# Proving Theorems about Java and the JVM with ACL2

J STROTHER MOORE
*Department of Computer Sciences*,
University of Texas at Austin,
Taylor Hall 2.124,
Austin, Texas 78712

**Abstract.** We describe a methodology for proving theorems mechanically about Java methods. The theorem prover used is the ACL2 system, an industrial-strength version of the Boyer-Moore theorem prover. An operational semantics for a substantial subset of the Java Virtual Machine (JVM) has been defined in ACL2. Theorems are proved about Java methods and classes by compiling them with `javac` and then proving the corresponding theorem about the JVM. Certain automatically applied strategies are implemented with rewrite rules (and other proof-guiding pragmas) in ACL2 "books" to control the theorem prover when operating on problems involving the JVM model.

## 1 Background

The Java Virtual Machine or JVM [27] is the basic abstraction Java [17] implementors are expected to respect. We speculate that the JVM is an appropriate level of abstraction at which to model Java programs with the intention of mechanically verifying their properties. The most complex features of the Java subset we handle – construction and initialization of new objects, synchronization, thread management, and virtual method invocation – are all supported directly and with full abstraction as single atomic instructions in the JVM. The complexity of verifying JVM bytecode program stems from the complexity of Java's semantics, not generally from implementation details in the JVM.

We model the JVM operationally. That is, we define an interpreter for JVM programs within a formal logic. At the moment, we model only a subset of the JVM. Our subset does not include dynamic class loading or exceptions. Our thread model assumes sequential consistency and atomicity at the bytecode level. These simplifications are not indicative of the limits of our modeling or proof technology, only of the amount of time we have invested in the model and (in the case of threads) uncertainty in the Java community as to the desired standard.

We model the JVM in ACL2, A Computational Logic for Applicative Common Lisp, by Matt Kaufmann and J Moore. ACL2 [25] is the industrial-strength successor to the Boyer-Moore theorem prover, Nqthm [6]. 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 Nqthm. Our JVM model in ACL2 may be thought of as a functional Lisp implementation of (a subset of) the JVM. It is *executable* in the sense that given a specific class file and thread schedule, the model can be run in Lisp to determine the state of the JVM at the end of the schedule.

Theorems can be proved about the execution of a Java method by translating the method JVM bytecode class file using a standard tool such as `javac` of Sun Microsystems, Inc., and then proving a theorem about the JVM model's behavior when interpreting that bytecode. The practicality of this approach to code verification was first demonstrated for a commercially interesting programming language by Yuan Yu [7]. He verified 21 of the 22 Berkeley C String Library programs by translating them into machine code for the Motorola 68020, using `gcc -o`, and verifying the resulting binary images with respect to an operational semantics for the 68020. Nqthm was used for the proofs.

We find this approach to code verification appealing for two reasons. First, modeling machines at the instruction-set architecture (ISA) level can generally be done with more confidence than modeling modern programming languages. This confidence stems in part from the simplicity of the typical ISA and in part from the precision and completeness of the informal ISA specifications provided by the manufacturers. Of course, ISAs and their specifications are not simple, precise or complete; but they exhibit those qualities to a higher degree than commercially supported programming languages. The second reason we find this approach appealing is that the practitioner need not assume the correctness of the compiler, be it `gcc` or `javac`, since the code verified is the code executed.

The main drawback of this approach is that the resulting proofs may be complicated by implementation details. While this phenomenon is real, it can be difficult to measure in the absence of any comparably precise and detailed formal model of the high level language. In many languages, side-effects, aliasing, pointer manipulation, order-of-evaluation and other "implementation details" are in fact standard features of the high-level semantics and are ignored at the programmer's peril.

While our approach to verifying Java suffers some unnecessary complexity due to implementation details in translation to the JVM, the JVM is a well-designed abstract ISA for Java and introduces few such details. A glaring omission is the absence of structured iteration in the JVM; but this omission can be remedied by inspection of the Java source.

The outline of our presentation is as follows.

In Section 2 we introduce ACL2 as a logic. In Section 3 we introduce the mechanized theorem prover for it. In Section 4 we describe our operational semantics of the JVM. In Section 5 we show a Java factorial method and show the formalization of its bytecode in detail. Also in that section we illustrate the use of the model as a JVM simulator. In Section 6 we describe how we configure ACL2 to reason about bytecode programs and in Section 7 we illustrate the method by proving the factorial method correct. In Section 8 we explain how we deal with more complicated examples, including virtual methods, the heap, and object manipulation. In this section the main example is an applicative linked-list insertion sort routine coded in Java. We prove that the method returns an ordered permutation of its input, in a suitable sense.

In Section 9 we describe the proof of a safety property of a multi-threaded Java system. We deal with a class that spawns an unbounded number of threads, each in contention for a single object in the heap, and we prove that mutual exclusion is achieved using the JVM's monitors.

In Section 10 we very briefly note related, ongoing developments in the ACL2 community. We comment on other related work throughout the document. We conclude in Section 11.

We assume the reader is somewhat familiar with Java and comfortable with formal mathematical logic. Readers should see the author's home page, `http://www.cs.utexas.edu/users/moore` for many of the publications cited here. In addition, the ACL2 home page is linked to the author's page. The ACL2 files containing our definition of the JVM

model and the proofs discussed here will also be posted under the author's Publications link.

## 2 ACL2

To a first approximation, ACL2 is the largest applicative subset of Common Lisp; it is described as a mathematical logic with a formal syntax, some axioms, some rules of inference, and a semantic model. We also provide a mechanized theorem proving system for the logic. A description of ACL2 – the programming language, the logic, and the theorem proving system – is provided in [25].

The ACL2 system is distributed without fee under the GNU General Public License from the ACL2 home page, `http://www.cs.utexas.edu/users/moore/acl2`. As of this writing, the latest release is Version 2.6. The source code, many files of definitions and lemmas, and online documentation are available under the same terms. In addition, the documentation may be searched and browsed on the web. See the link from the home page.

To make this paper more self-contained, we briefly discuss ACL2. But a precise description is beyond the scope of this paper. We refer the student to [25] and to the ACL2 system.

ACL2 is a quantifier-free first-order logic with induction and recursive definition.

### *2.1 The Semantic Domain*

The *ACL2 objects* are partitioned into five sets.

- numbers, including integers such as `123` and `-5`, non-integral rationals such as `22/7`, and complex rationals such as `#c(1,5)` ($1 + 5i$). We only use integers in this paper.
- characters, including `#\A` (uppercase "A") `#\a` (lowercase "A"), and `#\Space` (a space). We do not use character data in this paper, except as the constituents of other data objects such as strings and symbols.
- strings, including `"monitor"` and `"java.lang.Object"`.
- symbols, including `t`, `nil`, `x`, `monitor`, `top-frame`, `+`, and `>=`. Typically, symbols consist of finite sequences of alphabetic characters (`A`–`Z`), digits (`0`–`9`), and certain signs including `+`, `-`, `<`, and `=`, that are not conventionally interpreted as numbers. For example `+A1` is a symbol but `+81` is a number. Signs do not terminate symbols, e.g., `top-frame` is a symbol, not two symbols, `top` and `frame`, separated by a minus sign.
- ordered pairs of ACL2 objects, including `("monitor" . 0)` and `(1 2 3)`. The first of these is the ordered pair containing the string `"monitor"` as its left item and the integer `0` as its right. The second of these is an ordered pair containing the integer `1` as its left item and the ordered pair `(2 3)` as its right. The pair `(2 3)` contains the integer `2` as its left item and the pair `(3)` as its right. The pair `(3)` contains the integer 3 as its left item and the symbol `nil` as its right. This pair could also be written `(3 . nil)`. Indeed, `(1 2 3)` could be written `(1 . (2 . (3 . nil)))`. Ordered pairs are sometimes called "conses." The right item in an ordered pair is said to be the "car" and the left item is said to be the "cdr."

Note that these sets are disjoint. In particular, strings are different from symbols, e.g., `monitor` is a different object than `"monitor"`.

A *true-list* or *linear list* is either `nil` or an ordered pair whose right item is a true-list.

In this paper, case is unimportant when symbols are written, e.g., `Monitor`, `monitor`, and `MONITOR` are the same symbol.

Lisp is unusual in that the objects in the domain above are used to represent the terms and formulas of the language. The data objects we call symbols are also the symbols of the syntax, they are our function and variable names.

Symbols are structured objects, containing a "package name" and a "symbol name." The "full name" of a symbol is obtained by writing the package name and the symbol name, separated by two colons. But there is a convention that allows the package name to be omitted when it is the same as some previously selected "current package." In this paper, we write as though the current package were `"JVM"`. Thus, `JVM::top-frame` is the "full name" of the symbol whose "short name" (in this context) is `top-frame`.

Packages allow authors to have disjoint name spaces. For example, `ALICE::swap` is a different symbol than `BOB::swap` so Alice can define her symbol `swap` one way and Bob can define his another. By selecting `"ALICE"` as her current package, Alice can refer to her symbol `swap` by writing `swap`. To refer to Bob's, she must write `BOB::swap`.

But authors must frequently refer to symbols from other packages, especially the `"LISP"` package where all the primitive symbols are defined. That is, Alice will frequently wish to refer to the symbol `LISP::car`. It is inconvenient to have to write the package name of commonly used symbols. So symbols from one package can be "imported" into another. By importing `LISP::car` the `"ALICE"` package, Alice can refer to it simply as `car` while in the `ALICE` package.

So the notation for symbols is always relative both to a selected package and to the entire "package structure" – an acyclic directed graph specifying the imports of symbols between packages. This package structure is fixed for any use of ACL2. In addition to the `"LISP"` package, every package structure includes the `"KEYWORD"` package. Symbols such as `KEYWORD::pc` and `KEYWORD::stack` may be written more simply as `:pc` and `:stack` and are called *keywords*.

For this work we created a `"JVM"` package and we imported into it most of the commonly used `"LISP"` symbols.

Such details are unimportant to the casual reader. But the careful student might otherwise wonder how we can define the function `step`. Why? The symbol `LISP::step` is pre-defined in Common Lisp and symbols cannot be redefined in ACL2. But we select `"JVM"` as the current package and we do not import `LISP::step` into it, so when we define `step` we are actually defining `JVM::step`, which is initially undefined.

We take for granted the usual mathematical operations and relations on the mathematical objects we have been discussing. For example, just as we expect the reader to understand what we mean when we talk about about the sum or product of two numbers or whether two objects are equal, we also expect the reader to understand what we mean by the car and cdr of an ordered pair or whether an object is a string or a symbol. A precise semantics for the ACL2 logic can be described in terms of these mathematical operations and relations. What is perhaps surprising is that we use these mathematical notions to describe the syntax too.

