Verification of an In-place Quicksort in ACL2

ACL2 Workshop
Grenoble, France
April 9, 2002

Sandip Ray
sandip@cs.utexas.edu

Rob Sumners
robert.sumners@amd.com
Brief Introduction

- Goal: to demonstrate techniques for proving properties of stobj-based functions

- We chose in-place Quicksort as a common, well-understood example
  - The in-place Quicksort may also be of practical use in writing ACL2 programs which need to efficiently sort a large list of objects

- Supporting materials for this paper include the necessary definitions and proofs
[ In-place Quicksort ]

- Sort an array in-place by recursively dividing the problem and subsequently merging the results from this division

  - Choose a splitter object from the array and partition the input array into two halves and recursively sort the two halves
  - No subsequent merging is necessary since all elements in the upper half will be greater than the elements in the lower half

- We would like our definition of Quicksort to map an unsorted list to a sorted list (as opposed to using arrays)

  - So, in order to have the efficiency of array access and update, we will need to use a (local) single-threaded object
[Single-Threaded Objects (stobjs)]

- Stobjs were introduced in ACL2 2.4
  - Stobjs consist of a fixed set of fields, some of which may be arrays
  - The use of stobjs is syntactically restricted to ensure that the applicative semantics coincides with the destructive implementation

- In ACL2 2.6, several enhancements were made to stobjs:
  - Stobjs are more efficient, arrays can be resized, and stobjs may now be local to a given function
  - Stobj arrays in ACL2 are comparable in efficiency to arrays in C
    - Stobj array access and update (essentially) add the overhead of a function call
Definition of In-Place Quicksort-1

- Definition of main \texttt{qsort} wrapper function:

\begin{verbatim}
(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)))
\end{verbatim}

- Definition of recursive sorting function \texttt{sort-qs}:

\begin{verbatim}
(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)))))))
\end{verbatim}
● Definition of In-Place Quicksort-2

● Definition of array splitting function \texttt{split-qs}:

\begin{verbatim}
(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)))))
\end{verbatim}

● Definition of the stobj \texttt{qstor}:

\begin{verbatim}
(defun qstor
  (objs :type (array T (0))
       :resizable t
       :initially nil))
\end{verbatim}
The output of \texttt{qsort} is an ordered permutation of the input

- We use the ACL2 total order \texttt{<<}

We prefer to first define a simple insertion sort \texttt{isort} and:

- Prove that this function returns an ordered permutation (standard ACL2 exercise)

- Prove \texttt{(thm (equal (qsort x) (isort x)))}

  - In the paper we add a \texttt{(true-listp x)} hypothesis, but this is only a matter of convenience

\texttt{isort} can be viewed as an \textit{intermediate} function which separates the specification from the implementation

- We will introduce additional intermediate functions to aid in the proof
Reasoning about stobj functions

- Proofs about stobj functions encounter some common problems
  - Stobjs are frequently parameters (and return values) of every component function
  - Various properties will need to be proven to commute over the operations which update the stobj
    - For example (with \([a, b] \cap [x, y] = \emptyset\)):
      \[
      \begin{align*}
      \text{(equal (extract-qs a b (sort-qs x y qs))} \\
      \text{(extract-qs a b qs))}
      \end{align*}
      \]
  - Various invariants may need to be defined and proven to hold of the functions which update the stobj
  - The “logic” definition of the function will often be unwieldy to work with directly
    - Intermediate functions are often needed to factor the complexity of the proof into more manageable pieces
Our first intermediate function is an applicative Quicksort function:

\[
\text{(defun qsort-split (lst)}
\text{ (mv (lower-part lst (first lst))}
\text{ (upper-part lst (first lst)))}
\]

\[
\text{(defun qsort-fn (lst)}
\text{ (if (endp lst) nil}
\text{ (if (endp (rest lst))}
\text{ (list (first lst))}
\text{ (mv-let (lower upper)}
\text{ (qsort-split lst)}
\text{ (if (endp lower)}
\text{ (cons (first upper)}
\text{ (qsort-fn (rest upper)))}
\text{ (append (qsort-fn lower)}
\text{ (qsort-fn upper))))))))
\]
Intermediate Function #1, continued

- The definitions of lower-part and upper-part model split-qs

(defun lower-part (x s)
  (cond
   ((endp x) nil)
   ((and (<= s (first x))
        (<= (last-val x) s))
    (cons (last-val x)
          (lower-part (del-last (rest x)) s)))
   ((and (<= s (first x))
        (<= s (last-val x)))
    (lower-part (del-last x) s))
   ((and (<= (first x) s)
        (<= (last-val x) s))
    (cons (first x)
          (lower-part (rest x) s)))
   (t
    (cons (first x)
          (lower-part (del-last (rest x)) s)))))

- Relevant properties of upper-part and lower-part...
Refining the split function

- The definition of qsort-split is difficult to correlate directly with split-qs, so we introduce another refinement:

```lisp
(defun in-situ-qsort-split (lst)
  (let* ((merge (merge-func lst (first lst)))
         (ndx (walk lst (first lst)))
         (mv (first-n ndx merge)
              (last-n ndx merge))))
```

- We then define in-situ-qsort-fn to be qsort-fn with in-situ-qsort-split replacing qsort-split

  - The equivalence of in-situ-qsort-fn with qsort-fn easily reduces to proving the equivalence of in-situ-qsort-split with qsort-split
[ Relevant properties... ]

• Properties relating \texttt{in-situ-qsort-split} with \texttt{split-qs}:

\begin{verbatim}
(defun walk-split-qs-equal
  (implies (and (natp lo) (natp hi))
    (equal (mv-nth 0 (split-qs lo hi x qs))
            (+ lo (walk (extract-qs lo hi qs) x)))))

(defun merge-func-split-qs-equal
  (implies (and (natp lo) (natp hi))
    (equal (extract-qs lo hi
             (mv-nth 1 (split-qs lo hi x qs))
             (merge-func (extract-qs lo hi qs) x))))
\end{verbatim}

• Relating \texttt{sort-qs} with \texttt{in-situ-qsort-fn}:

\begin{verbatim}
(defun 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))))
\end{verbatim}
Concluding Remarks

• Previous work in Coq proved Quicksort using Hoare-style proof
  – i.e. loop invariants, preconditions, postconditions
  – Their proof is shorter, but comparison is difficult due to incongruences in libraries and definitions

• Quicksort is not the best example of the use of intermediate functions
  – This approach is more effective when stobjs are used to optimize the evaluation of applicative functions (e.g. hash tables, memoization, etc.)

• Future work:
  – Multi-threaded Quicksort with shared qstor
    • Proof requirements ensure that applicative semantics are still consistent with implementation
  – Develop library to aid in reasoning about stobjs