Verification of Array-Based Insertion Sort

ACL2 Lecture 3

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

Computer Science Department
University of Texas
2317 Speedway, M/S D9500
Austin, TX 78712-0233

May, 2021

Verification of Array-Based Insertion Sort

In this lecture, we explore how to prove the correctness of a *pointer-based*, in-memory, sorting procedure.

- We will model memory as a *fixed-length* list of integers.
- We access and update memory with *constant-time* operators.
- Using *pointers*, we access and update *in-memory* integers.
- We use *pointer* arithmetic to determine function termination.
- We prove the correctness of our *in-memory* insertion algorithm.
- We verify the correctness of our *in-memory* isort algorithm.

But, we must show more!

We show that our sorting algorithm doesn't alter other parts of the memory.

Representing our Memory as a List of Integers

Sometimes we wish to use routines that manipulate memory-based data, and we want to confirm that pointer-based routines behave properly.

Our memory is an `INTEGER-LISTP` list; for a "memory-level" view, we rotate a memory diagram somewhat counter-clockwise, so its visual representation is a right-associated tree is "laying on its side".

Memory, `M`, contains `(LEN M)` integers; addressed from `0` to `(1- END)`.

```
M --O--- o o o -----O-----O---- o o o ----O---- o o o ----O- NIL
    |               |     |               |               |
    E    . . .      a     b     . . .     z     . . .     W
 +-----+--    ---+-----+-----+--    ---+-----+--    ---+-----+
 |  E  |  o o o  |  a  |  b  |  o o o  |  z  |  o o o  |  W  |
 +-----+--    ---+-----+-----+--    ---+-----+--    ---+-----+
 0               L                           R               END
    Our Memory    <-- Vals from L up to R -->
```

One may think of `L` and `R` as *pointers* into memory `M`, where address `0` points to the start of the memory and address `(1- END)` to the last addressable location.

Characterizing Our Memory

We use ACL2's function INTEGER-LISTP to recognize memory as a fixed-length list of integers, and we use the LEN function to measure its size.

```
(defun int-listp (x)
  (if (atom x)
      (eq x nil)
    (and (integerp (car x))
         (int-listp (cdr x)))))
```

```
(defun len (x)
  (if (atom x)
      0
    (+ 1 (len (cdr x)))))
```

For each of our *in-memory* operations, we will prove that our memory remains an INTEGER-LISTP and that its LENgth remains unchanged.

```
(defun nth (x l)
  (if (endp l)
      nil
    (if (zp n)
        (car l)
      (nth (- n 1) (cdr l)))))
```

```
(defun !nth (pos val x)
  (if (zp pos)
      (cons val (cdr x))
    (cons (car x)
          (!nth (1- pos) val
                (cdr x)))))
```

And we prove various properties about our *arrays*; e.g.,

```
(implies (and (natp i) (< i (len x)))
         (equal (!nth i (nth i x) x) x))
```

Projecting a Sub-Sequence Out of our Memory

To compare memory configurations with our specifications, we define M-TO-L to project the contents of a range of memory locations.

```
(defun m-to-l (m l r)
  (declare (xargs :guard (and (integer-listp m)
                              (natp l)
                              (natp r)
                              (<= l r)
                              (<= r (len m)))
                  :measure (nfix (- r l)))) ;; must be a NATP
  (if (zp (- r l))
      nil
    (cons (nth l m)
          (m-to-l m (1+ l) r))))
```

Note the use of a :measure parameter: (nfix (- r l))

All recursive ACL2 functions must have a lexicographic measure that decreases with every recursive call.