### *2.2 Syntax*

In the technical treatment of ACL2's syntax, we define the notion of a "formal term." The formal terms are a subset of domain of objects above. The object $\tau$ is a *formal term* iff $\tau$ is (a) a variable symbol, (b) one of a small number of primitive constants, or (c) a true list of $n+1$ elements whose first element is a "function expression" of arity $n$ and whose remaining elements are formal terms. A *function expression* is either a function symbol or a lambda expression of the form `(lambda (`$v_1 \ldots v_n$`) `$\beta$`)`, where the $v_i$ are distinct variable symbols and $\beta$ is a formal term containing no free variables other than the $v_i$. Formal terms of kind (c)

are said to be function or lambda *applications*.

An example formal term is thus `(cons x (cons y nil))`. In more traditional systems this term might be written `cons`($x$, `cons`($y$, `nil`)).

Formal terms are relatively simple, which makes it straightforward to define the rules of inference for manipulating them. But formal terms are too cumbersome for practical use. Among the limitations are that there is no provision for writing arbitrary constants and every function must be supplied a fixed number of arguments.

To remedy this we define a *term* to be any object in the domain that *abbreviates* a formal term. We introduce two basic abbreviation mechanisms, one allowing the use of constant terms and the other allowing the use of macros.

The constant terms are the primitive constants (including `0`, `1`, and `nil`), the symbol `t`, the keywords, the numbers, the characters, the strings, and all objects of the form `(quote` $c$`)`, where $c$ is any object. Constant terms of the last kind may be written `'`$c$. The constant abbreviation convention specifies for each of these constant terms a formal term abbreviated by the constant. For example, the constant term `3` abbreviates the same formal term abbreviated by `(+ 1 1 1)`. The constant term `'(1 2 3)` abbreviates the same formal term abbreviated by `(cons 1 (cons 2 (cons 3 nil)))`. Essentially, we give a formal construction for each object in the ACL2 domain.

The macro convention is as follows. Associated with certain symbols, called *macro symbols*, are functions on true lists of ACL2 objects. Suppose $m$ is a macro symbol with associated function $f_m$. Then if the object `(`$m\ o_1 \ldots o_n$`)` is used as a term it is understood to abbreviate the term obtained by applying $f_m$ to the $o_i$. For example, `list` is a macro symbol and associated with it is a function that, for example, transforms the linear list `(x y)` into the object `(cons x (cons y nil))`. That is, when given the list of two elements containing the symbols `x` and `y`, the function corresponding to `list` produces a list of three objects whose first element is the symbol `cons`, whose second element is the symbol `x`, and whose third element is a list of three elements, namely `cons`, `y` and `nil`, respectively. That is, `list` is defined as a list processing function on objects in our domain. But by declaring it a macro we can then use `(list x y)` as a term abbreviating `(cons x (cons y nil))`. Macros allow us to abbreviate terms by describing how to compute the terms we mean.

A very commonly used macro is `cond`, which abbreviates a nest of `if`s. For example,

```
(cond ((equal sync-status 'LOCKED)
       (unlock-object th obj-ref (heap s)))
      ((equal sync-status 'S_LOCKED)
       (unlock-object th ret-ref (heap s)))
      (t (heap s))).
```

abbreviates

```
(if (equal status 'LOCKED)
    (unlock-object th ref1 (heap s))
    (if (equal status 'S_LOCKED)
        (unlock-object th ref2 (heap s))
        (heap s))).
```

Observe that the conditions tested in a `cond` expression are tested in the order written and the first true one determines the result. If no condition is true, `cond` returns `nil`. It is conventional for the last condition to be `t`, thereby insuring that the `cond` does not "fall off the end."

With a little license, we can read the `if` and `cond` expressions above as: if `status` is the symbol `LOCKED`, unlock `ref1` in the heap of `s` and return that heap; if `status` is `S_LOCKED`, unlock `ref2` in the heap of `s` and return that heap; otherwise, return the heap of `s`. This reading ignores the role of `th` in unlocking.

A *formula* is either an *atomic formula* of the form $\tau_1 = \tau_2$, where the $\tau_i$ are terms, or else of the form $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\phi_1 \rightarrow \phi_2$, or $\phi_1 \leftrightarrow \phi_2$, where the $\phi_i$ are formulas.

### *2.3 Quantifier-Free First-Order Logic*

We define quantifier-free first-order logic in a traditional way with several axiom schemas and primitive rules of inference. A *formal proof* is a finite tree of formulas, each of which is either an axiom or is derived from its immediate ancestors in the tree by one of the rules of inference. A *theorem* is any formula in a proof, but most especially the root of the tree. We derive a variety of more elaborate rules of inference, including the tautology theorem (every propositional tautology has a proof), proof by cases, and substitution of equals for equals.

### *2.4 Axioms*

The axioms of ACL2 characterize the primitive function symbols and constants. Among the axioms (or easily proved theorems) are the following. We can paraphrase these as saying the symbols `t` and `nil` are distinct, that `equal` is the characteristic function for equality, and `if` is a 3-place if-then-else that tests against `nil`.

**Axioms or Basic Theorems:**

```
t ≠ nil
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
```

"Predicates" in ACL2 are Boolean functions that return `t` or `nil`.

We characterize the "propositional functions," e.g., `not` and `implies`

**Axioms or Basic Theorems:**

```
(not p) = (if p nil t)
(implies p q) = (if p (if q t nil) t)
```

and we define `and` and `or` as macros that abbreviate `if` expressions. Thus `(and p q r)` abbreviates `(if p (if q r nil nil))`.

Among the theorems related to list processing are that `consp` is a Boolean function that recognizes ordered pairs constructed by `cons`, that `cons` constructs a pair with the appropriate `car` and `cdr`, that `car` and `cdr` return `nil` on non-conses, and that `nil` is not an ordered pair.

**Axioms or Basic Theorems:**

```
(consp x) = t ∨ (consp x) = nil
(consp (cons x y)) = t
(car (cons x y)) = x
(cdr (cons x y)) = y
(consp x) = nil → (car x) = nil
(consp x) = nil → (cdr x) = nil
(consp nil) = nil
```

We make the convention that when a term $\tau$ is used as a formula it is an abbreviation for the formula $\tau \neq$ `nil`. We can then prove the metatheorem that allows us to write any formula as an "equivalent" term. For example, using convention we might say that

```
(implies (consp x) (not (symbolp x)))
```

is a theorem, meaning

`(implies (consp x) (not (symbolp x))) ≠ nil`

is a theorem, which is provably equivalent to the axiom
`(consp x) = t → (symbolp x) = nil.`

We tend henceforth to exhibit all of our theorems as terms.

### *2.5 Definitions*

Following Gentzen [15], we embed the ordinals up to $\epsilon_0$ into our universe by defining a function that recognizes when certain lists and numbers represent ordinals. For example, the list `(4 2 2 2 . 7)` represents the ordinal $\omega^4 + \omega^2 \times 3 + 7$. We define well-founded "less than" relation on these ordinals.

We provide a definitional principle that permits us to define new function symbols recursively, provided we can prove that a certain ordinal measure of the arguments decreases in each recursive call.

Here are three definitions.

```
(defun push (obj stack) (cons obj stack))
(defun top (stack) (car stack))
(defun pop (stack) (cdr stack))
```

These definitions the following axioms.

**Definitional Axioms:**

```
(push obj stack) = (cons obj stack)
(top stack) = (car stack)
(pop stack) = (cdr stack)
```

Clearly, we are representing stacks as lists. The top of a stack is its car and the rest of the stack is its cdr. The stack obtained by pushing `4 4` onto the stack `(3 2 1)` might be written `(push 4 '(3 2 1))`. We can prove the theorem:

**Theorem:**

```
(push 4 '(3 2 1)) = '(4 3 2 1)
```

The proof of this theorem illustrates how we can use the axioms to *compute* the values of or *evaluate* certain terms. Note that `push` does not *change* the list `(3 2 1)` into the list `(4 3 2 1)`. In a logical setting, that remark is as unnecessary as saying that adding 4 to 3 does not change 3 to 7.

Some other simple theorems about these stack manipulation functions are:

**Theorems:**

```
(top '(3 2 1)) = 3
(pop '(3 2 1)) = '(2 1)
(top (pop '(3 2 1))) = 2
(pop (pop (pop '(3 2 1)))) = nil
```

We can also prove the following important theorems

**Theorems:**

```
(top (push x stack)) = x
(pop (push x stack)) = stack
```

These proofs are obvious from the definitions of `push`, `top`, and `pop` and the axioms about `cons`, `car`, and `cdr`.

Other axioms permit us to prove that when `top` and `pop` are applied to the empty stack or a non-list, the result is `nil`. If an empty stack or non-list is popped, `nil` is returned.

Here is another definition.

```
(defun popn (n stack)
  (if (zp n)
      stack
      (popn (- n 1) (pop stack))))
```

This one involves recursion and so a measure argument is necessary. The predicate `(zp n)` is true when `n` is `0` or when `n` is not a natural number. Thus, the first argument, `n`, is decreased each time the recursive branch is taken. Once this argument is made, the function definition for `popn` is *admitted* and a new axiom is added.

**Definitional Axiom:**

```
(popn n stack)
  =
(if (zp n)
    stack
    (popn (- n 1) (pop stack)))
```

Clearly, `popn` is the function that pops a stack `n` times. If the stack is insufficiently deep, `nil` is returned.

It is convenient at this point to display a few other recursively defined functions. We choose to display some that are manipulate a very common data structure in our JVM modeling work: association lists. An *association list* or *alist* is a list of pairs, `(`$k$ `.` $v$`)`. Each such pair is said to *bind* its *key*, $k$, to its *binding* or *value* $v$. The association list is treated as a table. The first pair binding a given key in a table specifies the value of that key in the table. For example, `((A . 1) (C . 3) (B . 2))` binds `A` to `1`, `B` to `2`, and `C` to `3`.

To look up a key, we use the function `assoc-equal`, which actually returns the pair binding the given key. Here is the recursive definition.

```
(defun assoc-equal (key alist)
  (cond ((endp alist) nil)
        ((equal key (car (car alist))) (car alist))
        (t (assoc-equal key (cdr alist)))))
```

The function `endp` returns `t` if its argument is `nil` or any other non-cons. Thus, the definition above may be paraphrased as follows: if `alist` is empty, return `nil`; if `key` is the key of the first pair in `alist`, return that pair; else, look for `key` in the cdr of `alist`. It is not uncommon for us to abbreviation `(car (car alist))` as `(caar alist)`; combinations of `car` and `cdr` are often so abbreviated.

The two functions

```
(defun bound? (key alist) (assoc-equal key alist))
(defun binding (key alist) (cdr (assoc-equal key alist)))
```

determine whether `key` is bound in `alist` and, when it is, what its binding is, respectively.

