Single-Threaded Objects

Robert S. Boyer
J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Office: TAY 4.140A
Email: moore@cs.utexas.edu

http://www.cs.utexas.edu

Abstract

To support efficient execution of certain kinds of models, especially models of microprocessors, ACL2 provides "single-threaded objects," structures with the usual copy on write applicative semantics but for which writes are implemented destructively. Syntactic restrictions insure consistency between the formal semantics and the implementation. The design of single-threaded objects has been influenced both by the need to make execution efficient and the need to make proofs about them simple. We discuss the issues.

Acknowledgments: The work on single-threaded objects was a straightforward generalization by Moore of the joint work of Boyer and Moore on the handling of `state` in all versions of ACL2 since its beginning in 1989. Matt Kaufmann and Moore have been jointly responsible for ACL2 since 1993. We would not have generalized our `state`-handling had we not been shown the work of Hardin, Greve, Wilding and Cowles at Rockwell-Collins. Thanks to all.

Standard ACL2 Application

```
(defun run (MS n)
  (if (zp n)
      MS
    (run (step MS) (- n 1))))

(defun step (MS)
  (let ((inst (next-inst MS)))
    (case (op-code inst)
     (ADD           (execute-ADD inst MS))
     (LOAD          (execute-LOAD inst MS))
     ...)))

(defun execute-ADD (inst MS)
  (let ((a1 (arg1 inst))
        (a2 (arg2 inst)))
    (update-nic (+ 1 (nic MS))
                (update-memi a1
                             (+ (memi a1 MS) (memi a2 MS))
                             MS))))
```



```
(defun memi (i MS)
  (nth i (nth 1 MS)))

(defun update-memi (i v MS)
  (update-nth 1
              (update-nth i v (nth 1 MS))
              MS))
```

where the fundamental access and update operations are:

```
(defun nth (n x)
  (cond ((zp n) (car x))
        (t (nth (- n 1) (cdr x)))))

(defun update-nth (n v x)
  (cond ((zp n) (cons v (cdr x)))
        (t (cons (car x)
                 (update-nth (1- n) v (cdr x))))))
```

Observe that `update-nth` copies x down to the n^{th} element.

```
(defun update-nth (n v x)
  (cond ((zp n) (cons v (cdr x)))
        (t (cons (car x)
                 (update-nth (1- n) v (cdr x))))))
```

Thus, the model of updating memory is "copy on write" (down to the address at which the write occurs),

```
(defun update-memi (i v MS)
  (update-nth 1
              (update-nth i v (nth 1 MS))
              MS)),
```

which makes our models run more slowly.

What if we implemented `update-memi` so that it modified *MS* destructively as would happen in a C-program?

Standard ACL2 Application

```
(defun run (MS n)
  (if (zp n)
      MS
    (run (step MS) (- n 1))))
(defun step (MS)
  (let ((inst (next-inst MS)))
    (case (op-code inst)
     (ADD            (execute-ADD inst MS))
     (LOAD           (execute-LOAD inst MS))
     ...)))
(defun execute-ADD (inst MS)
  (let ((a1 (arg1 inst))
        (a2 (arg2 inst)))
    (update-nic (+ 1 (nic MS))
                (update-memi a1
                             (+ (memi a1 MS) (memi a2 MS))
                             MS))))
```

Here, destructive implementation of *MS* updates would be ok!

Single-Threaded Objects

In ACL2 Version 2.4, a "single-threaded object" is a structure whose use is so restricted syntactically that we guarantee that there is exactly one pointer to it.

Thus, updates to the structure can be carried out destructively.

Issues

- How is a single-threaded objected defined?
- What are the axiomatic semantics of access and update?
- What syntactic restrictions are enforced?
- How are the semantics implemented?
- What is an example of the use of single-threaded objects?

Defstobj

The following new event declares *MS* to be a single-threaded object.

```
(defstobj MS
  (ptr :type (integer 0 *) :initially 0)
  (mem :type (array t (5)) :initially nil))
```

Logically speaking a "typical value" of *MS* might be `(3 (0 1 TWO (X X X) "four"))`.

```
(defstobj MS
  (ptr :type (integer 0 *) :initially 0)
  (mem :type (array t (5)) :initially nil))
```

Gives rise to the following definitions.

...

