 1 \leq i \leq n\}$ is a set of assumptions about the procedures then $A' = \{\{\exists z : p_i\}P_i\{\exists z : q_i\} \mid 1 \leq i \leq n\}$.

Theorem 1 (**Deleting Auxiliaries**) *For all assertions p, q and statements S: If $A \vdash_D \{p\}S\{q\}$ is derivable using the proof rules for recursive programs given in* [2] *without the conjunction rule, then so is $A' \vdash_{D'} \{\exists z : p\}S'\{\exists z : q\}$.*

Proof By induction on the length of the derivation of $A \vdash_D \{p\}S\{q\}$, followed by a case analysis on the last rule applied in the derivation. □

Remark 1 The above theorem does not extend to proofs that use the conjunction rule: $A' \vdash_{D'} \{\exists z : p_1\}S\{\exists z : q_1\}$ and $A' \vdash_{D'} \{\exists z : p_2\}S\{\exists z : q_2\}$ do not imply $A' \vdash_{D'} \{\exists z : p_1 \wedge p_2\}S\{\exists z : q_1 \wedge q_2\}$ since $\exists$ does not distribute over conjunctions. But even without the conjunction rule, the proof system is relative complete [3], so it is always possible to avoid the conjunction rule.

Remark 2 The translation is automatic, and thus no new invariants and contracts have to be devised.

Remark 3 The resulting translated proof does not necessarily contain only first-order assertions anymore. The translation introduces an existential quantification over the auxiliary variables and in our Radix sort proof, the auxiliary variable idx is of an array type: thus the existential quantification ranges over all unary functions on the domain and is second-order.

3.1 General Auxiliary Functions

For a human readable proof of Counting sort, and for both the specification and proof of Radix sort, it is absolutely crucial to introduce suitable abstractions. We therefore define the following auxiliary functions:

Name	Meaning
$\mathrm{val}(b, r, d, a)$	$\sum_{i=0}^{d}(a[r][i] * b^i)$
$\mathrm{cntEq}(x, r, a)$	$\lvert\{i \mid 0 \leq i \leq r \wedge a[i] = x\}\rvert$
$\mathrm{cntLt}(x, r, a)$	$\lvert\{i \mid 0 \leq i \leq r \wedge a[i] < x\}\rvert$
$\mathrm{pos}(x, e, l, a)$	$\mathrm{cntEq}(x, e, a) + \mathrm{cntLt}(x, l, a)$

Intuitively, val(b, r, d, a) is the large number represented in base b which is stored in row r of the array of large numbers a (and this number has $d + 1$ digits). The functions cntEq and cntLt count the number of elements in the array segment $a[0 \ldots r]$ respectively equal to or smaller than x. As a consequence of these definitions, pos$(a[i], i, N - 1, a) - 1$ is the position of $a[i]$ in the sorted version of a.

The function val can easily be implemented in JML using the built-in constructs `\sum` and `\product`. The value of cntEq(x, r, a) (and similarly cntLt(x, r, a)) can be represented in JML by

```
\sum int i; 0<=i && i<=r; (x==a[i]) ? 1 : 0
```

3.2 Counting Sort Proof

With the above definitions in place, we are ready to prove that the implementation of Counting sort satisfies the contract in Listing 3. To this end, we devise the loop invariants of Counting sort. The first loop (Listing 1, line 4-5) sets $c[i]$ to the number of occurrences of the value i in $a[0 \ldots j - 1]$. Thus we use the invariant:

```
\forall int i; c[i] == cntEq(i, j-1, a);
```

The second loop replaces each $c[i]$ with its partial sum. We formalize this by the following invariant:

```
  (\forall int i; 0 <= i && i <= j-1;
     c[i] == cntEq(i, N-1, a) + cntLt(i, N-1, a))
&& (\forall int i; j <= i && i < k;
     c[i] == cntEq(i, N-1, a))
```

The first conjunct ranges over the elements in c which have already been replaced by their partial sum, the second conjunct ranges over the elements which have not been processed yet (and hence, obey the invariant of the first loop).