The following function "changes" the binding of `k` to `v` in `alist`. To be more precise, it constructs a new alist in which `k` is bound to `v` and the bindings of all other keys are preserved.

```
(defun bind (k v alist)
  (cond ((endp alist) (list (cons k v)))
        ((equal k (car (car alist)))
         (cons (cons k v) (cdr alist)))
        (t (cons (car alist) (bind k v (cdr alist))))))
```

The function unnecessarily puts the binding for `k` "in the same place" in the alist. For example,

```
(bind 'B
      22
      '((A . 1) (C . 3) (B . 2)))
```

is `((A . 1) (C . 3) (B . 22))`.

### *2.6 Induction*

We provide an induction principle that permits us to prove a conjecture by splitting it into cases and inductively assuming instances of the conjecture that are smaller according to some ordinal measure. For example, to prove `(p x y)` it suffices, by induction, to prove

```
(and (implies (not (consp x)) (p x y))     ; base case
     (implies (and (consp x)               ; induction step
                   (p (cdr x) y))
              (p x y)))
```

Here the ordinal measure is the number of conses in `x`. Our induction principle is quite general and would permit us to also assume, as an induction hypothesis, `(p (car x) y)`, since the number of conses in `(car x)` is less than that in `x` when `x` is a cons. We could also assume `(p (cdr x) (+ y 3))` – indeed, we can assume the theorem for as many instances of `y` as we wish, as long as `x` is replaced by something with fewer conses.

We could choose to induct using a different measure, e.g., the difference between `x` and `y` when x is bigger. The induction would be:

```
(and (implies (not (and (integerp x)       ; base case
                        (integerp y)
                        (< y x)))
              (p x y))
     (implies (and (integerp x)            ; induction step
                   (integerp y)
                   (< y x)
                   (p x (+ y 1)))
              (p x y))).
```

Recall the function `popn`.

```
(defun popn (n stack)
  (if (zp n)
      stack
      (popn (- n 1) (pop stack))))
```

Here is another recursive function. It computes the length of a list (or depth of a stack).

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

`Len` is admissible because the number of conses in `x` decreases each time the recursive branch is taken.

Here is a theorem relating `len` and `popn`.

**Theorem:**

```
(<= (len (popn n stack)) (len stack))
```

This theorem is easily proved by the following induction argument:

```
(and (implies (zp n)
              (<= (len (popn n stack)) (len stack)))
     (implies (and (not (zp n))
```

```
        (<= (len (popn (- n 1) (pop stack)))
            (len (pop stack))))
(<= (len (popn n stack)) (len stack))))
```

Note how the induction is suggested by the recursion in `popn`. The proof of each case follows easily, expanding the definitions of `popn` and `len` and simplifying, using properties of `<=`. In fact, the mechanical theorem prover will select this induction and do the simplification automatically.

## 3 ACL2 as a Mechanical Theorem Prover

### *3.1 Simplistic Models of the System*

We provide a theorem prover for the ACL2 logic. The ACL2 theorem prover is an industrial-strength version of the Boyer-Moore Nqthm prover, adapted to the ACL2 logic. ACL2 is the work of Kaufmann and Moore.

A grossly inaccurate but still useful model of the theorem prover is as follows. To *try to prove* a formula, $\phi$, "simplify" the formula to $\phi'$. If $\phi'$ is `t`, then $\phi$ is a theorem. Otherwise, use the recursive functions occurring in $\phi'$ to suggest an induction scheme producing new subgoals $\phi_1 \ldots \phi_n$. If no scheme is suggested, the proof attempt fails. Otherwise, recursively try to prove each $\phi_i$.

A grossly inaccurate model of *simplification* is that it is just the exhaustive, inside-out rewriting of the formula, with axioms, definitions, and user-supplied but previously proved theorems.

Recall the main axiom about `car`.

```
(car (cons x y)) = x
```

The rewrite rule generated causes the following behavior. When the rewriter encounters a term, $\tau$, that is an instance of the left-hand of the rule under some variable substitution $\sigma$, it replaces $\tau$ by the $\sigma$ instance of the right-hand side. The term $\tau$ is called the *target* of the rule and the rule is said to have *fired* on the target. So for example, the simplifier would transform `(p (car (car (cons (cons a b) c))))` first to `(p (car (cons a b)))` and then to `(p a)` by firing first on the inner `car` expression and then on the outer.

Every time a definition is made or a theorem is proved, rules are generated from the formula. There are about a dozen kinds of rules. The user specifies which kinds of rules to generate from a formula. Each rule is initially *enabled*, meaning it is available for automatic use by the system. But the user can *disable* a rule to prevent it from firing automatically. By default, *rewrite rules* are generated from each definition and theorem. We will limit our discussion here to rewrite rules.

### *3.2 Rewrite Rules*

Recall the definition of `bind`. It is used as a rewrite rule to replace applications of `bind` by the instantiated body of `bind`. We call this *opening* or *expanding* the application of `bind`. For example, `(bind key val (cons (cons a b) c))` can be expanded to

```
(if (equal key a)
    (cons (cons key val) c)
    (cons (cons a b) (bind key val c)))
```

using the definition of `bind` and the axioms about `car`, `cdr`, and `cons`. Note that the result still contains an application of `bind` and that application could be expanded, replacing it with the definition of `bind`. Since `bind` is recursive, this expansion could go on forever; the

system has heuristics to control the use of recursive functions. In some particular proof, the user may not want `bind` expanded. In that case, the user will disable the rule generated from the definition of `bind`.

Here is a theorem about `bind`. The syntax shown below includes the `defthm` command used to pose and name the conjecture. If the equality below is proved, a rewrite rule is generated from it and given the name `bind-bind`.

```
(defthm bind-bind
  (equal (bind x v (bind x w a))
         (bind x v a)))
```

The effect of the rule is to replace every instance of the left-hand side, `(bind x v (bind x w a))`, by the corresponding instance of the right, `(bind x v a)`. The left/right orientation of the rule is that given in the statement of conjecture. That orientation is irrelevant to the proof of the rule – equality is symmetric and the theorem prover "knows" that – but the user must be aware of how the expression of a conjecture affects the rules generated from it.

Before we leave `bind-bind`, it is worth pointing out what it means semantically. We can read the composition of the two bind expressions as chronological sequencing. That is, if the binding of `x` is changed to `w` and then it is changed to `v`, the effect is as though `x` was set to `v`.

Typically, the ACL2 theorem prover fails to simplify the input conjecture $\phi$ to `t` and induction is inappropriate. The user inspects the output of simplification, $\phi'$, and looks for compositions that are "unnecessarily complex," as is the left-hand side of `bind-bind`. The user then proves a theorem like `bind-bind` that shows a simpler form of the offending term and gets the theorem prover to prove it, recursively solving whatever problems arise in the process. When the simplifying theorem is finally proved, the user then tries to prove $\phi$ again, expecting to see the new rule fire and simplify $\phi'$. The iterated invention of new theorems that give rise to rules for proving a given goal is called *The Method* and is the most common way user's interact with ACL2.

`Bind-bind` is an example of an unconditional rewrite rule. ACL2 supports conditional rewriting too. Here is a theorem from which a conditional rule is produced.

```
(defthm binding-append1
  (implies (bound? key a)
           (equal (binding key (append a b))
                  (binding key a))))
```

Here `append` is the function that concatenates two lists. The theorem says that if `key` is bound in `a`, then the binding of `key` in the concatenation of `a` and `b` is just the binding of `key` in `a`. The rewrite rule generated from this theorem causes the following behavior by the rewriter. Suppose the rewriter encounters a potential target term that is a $\sigma$ instance of `(binding key (append a b))`. Then the rewriter *backchains* to the $\sigma$ instance of `(bound? key a)` and tries, recursively, to rewrite it to `t`. If successful, it replaces the target by the $\sigma$ instance of `(binding key a)`. Otherwise, the rule does not fire. A more succinct way to read this rule is: replace `(binding key (append a b))` by `(binding key a)`, provided `(bound? key a)` rewrites to `t`.

There is a dual theorem we could prove:

```
(defthm binding-append2
  (implies (and (alistp a)
                (not (bound? key a)))
           (equal (binding key (append a b))
                  (binding key b))))
```

The rule can be read: replace `(binding key (append a b))` by `(binding key b)`, provided `(alistp a)` and `(not (bound? key a))` rewrite to `t`. The hypothesis that `a` be an `alist`, a true list of pairs, is necessary for this to be a theorem.

Some ACL2 users might prefer the following rule instead of `binding-append1` and `binding-append2`.

```
(defthm binding-append
  (implies (alistp a)
           (equal (binding key (append a b))
                  (if (bound? key a)
                      (binding key a)
                      (binding key b)))))
```

This rule rewrites `(binding key (append a b))` to the indicated `if`-expression, provided `(alistp a)` rewrites to `t`. The `if` expression introduces into the conjecture the question `(bound? key a)`. This `if` will cause a case split, unless the question can be settled by rewriting.

Note that `binding-append` combines the other two `binding`/`append` rules shown. In a way, it is more powerful than either of those rules: it can fire even if the `bound?` question cannot be settled by rewriting. But it is not strictly more powerful because there are circumstances under which `binding-append` will not fire while `binding-append1` would fire: if the `bound?` question rewrites to `t` but `(alistp a)` cannot be rewritten to `t`. That is, the `alistp` hypothesis actually weakens `binding-append` in the case where `binding-append1` fires.

It is up to the user to decide which behavior is desired: that induced by `binding-append` or that induced by the combination of the other two rules. It is possible the user might prove all three rules and switch between the two behaviors by enabling and disabling rules.

It is possible to attach pragmatic advice to a rule. The theorem below is logically equivalent to `binding-append` above because the function `force` is logically defined to be the identity function.

```
(defthm forced-binding-append
  (implies (force (alistp a))
           (equal (binding key (append a b))
                  (if (bound? key a)
                      (binding key a)
                      (binding key b)))))
```

But the rule generated from it will fire even if `(alistp a)` does not rewrite to `t`. When that backchaining fails, `(alistp a)` is *assumed* and the rule is fired anyway! If the main proof succeeds, the theorem prover starts a *forcing round* during which it brings all of its power to bear on proving (the appropriate instance of) `(alistp a)` (under the assumptions governing the occurrence of the target). For example, induction might be used. Users often *force* hypotheses that are *type-like*. If we believe that `binding` is always applied to an alist, then that hypothesis "should" not prevent the rule from firing. If the main proof succeeds but the forcing round fails, it indicates a possible "type" violation.

More arbitrary heuristic advice may be attached to a rule using the function `syntaxp`. `Syntaxp` is logically defined to be the constant function that returns `t`. The theorem below is logically equivalent to `forced-binding-append` and `binding-append`.

```
(defthm forced-binding-append-on-locals
  (implies (and (force (alistp a))
                (syntaxp (equal a '(locals (top-frame s)))))
           (equal (binding key (append a b))
                  (if (bound? key a)
                      (binding key a)
                      (binding key b))))).
```

But it will only fire on targets in which the instance of `a` is literally the term `(locals (top-frame s))`. `Syntaxp` allows the user to use ACL2 as a programming language to

query the context in which the potential target occurs and to there decide whether to fire the rule.

