Single-Threaded Objects in ACL2

Robert S. Boyer*
J Strother Moore†

April 13, 1999

Abstract

ACL2 is a first-order applicative programming language based on Common Lisp. It is also a mathematical logic for which a mechanical theorem-prover has been implemented in the style of the Boyer-Moore theorem prover. The ACL2 system is used primarily in the modeling and verification of computer hardware and software, where the executability of the language allows models to be used as prototype designs or "simulators." 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.

1 Background

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." We use the name both for a mathematical logic based on applicative Common Lisp [24] and for a mechanized theorem proving system for that logic developed by Matt Kaufmann and author Moore. ACL2 is closely related to the Boyer-Moore logic and system and its interactive enhancement [2, 3, 4]. ACL2's primary use is in modeling microprocessors and proving theorems about those models. The key reason we abandoned the Nqthm logic and adopted applicative Common Lisp is that the latter can produce extremely efficient runtime code. Execution efficiency is important because our microprocessor models are often run as simulators.

*Department of Computer Sciences, University of Texas, Austin, TX 78712, boyer@cs.utexas.edu.

†Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu.

In ACL2, a single-threaded object is a structure whose use is syntactically restricted so as to guarantee that there is exactly one reference to the structure. A perfect example of a single-threaded object is the "current state" in a microprocessor model. The fact that only one reference to the object exists allows updates to the structure to be performed destructively even though the axiomatized semantics of update is "copy on write."

This work is thus addressing the classic problem of how to implement updates efficiently in an applicative setting. In that sense, our work is akin to that of [23, 13, 26, 27]. Indeed, Schmidt introduced the term "single threaded" in [23]. [27] contains a good survey of the most popular alternative in applicative languages, Haskell's "monads". But ACL2 is unusual among purely applicative programming languages in that it is focused as much on using the language as a specification language and on mechanically produced proofs as on execution efficiency. We find that these other concerns influenced our treatment of single-threaded objects. We do not regard the addition of single-threaded as objects as having changed the logic but rather just restricted its executable subset for efficiency reasons. The unrestricted logic is available for specification and proof.

The ACL2 theorem prover is used primarily in hardware and software verification. For example, the correctness of floating point division and square root on the AMD K5 microprocessor was proved using the theorem prover [18, 20]. ACL2 has been used to prove the correctness of hardware designs for floating point addition, subtraction, multiplication, division, and square root on the AMD K7 [21]. It has been used to model the Motorola CAP digital signal processor, to prove that the CAP pipeline architecture correctly implements the instruction set architecture, and to prove properties of CAP microcode programs [6, 7]. ACL2 has been used to study the problem of specifying advanced microprocessor architectures, in particular the interaction of such features as multi-issue, speculative execution and exceptions and has been used to prove that one such design correctly implements a sequential architecture [22]. ACL2 was used to model the Rockwell-Collins JEM1, the world's first silicon Java Virtual Machine [9, 10, 11, 12]. The use of ACL2 to prove theorems about simple Java-like byte code programs is reported in [17].

One of the main reasons ACL2 has found industrial application is that it is both a logic and an efficient applicative programming language. Once a formal model is created it is possible to test it on concrete examples and to prove properties of it. Why might one want to run a formal model? Testing the model is a relatively inexpensive way to find the "easy" bugs. In addition, such a model can be used as a prototype for the intended component, allowing engineers to assess its appropriateness given the informal requirements. Another motivation for such testing is to corroborate the formal model against some other model, e.g., a circuit-level simulation or even an existing physical artifact like a chip or gate-array implementation. At AMD, ACL2 formal models have been executed on many millions of "test vectors" to corroborate them against other models.

The most common industrial applications of ACL2 involve models of mi-

croprocessors. Our motivation for adding "single-threaded objects" to ACL2 comes largely from the desire to speed up the execution of such models. Before further describing our work, it is helpful to look closely at how microprocessor models are written in ACL2. We assume a modicum of familiarity with Lisp.

Typically, the state of a microprocessor is modeled as an n-tuple containing fields representing memories of various kinds. Here we will imagine a state, *MS*, to be a triple containing a "next instruction counter" and two memories, one used for read/write and the other used to hold "execute-only" programs. The "state-transition" function, here called **step**, is a Lisp function that creates the next state from a given state, usually as a function of the "next instruction" indicated by the program counter and memory. The machine's fetch-execute cycle is then modeled by the simple recursive function

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

This Lisp command defines the function **run** so that, when applied to *MS* and *n*, it successively **step**s *MS* *n* times and returns the final result. In Lisp, the application of **run** to *MS* and *n* is written **(run** *MS* *n***)** instead of **run**(*MS*,*n*).

We might define **step** so that it fetches the next instruction from *MS* and then "does" that instruction to *MS*,

```
(defun step (MS)
  (do-inst (next-inst MS) MS))
```

where the function **next-inst** fetches the instruction indicated by the next instruction counter and **do-inst** is defined as a "big switch" that invokes the appropriate transition function depending on the opcode of the next instruction.

```
(defun do-inst (inst MS)
  (case (op-code inst)
    (LOAD           (execute-LOAD inst MS))
    (STORE          (execute-STORE inst MS))
    (ADD            (execute-ADD inst MS))
    (GOTO           (execute-GOTO inst MS))
    ...
    (otherwise MS)))
```

Each instruction modeled, e.g., **ADD**, has a logical counterpart that specifies the transition, e.g., **execute-ADD**. Here is one such definition:

```
(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 we are imagining that an **ADD** instruction has a "2-address" format. In the definition above we bind the variables a_1 and a_2 to the two addresses from which the instruction is to get its operands. We then construct a new state from *MS* by two "sequential" (i.e., nested) updates. The first (innermost) replaces the contents of memory location a_1 by the sum of contents of memory locations a_1 and a_2. The second update increments the next instruction counter, **nic**, by one. The program component of the state *MS* is unchanged.[1]

Suppose states are represented as triples and the memory component of a state *MS* is **(nth 1 *MS*)**, i.e., the 1st element of the triple. Suppose that the memory is itself represented as a linear list. Then **memi** and **update-memi** are defined as shown below.