The invariant of the last loop is as follows:

```
  (\forall int i; j+1 <= i && i < N;
       (res[pos(a[i], i, N-1, a)-1] == a[i]
    && idx[pos(a[i], i, N-1, a)-1] == i))
&& (\forall int i; 0 <= i && i < N;
      c[a[i]] == pos(a[i], j, N-1, a))
```

Recall that pos$(a[i], i, N - 1, a) - 1$ is the position of $a[i]$ in the sorted version of a. Thus the last conjunct intuitively means that $c[a[i]] - 1$ points to the position in which $a[i]$ should be stored in the sorted array. The assertion about *res* expresses that *res* is the sorted version of a, while the assertion about idx expresses that *idx* is the 'inverse' (modulo an offset of 1) of pos. The invariant gives rise to several verification conditions.[4] We discuss the most interesting ones. For readability we abbreviate the invariant by I. Furthermore, whenever it is clear from the context we denote pos$(x, i, N - 1, a)$ by pos(x, i) and pos$(a[i], i)$ by pos(i). Thus for example, pos$(i) - 1$ is the index of $a[i]$ in the sorted version of a.

[4] Verification conditions for a program are a set of purely logical formulae (i.e., not containing any program fragments) whose truth imply the correctness of that program.

Our first proof obligation states that pos satisfies a weak form of injectivity.

$$\forall i \in [j : N-1] : \mathrm{pos}(i) = \mathrm{pos}(j) \rightarrow a[j] = a[i]$$

This follows directly from the definitions of pos, cntEq and cntLt. The next verification condition characterizes the behavior of pos.

$$\begin{aligned}\forall i \in [0 : N-1] :\ & a[i] = a[j] \rightarrow \mathrm{pos}(a[j], j) - 1 = \mathrm{pos}(a[i], j-1) \\ & \wedge\ a[i] \neq a[j] \rightarrow \mathrm{pos}(a[i], j) = \mathrm{pos}(a[i], j-1)\end{aligned}$$

The truth of the first conjunct follows from the fact that $\mathrm{cntEq}(a[j], j, a) - 1 = \mathrm{cntEq}(a[j], j-1, a)$. The second conjunct holds since $\mathrm{cntEq}(x, j, a) = \mathrm{cntEq}(x, j-1, a)$ whenever $x \neq a[j]$. After the execution of the loop (i.e., when $j = -1$) *res* must be sorted:

$$\forall i \in [0 : N-2] : I \wedge j = -1 \rightarrow res[i] \leq res[i+1]$$

This is true since the invariant implies $res[\mathrm{pos}(i) - 1] = a[i]$ for $i \in [0 : N-1]$. But as remarked above, $\mathrm{pos}(i) - 1$ is the position of $a[i]$ in the sorted version of a, hence *res* is sorted. Finally, *idx* stores the 'inverse' of pos:

$$I \wedge j = -1 \rightarrow \forall i \in [0 : N-1] : res[i] = a[idx[i]]$$

This can be deduced from the fact that pos (modulo an offset of 1) and *idx* are bijections on $[0 : N-1]$, together with the first part of the invariant:

$$\forall i \in [0 : N-1] : res[\mathrm{pos}(i) - 1] = a[i] \wedge idx[\mathrm{pos}(i) - 1] = i)\,.$$