### *3.3 Books*

Collections of definitions and theorems (and the rules they generate) can be assembled into files called *books*. The user can include a book in a session and thereby extend the database with all the rules in that book. It is also possible to include selected rules from a book and to include multiple books.

It is via books that we have implemented a variety of proof techniques for the JVM.

## 4 Our Formal Model of the JVM

We model the JVM operationally. Formally modeling computing machines operationally has a long tradition. McCarthy [31] said "The meaning of a program is defined by its effect on the state vector."

The first mechanized formal model of the JVM was Cohen's "defensive JVM" [10] in ACL2. Cohen's machine includes type tags on all data objects so that type errors can be detected and signaled at runtime. It was designed for use in verifying the bytecode verifier. Cohen's machine does not include as many bytecodes as the model described here, nor does it include threads.

The model described here is the fifth machine in a series of models developed, with Cohen's help, so that the author could teach an undergraduate course at the University of Texas at Austin on modeling the JVM in ACL2. The series starts with a simple machine for executing straight-line stack-based integer code. Successively more complicated machine models are then introduced, adding new control and primitive data instructions, a subroutine call, a heap with instance objects, method resolution and invocation, and threads. The machine described here is named M5 and is the last machine in the series above.

A sequential predecessor of M5, named M3, is discussed in [33], including how we use ACL2 to prove theorems about sequential bytecode programs. (M4 was a multi-threaded version that supported only a few bytecodes.) That paper briefly presents some of the ideas discussed here at greater length.

M5 is an incomplete and inaccurate model of the JVM. M5 omits support for syntactic typing (including the role of method signatures in method resolution), field and method access modes, class loading and initialization, exception handling, and errors. In addition, our semantics for threading is interleaved bytecode operations (and thus assumes sequential consistency). We are working on elaborations of M5 to remedy these omissions.

To describe our JVM model, we start by describing the representation of JVM states in ACL2. The state is an ACL2 object. In this paper, we capitalize the word "object" when referring to a Java instance Object and we use lowercase when referring to an ACL2 object. Thus, every Java Object is represented by an ACL2 object, in fact, a list. Every Java item of primitive type, e.g., each `char` and `int`, is also represented by an ACL2 object.

After presenting our representation of states, we discuss our models of several important bytecode instructions. M5 includes models of 138 instructions. Each is modeled by a "semantic function" that describes the state transition induced by executing a given bytecode instruction. We then explain how we deal with thread scheduling and the execution model. We conclude this section with a brief summary of related modeling work. The full M5 model can be found at `http://www.cs.utexas.edu/users/moore/publications/m5`.

A state is a triple, containing a thread table, a heap, and a class table. The accessors for the components of a state are

- `thread-table` – a map from thread identifiers to threads.
- `heap` – a map from heap addresses to instance objects.
- `class-table` – a map from class names to class declarations.

All of the maps in question are represented as alists. We discuss each component in turn in the following subsections.

The functions for constructing and accessing the components of states are shown below. The function `nth` fetches the element at a given position in a list.

```
(defun make-state (thread-table heap class-table)
  (list thread-table heap class-table))
(defun thread-table (s) (nth 0 s))
(defun heap         (s) (nth 1 s))
(defun class-table  (s) (nth 2 s))
```

We define these four functions for reasons of abstraction and convenience. We could simply write `cons`, `car`, and `cdr` expressions.

### 4.1.1 Thread Table

The thread table is a map from thread identifiers to threads. This map is represented as an alist. Thus, `(binding th (thread-table s))` is the standard idiom for fetching the thread associated with a given thread identifier, `th`, in a state `s`. Thread identifiers are natural numbers.

A thread is a triple.

- `call-stack` – sometimes called the method invocation stack, this is a stack of frames. Each frame records the invocation of one method and the top frame is the currently running method in the thread. Each other frame is suspended awaiting the termination of some method invoked by it.
- `call-stack-status` – either `SCHEDULED` or `UNSCHEDULED`.
- `call-stack-rref` – a reference to an object in the heap representing the current thread. We discuss this reference later (see page 16).

In our model it is rare for a function to have a thread "in hand" and need to fetch one of these components. But it is very common for a function to have a thread identifier and state "in hand" and need one of these components for the corresponding thread. So the standard idiom for accessing the call stack of thread `th` in state `s` is `(call-stack th s)`.

```
(defun call-stack (th s)
  (nth 0 (binding th (thread-table s))))
```

The call stack of a thread is a stack of frames. Each frame other than the deepest corresponds to the invocation of a method by the method in the frame below it. The deepest frame is typically executing the `run` method of the thread. The top frame of a call stack is that for the currently active method. We use it so frequently we define a function for accessing it.

```
(defun top-frame (th s) (top (call-stack th s)))
```

Each frame contains six components.

- `pc` – a natural number that is the byte offset of the current bytecode instruction in the method body of the method being executed.
- `program` – a list of the bytecode instructions of the current method
- `locals` – a list specifying the values of the method's local variables. In the JVM, locals are referenced positionally. For example, local 0 is the "self" Object (`this`) for an instance method, local 1 is the first formal parameter, etc., local variables other than parameters are included at the end. The $i^{th}$ element of the `locals` list of a frame holds the object representing the value of the $i^{th}$ local variable. Our model supports double word variables (which consume two positions) by using one slot to hold the entire object representing the value and the other slot to hold an irrelevant dummy value.
- `stack` – sometimes called the "operand stack" to distinguish it from the call stack, this is a stack objects used in the evaluation of expressions and to pass arguments to other methods.
- `sync-flg` – one of three symbols, `LOCKED`, `S_LOCKED`, or `UNLOCKED`, indicating whether the invocation of the current method locked the self Object (in the case of an instance method), the method's class Object (in the case of static method) or neither.
- `cur-class` – the name of class in which the current method was defined.

In the actual JVM, programs are just sequences of bytes; most instructions are one byte long but some are longer. The `ALOAD` instruction, which pushes the value of the $n^{th}$ local variable onto the operand stack, is coded as two successive bytes, the byte for the opcode `ALOAD` and a byte for $n$. In our representation, this instruction is represented in parsed form by the object `(ALOAD` $n$`)`. We discuss the bytecode instructions at length below.

### 4.1.2 Heap

In Java and the JVM, new Objects are allocated in a heap. Pointers to these Objects, called "references," are passed around in JVM programs.

In our model, the `heap` of a state is a map, represented as an alist, from heap addresses to objects representing instance Objects. If $a$ is the heap address of an object representing some Object, then `(REF` $a$`)` is the object representing a reference to that Object.

Our standard idiom for retrieving the representation of the instance Object at a given reference `ref` in the heap of a state `s` is `(deref ref (heap s))`, where
`(defun deref (ref heap) (binding (cadr ref) heap))`.

We use natural numbers to represent heap addresses. This allows a simple scheme for generating new heap addresses: as Objects are allocated, they are assigned successive natural numbers as heap addresses. Thus, in a sequential process, the heap addresses indicate the order in which Objects were created. This makes the definition of certain heap invariants and other properties easier to express, because arithmetic on heap addresses can be used in the specification language (ACL2).

Natural numbers are also used to represent the int primitive type in Java. But our representation is unambiguous. If, in our model, a JVM method finds `27` on top of the operand stack, it represents the int 27. If a method finds `(REF 27)` on top of the stack, it represents a reference to the object with heap address `27`.

It is not logically necessary to garbage collect our heap.

The concreteness of this representation of heap addresses may offend some readers. But heap addresses have to be represented by *some* ACL2 objects, since they must somehow be involved in references, and references *must* be objects since they are part of the state. What other ACL2 objects might we have used for heap addresses? We could have used lists, symbols, strings, or numbers of some kind, e.g., any objects chosen from some infinite set of ACL2 objects. But natural numbers have the advantages discussed above. They should not be thought of as connected in any way to the physical addresses at which an implemented JVM allocates an object. While we can use arithmetic on them in the specification language, Java and JVM programs cannot exploit their arithmetic properties. Heap addresses and references are just abstractions.

We have explained that the heap is a map from heap addresses to objects representing instance Objects. We now explain how we represent the latter.

An instance Object is represented as a map from class names to maps from field names to values. Suppose, for example, that the class `Point` extends the built in `Object` class and has fields named `x-coord`, `y-coord`, and `radius`. Suppose that `ColoredPoint` extends `Point` and has fields named `color` and `radius`. Then the object representing a `ColoredPoint` looks like this

```
(("ColoredPoint"     . (("color"   . v1)
                        ("radius"  . v2)))
 ("Point"           . (("x-coord" . v3)
                        ("y-coord" . v4)
                        ("radius"  . v5)))
 ("java.lang.Object" . ( ... )))
```

where the $v_i$ are the values of the indicated fields and we have elided away the fields for the primitive `"Object"` class. We call the object above an *instance object*. Observe that there are two fields with name `"radius"` in the instance object above. One is the `"radius"` field of the `"ColoredPoint"` class, which has value $v_2$, and the other is the `"radius"` field of the `"Point"` class, which has value $v_5$.

Here are the functions for accessing and setting the fields of an instance object.

```
(defun field-value (class-name field-name instance)
  (binding field-name
           (binding class-name instance)))
(defun set-instance-field
  (class-name field-name value instance)
  (bind class-name
        (bind field-name value
              (binding class-name instance))
        instance))
```

For example, if the variable `instance` has as its value the instance object above, then `(field-value "Point" "radius" instance)` is equal to $v_4$. The term `(set-instance-field "Point" "radius"` $v_4'$ `instance)` returns an instance object just like the one above, but with $v_4$ replaced by $v_4'$.

In our model, arrays are represented as though they were ordinary instance Objects extending `Object` with one field named `"<array>"`. However, the contents of this field is a 4-tuple consisting of the constant symbol `*array*`, an array type, an array bound, and the array content. The last is a list of objects.

We now explain the `call-stack-rref` field of a thread. In Java, threads are Objects in the heap, references to them are created by `new` and those references can be passed to methods, such as the `start` method. But threads are also processes. Our model uses the heap to store the "object manifestation" of a thread and the thread table to store the "process manifestation". To every thread in the thread table (except the first one), there corresponds

a unique "java.lang.Thread" object in the heap. The call-stack-rref of the thread contains the reference to that thread object.

For example, suppose new is used to create a new object in the heap. The object contains the fields specified by its class hierarchy and is allocated to some heap address $a$ and has reference (REF $a$). But if the object is a Thread, i.e., if the class hierarchy includes "java.lang.Thread", then new also creates a new entry in the thread table and stores (REF $a$) in its call-stack-rref component. JVM programs pass (REF $a$) to refer to the Thread object, but when it is necessary to determine what is the corresponding process, we can search the thread table for the thread with that call-stack-rref.

