Verification of an In-place Quicksort in ACL2

Sandip Ray
sandip@cs.utexas.edu

Rob Sumners
robert.sumners@amd.com

Department of Computer Science
The University of Texas at Austin
Austin, Texas, USA

Abstract

We present a proof of an efficient, in-place Quicksort implementation [1] using single-threaded objects (stobjs) in ACL2 [3, 4]. We demonstrate that the Quicksort implementation is equivalent to a simple insertion-sort function that is shown to produce an ordered permutation of its input. For ease of reasoning, the demonstration is carried out by verifying a series of "intermediate" sorting functions. The intermediate functions are equivalent to the efficient Quicksort implementation, but written in a more applicative style, and hence easier to reason about. We then decompose the proof into a verification of the equivalence of the efficient implementation with an intermediate implementation, and a proof of correctness of the intermediate implementation. We show how this decomposition allows us to simplify our reasoning about stobjs and obtain a cleaner proof of the implementation.

1 Introduction

In ACL2 version 2.4, support for efficient array manipulation via the use of single-threaded objects (stobjs) was introduced. The syntactic restrictions ACL2 enforces on the use of stobjs ensures that the underlying efficient destructive implementation of arrays coincides with the applicative semantics of ACL2 functions. In this paper, we show how to implement and verify an efficient, in-place Quicksort function which uses ACL2 stobjs. Unfortunately, the efficient Quicksort function is difficult to reason about directly in ACL2. We show how to decompose the verification task into two tractable pieces. In one piece, we prove the equivalence of the efficient Quicksort with a stobj-free function which mimics its effects. In the second piece, we prove correctness of the latter function. We observe that constraining the amount of reasoning required about recursive functions containing single-threaded objects makes the proof much simpler. To this end, we design the stobj-free versions of Quicksort by looking at the induction schemes suggested by the efficient recursive functions.

The remainder of the paper is organized as follows: in section 2, we provide an overview of the efficient Quicksort implementation that we verify. In section 3, we give an overview of the correctness proof and motivation for the auxiliary functions we define for this proof. Finally, we provide some concluding remarks in section 4.

2 Quicksort Implementation and Specification

Our Quicksort implementation sorts its input according to the total order `<<` on all ACL2 objects [5] defined in the book `''books/misc/total-order''`. While the choice of this order is not critical for our implementation of Quicksort, the fact that the total order applies to all ACL2 objects does enable us to design a function that can be proved to sort arbitrary ACL2 objects without any assumption about their types.

The implementation uses the stobj `qstor` as a temporary store in which to sort the input list. The stobj `qstor` has a resizable array called `objs`, providing random-access to its elements. We define functions which load the input into this array, and extract an output list from the sorted array. The main in-place sorting operations for the array `objs` occurs in two functions, namely, `split-qs` and `sort-qs`. Their definitions are provided in Figures 1 and 2.

```
(defun natp (x)
  (and (integerp x) (>= x 0)))

(defun ndx< (x y)
  (implies (and (natp x) (natp y))
           (< x y)))

(defstobj qstor
  (objs :type (array T (0))
        :resizable t))

(defun split-qs (lo hi splitter qstor)
  (declare (xargs :stobjs qstor))
  (if (ndx< hi lo) (mv lo qstor)
    (let* ((swap-lo (<<= splitter (objsi lo qstor)))
           (swap-hi (<< (objsi hi qstor) splitter))
           (qstor (if (and swap-lo swap-hi)
                      (swap lo hi qstor)
                    qstor)))
      (split-qs (if (implies swap-lo swap-hi)
                    (1+ lo)
                  lo)
                (if (implies swap-hi swap-lo)
                    (1- hi)
                  hi)
                splitter qstor))))
```

Figure 1: ACL2 function for splitting the array, given a pivot element, `splitter`.