The Accessor and Updater for `mem` *Elements*

```
(defun memi (i MS)
  (declare (xargs :guard
                  (and (integerp i) (<= 0 i) (< i 5) (msp MS))))
  (nth i (nth 1 MS)))

(defun update-memi (i v MS)
  (declare (xargs :guard
                  (and (integerp i) (<= 0 i) (< i 5) (msp MS))))
  (update-nth 1
              (update-nth i v (nth 1 MS))
              MS))
```


Semantics

```
(defstobj MS
  (ptr :type (integer 0 *) :initially 0)
  (mem :type (array t (5)) :initially nil))
```

Gives rise to the following definitions.

The Recognizer

```
(defun msp (MS)
  (declare (xargs :guard t))
  (and (true-listp MS)
       (= (length MS) 2)
       (ptrp (nth 0 MS))       ; (integerp ptr) ∧ 0 ≤ ptr
       (memp (nth 1 MS))))     ; (true-listp mem) ∧ (len mem) = 5
```


Syntactic Restrictions

Draconian Idea: In executable code, the only reference to the *MS* object shall be in the variable named *MS*.

Consequences

- *MS* is a top-level global variable that contains the current object, *ms*.
- If a function uses the formal parameter *MS*, the only "actual expression" that can be passed into that slot is *MS*.
- The accessors and updaters have a formal parameter named *MS*, thus, those functions can only be applied to the current object.
- The ACL2 primitives, such as `cons`, `car` and `cdr`, may not be applied to the variable *MS*. Thus, for example, *MS* may not be consed into a list (which would create another pointer to it) or accessed or copied via "unapproved" means.


```
(defstobj MS
  (ptr :type (integer 0 *) :initially 0)
  (mem :type (array t (5)) :initially nil))
```

Gives rise to the following definitions.

...

The Accessor and Updater for the `ptr` Field

```
(defun ptr (MS)
  (declare (xargs :guard (msp MS)))
  (nth 0 MS))

(defun update-ptr (v MS)
  (declare (xargs :guard
                  (and (and (integerp v) (<= 0 v))
                       (msp MS))))
  (update-nth 0 v MS))
```

- The updaters return a "new *MS* object", i.e., ms'; thus, when an updater is called, the only variable which can hold its result is *MS*.
- If a function calls an *MS* updater, it must return *MS*.
- When a top-level expression involving *MS* returns an *MS* object, that object becomes the new current value of *MS*.

Signatures

To enforce the restrictions, ACL2 deduces the "signatures" of functions.

- `((CONS * *) => *)`
- `((CAR *) => *)`
- `((PTR MS) => *)`
- `((MEMI * MS) => *)`
- `((UPDATE-PTR * * MS) => MS)`
- `((FOO * MS * STATE) => (MV * * STATE)`

Examples of the Use of *MS* Primitives

Initially, *MS* has the value `(0 (NIL NIL NIL NIL NIL))`.

```
ACL2 !>MS
<ms>
ACL2 !>(ptr MS)
0
ACL2 !>(update-ptr 3 MS)
<ms>
ACL2 !>(ptr MS)
3
ACL2 !>(memi 2 MS)
NIL
ACL2 !>(update-memi 2 'abc MS)
<ms>
ACL2 !>(memi 2 MS)
ABC
ACL2 !>(car MS)
ACL2 Error in TOP-LEVEL:  A single-threaded object, namely MS, is being
used where an ordinary object is expected.
```


The Logic Remains the Same

```
ACL2 !>(thm (equal (ptr MS) (car MS)))
...
Q.E.D.

ACL2 !>(thm (equal (ptr x) (car x)))
...
Q.E.D.
```

An Additional Restriction

- When a non-top-level expression returns an *MS* object, the result must be bound to the local variable named *MS* (rather than passed as an actual to a function with a formal parameter named *MS*).

Consider `(f (smash MS) (g MS))`, where

- `((g MS) => *)`
- `((smash MS) => MS)`
- `((f MS *) => MS)`

If `smash` is destructive, then Lisp would apply `g` to the "new" *MS*, contrary to the meaning of the term.

The new rule disallows

- `(f (smash MS) (g MS))`.

You must write either

```
(let ((MS (smash MS)))
  (f MS (g MS))).
```

or

```
(let* ((v (g MS))
       (MS (smash MS)))
  (f MS v)).
```

depending on what you mean.