```
(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 **(nth *n* *x*)** is the n^{th} element (0-based) of the list x and **(update-nth *n* *v* *x*)** copies the list x, replacing the n^{th} element with v. The definition of the latter is

```
(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, for example, **(update-nth 3 'G '(A B C D E))** is equal to **(A B C G E)**.

In principle, given a concrete microprocessor state and a particular number of steps to take we can compute the final state. This just requires executing **run** and the above subroutines on the concrete data. But if we actually implement **memi** and **update-memi** as shown above, the time taken to execute memory reads and writes in our model is linear in the address. This is because **nth** and **update-nth** "chase links" in the linked list representation of memories. In addition, because the formal semantics of **update-nth** is "copy on write," storage (in an amount proportional to the address) is allocated on memory writes.

[1] Machines of commercial interest often have more complicated instruction semantics, e.g., the +-expression might be replaced by (mod (+ (memi a_1 *MS*) (memi a_2 *MS*)) (expt 2 32)), but this example is suggestive of the essential character of such models.

But inspection of the nest of functions starting with `run` and proceeding through `step` to the individual semantic functions like `execute-ADD`, reveals that we could in principle do this computation by destructively re-using the representation of the initial state, provided we never needed the top-level state again. If `update-memi` were implemented destructively, modifying the existing representation of the current state to obtain the next one, no harm would come because in the functions above no function references the "old" state after any update to any part of it. This is a syntactic property of the definitions and depends, in part, on order of evaluation.

This observation has led us to incorporate into ACL2 the notion of a user-defined *single-threaded object*. Such an object is a structure, possibly containing linear lists accessed positionally and usually quite large. Accessor functions, such as `nic` and `memi`, are provided, as are update functions, such as `update-nic` and `update-memi`. The axiomatic descriptions of the functions are as indicated by the definitions of `memi` and `update-memi` above. This permits us to state and prove properties of functions using the single-threaded object. However, syntactic restrictions are enforced that insure that it is sufficient to allocate only one "live" copy of the object. Updates are performed destructively on the live object. The syntactic restrictions — which actually require that we make minor changes to some of the definitions above — insure that no well-formed code executed on the live object can detect the difference between the axiomatic and implemented semantics of updates.

The history of single-threaded objects in ACL2 is rather long. The initial design of the kernel of ACL2 was done in 1989 by Boyer and Moore. The system is coded almost entirely in its own applicative language. That forced us to provide ourselves an explicit notion of "state" into which we would accumulate the effects of a session with the user. The state of the ACL2 theorem prover, for example, includes a list of the definitions added by the user, the rewrite rules proved, etc. This notion of state also provides streams and files so that the input/output functions, proof descriptions, error handling, and the read-eval-print loop could be coded applicatively in the language. To make this applicative state efficient, we implemented it destructively, and we enforced certain draconian syntactic rules on the use of the name `state` so that the applicative semantics was not violated by the destructive implementation. We added axioms which characterized the semantics of the accessors and updaters of our state and proved theorems about state in order to bootstrap the system. This "single-threaded" notion of state has been present and available to the user in all versions of the system. Matt Kaufmann joined the ACL2 project soon after the treatment of `state` had stabilized; since 1993 most ACL2 development has been the joint work of Kaufmann and Moore.

However, while the ACL2 user could write functions that used our `state`, provided the syntactic rules were followed, and the user could prove theorems about those functions with ACL2, the user could not introduce his or her own single-threaded object.

Researchers at Rockwell-Collins, namely David Hardin, David Greve, and Matt Wilding, demonstrated the need for user-defined single-threaded objects by "cheating:" they implemented destructive state manipulation functions and then used them as though they were applicative – being careful to obey the syntactic restrictions. Their results are reported in [12]. When John Cowles, of the University of Wyoming, spent a sabbatical at Rockwell-Collins, he implemented macros that enforced their restrictions.

When they told us what they were doing, we recognized their approach as a straightforward generalization of what we were already doing for our `state` and added single-threaded objects as described here to ACL2 Version 2.4.

2 Introduction to ACL2

ACL2 is both the name of an applicative programming language and a theorem proving system for it. ACL2 is largely the work of Matt Kaufmann and Moore, building on work by Boyer and Moore. This section therefore describes joint work in which Kaufmann was a major contributor.

2.1 The Logic

The kernel of the ACL2 logic consists of a syntax, some rules of inference, and some axioms. The kernel logic is given precisely in [16]. The logic supported by the mechanized ACL2 system is an extension of the kernel logic.

The kernel syntax describes terms composed of variables, constants, and function symbols applied to fixed numbers of argument terms. Thus, `(*` x `(fact` n`))` is a term that might be written as $x \times n!$ in more traditional syntactic systems. After introducing Lisp-like terms, the kernel logic introduces the notion of "formulas" composed of equalities between terms and the usual propositional connectives. The kernel language is first order and quantifier free.

The ACL2 axioms describe the properties of certain Common Lisp primitives. For example,

Axioms.
x = y → `(equal` x y`)` = `t`

x ≠ y → `(equal` x y`)` = `nil`

x = `nil` → `(if` x y z`)` = z

x ≠ `nil` → `(if` x y z`)` = y

The expression `(cond (`p_0 x_0`) ... (`p_n x_n`) (t` x_{n+1}`))` is just an abbreviation for `(if` p_1 x_1 ... `(if` p_n x_n x_{n+1}`)...)`. Using the function symbols `equal` and

`if` we "embed" propositional calculus and equality into the term language of the logic and generally write terms instead of formulas.

The kernel logic includes axioms that characterize the primitive functions for constructing and manipulating certain Common Lisp numbers, characters, strings, symbols, and ordered pairs.

Of special importance here, besides `equal` and `if`, are `cons`, `car`, and `cdr`, which, respectively, construct a new ordered pair and return the left and right components of such a pair. The predicate `consp` "recognizes" `cons`-pairs by returning one of the symbols `t` or `nil` according to whether its argument is a `cons` pair.

The rules of inference are those for propositional calculus with equality, instantiation, an induction principle and extension principles allowing for the definition of new total recursive functions, new constant symbols, new "symbol packages," and the declaration of the "current package" (used to support possibly overlapping name spaces). Our extension principles specify conditions under which the proposed extensions are admissible. For example, recursive definitions must be proved to terminate. The admissibility requirements insure the consistency of the resulting extensions.

For example, here is the definition of the previously mentioned function `nth`.

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

The predicate `zp` is true if its argument is either 0 or not a natural number. Thus `nth` effectively "coerces" n to be a natural, by using `zp` as the "test against 0." All values of n other than natural numbers are treated as though they were 0. Termination of the recursion above is easy: when the recursive branch is taken, n is a non-0 natural number and the function decreases it in the recursion.

The logic supported by the ACL2 theorem prover is somewhat richer than the kernel logic sketched above. The full logic is obtained from the kernel by (a) a syntactic extension and some syntactic restrictions (b) the inclusion of an extension principle called "encapsulation" and a derived rule of inference called "functional instantiation," and (c) the inclusion of an extension principle called "`defchoose`" which provides the power of first-order quantification in ACL2. The syntactic extension is provided via the incorporation of Common Lisp's notion of macros, whereby new syntactic forms are implemented by functions that translate those forms into terms in the kernel syntax. The syntactic restrictions have to do with syntactic limitations on the use of certain primitives so as to allow efficient execution, as discussed in this paper. Encapsulation and related issues are discussed in [14], where admissibility requirements are extended to the full logic and insure not just consistency but conservativity.

2.2 The Relation to Common Lisp

Logically speaking, all ACL2 functions are total, but not all Common Lisp functions are total. For example, in Common Lisp, `cdr` is defined to be the right component of a `cons` pair and to be `nil` on the symbol `nil`. But ACL2 has the axiom

`(consp` x`)` $=$ `nil` $\rightarrow$ `(cdr` x`)` $=$ `nil`

Thus, in both ACL2 and Common Lisp, `(cdr nil)` is `nil`. But according to the axiom above, in ACL2 `(cdr 23)` is `nil` while in Common Lisp it is undefined and might signal an error or behave in some erratic or arbitrary way.

Our "completion" of Common Lisp makes the task of writing a theorem prover for it simpler, because the language is untyped and the axioms are strong enough to let us reduce to a constant any variable-free expression involving recursively defined functions in the primitives.

But only certain ACL2 expressions have their axiomatically described values under Common Lisp. The expressions in question are ones in which each function, f, is applied only to arguments within the domain prescribed for f by the Common Lisp specification [24]. The formalization of this notion of "prescribed domain" of a function is ACL2's notion of *guard*, a formula that describes the intended inputs to the function.

The guard for `nth`, above, requires that n be a natural number and x be a linear list or "true list". A linear list is a binary tree whose rightmost tip is `nil`. ACL2 uses an extension of Common Lisp's `declare` statement to allow the user to annotate definitions with their guards. Here is the definition of `nth` with its guard:

```
(defun nth (n x)
  (declare (xargs :guard (and (integerp n)
                              (<= 0 n)
                              (true-listp x))))
  (cond ((zp n) (car x))
        (t (nth (- n 1) (cdr x)))))
```

A guard may be any ACL2 formula in the formal parameters of the function. Often guards are type-like and the system supports the use of Common Lisp's `type` declaration in conjunction with guards declared as above.

We say a function is *Common Lisp compliant* if, when its guard is satisfied by the function's inputs, the guards of all subroutines are satisfied by their inputs. The process of verifying that a function is Common Lisp compliant is called *guard verification*. Since ACL2 has a mechanical theorem prover associated with it, guard verification is elegantly implemented. Formulas expressing the conditions above are generated and handed over to the theorem prover for proof. Roughly speaking, in the definition of a function f there is a *guard conjecture* for each occurrence of a call of a subroutine g. The guard conjecture says "if

the formal parameters of f satisfy the guard for f and the tests governing this call of g are true, then the actuals of the call of g satisfy the guard of g."

If an ACL2 function is Common Lisp compliant then any execution of it on inputs satisfying its guard is correctly calculated by executing the function in Common Lisp.

When the user submits an admissible function definition to ACL2, two functions are actually defined in the underlying Common Lisp. The first definition is called the *raw definition* and corresponds to what the user actually typed. The second is the *completed definition*. This definition is obtained from the given one by replacing all function names by the names of their completed counterparts as per the ACL2 axioms. For example, the primitive Common Lisp function named `cdr`, which is undefined on 23, is replaced by another symbol – actually the symbol `cdr` in another package – defined as our `cdr` is axiomatized. Both definitions can be compiled. Generally, the raw definitions are faster than the completed ones, because the latter do runtime type checks and the former do not.

When the user submits a form to be evaluated, the system runs the guards on the form and if they are satisfied, the form is evaluated in Common Lisp, i.e., the faster, raw definitions are run. Otherwise, the slower, completed form is evaluated. Note that the guard is irrelevant to the logical meaning of a function; it only affects the efficiency with which ACL2 can compute the value of the function. If a large system of definitions has been proved to be Common Lisp compliant and some function in that system is called, e.g., to simulate a test run of a microprocessor, then the guard of that top-level function call is tested once and all subsequent execution is of fast, raw code.

Because of guards, calls of compliant ACL2 functions can be replaced by raw Lisp that is more efficient than their logical definitions suggest. Consider the expression `(zp` n`)`. Logically this tests whether n is a non-0 natural number. One might think that the execution of `(zp` n`)` therefore required a runtime type check on n and the test $0 \leq n$. But the guard for `zp` is that n is a natural number. Hence, the compiled code for `(zp` n`)` can test just whether $n = 0$.

`Nth`, as shown above, is Common Lisp compliant. On inputs satisfying its guard, the compiled code repeatedly decrements n and `cdr`s x until $n = 0$ and then returns the `car` of x. We will use `nth` often in this paper.

2.3 About the Theorem Prover

The ACL2 theorem prover is an improved and extended descendent of the Boyer-Moore theorem prover, NQTHM, [2, 3, 4]. ACL2 presents itself to the user as a read-eval-print loop. In addition to the typical commands of defining functions and evaluating forms, ACL2 permits the user to pose theorems to be proved. The theorem prover is fully automatic but its behavior is determined, in part, by its state, which is in turn affected by the theorems it has already proved. We regard the theorem prover as interactive: it is led to the proofs of complicated

theorems by the user, who formulates appropriate intermediate results to prove first. These results are designed by the user to lead the system to the proof of the main result.

Here is some sample input to the theorem prover:

```
(defthm nth-update-nth
       (equal (nth i (update-nth j v x))
              (if (equal (nfix i) (nfix j))
                  v
                  (nth i x))))
```

This form directs the system to prove the above formula and then build it in as a rewrite rule with the name **nth-update-nth**.

Consider the theorem above. It is an equality and the left-hand side is the term denoting the i^{th} element in the result of updating x so that its j^{th} element is v. The right-hand side tells us what that element is. The expression `(nfix` i`)` "coerces" i to a natural: if i is a natural number, `(nfix` i`)` is identically i; otherwise, it is 0. If i and j are the same (when coerced to natural numbers), the answer is v; otherwise, the answer is the i^{th} element of x.

If we think of the update as a destructive operation on x, then this theorem relates the i^{th} element *after* the update to the i^{th} element *before* the update. But update is not destructive; x does not change. We are dealing with a logic here, not a programming language.

Logically speaking, there is no "before" or "after.". There is no such "event" as the "updating of x." Instead, the logical expressions x and `(update-nth` j v x`)` both denote objects and the theorem relates the i^{th} element of the object denoted by the first to the i^{th} element of the object denoted by the second.

The ACL2 theorem prover proves **nth-update-nth** automatically, by induction on i and the structure of the list x. After setting up a suitable base case and induction step, the theorem prover proves both cases by simplification, applying such axioms as the definitions of `nth` and `update-nth` and the fact that `(car (cons` x y`))` $= x$.

Once proved, the theorem is built into ACL2's simplifier as a rewrite rule. Suppose that the system later tries to prove a formula ϕ involving the term `(nth` i `(update-nth` j v x`))`. We will denote such a formula as ϕ[`(nth` i `(update-nth` j v x`))`]. The rewrite rule **nth-update-nth** will split this goal into two goals. In the first, the goal becomes $\phi[v]$ and has an additional hypothesis equating (as above) i and j; in the second, the goal becomes ϕ[`(nth` i x`)`] and has an additional hypothesis asserting that i and j are different. Of course, ϕ or the particular instantiations of i and j may make a case impossible (e.g., as when i and j are different constants or identical expressions).

By proving lemmas such as the one above, the user can configure ACL2 to do case splits and simplifications designed to prove certain classes of theorems of interest. The user can augment or control ACL2's proof search in a variety of other ways as well.

ACL2 is available without fee from the ACL2 home page, http://www.-cs.utexas.edu/users/moore/acl2. Five megabytes of hypertext documentation can be browsed there. The documentation can be downloaded with the ACL2 sources.

3 Single-Threaded Objects

In ACL2, a "single-threaded object" is a data structure whose use is so syntactically restricted that only one instance of the object need ever exist and its fields can be updated by destructive assignments.

From the logical perspective, a single-threaded object is an ordinary ACL2 object, e.g., composed of integers, symbols and conses. Logically speaking, ordinary ACL2 functions are defined to allow the user to "access" and "update" its fields. Logically speaking, when fields in the object, *obj*, are "updated" with new values, a new object, *obj'*, is constructed.

But by syntactic means we insure that after an updated version of the object is created there are no more references to the "old" object, *obj*. Then we can create *obj'* by destructively modifying the memory locations involved in the representation of *obj*. The syntactic means is pretty simple but draconian: the only reference to *obj* is in the variable named *OBJ*, where that is a "name" for the object introduced when the original instance was created.

The consequences of this simple rule are far-reaching and require some getting used to. For example, if *OBJ* has been declared as a single-threaded object name, then:

- *OBJ* is a top-level global variable that contains the current object, *obj*.
- If a function uses the formal parameter *OBJ*, the only "actual expression" that can be passed into that slot is *OBJ*; thus, such functions can only operate on the current object. Note that since the formal parameters of a function must be distinct, this rule prevents a single-threaded object being passed into a function in two or more argument positions, eliminating the possibility of aliasing.
- The accessors and updaters have a formal parameter named *OBJ*, 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 *OBJ*. Thus, for example, *OBJ* may not be consed into a list (which would create another pointer to it) or accessed or copied via "unapproved" means.
- The updaters return a "new *OBJ* object", i.e., *obj'*; thus, when an updater is called, the only variable which can hold its result is *OBJ*.

- If a function calls an *OBJ* updater, it must return *OBJ*.
- When a top-level expression involving *OBJ* returns an *OBJ* object, that object becomes the new current value of *OBJ*.

To avoid dependence on the left-to-right order of evaluation in Common Lisp, we impose another rule

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

Consider the term `(f (smash OBJ) (g OBJ))`, where `smash` is a function that takes *obj* as input and returns a modified version of it, *obj'*. Observe that `f` has two arguments and that both actuals mention *OBJ*. Logically speaking, the two occurrences of the variable *OBJ* refer to the same object, *obj*, which is, of course, the value of *OBJ* "before" the modification. But with Lisp's left-to-right order of evaluation and our surreptitious destructive modification of *obj* to produce *obj'*, the Lisp evaluation of this expression would apply `g` to *obj'*. Hence, the rule above disallows this term and requires us to write

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

If we mean to apply `g` to *obj* instead of *obj'* we must write

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

Note that `f` must return *OBJ* for these expressions to be legal under our rules.

In ACL2, `(let ((`v_1 `x1) ... (`v_n `xn)) body)`, where the v_i are distinct variable symbols and the x_i and *body* are terms, is logically equivalent to the term obtained by simultaneously and uniformly replacing the v_i by the corresponding x_i, i.e., $body_{[v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n]}$. The raw implementation of `let` binds the variables to the values of the terms and then evaluates the body in the extended binding environment. Because ACL2 is applicative, these two "meanings" of the expression are equivalent. Lisp's `let*` construct is similar but does sequential assignments (nested substitutions).

The above restrictions on the use of single-threaded objects are enforced by the ACL2 syntax checker. When a form is submitted to the read-eval-print loop, the terms in it are checked for well-formedness. This includes, for example, the expansion of macros, the check that functions are defined and the check that function calls are given the correct number of arguments.

To enforce the syntactic rules on single-threaded objects, we must know the "signature" of every function symbol. For example, `memi` is known to take an "ordinary" argument and a single-threaded object of type *MS* as input and to

yield an ordinary object as the single result. This is written `((memi * MS) ⇒ *)`. The signature of `update-memi` is `((update-memi * * MS) ⇒ MS)`. The signature of `cons` is `((cons * *) ⇒ *)`. Thus, the syntax checker is able to insure such things as that *MS* is passed into the second argument of `memi`, that *MS* is passed into the third argument of `update-memi`, and that *MS* is *not* passed into the second argument of `update-memi` or into either argument of `cons`. The last restriction prevents single-threaded objects from being referenced by other objects. The syntax checker also uses signatures to insure that the result of `update-memi` is immediately `let`-bound or else returned as the final answer.

ACL2 supports "multi-valued" functions, i.e., functions that return a vector of results. The rules above are generalized to handle such functions. For example, the function `func` might take an ordinary first argument, the single-threaded object *MS* as its second argument, and the single-threaded object *STATE* as its third argument and it might return an ordinary object and a modified *STATE*. Such a signature is written `((func * MS STATE) ⇒ (mv * STATE))`. The syntax checker insures that when `func` is called its last two arguments are the proper single-threaded objects and that its vector of two results is either returned immediately or is `mv-let`-bound to a vector of two variables, the second of which is *STATE*.

The hardest part of the syntax checking is inducing the signature of a newly defined function. The input signature is obvious from the formal parameters[2] The output signature can be determined by examining some output branch, where the function returns an explicit formal parameter (or vector of values) or calls a subroutine whose output signature is known. The support for recursive and mutually-recursive definitions, however, complicates this process, as it may be necessary to enforce the restrictions on recursive calls before the type of the output has been determined.

The ACL2 read-eval-print loop does not allow the use of any global variable except single-threaded objects. For example, while `(car '(a b c))` is allowed, `(car` *x*`)` is not, because in our applicative setting there is no binding environment assigning a value to *x*. We make an exception for single-threaded object names. If *MS*, for example, is a single-threaded object, and `fn` is a function which expects *MS* as its only argument, then `(fn MS)` is a legal top-level form. If the signature of `fn` indicates that `(fn MS)` returns an updated copy of *MS*, then that value becomes the new "current" *MS* after the evaluation. According to our rules above, `fn` must return an updated *MS* if `fn` (or any of its subfunctions) updates *MS*. We illustrate these restrictions in the next section.

What makes ACL2 different from other functional languages supporting such operations (e.g., Haskell's "monads" [26] and Clean's "uniqueness type system" [1]) is that ACL2 also implements an explicit axiomatic semantics so that theorems can be proved about them. In particular, the syntactic restrictions noted

[2]Actually, for reasons explained later, we require the user to declare explicitly that a given formal is being used as a single-threaded object rather than as an ordinary object.

above are enforced only when single-threaded objects are used in function definitions (which might be executed outside of the ACL2 read-eval-print loop in Common Lisp). The accessor and update functions for single-threaded objects may be used without restriction in formulas to be proved. Since function evaluation is sometimes necessary during proofs, ACL2 must be able to evaluate these functions on logical constants representing the object, even when the constant is not "the current object." Thus, ACL2 supports both the efficient von Neumann semantics and the clean applicative semantics, and uses the first in contexts where execution speed is paramount and the second during proofs.

4 An Example

We describe our implementation of single-threaded objects largely by example. For simplicity, we do not model a microprocessor state here, but rather a much simpler structure containing a "pointer" and a small memory. The following command to the ACL2 read-eval-print loop defines a new single-threaded object named *MS*. The name "**defstobj**" comes from the phrase "define single-threaded object."

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

This constructs a single-threaded object named *MS* with two fields. The first, named **ptr**, contains a positive integer and is initially 0. The second, named **mem**, is a list of five arbitrary (i.e., of Common Lisp type **t**) objects, indexed sequentially from 0 through 4. Initially **mem** contains five occurrences of **nil**.

Logically speaking, the top-level global value of the variable symbol *MS* is now **(0 (nil nil nil nil nil))**.[3]

The **defstobj** command above introduces several function definitions. These definitions extend the logic to include the corresponding axioms and they extend the underlying Common Lisp to include the completed versions of these axiomatic definitions. The completed definitions, recall, are used by ACL2 when it must apply a logically defined function outside of its guarded domain. After making these extensions, **defstobj** introduces raw definitions for the functions. These definitions are destructive and will be discussed after we have clearly described the intended semantics.

The axiomatic definition of the "recognizer" for *MS* objects is