The function `split-qs` implements the main partitioning task for Quicksort. Specifically, it takes two indices `lo`, and `hi`, and an array element called the `splitter`, rearranges (permutes) the array `objs`, and returns the rearranged store and an `index`. Informally, the rearranged array has the property that elements in the sub-array from `lo` to `(1- index)` (if such elements exist) must be smaller (according to `<<`) than `splitter`, and elements in the sub-array from `index` to

```
(defun sort-qs (lo hi qstor)
  (declare (xargs :stobjs qstor))
  (if (ndx<= hi lo)
      qstor
    (mv-let (index qstor)
        (split-qs lo hi (objsi lo qstor) qstor)
      (if (ndx<= index lo)
          (sort-qs (1+ lo) hi qstor)
        (let ((qstor (sort-qs index hi qstor)))
          (sort-qs lo (1- index) qstor))))))
```

Figure 2: Function for sorting the stobj `qstor` in the index range from `lo` to `hi`.

`hi` must be at least as large (according to `<<`) as `splitter`.

The function `sort-qs` sorts the array `objs` between the indices `lo` and `hi`. We call `split-qs` with `splitter` as the element of the array in position `lo`. Then we recursively sort the rearranged array returned by `split-qs` by sequentially sorting the sub-array from `lo` to `(1- index)` and the sub-array from `index` to `hi`.

```
(defun qsort (x)
  (with-local-stobj qstor
      (mv-let (rslt qstor)
          (let* ((size (length x))
                 (qstor (alloc-qs size qstor))
                 (qstor (load-qs x 0 size qstor))
                 (qstor (sort-qs 0 (1- size) qstor)))
            (mv (extract-qs 0 (1- size) qstor)
                qstor))
        rslt)))
```

Figure 3: The main qsort function

The main sorting function `qsort` (see figure 3) maps any given list of objects to the corresponding sorted list of objects. `qsort` performs this operation efficiently by creating a local stobj using the form `with-local-stobj` which was added in ACL2 version 2.6. This local stobj `qstor` is first initialized by the function `alloc-qs` which resizes the `objs` array in `qstor` to allocate enough room to store the elements in `x`. The `objs` array is then loaded with the input list `x` by the function `load-qs`. This local stobj `qstor` is sorted in place by the function `sort-qs` and finally, the sorted list is then extracted from `qstor` by the function `extract-qs`. After the resulting list is extracted, the scope of the `qstor` object is terminated by `with-local-stobj`. This allows the function `qsort` to use the local stobj `qstor` as an efficient workspace without introducing the stobj into the signature of the function (i.e. `qsort` will have the same signature as our insertion-sort function `isort` and thus, we can safely replace any call of `isort` with a call of `qsort`, assuming we prove they are equivalent functions).

The verification task is then to prove that `qsort` is correct. A sorting function is usually deemed correct if it returns an ordered permutation of its input. We prove this fact about `qsort`

```
(defun insert (e x)
  (if (or (endp x)
          (<< e (first x)))
      (cons e x)
    (cons (first x)
          (insert e (rest x)))))

(defun isort (x)
  (if (endp x) ()
    (insert (first x)
            (isort (rest x)))))
```

Figure 4: The Specification Function for Quicksort

by first proving that `qsort` is equivalent to a simple insertion-sort function `isort` (see figure 4) and then proving that `isort` returns an ordered permutation. The latter is a standard ACL2 exercise, so we turn our focus instead to the following main theorem in our verification task[1]:

```
(DEFTHM qsort-is-correct
  (implies (true-listp x)
           (equal (qsort x) (isort x))))
```

3 Overview of The Proof

At a high level, we reason as follows. Both functions `qsort` and `isort` produce an ordered permutation of the input, and the ordered permutation of a list is unique. Hence `qsort` and `isort` must be equal. However, while it is a simple exercise to prove that `isort` produces an ordered permutation, it is not so clear that `qsort` does the same. For reasoning about `qsort` we need to reason about two different aspects of the function. We need to be able to prove that the function `sort-qs` permutes (rearranges) the portion of the array between `hi` and `lo` respecting the ACL2 total order `<<`, and further, that the list extracted out of the array (using the function `extract-qs`) preserves the relative ordering of the elements in the array.