Implementation

```
(defstobj MS
  (ptr :type (integer 0 *) :initially 0)
  (mem :type (array t (5)) :initially nil))
```

The actual initial value of *MS* is

- `(#(0) #(NIL NIL NIL NIL NIL))`

This non-ACL2 Common Lisp object is assigned to the global constant `*the-live-ms*`, which is unavailable to the ACL2 user but which is understood to be the top-level value of the ACL2 variable *MS*.

The raw Lisp definitions of `memi` and `update-memi` are

```
(defun memi (i MS)
  (cond
   ((eq MS *the-live-ms*)
    (aref (car (cdr MS)) i))
   (t (nth i (nth 1 MS)))))

(defun update-memi (i v MS)
  (cond
   ((eq MS *the-live-ms*)
    (cond
     (*wormholep* (wormhole-er 'update-memi (list i v MS)))
     (t (setf (aref (car (cdr MS)) i)
              v)
        MS)))
   (t (update-nth 1 (update-nth i v (nth 1 MS)) MS))))
```

(We make extensive use of `declare` and `the` to make these defuns compile more efficiently.)

Proofs

```
(defthm show-insert
  (implies (< (ptr MS) 5)
           (equal (show (insert x MS))
                  (cdr (append (show MS) (list x))))))
```

Note this formula violates the rules on executable syntax, because it relates the show of *MS* before and after a modification.

But we have to allow this to use the language as a specification language.

Furthermore, we have to be able to execute single-threaded functions on non-live objects because such applications arise in proofs.

These features distinguish ACL2 from other applicative languages.

Using *MS*

```
(defun insert (x MS)
   (update-ptr (inc (ptr MS))
               (update-memi (ptr MS) x MS)))
```

```
(defun insert (x MS)
  (declare (xargs :stobjs (MS)))
  (let ((MS (update-memi (ptr MS) x MS)))
   (update-ptr (inc (ptr MS)) MS)))
```

or, if you define the macro `sequentially` to do nested `let`-bindings of *MS*

```
(defun insert (x MS)
  (declare (xargs :stobjs (MS)))
  (sequentially
   (update-memi (ptr MS) x MS)
   (update-ptr (inc (ptr MS)) MS)))
```


Another Definition

```
(defun scan (x MS)
  (declare (xargs :stobjs (MS)))
  (if (consp x)
      (sequentially
       (scan (car x) MS)
       (scan (cdr x) MS))
    (insert x MS)))
```

Let τ be `'(((A . B) . C) . (D . ((E . (F . G)) . (H . I))))`.

Then `(show (scan` τ `MS))` is `(E F G H I)`.

No new storage is allocated to compute `(scan` τ `MS)`.

Suppose *MS* is in its initial configuration and you do

```
ACL2 !>(sequentially
        (insert 'A MS)
        (insert 'B MS)
        (insert 'C MS)
        (insert 'D MS))
<ms>                        ;  but the logical value is (4 (A B C D NIL)).

ACL2 !>(insert 'E MS)
<ms>                        ;  but the logical value is (0 (A B C D E))

ACL2 !>(insert 'F MS)
<ms>                        ;  but the logical value is (1 (F B C D E))
```

It is convenient to define `((show MS) => *)` to convert the *MS* object to some ordinary object.

```
ACL2 !>(show MS)
(B C D E F)
```

```
(defthm show-scan
  (implies (< (ptr MS) 5)
           (equal (show (scan x MS))
                  (lastn 5 (append (show MS) (fringe x))))))

(defthm show-scan-corollary
  (implies (and (< (ptr MS) 5)
                (<= 5 (len (fringe x))))
           (equal (show (scan x MS))
                  (lastn 5 (fringe x)))))
```


Evidence of Practicality

ACL2 is coded largely in itself (7% of the .o files are generated from non-applicative Common Lisp and the rest (93%) is generated from ACL2).

The 93% constitutes five megabytes of applicative code.

We use single-threaded objects, namely, `state`, in all our applicative code and it is handling all the work of rewriting, bdds, linear arithmetic, heuristics, io, book handling, error checking, etc.

This is powerful evidence our support for single-threaded objects permits programs of practical size and complexity.

Conclusion

We sometimes call single-threaded objects *von Neumann bottlenecks* because they force sequentialization. Parallelization of the rest is still possible.

An important limitation of single-threaded objects is that we do not allow one within another. It would be nice to find syntactic rules to permit "hierarchical single-threaded objects."

Our design is much simpler than "competing" designs in other functional languages, e.g., Haskell's monads.

Our design is less powerful too.

But it gets the job done and is very efficient.