```
(defun msp (MS)
  (declare (xargs :guard t))
```

[3] If the ACL2 user were to print the value of the variable *MS*, the result is displayed as **<ms>**. Single-threaded objects are generally so large that it is counterproductive to display their values and yet by the nature of our syntactic conventions it is necessary that functions return such values.

```
  (and (true-listp MS)
       (= (length MS) 2)
       (ptrp (nth 0 MS))
       (memp (nth 1 MS))))
```

The sub-functions `ptrp` and `memp` are defined as informally sketched above, to check that their arguments are, respectively, a positive integer and a list of five objects.

The accessor and updater for the `ptr` field are

```
(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))
```

Note that the guard ensures that we do not run the raw code (shown later) unless *MS* satisfies `msp` and, in the case of the updater, *v* is a positive integer.

We do not provide the user with an accessor or updater for the `mem` field. Instead, we provide an accessor and updater for the elements of that field. This allows us to implement the contents of the field itself as a (non-applicative) Common Lisp array without exposing that implementation decision. The two functions provided are

```
(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))
```

Note that these names have an "`i`" suffix to remind the reader that they access and update elements of the `mem` component of *MS*, not the component itself.

So much for the axiomatic semantics of the new functions.

The initial value of the *MS* object is not `(0 (nil nil nil nil nil))` but a Common Lisp object outside the applicative domain of the ACL2 logic by virtue of the use of destructively modified arrays. The initial value is `(#(0) #(NIL NIL NIL NIL NIL))`. The hash marks are Common Lisp's notation for arrays. The two arrays serve two purposes. They will be destructively modified to update the current *MS* object and they sometimes allow us to avoid "boxing," the allocation of additional storage to represent runtime type tags. A pointer to this initial value is stored in the Lisp constant symbol named `*the-live-ms*`, which is not directly accessible to the ACL2 user. However, when the user evaluates a top-level ACL2 form containing the global variable *MS*, that variable is given the value of `*the-live-ms*`.

`Defstobj` introduces efficient raw definitions for these functions. We show below the raw definitions for `memi` and `update-memi`.[4] This is legal Common Lisp, but not within the applicative ACL2 subset.