To insulate the reasoning about permutations from the arguments about `extract-qs`, we write a simple, stobj-free version of qsort, called `qsort-fn`. Logically, `qsort-fn` is equivalent to `qsort`. However, the function is written in an applicative style, and the reasoning is simplified by a much simpler partitioning step. The partitioning step, instead of returning a rearranged list and an index, will now return two different lists, called `lower-part` and `upper-part`, modeling the portions of the array from `lo` to `(1- index)` and from `index` to `hi` respectively. We call `qsort-fn` recursively on the two lists and `append` the results to get the final sorted list. We provide the definition of `qsort-fn` in figure 5. It is easy to prove that `qsort-fn` produces an ordered permutation of its input, and hence is equivalent to `isort`.

[1]The main theorem could have simply been `(equal (qsort x) (isort x))`, but includes the hypothesis of `(true-listp x)` to allow certain reductions during the proof to be performed. While we believe the `(true-listp x)` is ultimately unnecessary, we also believe it is an uncontroversial assumption to make.

```
(defun qsort-fn (lst)
  (if (endp lst) nil
    (if (endp (rest lst))
        (list (first lst))
      (let ((lower (lower-part lst (first lst)))
            (upper (upper-part lst (first lst))))
          (if (endp lower)
              (cons (first lst)
                    (qsort-fn (rest lst)))
            (append (qsort-fn lower)
                    (qsort-fn upper)))))))
```

Figure 5: Intermediate Applicative-style Quicksort function `qsort-fn`

The proof that `qsort` is equivalent to `qsort-fn` is still complicated, however. Informally, the problem is that `qsort-fn` does not utilize a single function that is equivalent to `split-qs`. In fact, the configuration of the array after a call to `split-qs` is given (roughly) by the `append` of the two lists `lower-part` and `upper-part`. To rectify this situation, we define another applicative-style function that has an even closer correspondence to the state of affairs in the efficient `qsort` function. To achieve this, we need to define functions that closely simulate the function `split-qs`. The functions `merge-func` and `walk` in figure 6 define this correspondence. The theorems connecting `split-qs` and the two functions `merge-func` and `walk` that we prove are as follows:

```
(defthm walk-split-qs-equal
  (implies (and (natp lo)
                (natp hi))
           (equal (mv-nth 0 (split-qs lo hi splitter qs))
                  (+ lo (walk (extract-qs lo hi qs) splitter)))))

(defthm merge-func-split-qs-equal
  (implies (and (natp lo)
                (natp hi))
           (equal (extract-qs lo hi (mv-nth 1 (split-qs lo hi splitter qs)))
                  (merge-func (extract-qs lo hi qs) splitter))))
```

Using the definitions of `merge-func` and `walk`, we define a function `in-situ-qsort-fn` which is similar to `sort-qs`. The definition of this function is given in figure 7. We discuss how we arrived at this definition in a moment, but for now, notice that the function almost exactly mimics the operations of `sort-qs`. Further note that the proof of equivalence of `in-situ-qsort-fn` and `qsort-fn` is a simple matter in ACL2, after the following theorem connecting `merge-func` with the functions `lower-part` and `upper-part` has been proven.

```
(defthm merge-func-lower-upper-reduction
  (equal (merge-func x splitter)
         (append (lower-part x splitter)
                 (upper-part x splitter))))
```

```
(defun snoc (x a) (append x (list a)))

(defun last-val (x) (first (last x))

(defun del-last (x)
  (if (endp (rest x)) nil
    (cons (first x) (del-last (rest x)))))

(defun merge-func (x splitter)
 (if (endp x) nil
   (if (and (<<= splitter (first x))
            (<< (last-val x) splitter))
       (cons (last-val x)
             (snoc (merge-func (del-last (rest x)) splitter)
                   (first x)))
    (if (and (<<= splitter (first x))
             (<<= splitter (last-val x)))
        (snoc (merge-func (del-last x) splitter)
              (last-val x))
      (if (and (<< (first x) splitter)
               (<< (last-val x) splitter))
          (cons (first x)
                (merge-func (rest x) splitter))
        (cons (first x)
              (snoc (merge-func (del-last (rest x)) splitter)
                    (last-val x))))))))

(defun walk (x splitter)
  (if (endp x) 0
    (if (and (<<= splitter (first x))
             (<< (last-val x) splitter))
        (1+ (walk (del-last (rest x)) splitter))
      (if (and (<<= splitter (first x))
               (<<= splitter (last-val x)))
          (walk (del-last x) splitter)
        (if (and (<< (first x) splitter)
                 (<< (last-val x) splitter))
            (1+ (walk (rest x) splitter))
          (1+ (walk (del-last (rest x)) splitter)))))))
```