### 4.1.3 Class Table

The class-table component of a state is a map from class names to class declarations. This map is represented by an alist. Class declarations are 7-tuples. The components are briefly described below.

- class-decl-name – the name of the class, which is always some ACL2 string.
- class-decl-superclasses – the list of superclass names, from most specific to least. The class name "java.lang.Object" is always the last element of the list.
- class-decl-fields – the list of field names for instance objects in this class. Field names are always ACL2 strings.
- class-decl-sfields – the list of static field names associated with this class.
- class-decl-cp – the constant pool for the class. In our model, the constant pool is a list of objects, each of which is either of the form (INT $n$), where $n$ represents an int, or (STRING (REF $a$) $string$), where $string$ is an ACL2 string and $a$ is either -1 or the heap address at which a "java.lang.String" instance object has been allocated. The "java.lang.String" class contains a field called the "strcontents" and if $a$ is a heap address then the object at that address contains $string$ in its "strcontents" field.
- class-decl-methods – a list of the methods associated with this class. See below.
- class-decl-heapref – the reference to the object in the heap representing this class.

A method declaration has four components.

- method-name – the name of the method, a string.
- method-formals – a list of types in 1:1 correspondence with the formals of the method.
- method-sync – t or nil indicating whether the method is synchronized.
- method-program – the list of bytecode instructions in the method body.

Normally we would represent such a tuple as ($name$ $formals$ $sync$ $program$) but in this case we use ($name$ $formals$ $sync$ . $program$).

A method is said to be *native* if its program is (nil).

```
(defun method-isNative? (m)
  (equal '(NIL) (method-program m)))
```

### *4.2 Constructing Modified States*

For each JVM bytecode instruction in our subset, we define a state transition function that takes the instruction, a thread identifier, and a state, and returns the new state produced by executing the given instruction in the given thread of the given state. Typically, the new state is just like the old state except a few fields of the top frame of given thread are changed.

We define a macro to make it it easy to read and write such state transitions. Rather than define the macro here — which would require a digression into the fine points of macro definition — we merely show how the macro expands in common uses.

The expression (`modify th s :pc` $pc$) is equivalent to

```
(make-state
 (bind th
       (make-thread
        (push (make-frame pc
                          (locals (top-frame th s))
                          (stack (top-frame th s))
                          (program (top-frame th s))
                          (sync-flg (top-frame th s))
                          (cur-class (top-frame th s)))
              (pop (call-stack th s)))
        'SCHEDULED
        (call-stack-rref th (thread-table s)))
       (thread-table s))
 (heap s)
 (class-table s))
```

That is, the result is a state that is like `s` except for the top frame of the call stack of thread `th`, which in the new state has the program counter $pc$ instead of its old value. In addition, the status of thread `th` is set to `SCHEDULED`. It might have been more appropriate to leave the status flag unchanged, but in fact it will always be `SCHEDULED` when the thread is executing.

The `modify` macro takes a variety of keywords to indicate which slot of the state to change. Their order does not matter.

The term

```
(modify th s
   :heap heap
   :pc pc
   :sync-flg flg
   :stack stk)
```

is equivalent to

```
(make-state
 (bind th
       (make-thread
        (push (make-frame pc
                          (locals (top-frame th s))
                          stk
                          (program (top-frame th s))
                          flg
                          (cur-class (top-frame th s)))
              (pop (call-stack th s)))
        'SCHEDULED
        (call-stack-rref th (thread-table s)))
       (thread-table s))
 heap
 (class-table s))
```

and the term

```
(modify th s :heap heap :call-stack cs)
```

is equivalent to

```
(make-state
 (bind th
       (make-thread cs
                    'SCHEDULED
                    (call-stack-rref th (thread-table s)))
       (thread-table s))
 heap
 (class-table s))
```

Not all combinations of keywords make sense. For example, it makes no sense to set the `:call-stack` of the thread and also set the `:pc` of the top frame of the old call stack.

### *4.3 Bytecode Instruction Semantics*

We are now ready to characterize the effects of each bytecode instruction in our subset. For each instruction class we will define an ACL2 function, called the *semantic function*, which describes the state change caused by executing a given instruction in a given thread of a given state. After reading a few it should be straightforward to read the entire list of 138.

#### 4.3.1 AALOAD

The following description of the `AALOAD` bytecode instruction is taken verbatim from [27].

> `aaload`
> **Operation**
> Load reference from array
> **Format**
> `aaload`
> **Forms**
> `aaload` = 50 (`0x32`)
> **Operand Stack**
> *..., arrayref, index* ⇒ *..., value*
> **Description**
> The *arrayref* must be of type reference and must refer to an array whose components are of type reference. The *index* must be of type `int`. Both *arrayref* and *index* are popped from the operand stack. The reference *value* in the component of the array at *index* is retrieved and pushed onto the operand stack.
> **Runtime Exceptions**
> If *arrayref* is `null`, `aaload` throws a `NullPointerException`.
> Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the `aaload` instruction throws an `ArrayIndexOutOfBoundsException`.

In Java, every data item is either of a primitive type or is a reference (to some Object). JVM instructions are typed in the sense that it is possible to determine, for example, the type of object on top of the operand stack after each instruction.

The meaning of the informal specification of `aaload` is as follows. First, we are told that it leaves a reference on top of the stack. Then we are told it is a one-byte instruction whose opcode is 50 (32 in hexadecimal). When it is executed, two items are expected on the operand stack, an *index* and an *arrayref*, with the index on top. The top item is a 32-bit integer, and the next item is a reference to an array Object in the heap. Furthermore, the elements of the array are themselves references. The two items are popped off the operand stack, the item, called *value*, at position *index* is fetched from the array indicated by *arrayref* and *value* is pushed onto the operand stack. Certain exceptions are caused if the expectations are not met. Our current model of the JVM does not deal with exceptions, though we are working on a model that does.

Our formalization of the specification above is shown below. This is the *semantic function* for `AALOAD`.

```
(defun execute-AALOAD (inst th s)
  (let* ((index (top (stack (top-frame th s))))
         (arrayref (top (pop (stack (top-frame th s)))))
         (array (deref arrayref (heap s))))
    (modify th s
     :pc (+ (inst-length inst) (pc (top-frame th s)))
     :stack (push (element-at index array)
                  (pop (pop (stack (top-frame th s))))))))
```

It takes as its arguments an `AALOAD` instruction, `inst`, a thread identifier, `th`, and a state, `s`. It returns a modified state.

The `let*` expression above is a convenient abbreviation in ACL2. It sequentially binds the local variables `index`, `arrayref`, and `array` to the values of the corresponding expressions and then returns the modified state. The meaning of the definition above should be obvious by now. Note that we advance the `pc`, even though the informal specification omitted mention of it.

### 4.3.2 AASTORE

Here is the `AASTORE` instruction pops three items from the stack, treats the top one as a reference and stores it into the array position indicated by the other two.

```
(defun execute-AASTORE (inst th s)
  (let* ((value (top (stack (top-frame th s))))
         (index (top (pop (stack (top-frame th s)))))
         (arrayref (top
                    (pop
                     (pop (stack (top-frame th s)))))))
    (modify th s
     :pc (+ (inst-length inst) (pc (top-frame th s)))
     :stack (pop (pop (pop (stack (top-frame th s)))))
     :heap (bind (cadr arrayref)
                 (set-element-at value
                                 index
                                 (deref arrayref (heap s))
                                 (class-table s))
                 (heap s)))))
```

The term `(cadr arrayref)` is the heap address, $a$, inside the reference `(REF` $a$`)`.

### 4.3.3 IADD

The IADD instruction pops two ints off the operand stack and pushes their sum. According to [27], "The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type int." The int input is represented in ACL2 by integers satisfying the predicate intp.

```
(defun intp (x)
  (and (integerp x)
       (<= (- (expt 2 31)) x)
       (< x (expt 2 31))))
```

The result is the true mathematical sum of the two integers "fixed" with the function int-fix.

```
(defun s-fix (x n)
  (let ((temp (mod (ifix x) (expt 2 n))))
    (if (< temp (expt 2 (1- n)))
        temp
      (- temp (expt 2 n)))))
(defun int-fix (x)
  (s-fix x 32))
```

The function ifix above is an ACL2 primitive; it is the identity function on integers and returns 0 on all other objects. We use it in s-fix to insure that the output is an integer, regardless of the input's type. We use s-fix to manipulate 8-bit and 16-bit data also.

Here is the semantic function for IADD. The other arithmetic instructions are similar.

```
(defun execute-IADD (inst th s)
  (modify th s
   :pc (+ (inst-length inst) (pc (top-frame th s)))
   :stack (push (int-fix
                 (+ (top (pop (stack (top-frame th s))))
                    (top (stack (top-frame th s)))))
              (pop (pop (stack (top-frame th s)))))))
```

### 4.3.4 GETFIELD

The GETFIELD instruction is a 3-byte instruction. In the actual JVM it consists of the opcode (180) and two bytes, which together form an index into the constant pool of the current class. The object at that location in the constant pool "gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found." In our formalization, we let the GETFIELD instruction take three operands, class-name, field-name, and long-flag. These values can be determined statically from the indices of the instruction and the current class. The long-flag indicates whether the type of the field (included in its "descriptor") is a double word item. The instruction pops a reference off the operand stack, dereferences it through the heap to an instance object, fetches the value of the class-name field named field-name, and pushes it onto the operand stack.

```
(defun execute-GETFIELD (inst th s)
  (let* ((class-name (arg1 inst))
         (field-name (arg2 inst))
         (long-flag  (arg3 inst))
         (instance (deref (top (stack (top-frame th s)))
                          (heap s)))
         (field-value (field-value class-name field-name
                                   instance)))
    (modify th s
```

```
    :pc (+ (inst-length inst) (pc (top-frame th s)))
    :stack (if long-flag
               (push 0
                     (push field-value
                           (pop (stack (top-frame th s)))))
               (push field-value
                     (pop (stack (top-frame th s))))))))
```

### *4.4 MONITORENTER*

Each Object in Java has a `monitor` and an `mcount` field inherited from primitive class `java.lang.Object`. These fields are used to provide synchronization between threads. The `monitor` of an Object is either set to `0` or the thread identifier of the thread that "owns" the "lock" on the Object. The `mcount` of an Object is the number of times the Object has been locked (reentrant locking is allowed). Special instructions `MONITORENTER` and `MONITOREXIT`, are available for manipulating these fields.

```
(defun execute-MONITORENTER (inst th s)
  (let* ((obj-ref (top (stack (top-frame th s))))
         (instance (deref obj-ref (heap s))))
    (cond
     ((objectLockable? instance th)
      (modify th s
       :pc (+ (inst-length inst) (pc (top-frame th s)))
       :stack (pop (stack (top-frame th s)))
       :heap (lock-object th obj-ref (heap s))))
     (t s))))
```

This instruction increments the lock on the Object on top of the operand stack, if it is not already owned by another thread. Note that if the lock is owned by another thread, the instruction is a no-op: we return the given state `s`. Thus, the `pc` continues to point to the `MONITORENTER` instruction, which will be executed again when the thread is next scheduled.