```
(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))))
```

Observe that when `memi` is applied to the "live" instance of *MS* it does an array access, `aref`, to get the appropriate element. When `update-memi` is used to update the `mem` field of the live instance of *MS*, it does so destructively. The clause dealing with "wormholes" has to do with an interactive environment in which ACL2 does not allow single-threaded objects to be altered and is not germane here. When the functions are applied to values that are not the "live" one, they behave as per the axiomatic definitions.[5]

[4] The definitions introduced actually contain heavy use of Common Lisp `declare` and `the` forms so the compiler will produce more efficient code.

[5] These functions may be applied to "non-live" values by the theorem prover itself. In the course of proving a theorem about (`update-memi` *i v MS*) it is possible that the three variables get instantiated to constants and ACL2 will run the definition of `update-memi` to reduce the expression to a constant. If the particular values satisfy the guard on `update-memi`, the raw definition is run, even though the particular value of *MS* may not be the live one.

Finally, `defstobj` gives these functions signatures that indicate that they traffic in single-threaded objects. For example, the signature of `memi` is `((memi * MS) ⇒ *)` and that of `update-memi` is `((update-memi * * MS) ⇒ MS)`. Thus, both functions *must* be passed the current *MS* (in the appropriate argument position) when they are called. Furthermore, the output of `update-memi` *must* be either returned or bound to the `let` variable named *MS*.

Suppose that the `defstobj` above has just been admitted (i.e., evaluated successfully). Here is a sequence of interactions with the read-eval-print loop. The "`ACL2 !>`" is the ACL2 prompt.

```
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
```

The live version of *MS* is printed simply as `<ms>`. Case and font are irrelevant here.

The following theorem is proved immediately (provided **nth-update-nth** has been proved).

```
(defthm memi-update-memi
  (equal (memi i (update-memi j v x))
         (if (equal (nfix i) (nfix j)) v (memi i x))))
```

There are several noteworthy points about this theorem. First, it uses the variable x where the single-threaded object *MS* might have been expected. Our syntactic restrictions apply only to functions to be executed in Common Lisp, not to logical formulas to be proved as theorems. It is often necessary to break the single-threaded rules simply to state the desired properties of functions that manipulate these objects. For example, one might wish to pose a conjecture that relates components of the state before and after a change. Secondly, the theorem has no hypotheses restricting its use to `(msp x)` or legal i, j, and v. Those restrictions are reflected in the guards to the functions, not their logical meanings. The upshot of this is that powerful general theorems can often be proved — theorems without hypotheses which may encumber their subsequent use by the automatic theorem prover. However, to use the single-threaded

objects in the most efficient way – i.e., to gain access to the raw code produced for them – they must be applied to their intended domains. When functions are defined in terms of **memi**, **update-memi**, and the other single-threaded primitives here, those functions should be proved to be Common Lisp compliant to gain maximal efficiency.

5 Using Single-Threaded Objects

To illustrate the use of single-threaded objects, we use the *MS* object to implement a ring buffer. We wish to define **(insert** *x MS***)** so that it writes *x* to the **mem** location indicated by **ptr** and increments the pointer. Logically speaking we mean

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

where **inc** increments its argument modulo 5. However, this violates our syntactic rules because the output of **update-memi** is not immediately **let**-bound to the variable *MS*. In addition, we have found it desirable to require the user to declare explicitly the intention to use single-threaded objects. (Otherwise, the raw definition produced by an acceptable **defun** would be dependent on whether its formals had been defined to be single-threaded objects. This would open the user to the possibility that the inclusion of another user's library into a session would change effect of legal definitions.) The following definition of **insert** is legal in our system, provided *MS* has been introduced as above.

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

Logically, this definition is provably equivalent to the earlier one, but, we think, makes the sequencing more explicit. Finally, the user may define define a macro to produce a nest of **let**-bindings of the variable *MS*. We call that macro "**sequentially**" below. With such a macro we could write the **defun** above as

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

Suppose then that we have the initial *MS* (with **ptr** 0 and **mem** consisting of five **nil**s). If we execute the following

```
(sequentially
 (insert 'A MS)
 (insert 'B MS)
 (insert 'C MS)
 (insert 'D MS))
```

then the logical value of *MS* is `(4 (A B C D NIL))`. If we then do `(insert 'E` *MS*`)` the logical value is `(0 (A B C D E))`. Finally, if we do `(insert 'F` *MS*`)`, the logical value `(1 (F B C D E))`. However, if we ask ACL2 to print the value of *MS* the result is always the same `<ms>`.

It is convenient to define a function to display that part of the object in which we are interested. In the case of *MS*, we define `(show` *MS*`)` so that it returns the five elements in the ring buffer, starting with the oldest. Thus, for the state of *MS* shown above, `(show` *MS*`)` is `(B C D E F)`. So far we have only executed `insert`. What theorems can we prove about it?

We have proved

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

Again, note the relatively weak hypothesis, which makes this lemma easier to apply in the future. We do not need to know that *MS* is a well-formed ring buffer, only that its pointer is less than five. [6] This lemma is proved by reasoning about `nth` and `update-nth`.

Now suppose we wish to scan a binary tree and keep track of the last five tips seen. We can write this as follows:

```
(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))))`, i.e., a binary tree whose fringe consists of the nine symbols `A`, `B`, `C`, ..., `I`. Then `(show (scan` τ *MS*`))` is `(E F G H I)`. No new storage is allocated to compute `(scan` τ *MS*`)`.