Figure 6: Applicative Functions Equivalent to `split-qs`

```
(defun in-situ-qsort-fn (lst)
  (if (endp lst) nil
    (if (endp (rest lst))
        (list (first lst))
      (let ((merge (merge-func lst (first lst)))
            (ndx (walk lst (first lst))))
        (if (zp ndx)
            (cons (first merge)
                  (in-situ-qsort-fn (rest merge)))
          (let ((upper (in-situ-qsort-fn (last-n ndx merge)))
                (lower (in-situ-qsort-fn (first-n ndx merge))))
             (append lower upper)))))))
```

Figure 7: Quicksort Function Corresponding to `sort-qs`

With the proof of equivalence of `in-situ-qsort-fn` with `qsort-fn` and the proofs of correspondence of `qsort-fn` with `isort`, the only remaining piece of proof is the correspondence of `in-situ-qsort-fn` with `qsort`. The main theorem in the proof of this correspondence is the theorem below on equivalence of `in-situ-qsort-fn` and `sort-qs`.

```
(defthm sort-qs-equal-in-situ-qsort-fn
  (implies (and (natp lo)
                (natp hi)
                (<= lo hi))
           (equal (extract-qs lo hi (sort-qs lo hi qs))
                  (in-situ-qsort-fn (extract-qs lo hi qs)))))
```

To understand the approach for the proof of this theorem, it is instructive to observe the induction scheme suggested by `sort-qs`. The definition of the function `sort-qs` has a recursive call of the form `(sort-qs lo (1- index) (sort-qs index hi qs))`. Thus, an effective way of using this function as the induction scheme would be to come up with an auxiliary function where, in the induction step, we have a recursive call to a function that matches this term. This is the motivation behind the definition of `in-situ-qsort-fn` as it is. In the definition of `in-situ-qsort-fn`, we have a recursive call of the form `(in-situ-qsort-fn (first-n ndx merge))`, which, informally, "matches up" with the recursive step of `sort-qs`. Indeed, with the `in-situ-qsort-fn` as defined, we can easily match up the induction scheme with `sort-qs` and the result is proved in a rather simple manner.

It is legitimate at this point, to ask whether two distinct intermediate-level functions namely `qsort-fn` and `in-situ-qsort-fn` were really necessary for the proof, or whether the proof would have been simplified by using a single intermediate-level applicative-style quicksort function. While it would certainly be interesting to see a simpler proof using one intermediate-level function instead of two, coming up with such a function does not appear to be easy. Informally, the two functions serve two different purposes. The function `in-situ-qsort-fn` has been specifically designed to keep the proof of correspondence with `sort-qs` simple. In other words, such a proof now only requires an induction based on the scheme suggested by `sort-qs`, and the proof does not need to do any reasoning based on permutations or ordering. However, `qsort-fn` has been designed so that the fact that it produces an ordered permutation is simple to prove. At a higher level, the proof that the array is ordered depends on the property that the elements in one part of the array are "less" (according to <<) than each element in the other part. The formal statement is the theorem below.

```
(defthm ordp-lessp-not-lessp-reduction-1
  (implies (and (ordp x)
                (ordp y)
                (lessp e x)
                (not-lessp e y))
           (ordp (append x y))))
```

However, the function `merge-func`, used by `in-situ-qsort-fn`, returns a list in which the "less" and "not-less" portions are appended together. The proof that `in-situ-qsort-fn` returns an ordered list necessarily needs to be able to decompose the list returned by `merge-func` into two parts such that the first part is "less" and the second part is "not-less" than the splitter element. The simplest way of achieving that goal is to define an explicit function that has the lists explicitly split into two parts, and derive the correspondence of that function with `in-situ-qsort-fn`. Thus, a single intermediate applicative function that achieves all the goals does not seem plausible without complicating the proof by a large factor. However, it would be interesting to investigate a simpler proof of quicksort using a less number of intermediate functions.