The two important functions used above are shown below.

```
(defun objectLockable? (instance th)
  (let* ((obj-fields (binding "java.lang.Object" instance))
         (monitor (binding "monitor" obj-fields))
         (mcount (binding "mcount" obj-fields)))
    (or (zp mcount)
        (equal monitor th))))
(defun lock-object (th obj-ref heap)
  (let* ((obj-ref-num (cadr obj-ref))
         (instance (binding (cadr obj-ref) heap))
         (obj-fields (binding "java.lang.Object" instance))
         (new-mcount (+ 1 (binding "mcount" obj-fields)))
         (new-obj-fields
            (bind "monitor" th
               (bind "mcount" new-mcount obj-fields)))
         (new-object
            (bind "java.lang.Object" new-obj-fields
                  instance)))
    (bind obj-ref-num new-object heap)))
```

### 4.4.1 INVOKEVIRTUAL

The most complicated JVM instruction is probably `INVOKEVIRTUAL`, which invokes a method on an Object and passes it some actuals. The Object and the actuals are pushed onto the operand stack of the caller before `INVOKEVIRTUAL` is executed. If there are $n$ actuals, then the Object is the item at depth $n + 1$ on the operand stack. In the semantic function below, $n$ is called `nformals`. The top item on the operand stack is the last of the method's actuals.

The format of our `INVOKEVIRTUAL` instruction is

`(INVOKEVIRTUAL"`*`class`*`" "`*`method-name`*`" "`*`nformals`*`).`

In implemented JVMs, the instruction includes indices into the constant pool through which the machine can determine *class*, *method-name*, and the type signature of each formal and the result. In our current model we ignore types; we abstract the signature information merely to the number of formals involved.

The method invoked depends upon the class hierarchy of Object and upon *name* and *nformals*. Recall that each class contains method definitions and superclasses; this gives rise to a hierarchy of alternative method definitions (some of which may have the same name) associated with the class of the Object. Roughly speaking, `INVOKEVIRTUAL` searches this hierarchy for the first (or "closest") method with the given name and type signature. The reason the instruction format includes a *class* is that the search can be optimized by means of a "method dispatch vector," but the result of that optimization must be as specified below.

Once a particular method definition is identified, the Object and the actuals are popped from the operand stack of the caller, a new frame is constructed containing the bytecode for the identified method and local variable bindings containing the Object and the actuals, and that frame is pushed onto the call stack of the thread in question.

There are myriad details, of course. These details include the possibility that the method is "native," which means bytecode for it is not available. In an implemented JVM, native methods are usually coded in machine code. But two native methods are especially important and "ought" to be part of the JVM: `start` and `stop`, the methods for beginning and ending the life of a thread Object as a process. They are handled specially below. Recall that each thread created by `NEW` has a object manifestation in the heap and a process manifestation in the thread table. Upon creation, the process manifestation has the status `UNSCHEDULED` which may be thought of as meaning "not allowed to run." The `start` method is invoked on the reference to the object manifestation, it identifies the corresponding entry in the thread table and sets its status to `SCHEDULED`.

Another detail is that of synchronization. Some method definitions specify that the Object on which they are to be invoked must be locked upon invocation. These "synchronized methods" are also handled specially below. The `t` clause of the `cond` below describes the "normal" case of unsynchronized method invocation.

The semantic function below constructs its answer state in two steps. First, the input state `s` is transformed to `s1` and then `s1` is transformed to the answer. State `s1` may be thought of as `s` with the program counter of the caller's frame advanced past the `INVOKEVIRTUAL` instruction and the Object and actuals removed from the operand stack.

Do not be put off by the length of the definition below. Each part should be clear now.

```
(defun execute-INVOKEVIRTUAL (inst th s)
  (let* ((method-name (arg2 inst))
         (nformals (arg3 inst))
         (obj-ref
          (top (popn nformals (stack (top-frame th s)))))
         (instance (deref obj-ref (heap s)))
         (obj-class-name
```

```
       (class-name-of-ref obj-ref (heap s)))
      (closest-method
       (lookup-method method-name
                      obj-class-name
                      (class-table s)))
      (prog (method-program closest-method))
      (s1
       (modify th s
               :pc (+ (inst-length inst)
                      (pc (top-frame th s)))
               :stack (popn (+ nformals 1)
                            (stack (top-frame th s)))))
      (tThread (rrefToThread obj-ref (thread-table s))))
(cond
 ((method-isNative? closest-method)
  (cond ((equal method-name "start")
         (modify tThread s1 :status 'SCHEDULED))
        ((equal method-name "stop")
         (modify tThread s1
                 :status 'UNSCHEDULED))
        (t s)))
 ((and (method-sync closest-method)
       (objectLockable? instance th))
  (modify th s1
   :call-stack
   (push
    (make-frame 0
                (reverse
                 (bind-formals (make-list (+ nformals 1))
                               (stack (top-frame th s))))
                nil
                prog
                'LOCKED
                (arg1 inst))
    (call-stack th s1))
   :heap (lock-object th obj-ref (heap s))))
 ((method-sync closest-method)
  s)
 (t
  (modify th s1
   :call-stack
   (push
    (make-frame 0
                (reverse
                 (bind-formals (make-list (+ nformals 1))
                               (stack (top-frame th s))))
                nil
                prog
                'UNLOCKED
                (arg1 inst))
    (call-stack th s1)))))))
```

### 4.4.2 NEW

We briefly deal with the `NEW` instruction. The basic idea is to construct a new instance object of a given class, assign it a heap address, `new-address`, (namely, the length of the current heap), and push a reference to that address on the operand stack. The reference is, of course,

constructed by (list 'REF new-address). The construction of the instance object itself is done by the function build-an-instance.

```
(defun build-class-field-bindings (field-names)
  (if (endp field-names)
      nil
    (cons (cons (car field-names) 0)
          (build-class-field-bindings (cdr field-names)))))
(defun build-immediate-instance-data (class-name class-table)
  (cons class-name
      (build-class-field-bindings
       (class-decl-fields
        (bound? class-name class-table)))))
(defun build-an-instance (class-names class-table)
  (if (endp class-names)
      nil
    (cons (build-immediate-instance-data (car class-names)
                                         class-table)
          (build-an-instance (cdr class-names)
                             class-table))))
```

The last function takes a list of classes (the class to build and its superclass chain) and a class table and constructs an alist mapping each class to an alist mapping each field of the class to the initial value 0. The object thus constructed is assigned to new-address in the new heap.

The NEW instruction is complicated, however, by the need to create the process manifestation of any new thread Object and store it in the thread table. The semantics of Java specifies that when a new thread is created the process should invoke the run method of the class. Thus, in building the new thread table entry, NEW resembles INVOKEVIRTUAL. The reference to the new object is passed as the $0^{th}$ formal (aka "this") to the run method and that reference is also stored as the entry's call-stack-rref. The entry's status is UNSCHEDULED.

```
(defun execute-NEW (inst th s)
  (let* ((class-name (arg1 inst))
         (class-table (class-table s))
         (closest-method
          (lookup-method "run" class-name class-table))
         (prog (method-program closest-method))
         (new-object
          (build-an-instance
           (cons class-name
                 (class-decl-superclasses
                  (bound? class-name class-table)))
           class-table))
         (new-address (len (heap s)))
         (s1
          (modify th s
           :pc (+ (inst-length inst) (pc (top-frame th s)))
           :stack (push (list 'REF new-address)
                        (stack (top-frame th s)))
           :heap (bind new-address new-object (heap s)))))
    (if (isThreadObject? class-name class-table)
        (modify nil s1
         :thread-table
         (addto-tt
          (push
           (make-frame 0
                       (list (list 'REF new-address))
                       nil
                       prog
```

```
                    'UNLOCKED
                    class-name)
          nil)
         'UNSCHEDULED
         (list 'REF new-address)
         (thread-table s1)))
    s1)))
```

### *4.5 Execution*

We have defined semantic functions for 138 instructions in the style shown above. Now we put it all together as a state transition function for the JVM.

The function, named `step`, takes a thread identifier and a state and returns the next state.

```
(defun step (th s)
  (if (equal (call-stack-status th s) 'SCHEDULED)
    (do-inst (next-inst th s) th s)
    s))
```

If the thread to be stepped has status `SCHEDULED`, then it is eligible for running. In that case, `step` uses `next-inst` to fetch the next bytecode instruction from the thread.

```
(defun next-inst (th s)
  (index-into-program (pc (top-frame th s))
                      (program (top-frame th s))))
```

Of course, `next-inst` just uses the program counter and program in the top frame of the thread's call stack and fetches the appropriate instruction. The function `index-into-program` counts the length of each instruction in bytes.

Once the next instruction is obtained, `step` uses `do-inst` to execute the appropriate semantic function. `Do-inst` is just a big `case`-statement on the opcode of the instruction.

```
(defun do-inst (inst th s)
  (case (op-code inst)
    (AALOAD         (execute-AALOAD inst th s))
    (AASTORE        (execute-AASTORE inst th s))
    ...     ...
    (otherwise s)))
```

Unrecognized opcodes are treated as no-ops in our model.

We define `run` to take a schedule and a state and return the state obtained by stepping as per the schedule. In our model, a schedule is just a list of thread identifiers indicating the order in which the threads are to be stepped.

```
(defun run (sched s)
  (if (endp sched)
      s
      (run (cdr sched) (step (car sched) s))))
```

Observe that we do not model thread scheduling; like garbage collection, that is up to the JVM implementor and not specified by [27]. Suppose that `s0` is defined as a function that takes some input, `n`, and returns some initial state poised to run some method on input `n`. Suppose also that `p` is some predicate. Then if

```
(p n (run sched (s0 n)))
```

is a theorem, then every state reachable from the initial state `(s0 n)` has the property `p`.

More commonly, we might prove a theorem of the form

```
(implies (good-sched sched n) (p n (run sched (s0 n))))
```

which says that all states reachable under "good schedules" have property `p`. Here `good-sched` formalizes some constraint on the schedules we consider. Typical constraints might say that a given thread is executed a certain number of times or that every thread is executed at least a certain number of times. The latter constraint is a finite approximation to fairness.