3.3 Radix Sort Proof

The correctness of Radix sort relies on the correctness of the stable sorting algorithm used in Radix sort. In the proof below, we assume only the contract of the generic stable sorting algorithm, instead of a particular implementation. This has the advantage that instead of being tied to Counting sort, *any* stable algorithm can be used within Radix sort, as long as it satisfies the contract of `stableSort`. There is one issue: while Radix sort processes arrays of large numbers, each represented as an array, the above contract of `stableSort` concerns arrays of integers. The adaptation to large numbers is as follows:

```
/*@ public normal_behavior
  @   requires
  @        N >= 0 && k > 0 && col >= 0
  @     && (\forall int i; 0 <= i && i < N; a[i][col] < k);
  @   ensures
  @        \seqPerm(\old(a), \result)
  @     && (\forall int i; 0 <= i && i < N;
  @             \result[i][col] <= \result[i+1][col])
  @     && \seqNPerm(idx)
  @     && (\forall int i; 0 <= i && i < N;
  @             \result[i] == \old(a)[idx[i]])
  @     && (\forall int i; 0 <= i && i < N;
  @             \result[i][col] == \result[i+1][col]
  @                             ==> idx[i] < idx[i+1]);
  @*/
public int[][] stableSort(int[][] a, int N, int col, int k);
```

Given the definitions of the auxiliary functions, the specification of Radix sort is as follows:

```
/*@ public normal_behavior
  @   requires
  @        N >= 0 && k > 0 && M >= 0
  @     && (\forall int i, col; 0 <= i && i < N && 0 <= col
  @             && col < M; a[i][col] < k);
  @   ensures
  @        \seqPerm(\old(a), \result)
  @     && (\forall int row; 0 <= row && row < N;
  @                 \val(k,row,M-1,\result) <=
  @                 \val(k,row+1,M-1, \result)));
  @*/
public int[][] radixSort(int[][] a, int N, int M, int k);
```

The formula `\forall int row (...)` expresses that the large number in each row of the returned array is smaller or equal to the number in the next row, when interpreted in base k up to the digit at position $M-1$.

The correctness proof then is based on the following loop invariant I:

```
   \seqPerm(a, \old(a))
&& (\forall int row; 0 <= row && row < N;
        \val(k,row,i-1,a) <= \val(k,row+1,i-1,a))
&& (i>0 ==> \seqNPerm(idx));
```

Intuitively, the formula `\forall int row (...)` states that a is sorted with respect to the first i digits. When proving that the body of the loop preserves I, the main verification condition that arises states that the invariant follows from the postcondition of the procedure call, provided that the invariant was true initially. There are two problems: how to deal with `\old`, and how to refer to the contents of a before the call. Both problems can be solved by introducing a logical variable A in the contract of `stableSort` as follows: we add $A = a$ to the precondition and substitute A for `\old(a)` in the postcondition. Let $post'$ be the resulting postcondition. Formally the verification condition is then as follows:

$$I[a := A] \wedge post'[\texttt{\result} := a] \rightarrow \\ \forall row \in [0 : N-1] : \mathrm{val}(k, row, i, a) \leq \mathrm{val}(k, row+1, i, a)$$

To see that this formula is valid, consider an arbitrary row $r \in [0 : N-1]$. Given the assumption $I[a := A] \wedge post'[\texttt{\result} := a]$, we must prove $\mathrm{val}(k, r, i, a) \leq \mathrm{val}(k, r+1, i, a)$. From $post'[\texttt{\result} := a]$ we can infer $a[r][i] \leq a[r+1][i]$. We distinguish two cases.

- $a[r][i] < a[r+1][i]$. Then also $a[r][i] * k^i < a[r+1][i] * k^i$. Clearly $\mathrm{val}(k, r, i-1, a) < k^i$, since $\mathrm{val}(k, r, i-1, a)$ is a number with i digits in base k, while k^i has $i+1$ digits in base k. But $\mathrm{val}(k, r, i, a) = \mathrm{val}(k, r, i-1, a) + a[r][i] * k^i$, hence $\mathrm{val}(k, r, i, a) \leq \mathrm{val}(k, r+1, i, a)$.
- $a[r][i] = a[r+1][i]$. Then it suffices to prove $\mathrm{val}(k, r, i-1, a) \leq \mathrm{val}(k, r+1, i-1, a)$. But $post'[\texttt{\result} := a]$ implies that $a[r] = A[idx[r]]$, and analogously for row $r+1$, so it suffices to prove $\mathrm{val}(k, idx[r], i-1, A) \leq \mathrm{val}(k, idx[r+1], i-1, A)$. Observe that since $a[r][i] = a[r+1][i]$, we may deduce $idx[r] < idx[r+1]$ from $post'[\texttt{\result} := a]$ (this is where stability is used!). But the invariant implies $\mathrm{val}(k, r_1, i-1, A) \leq$

$\mathrm{val}(k, r_2, i-1, A)$ if $r_1 < r_2$. Instantiating r_1 with $idx[r]$ and r_2 with $idx[r+1]$ gives the desired result.