Observations about our Projection Function

Extracting a range of memory values produces an `integer-listp` result.

```
(defthm integer-listp-m-to-l
  (implies (and (integer-listp m)
                (natp l) (<= r (len m)))
           (integer-listp (m-to-l m l r))))
```

For the lemma above, why don't we need `(natp r)` as a hypothesis? Consider:

```
(defthmd reason-r-is-natp-greater-than-0
  (implies (and (natp l)
                (not (zp (+ r (- l)))))
           (and (natp r)
                (< 0 r))))
```

Inductive fact: writing below the start address doesn't effect the projection.

```
(defthm m-to-l-!nth-above
  (implies (and (natp l)
                (natp l+)
                (< l l+))
           (equal (m-to-l (!nth l e m) l+ r)
                  (m-to-l m l+ r))))
```

Comparison of In-Memory Operations to List-Based Operations

Imagine we wish to sum the elements of a list of integers.

```
(defun sum-list (x)
  (declare (xargs :guard (integer-listp x)))
  (if (atom x)
      0
    (+ (car x)
       (sum-list (cdr x)))))
```

Similarly, imagine a function that sums a vector of in-memory integers.

```
(defun sum-sub-array (m l r)
  (declare (xargs :guard (and (integer-listp m)
                              (natp l) (natp r)
                              (<= l r)
                              (<= r (len m)))
                  :measure (nfix (- r l))))
  (if (zp (- r l))
      0
    (+ (nth l m)
       (sum-sub-array m (1+ l) r))))
```

Is the SUM-LIST of a projection equal to SUM-SUB-ARRAY of the same range?

The Correctness of our *In-Memory* Summation Function

Summing a range of in-memory elements is same as collecting the same range of elements and summing this collection.

```
(defthm sum-sub-array-is-same-as-project-and-sum-list
  (implies (and (integer-listp m)
                (natp l) (natp r)
                (<= l r) (<= r (len m)))
           (equal (sum-sub-array m l r)
                  (sum-list (m-to-l m l r)))))
```

(SUM-SUB-ARRAY Mem L R) is equal to the (SUM-LIST (LIST a b ... z)).

```
 (M-TO-L Mem L R) ---O-----O---- o o o ----O---- NIL
  projection         |     |               |
                     a     b     . . .     z
   Mem:
  +-----+--    ---+-----+-----+--    ---+-----+--    ---+-----+
  |  E  |  o o o  |  a  |  b  |  o o o  |  z  |  o o o  |  W  |
  +-----+--    ---+-----+-----+--    ---+-----+--    ---+-----+
  0               L                            R               END
```

We have *lifted* ourselves from a pointer-based, in-memory algorithm to list-based operations.

Insertion into an ORDEREDP Array

```lisp
(defun insert-e-in-m (m l r e)
  "Insert E into integer memory having one empty slot at L."
  (declare (xargs :guard (and (integer-listp m)
                              (natp l) (natp r)
                              (<= l r) (<= r (len m))
                              (integerp e))
                  :measure (nfix (- r l))))
  (if (zp (- r l))
      m  ;; Zero length array; nothing can be done
    (let ((l+1 (1+ l)))
      (if (= l+1 r)
          ;; Single-element array, perform insertion
          (!nth l e m)
        (let ((nx-e (nth l+1 m)))
          ;; Compare E with first element of array sub-sequence
          (if (<= e nx-e)
              ;; Place E if it is less than or equal NX-E
              (!nth l e m)
            ;; Otherwise, m[l] <- m[l+1], and we move on...
            (let ((updated-m (!nth l nx-e m)))
              (insert-e-in-m updated-m l+1 r e))))))))
```

Facts About Inserting an Element into an ORDEREDP Memory

To confirm our *memory contract*, we prove LEN and INTEGER-LISTP properties.

```
(defthm len-insert-e-in-m
  (implies (and (natp l)
                (<= r (len m)))
           (equal (len (insert-e-in-m m l r e))
                  (len m))))

(defthm integer-listp-insert-e-in-m
  (implies (and (integer-listp m)
                (natp l) (<= r (len m))
                (integerp e))
           (integer-listp (insert-e-in-m m l r e))))
```

And, importantly, we confirm no other part of memory is changed.

```
(defthm insert-e-in-m-does-not-alter-m-outside-sort-range
  (implies (and (natp l) (natp i)
                (or (< i l)
                    (and (<= r i)
                         (<= r (len m)))))
           (equal (nth i (insert-e-in-m m l r e))
                  (nth i m))))
```