Finally, we might also prove the special case

```
(p n (run (sched n) (s0 n)))
```

which says that `(sched n)` produces a schedule sufficient to drive `(s0 n)` to a state satisfying `p`. This is actually the form of a typical program correctness theorem in this setting. For example, imagine that `s0` constructs a state poised to execute the factorial method on `n` and that `p` checks that `n!` is the top item on the operand stack of top-frame of the call stack of a certain thread.

Some readers might be tempted to abstract the theorem above to

$\exists a$ `(p n (run` $a$ `(s0 n))).`

and be content that some schedule is sufficient to make `(s0 n)` compute the desired result. In ACL2, this statement is not easily made, because the logic does not provide quantifiers.[1]

But in fact we prefer the exhibition of a total recursive function, `sched`, for delivering the alleged schedule. This is a much stronger result than the existentially quantified one. Consider the program that runs forever, generating successively all `int` values. The bytecode for this program is

```
((ICONST_0)        ;       push 0
 (ICONST_1)        ; loop: push 1
 (IADD)            ;       pop two and push sum
 (GOTO -2))        ;       goto loop
```

We call this the "universal `int` program," largely in jest. The universal `int` program has the property that for any given `int`, there exists a schedule that makes the program leave that `int` on top of the stack. Thus, if one is content with existential scheduling, the universal `int` program computes all possible `int`-valued functions — you just have to know when to look at the answer.

We return to this point later.

### *4.6 Omissions and Inaccuracies*

We now reiterate the ways in which our model is incomplete and inaccurate. We are working on improving the accuracy of our model.

- The current model omits support for floating point data. ACL2 has been used extensively to do floating-point verification work [14, 41, 40, 39, 34] and adding floating-point to our JVM model would not be difficult.

- The current model omits support for syntactic typing but we currently have a prototype model that addresses this problem. In the model, class declarations have additional components that contain the type signatures of fields and methods; and data in locals, the operand stacks, and the heap is tagged to indicate its type. This allows us to redefine `lookup-method` — our function for finding the closest method — to do signature

[1] Actually, support for quantification exists. See the online documentation of `defchoose` and `defun-sk` ("define Skolem function").

matching, as required in Java method resolution. In addition, it allows us to define two JVM models, a "defensive" one that does runtime "type checks" and one (like this one) that does not. The defensive machine signals a type violation and halts when the runtime checks fail. It is possible to prove that when no violation is signaled, the two machines return the same state. Using the defensive machine, it is possible to investigate the formal correctness of the bytecode verifier: code approved by the verifier, when run on well-typed input, will not signal a type violation. Major parts of this proof have been checked with ACL2, under the direction of Hanbing Liu, a UT Austin graduate student. Liu continues to work on the project.

- Java permits fields and methods to have access modes limiting their use outside the defining class. We have not implemented them; the outline of the required modification should be clear.

- Class loading and initialization are not addressed. The formal model must have as an argument (or somehow accessible otherwise) the class descriptions of all classes that might be loaded, along with socket information to formalize the restrictions of the security manager. Class loading and initialization would move class declarations from this data structure to our class table.

- The current model ignores the notion of exception handling in Java and the JVM. This is a major omission and we are working on its addition. Hanbing Liu is also managing this effort. The JVM's facilities for exception handling are clearly specified but will affect our method declarations — which should include exception tables — as well as the handling of returns from method invocations.

- Our JVM model provides a sequentially consistent memory model. The official JVM memory model, which is described in Chapter 17 of [27], does not require this and probably will not require it for arbitrary programs. The official JVM memory model is under revision [30] and will probably require that any execution of a "correctly synchronized" program must be equivalent to some interleaved bytecode execution. The memory model is under revision . For details see `www.jcp.org/jsr/detail/133.jsp`.

### *4.7 Related Work*

We have already mentioned the ground breaking work of Rich Cohen [10] on the defensive JVM (dJVM) model, the first formal model of a significant portion of the JVM. That work was done at Computational Logic, Inc., where ACL2 was also first developed, and ACL2 was the modeling language used. As previously noted, the machine described here was evolved, with Cohen's help, from the dJVM.

ACL2 was also used to model the Rockwell JEM1 microprocessor, the world's first silicon JVM, now marketed by aJile Systems, Inc. The ACL2 model was at the microarchitectural level, not the higher level shown here, and was used as the standard test bench on which Rockwell engineers tested the chip design against the requirements by executing compiled Java programs. The ACL2 model executed at approximately 90% of the speed of the previously used C model [20, 21]. Greve, Wilding, and Hardin describe how microprocessor models in ACL2 are made to execute fast [19]. The model there executes at approximately 3 million simulated instructions per second on a 728 MHZ Pentium III host running Allegro Common Lisp.

At Rockwell not only are ACL2 models used for simulation purposes but microarchitectural models are related to one another by ACL2 proofs. See [18].

Similar work is being conducted at Advanced Micro Devices. For example, an executable ACL2 model of the RTL for the AMD Athlon$^{TM}$ floating-point square root was tested on 80 million floating-point vectors. The model computed the same answers as AMD's RTL simulator and this fact helped establish confidence in the formal model. ACL2 was then used to prove that the RTL for each elementary floating-point operation on the Athlon is compliant with the IEEE 754 floating-point standard [39].

We now turn to related mechanized formal JVM work other than that by the ACL2 community.

The Extended Static Checker (ESC) [11] is an example of a formal, practical and mechanized tool for establishing certain simple assertions about Modula-3 programs. It is the basis of the ESC/Java verifier [26] for Java, developed at the Compaq (now HP) Systems Research Center (SRC) in Palo Alto, CA. ESC/Java is being extended in a tool named Calvin, by Shaz Qadeer, at SRC, to support assume-guarantee style reasoning about Java [private communication].

The Java PathFinder [8, 44] (JPF) is an explicit-state model-checker for programs written in Java. It can check certain kinds of invariants and deadlock. A tool with similar functionality is Bandera [13]. Both of these model-checking tools have been used to check properties of a Java version of the DEOS real-time operating system kernel, a program involving approximately 20 classes, 6 threads, 91 methods, 41 instance fields, and 51 static fields. The property was a complex time-partitioning requirement. See our paper [35] for a discussion of the use of JPF to check a theorem also proved by ACL2.

There are other theorem-proving based approaches to Java verification. One such tool is the LOOP tool [2, 43] which translates Java and JML (a specification language tailored to Java) classes into their semantics in higher order logic. As such, LOOP can be used as a front end for such theorem provers as PVS [37] and Isabelle [36]. However, LOOP currently deals only with sequential Java.

Other related work include [38, 1], where models of the JVM are formalized in Isabelle and Coq [12]. In both efforts, the correctness of the bytecode verifier is addressed and the JVM models are largely concerned with type correctness rather than full functionality.

## 5 An Example

In this section we display some simple Java, the class file produced from it, the representation of that data in terms of a state object, and the use of `run` on that data.

### *5.1 Factorial in Java*

In Figure 1 we show a Java file named `Demo.java`.

The `main` method is trivial: it computes `fact` on `5` and stores the result, `120`, in the static class variable `ans`. It would be more common to print the result, but we have not formalized i/o, which is handled by native methods.

The result of running the Sun Java compiler, `javac`, on `Demo.java` and displaying the results with the Sun utility `javap -c` is shown in Figure 2.

The `ifle` instruction at pc `1` in `Method int fact(int)` is a branch instruction to pc `13` when the top of the stack is less than or equal to `0`. The `javap` utility displays the jump targets as absolute program counters. But the actual JVM `ifle` instruction takes an offset from the current pc as its operand. Thus, this line ought to read `1 ifle 12` to faithfully display the bytecode.

```
class Demo {
  static int ans;
  public static int fact(int n){
    if (n>0)
      {return n*fact(n-1);}
    else return 1;
  }
  public static void main(String[] args){
  int k = 4;
  ans = fact(k+1);
  return;
  }
 }
```

Figure 1: A Recursive Factorial Method

```
Compiled from Demo.java
synchronized class Demo extends java.lang.Object
    /* ACC_SUPER bit set */
    public static int fact(int);
    public static void main(java.lang.String[]);
    Demo();
Method int fact(int)
   0 iload_0
   1 ifle 13
   4 iload_0
   5 iload_0
   6 iconst_1
   7 isub
   8 invokestatic #4 <Method int fact(int)>
  11 imul
  12 ireturn
  13 iconst_1
  14 ireturn
Method void main(java.lang.String[])
   0 iconst_4
   1 istore_1
   2 iload_1
   3 iconst_1
   4 iadd
   5 invokestatic #5 <Method int fact(int)>
   8 putstatic #4 <Field int ans>
  11 return
Method Demo()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return
```

Figure 2: The Class File for Demo.java

The bytecode for the `fact` method should be obvious. Push local `0` (i.e., $n$) on the operand stack and test it against `0`. If it less than or equal to 0, branch to pc 13, push `1` on the operand stack and return that `int`. Otherwise (at pc 4), push $n$ on the stack, push $n$ on the stack again, push `1`, pop the top two `int`s and push their difference. Now we have $n$ and $n-1$ on the stack, with the latter on top. Recursively invoke `fact` at pc 8. That will pass the $n-1$ as the actual and leave an `int` result on the stack when it returns. Then, at pc 11, multiply the two `int`s on top of the stack and return that `int`.

The bytecode for the `main` method is also obvious. Push the constant `4` onto the operand stack. Pop and store it into local variable `1`. Push the value of local variable `1`. Push the constant `1`. Pop two `int`s and push their sum. (Now `5` is the only item on the operand stack.) Invoke the `fact` method on `5`, leaving `120` on the stack. Pop it and store it into the static field named `ans` (in the `"Demo"` class). Finally, return.

### *5.2 A State*

The constant `*Demo-state*`, below, is a JVM state about to execute the `main` method of the `Demo.java`. To explain it, we must explain its thread table, heap, and class table. These components are defined below as individual constants so that we can discuss them one by one.

```
(defconst *Demo-state*
  (make-state *Demo-thread-table*
              *Demo-heap*
              *Demo-class-table*))
```

Here is the thread table. There is one thread, with thread identifier `0`. The call stack of the thread has one frame, poised to execute the `main` method of the `Demo` class. The thread is `SCHEDULED` and has a `nil` `rref` because the main thread is not a heap object.

```
(defconst *Demo-thread-table*
   (list
    (cons 0                          ; thread identifier
          (make-thread               ; thread
           (push                     ;  call stack
            (make-frame              ;   frame
             0                       ;    pc
             nil                     ;    locals
             nil                     ;    operand stack
             '((ICONST_4)            ;    program
               (ISTORE_1)
               (ILOAD_1)
               (ICONST_1)
               (IADD)
               (INVOKESTATIC "Demo" "fact" 1)
               (PUTSTATIC "Demo" "ans" nil)
               (RETURN))
             'UNLOCKED               ;    sync flag
             "Demo")                 ;    current class
            nil)                     ;   end of call stack
           'SCHEDULED                ;  thread status
           nil))))                   ;  thread rref
```

The single frame has pc 0. The `locals` is `nil`, but when values are stored there it will grow. The operand `stack` is nil. The `program` is that for `main`. The frame is `UNLOCKED` and comes from the `"Demo"` class.