This concludes the proof of Radix sort.

4 KeY: An Experience Report

In this section we discuss our practical experience with KeY [5]. KeY is a state-of-the-art theorem prover for Java based on symbolic execution (to 'execute' the Java program under consideration), dynamic logic (DL) to specify properties of the program and generate verification conditions, and the sequent calculus to prove the verification conditions. KeY also supports JML: all JML annotations in the Java source (i.e., loop invariants, method contracts etc.) are translated to DL fully automatically.

The following table summarizes some statistics of the proofs in KeY:

	Counting Sort	Radix Sort
Rule applications	372.307	169.030
User interactions	2.228 (0.7 %)	838 (0.5 %)

"Rule applications" serves as a measure for the length of the proofs: this row contains the total number of proof rule applications used in the proofs, whereas "User interactions" indicates the number of proof rule applications that were applied manually by the authors (i.e., required creativity). The statistics show that the degree of automation of KeY for both algorithms was over 99 %.

The formal proofs are significantly larger than the high-level proofs, for several reasons. First, the above statistics for counting sort concern a version that sorts arrays of large numbers instead of arrays of integers. Second, we used the automatic proof strategies of KeY as much as possible, but the strategies do not always find the shortest proofs. Third, Java contains several features that were ignored in the high-level proofs but complicate the formal KeY proofs.

For example, different array variables in Java can refer to the same array (object), and the algorithms potentially break when this happens. To rule this form of aliasing out, we had to add the following precondition to Counting sort:

```
(\forall int row; 0 <= row && row < N;
    a[row] != idx && res[row] != idx)
```

Only a small number of rule applications concern this assertion, all of which were applied automatically by KeY.

Another Java feature, which leads to a larger increase in rule applications, is the fact that arrays are bounded. For example, to ensure that the assignment `res[c[a[j]]] = a[j]` does not lead to out-of-bounds exceptions, KeY generates *four* proof obligations: j must be between the bounds of the array a (this condition must be proven twice, since $a[j]$ occurs twice), and $a[j]$ and $c[a[j]]$ must be within the array bounds of respectively C and res. To avoid duplication in the proofs due to multiple references to the same array element, we changed the Java source to `int tmp = a[j]; res[c[tmp]] = tmp`. KeY was able to automatically prove that the array references to c in the first two loops did not violate the array bounds, and similarly for a in the third loop. The references to res and c in the third

loop required some user interactions. Still, after the transformations, less than 5 % of the rule applications concerned array bounds.

The part of the proof by far responsible for the most rule applications (over 60 %) surprisingly is unrelated to deriving validity of the verification conditions discussed in the previous section. Instead it concerns proving that the value of our auxiliary functions cntEq, cntLt, pos and val is the same in different heaps that arise during execution. Note first that these functions indeed depend on the contents of the heap, since their value depends on the contents of an array (the array object a passed as a parameter), and arrays are allocated on the heap. Since the heap is represented explicitly by a term in KeY, the actual KeY formalization of the definitions of these functions contain an additional parameter Heap. In fact, since a is the only parameter of the auxiliary functions which has a class type, the value of the auxiliary functions depends *only* on the part of the heap containing a; other parts of the heap are simply not visible. However, the program never changes the contents of a: only parts of the heap irrelevant to the value of the auxiliary functions are changed. Unfortunately, currently KeY cannot detect this without unrolling the definition of the auxiliary functions. After unrolling, KeY could prove in most (but not all) cases automatically that the heap was only changed in ways irrelevant to the value of the auxiliary functions, though at the expense of a huge number of rule applications due to the size of the involved heap terms. One workaround for this is to surround any reference to the auxiliary functions by `\old` in the loop invariants. This seemingly small change, which causes all occurrences of auxiliary functions to be evaluated in the same (old) heap, resulted in a reduction of the Radix sort proof from 169.030 rule applications to a little over 60.000! This change also reduced the number of manual user interactions by about 30 %.