Correctness of In-Memory Insertion

ACL2's ENCAPSULATE limits the visibility of the first lemma to this environment.

```
(encapsulate ()
  (local
   (defthm cons-is-same-as-insert-when-e-less-than-m-l+1
     (implies (and (integer-listp m)
                   (natp l)
                   (<= r (len m))
                   (integerp e)
                   (<= e (nth l m)))
              (equal (insert e (m-to-l m l r))
                     (cons   e (m-to-l m l r))))))

  (defthm insert-e-in-m-ok
    (implies (and (integer-listp m)
                  (natp l) (natp r)
                  (< l r) (<= r (len m))
                  (integerp e))
             (equal (m-to-l (insert-e-in-m m l r e) l r)
                    (insert e (m-to-l m (1+ l) r))))))
```

The lemma above says in-memory insertion works just like list-based insertion.

Sort From The End to the Front

Insert elements from right-to-left (end-to-start) into an ORDEREDP list.

```
(defun isort-in-m (m l r)
  "ISORT insertion iteration."
  (declare (xargs :guard (and (integer-listp m)
                              (natp l) (natp r)
                              (< l r) (<= r (len m)))
                  :measure (nfix (- r l))
                  :verify-guards nil))  ;; Guards not verified!
  (if (zp (- r l))
      m
    (let ((l+1 (1+ l)))
      (if (= l+1 r)
          ;; One-element array; do nothing
          m
        ;; Sort the rest (the tail) of the array
        (let ((e (nth l m))
              (m-updated (isort-in-m m l+1 r)))
          ;; Insert E in ordered array M-UPDATED
          (insert-e-in-m m-updated l r e))))))
```

Notice that the guards are not verified.

Facts About Our *In-Memory* Sorting Procedure

To verify the guards of ISORT-IN-M , we prove that it LEN is unchanged.

```
(defthm len-isort-in-m
  (implies (and (natp l)
                (<= r (len m)))
           (equal (len (isort-in-m m l r))
                  (len m))))

(defthm integer-listp-isort-in-m
  ;; This lemma needs to know the LEN of ISORT-IN-M
  (implies (and (integer-listp m)
                (natp l)
                (<= r (len m)))
           (integer-listp (isort-in-m m l r))))

(verify-guards isort-in-m)
```

Induction is needed to prove (integerp-listp (isort-in-m m l r)).

Once these facts are known, the ISORT-IN-M guards can be verified.

More Properties about *In-Memory* Sorting

We prove an inductive fact that sorting above `l` doesn't change value at `l`.

```
(defthm nth-isort-in-m
  (implies (and (integer-listp m)
                (natp l)
                (natp l+)
                (< l l+))
           (equal (nth l (isort-in-m m l+ r))
                  (nth l m))))

(defthm isort-in-m-does-not-alter-elements-outside-sort-range
  (implies (and (natp l)
                (<= l r)
                (natp i)
                (or (< i l)
                    (and (<= r i)
                         (<= r (len m)))))
           (equal (nth i (isort-in-m m l r))
                  (nth i m))))
```

Key property: `ISORT-IN-M` does not alter memory outside of its sort range.

Correctness of our *In-Memory* Insertion-Sort Procedure

We have established that `ISORT-IN-M`:

- does not change the size of the memory,
- does not change the memory outside of the sort range, and
- leaves the elements in the memory sorted.

```
(defthm isort-in-m-ok
  (implies (and (integer-listp m)
                (natp l)
                (natp r)
                (<= l r)
                (<= r (len m)))
           (equal (m-to-l (isort-in-m m l r) l r)
                  (isort (m-to-l m l r)))))
```

Inherently, pointer-based algorithms – where one has to keep track of memory usage – require more analysis effort than their list-based algorithms.

Challenge: Can you specify and verify an in-memory, quick-sort algorithm?