We can prove the following theorem,

[6] The equality by itself is not a theorem. If `(ptr` *MS*`)` exceeds four, then the `insert` on the left-hand side inserts x beyond the end of memory and then `show` on the left-hand side collects the first five elements of the buffer, ignoring the x altogether. Meanwhile, the `(show` *MS*`)` on the right-hand side collects the `nil` beyond the end of memory and then the first four elements of the buffer. The `nil` is `cdr`'d off and the final list of length five contains the first four elements of the buffer, followed by x.

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

where `(lastn` i x`)` returns the last n elements of list x and

```
(defun fringe (x)
  (if (consp x)
      (append (fringe (car x))
              (fringe (cdr x)))
    (list x)))
```

Note that this theorem "abuses" our syntactic restrictions on the use of *MS* in a completely unavoidable way. It relates the result of "showing" the object after a `scan` with the result of showing it before the `scan`. Such constructions must be legal if we are to use the language to specify our intentions. The theorem above is proved by reasoning inductively about the tree structure of x. The proof requires knowledge of the properties of `lastn`, `append`, etc., but no new knowledge about the primitives for our single-threaded object *MS*.

Obviously, if the fringe of x contains five or more elements, the initial contents of *MS* is irrelevant. That is, an easy corollary of the above, derived via a lemma about `lastn` and `append`, is

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

While these theorems are not fundamentally deep, we offer them to illuminate the claim that we can reason about "destructive" functions such as `scan` without much trouble thanks to the observation that their syntactic nature allows applicative semantics but von Neumann implementations.

6 Conclusion

Of course, single-threaded objects have not added anything to the expressive power of ACL2; since they are axiomatized entirely in terms of the predefined functions `nth` and `update-nth`, we cannot use them to model computing systems previously beyond our grasp. But the new models execute much faster. For example, researchers at Rockwell-Collins, Inc., have used ACL2 to model a microprocessor, both with and without using single-threaded objects. The models are logically identical. We can compare the speeds of these two models when executing test programs for the modeled microprocessor. It is convenient to measure speeds in simulated microprocessor instructions per second. The

model that does not use single-threaded objects executes about 278 instructions/second. The model with single-threaded objects executes at about 2,326 instructions/second. These tests are extremely sensitive to the program being run since the execution time of the first model is heavily dependent upon the memory addresses at which writes are being done. Furthermore, if the test is arranged so that all the integers created are Common Lisp fixnums, the first model still executes at about 278 instructions/second while the second improves to over 75,000 instructions/second because no "boxes" are created for the data. These tests were performed on a Sun Ultra 2 (160 MHz) running the beta release of ACL2 Version 2.4 under Gnu Common Lisp.

It should be noted that the use of single-threaded objects explicitly sequentializes (some of the updates in) any function using them. This reduces the opportunities to introduce parallelism, which is one of the potential payoffs of an applicative language. For this reason, we have sometimes used the name "von Neumann bottlenecks" instead of "single-threaded objects." However, until we realize the potential parallelism in ACL2, we feel that single-threaded objects are very useful.

Our single-threaded objects are similar in spirit to Haskell's "monads." A monad is a type constructor together with two operations that correspond, roughly, to the notions of "update" and "sequentially." Like our single-threaded objects, monads are state-holding objects that are understood applicatively but can be implemented destructively because of syntactic (type) checks analogous to those we implement. It is possible in Haskell for a function to temporarily create a monad for the purpose of some computation. The state-holding object "evaporates" when it is no longer referenced. ACL2 does not support such a use of single-threaded objects. Indeed, every single-threaded object must appear as an explicitly named actual to any expression using it. The storage for the object is allocated once and is never deallocated. Another limitation of our single-threaded objects is that they cannot be nested: it is against our syntactic rules for such an object to be a component of any object, including another single-threaded one. It is perhaps possible to find syntactic restrictions under which such hierarchies of objects may be safely used.

The paper [27] relates monads to other popular alternative approaches, including synchronized streams [25], continuations [19], linear logic [8], and side-effects. Our approach shares a lot with linear logic, but we do not regard the provision of single-threaded objects as having produced a new logic. Indeed, the situation is exactly the opposite: if one regards this paper having "added" single-threaded objects to our existing logic, we must stress that we did not alter the logic in any way. Our conventions are merely syntactic restrictions on the executable subset of a conventional first-order, quantifier free logic of recursive functions. We exploit the fact that the syntax of the logic is unchanged so that we can state theorems about the effects of updates.

In general, it is our opinion that our single-threaded objects are less expressive than monads and the alternative mechanisms but have the winning

attributes of being very simple, very efficient, and sufficient for our purposes.

As supporting evidence we cite the ACL2 system itself. ACL2 is largely coded in the ACL2 applicative programming language. The system constructed by Kaufmann and Moore consists of 5.5 megabytes of compiled code. Roughly 400,000 bytes is compiled non-applicative Common Lisp code implementing the single-threaded applicative state containing property lists, clocks, streams and files. Thus, we think of 7% of the code as being the non-applicative implementation of the primitives and the remaining 93%, or 5 megabytes, being pure applicative code. The applicative code includes all of the theorem prover, including its rule-based simplifier, various decision procedures including a bdd package and a linear arithmetic procedure, the induction mechanism and all the other heuristics and proof techniques. Many parts of the theorem prover access and update the applicatively formalized property list "world" containing tens of thousands of properties when the system is in its initial state and before it has loaded user-supplied "books" of previously proved theorems, which may add tens of thousands of additional properties. The applicative code also includes the error checkers and error handlers, all input/output including the generation of natural language proof descriptions, the syntax checking and macro expansion, and the read-eval-print loop. The theorem prover is not a toy; it is used to do industrial-scale verification projects. This is powerful evidence that our single-threaded **`state`** provides adequate expressiveness, convenience and efficiency for practical application.