One final point of discussion concerns the auxiliary predicates `\seqPerm` and `\seqNPerm`. Unlike the auxiliary functions cntEq, cntLt, pos and val, these auxiliary predicates cannot be defined directly in JML, since JML assertions are first-order. In theory they *can* be defined indirectly using a Gödel encoding (since JML contains enough arithmetic), but this leads to a very low-level specification. Moreover the corresponding verification conditions in the proof of the contract become so large and low-level that one can forget about automation. Thus, although it is possible in theory to reason about permutations in first-order logic by using an arithmetical encoding, this is infeasible in practice. KeY solves this by adding a sequence type to the logic, along with several high-level lemmas and theorems about the predicate `\seqPerm(s,s')`, with the intended meaning that the sequences s and s' are permutations of each other, and `\seqNPerm(s)`, with the intended meaning that the sequence s is a permutation of the sequence $0, \ldots, \texttt{\seqLen}(s) - 1$. The detailed KeY formalization of these predicates can be found in [4, Appendix A]. So far, little is known about the implications regarding automation. The present case study provides the first empirical results: about 20 % of the total manual interactions concerned reasoning about sequences.

The full proofs together with the relevant version of KeY are available online.[5]

Acknowledgments We would like to express our gratitude to Jeannette de Graaf and Marcello Bonsangue for helpful discussions, and Richard Bubel for technical support with KeY. The research of J. Rot has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

[5] http://www.liacs.nl/~jrot/sorting

References

1. Ahrendt, W., Mostowski, W., Paganelli, G.: Real-time Java API specifications for high coverage test generation. In: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '12, pp. 145–154. ACM, New York (2012). doi:10.1145/2388936.2388960
2. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd Edn. Texts in Computer Science. Springer-Verlag (2009). 502 pp, ISBN 978-1-84882-744-8
3. Apt, K.R., de Boer, F.S., Olderog, E.R., de Gouw, S.: Verification of Object-Oriented programs: A transformational approach. J. Comput. Syst. Sci. **78**(3), 823–852 (2012)
4. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Secure information flow for Java. A Dynamic Logic approach. Karlsruhe reports in informatics; 2013-10, KIT (2013). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000036786
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, Lecture Notes in Computer Science, vol. 4334. Springer (2007)
6. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transfer **7**(3), 212–232 (2005)
7. Filliâtre, J.C., Magaud, N.: Certification of sorting algorithms in the system Coq. In: Theorem Proving in Higher Order Logics: Emerging Trends. Nice, France (1999). http://www.lri.fr/filliatr/ftp/publis/Filliatre-Magaud.ps.gz
8. Foley, M., Hoare, C.A.R.: Proof of a recursive program: Quicksort. Comput. J. **14**(4), 391–395 (1971)
9. Mostowski, W.: Formalisation and verification of Java Card security properties in Dynamic Logic. In: M. Cerioli (ed.) Proceedings Fundamental Approaches to Software Engineering (FASE), Edinburgh, Lecture Notes in Computer Science, vol. 3442, pp. 357–371. Springer (2005). http://www.springerlink.com/link.asp?id=x0u5bj47bcqhy4b7
10. Mostowski, W.: Fully verified Java Card API reference implementation. In: VERIFY (2007)
11. Sternagel, C.: Proof Pearl - A mechanized proof of GHC's mergesort. J. Autom. Reasoning **51**(4), 357–370 (2013)