4 Conclusion

The efficient in-place implementation of Quicksort using arrays is usually covered in a first course on algorithms at the undergraduate level (if not sooner). While the higher-level intuition behind the algorithm is easy to grasp, it is surprising how subtle a proof of correctness of the algorithm turns out to be. Indeed, the stubbornness of ACL2 in refusing to accept the correctness of conjectures it was asked to prove led to a much clearer understanding of the implementation. We decided to carry out this proof and report on it to illustrate the difficulties in proving efficient implementations correct. There are numerous details forced upon us in a formal proof which are blissfully ignored in the informal analysis of correctness. Sometimes these details point to possible bugs in the function definitions or to misunderstandings in their interpretations; and sometimes these details are simply nuisances of formal reasoning. In either case, the details cannot be ignored in the formal proof. The proof was carried out in a top-down style using the approach outlined in Kaufmann's case study in [4]. The use of "skip-proofs" turns out to be of invaluable help in modularizing the proof.

This work also arose as a by-product of our attempts to define a multi-threaded model of stobjs (i.e. when is it acceptable to logically break the single-thread while maintaining correspondence with the destructive operations under-the-hood). Notice that the cumbersome induction scheme produced by `sort-qs` is chiefly due to the sequential updating of the single-threaded object `qstor`, requiring the `let` binding. However, in Quicksort, since the recursive calls sort different portions of the array, it should be possible for the calls to use the same stobj in a multi-threaded environment, and sort the different portions of the stobj array `objs` in parallel. Determining an effective model for this multi-threaded use of stobjs is a future area of work.

It is also a goal of this paper to motivate continued efforts to improve the efficacy of ACL2 in the verification of efficient implementations of algorithms using stobjs. Functions manipulating stobjs are akin to functions defining state machines, and proofs about the former share a lot of structure with proofs of the latter. In particular, we had to define an invariant of the stobj `qstor` and prove that this invariant held throughout the various updates of the stobj. Our approach in this verification effort is analogous to refinement proofs between state machines, especially proofs verifying concurrent shared-memory protocols. The main difference is that we are dealing with terminating functions which affords functional decomposition, where in the case of refinement proofs between state machines often require the definition of a refinement map which demonstrates that the desired correspondence holds over infinite runs.

Finally, one interesting point is to compare our proof with the proof of Quicksort carried out in the theorem prover Coq in [2]. They prove Quicksort correct in a more direct Hoare-style logic with preconditions, postconditions, and loop invariants. The proof follows a clearer decomposition and does not introduce intermediate functions to show the correctness of Quicksort, but some of the conditions and invariants are involved. Use of Intermediate functions is a well-established technique for decomposing complicated proofs in ACL2, and shows promise in simplifying proofs of stobj-based functions [6]. Nevertheless, it would be worthwhile to investigate a more direct ACL2 proof of `qsort` and then compare the proof to ours.

Acknowledgments

The authors gained insight from discussions with the ACL2 community at UT. The first author especially acknowledges discussions with Jeff Golden, that helped to give shape to the proof.

References

[1] T. H. Cormen, C. E. Leiserson, and R. E. Rivest: *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts London, England.

[2] J. C. Filliatre: "Proof of the Quicksort Algorithm",
URL – http://paulliac.inria.fr/coq/contribs/quicksort.html.

[3] M. Kaufmann, P. Manolios, J Moore: *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June 2000.

[4] M. Kaufmann, P. Manolios, J Moore: *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, June 2000.

[5] P. Manolios, M. Kaufmann: "Adding a Total Order to ACL2" submitted to ACL2 Workshop 2002.

[6] R. Sumners: "Correctness Proof of a BDD Manager in the Context of Satisfiability Checking", in ACL2 Workshop 2000,
URL – http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/.