The main contribution of this work is that we have connected an efficient implementation of state holding objects with an applicative programming language while preserving our ability to reason formally and mechanically about the functions. The "limitation" that single-threaded objects are explicitly named in expressions using them is useful in this setting because it allows us to state hypotheses about their current configuration, e.g., that **`(ptr`** *MS*`)` is less than five. These hypotheses may in fact be invariants that could be proved of the object; but more often in our work they are true restrictions on the space of possible states of the object – restrictions which allow one to address the situations of interest. For example, in our microprocessor work, a theorem might have a hypothesis that restricts the theorem to those states in which a given microcode program occupies a certain region of memory; the theorem might conclude with a relation between the initial state and final state of a **`run`**. That is, the theorem characterizes the correctness (or some other property) of the microcode program in question. Other memory configurations are possible under the model and are indeed studied with theorems about other programs. These theorems can then be combined to prove facts about systems of programs. See [5, 17] for some simple examples of how this is done and citations of applications of industrial interest. In [27] the question is raised, in regard to linear logic, whether "mentioning state explicitly" is a "pain" or a "boon;" [27] says additional experience is necessary to determine the answer. Our experience is that it is a boon when one wishes to state theorems about one's functions and

compose those theorems.

Our connection of an efficient implementation with a proof engine also exposes the need for an applicative programming language to support not just the execution of such functions on the "live" object but to support execution on "non-live" instances of the object as well. This is necessary since applications of the functions arise in proofs and must be calculated. Indeed, the syntactic restrictions on the use of single-threaded objects must be lifted when one is writing formulas for the purpose of expressing specifications.

7 Acknowledgments

We would like to thank the researchers at Rockwell-Collins, Inc., who brought to our attention the need to implement user-defined single-threaded objects, in particular John Cowles, David Greve, David Hardin and Matt Wilding. It is unlikely we would have done this without their convincing demonstrations of its utility in modeling realistic processors and we are very grateful for their efforts.

References

[1] E. Barendsen and S. Smetsers, Uniqueness typing for functional languages with graph rewriting semantics, *Mathematical Structures in Computer Science 6*, pp. 579–612, 1996.

[2] R. S. Boyer and J S. Moore, *A Computational Logic*. Academic Press: New York, 1979.

[3] R. S. Boyer and J S. Moore, *A Computational Logic Handbook, Second Edition*, Academic Press: London, 1997.

[4] R. S. Boyer, M. Kaufmann, and J S. Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement, *Computers and Mathematics with Applications*, **5**(2) (1995) 27–62.

[5] R. S. Boyer and J S. Moore. Mechanized Formal Reasoning about Programs and Computing Machines. In R. Veroff (ed.), *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos,* MIT Press, 1996.

[6] B. Brock, M. Kaufmann and J S. Moore, "ACL2 Theorems about Commercial Microprocessors," in M. Srivas and A. Camilleri (eds.) *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, pp. 275–293, 1996.

[7] B. Brock and J S. Moore, "A Mechanically Checked Proof of a Comparator Sort Algorithm" URL: http://www.cs.utexas.edu/users/-moore/publications/csort/main.ps.Z [submitted for publication] 1999.

[8] J.-Y. Girard, Linear logic, *Theoretical Computer Science*, Volume 50, pp. 1–102, 1987.

[9] D. A. Greve, Symbolic Simulation of the JEM1 Microprocessor, in G. Gopalakrishnan and P. Windley (eds.) *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, Springer-Verlag LNCS 1522, November, 1998.

[10] D. A. Greve, D. S. Hardin and M. M. Wilding, Efficient Simulation Using a Simple Formal Processor Model, Technical Report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, April, 1998.

[11] D. A. Greve and M. M. Wilding, Stack-based Java a back-to-future step, Electronic Engineering Times, Jan. 12, 1998, pp. 92.

[12] D. S. Hardin, M. M. Wilding, and D. A. Greve, Transforming the Theorem Prover into a Digital Design Tool: From Concept Car to Off-Road Vehicle, in A. J. Hu and M. Y. Vardi (eds.) *Computed Aided Verification: 10th International Conference, CAV '98*, Springer-Verlag LNCS 1427, pp. 39–44, 1998.

[13] P. Hudak, Continuation-based mutable abstract data types, or how to have your state and munge it too. Technical Report YaleU/DCS/RR-914, Department of Computer Science, Yale University, July, 1992.

[14] M. Kaufmann and J S. Moore, "Structured Theory Development for a Mechanized Logic", URL http://www.cs.utexas.edu/users/moore/-publications/encap-story.ps.Z [submitted for publication] 1999.

[15] M. Kaufmann and J S. Moore, "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp,"*IEEE Transactions on Software Engineering*, **23**(4), pp. 203–213, April, 1997.

[16] M. Kaufmann and J S. Moore, "A Precise Description of the ACL2 Logic," http://www.cs.utexas.edu/users/moore/publications/-km97a.ps.Z, April, 1998.

[17] J S. Moore, "Proving Theorems about Java-like Byte Code," in E.-R. Olderog and B. Steffen (eds.) *Correct System Design – Issues, Methods and Perspectives*, LNCS (to appear, 1999).

[18] J S. Moore, T. Lynch, and M. Kaufmann, A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Program. *IEEE Trans. Comp.* **47**(9), Sept. 1998, pp. 913–926. See also URL http://-devil.ece.utexas.edu/~lynch/divide/divide.html.

[19] G. Plotkin, Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, Volume 1, pp. 125–159, 1975.

[20] D. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD5$_K$86 Floating-Point Square Root Microcode," *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.

[21] D. M. Russinoff, A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithms of the AMD-K7^{TM} Processor URL http://www.onr.com/-user/russ/david/k7-div-sqrt.html.

[22] J. Sawada and W. Hunt, Jr., Processor Verification with Precise Exceptions and Speculative Execution, in A. J. Hu and M. Y. Vardi (eds.) *Computed Aided Verification: 10th International Conference, CAV '98*, Springer-Verlag LNCS 1427, 1998.

[23] D. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. Prog. Lang*, **7**, pp. 299–310, 1985.

[24] G. L. Steele, Jr, *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[25] W. Stoye, Message-based functional operating systems. *Science of Computer Programming*, **6**(3) pp. 291–311, 1986.

[26] P. Wadler, Monads for functional programming, in J. Jeuring and E. Meijer (eds.), *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.

[27] P. Wadler, How to declare an imperative, *ACM Computing Surveys* **29**(3), pp. 240–263, September, 1997.