Here is the heap. Each instance object in this heap is the object manifestation of a class. We have not discussed these objects but they are to the class table what the object manifestations of threads are to the thread table. These objects are used by synchronized static methods.

```
(defconst *Demo-heap*
  '((0 . (("java.lang.Class"
            ("<name>" . "java.lang.Object"))
           ("java.lang.Object"
            ("monitor" . 0)
            ("mcount" . 0)
            ("wait-set" . 0))))
    ...
    (5 . (("java.lang.Class"
            ("<name>" . "Demo")
            ("ans" . 0))
           ("java.lang.Object"
            ("monitor" . 0)
            ("mcount" . 0)
            ("wait-set" . 0))))))
```

The instance object with heap address `0` corresponds to the `"java.lang.Object"` class. It is an object of class `"java.lang.Class"` extending class `"java.lang.Object"`. It has the field `"<name>"` from its immediate class and inherits the fields `"monitor"`, `"mcount"` and `"wait-set"` from the `Object` class.

The elided entries in the heap are the instance objects for the primitive classes `ARRAY`, `java.lang.Thread`, `java.lang.String`, and `java.lang.Class`.

The instance object with heap address `5` corresponds to the `"Demo"` class. Its structure is analogous to the other instance object shown except that it has one static field, named `"ans"`.[2]

Here is the class table for the `Demo` class. Again, we have elided the entries for primitive classes above. The `"java.lang.Object"` class has no superclasses — it is the only such class. It declares three instance field names, `"monitor"`, `"mcount"`, and `"wait-set"`, and no static field names. It has an empty constant pool. It declares only one method, the `<init>` method. The method declaration lists the name, the formals, the synchronization flag, and then the bytecode. In this case, the name is `"<init>"`, the list of formals is empty, the synchronization flag is `NIL`, and the bytecode program is just a list containing the single bytecode instruction `(RETURN)`. We discuss a more interesting bytecode example later. The "*object in heap*" entry is a reference to the instance object representing this class in the heap. We have previously seen that the class corresponds to the object with heap address 0.

```
(defconst *Demo-class-table*
  '(("java.lang.Object"                ; Object class
     NIL                               ; superclasses
     ("monitor" "mcount" "wait-set")   ; instance fields
     NIL                               ; static fields
     NIL                               ; constant pool
     (("<init>" () NIL (RETURN)))      ; methods
     (REF 0))                          ; object in heap
    ...
    ("Demo"                            ; Demo class
     ("java.lang.Object")              ; superclasses
     NIL                               ; instance fields
     NIL                               ; static fields
     NIL                               ; constant pool
```

[2] We intend to change this representation of static fields in future versions, so that they do not appear as instance fields of the object but are buried within some built in field comparable to `"<name>"`.

```
(("<init>" NIL NIL                ; methods
  (ALOAD_0)
  (INVOKESPECIAL "java.lang.Object" "<init>" 0)
  (RETURN))
 ("fact" (INT) NIL
  (ILOAD_0)
  (IFLE 12)
  (ILOAD_0)
  (ILOAD_0)
  (ICONST_1)
  (ISUB)
  (INVOKESTATIC "Demo" "fact" 1)
  (IMUL)
  (IRETURN)
  (ICONST_1)
  (IRETURN))
 ("main" (java.lang.String[]) NIL
  (ICONST_4)
  (ISTORE_1)
  (ILOAD_1)
  (ICONST_1)
  (IADD)
  (INVOKESTATIC "Demo" "fact" 1)
  (PUTSTATIC "Demo" "ans" nil)
  (RETURN)))
(REF 5))))                        ; object in heap
```

Note in particular the methods declared in the `"Demo"` class. Three methods are declared, named `"<init>"`, `"fact"`, and `"main"`. Each declaration is of the form (*name formals sync . program*). That is, each declaration lists three elements, the method name, its formals, and whether it is synchronized, and then concludes with a list of the bytecodes in the program for the method.

The `<init>` method is the initializer for an instance object of the class. It has no formals and is not synchronized. This initializer just invokes the initializer for the superclass.

The `fact` method takes one formal, of type `int`, and is not synchronized. The bytecode for `fact` corresponds exactly to what `javap` displayed (given the caveat about `javap` displaying branch targets as absolute rather than relative addresses).

The `main` method takes one formal, of type `java.lang.String[]`, and is not synchronized. Its bytecode is as previously discussed.

It is convenient to define the constant `*fact-def*` to be the method declaration for `fact`. We have added comments to indicate the byte address of each instruction.

```
(defconst *fact-def*
  '("fact" (INT) NIL
    (ILOAD_0)                      ;;;  0
    (IFLE 12)                      ;;;  1
    (ILOAD_0)                      ;;;  4
    (ILOAD_0)                      ;;;  5
    (ICONST_1)                     ;;;  6
    (ISUB)                         ;;;  7
    (INVOKESTATIC "Demo" "fact" 1) ;;;  8
    (IMUL)                         ;;; 11
    (IRETURN)                      ;;; 12
    (ICONST_1)                     ;;; 13
    (IRETURN)))                    ;;; 14
```

### *5.3 A Schedule*

It is useful to execute a model to corroborate it against requirements, informal expectations or other experiments. But to execute our state, `*Demo-state*`, we must have a suitable schedule.

The code in `*Demo-state*` will run in thread `0`. So we content ourselves with a list of `0`'s sufficiently long to step that thread until we have returned from `main`. While we could try successively longer lists until the returned state stops changing, we prefer to be more analytic. How long should the schedule be?

Let us start with `fact`. How long does it take to compute `fact` on some integer `n`? Let us count instructions, starting with the `INVOKESTATIC` instruction that initiates the computation. If `n` is `0` (or less), we will take `5` steps, namely the initiating `INVOKESTATIC`, the `ILOAD_0` at pc `0`, the `IFLE` at pc `1`, the `ICONST_1` at pc `13`, and the `IRETURN` at pc `14`. By similarly tracing the code we see that if `n` is greater than `0` then `7` steps will bring us to the recursive `INVOKESTATIC` of `"fact"` on $n-1$ at pc `8` and when we return from that invocation, we will execute two more steps, the `IMUL` at pc `11` and the `IRETURN` at pc `12`.

The recursive construction just described can be formalized as follows. If `"fact"` is running on input `n` in thread `th`, then the following function computes a suitable schedule. In our state, `th` will be `0`.

```
(defun fact-sched (th n)
  (if (zp n)
      (repeat th 5)
      (append (repeat th 7)
              (fact-sched th (- n 1))
              (repeat th 2))))
```

The term `(repeat th 5)` makes a list containing five occurrences of `th`. (The order of the arguments in this `repeat` is the opposite of that used in our previous work.)

Given `fact-sched` it is easy to compute a suitable schedule for `main` running in thread `th`.

```
(defun main-sched (th)
  (append (repeat th 5)
          (fact-sched th 5)
          (repeat th 2)))
```

For example `(main-sched 0)` is a list of fifty-seven `0`s.

### *5.4 Execution*

To test our model, we can use ACL2's read-eval-print loop to evaluate the expression `(run (main-sched 0) *Demo-state*)` and inspect the resulting state. We would expect to find `120` in the static field `ans` associated with the object manifestation of the class `"Demo"` in the heap. We can actually phrase this expectation as a theorem if we wish.

```
(defthm example-execution
  (equal (static-field-value "Demo" "ans"
           (run (main-sched 0) *Demo-state*))
         120)
  :rule-classes nil)
```

This commands ACL2 to try to prove the indicated equality and, if successful, give it the name `example-execution`, and generate no rules from it. This equality is "trivial" to prove: it reduces to `t` by evaluation of the functions.

We quote the word trivial above because many theorem provers do not support evaluation, much less the scale of evaluation required here. The JVM model is quite large and so are the constants representing the fifty-seven successive states. One important aspect of ACL2 is the engineering that we have put into doing evaluation. As our evaluations go, this is a relatively small scale problem.

It is easy to define the function `fact-test` for testing the `fact` method on arbitrary input `n`: compute a schedule and a state poised to execute `fact` on `n`, `run` them, and fetch the result. `(fact-test 17)` requires a schedule of length 161. The final answer is `-288522240`. Why is our factorial method returning a negative answer? A still larger test is `(fact-test 1000)`, which requires a schedule of length 9008. The answer is 0. Why?

In the larger runs we can time the execution. The execution of `(fact-test 1000)` requires 0.16 seconds on a 728 MHz Pentium III running GCL under Redhat Linux. (This does not count the time taken to compute the schedule.) This means our model executes at about 56000 JVM bytecodes per second.

## 6 The Proof Methodology

Mechanically analyzing programs with respect to a formally defined operational semantics has a long tradition in the Boyer-Moore community, dating back to the 1970's when we used an early version of the Boyer-Moore theorem prover to formalize part of the Bendix 930 avionics processor and tried to prove a property of a snippet of machine code implementing context switching in SRI's software implemented fault-tolerance (SIFT) computer. Our "minority report" in the SIFT final report made it clear that the semantics of the Bendix 930 were not sufficiently clear to permit accurate modeling and, in addition, that the capacity of the theorem prover in the late 1970s was inadequate for such an undertaking.

But we persisted in trying to model machines operationally, largely because of the advantages already discussed: (a) such models have a clear correspondence to the artifact being studied, so discrepancies might be easier to spot, (b) such formal models are executable and so can be used as pre-fabrication simulation engines, and (c) with sufficient development of theorem proving strategies it ought to be possible to reason directly about such models and their interpretation of programs.

During the 1980s and 1990s the Boyer-Moore community modeled several microprocessors and other computing machines and developed techniques for managing proofs [22, 3, 46, 4, 23, 7, 45, 9]. The techniques are described in [5]. A good example of the application of these techniques to another commercial language is provided by Yu's work [7] in which 21 of the 22 Berkeley C String Library subroutines were verified by mechanically analyzing the binary code produced by `gcc -o` for a Motorola 68020 model in Nqthm. Those techniques, adapted to the JVM model, are the ones we use today.

In this paper we focus entirely on the use of ACL2 to prove theorems about bytecode programs.

### *6.1 Arithmetic*

No serious software verification application can be undertaken without devoting significant resources to arithmetic, especially integer arithmetic modulo powers of 2. ACL2 has several good integer arithmetic libraries and we are always trying to improve them. In the work reported here we use two of the standard libraries distributed with ACL2, namely
`books/arithmetic/top-with-meta` and
`books/ihs/quotient-remainder-lemmas`.

In addition, we prove a few lemmas establishing basic properties of `intp` and `int-fix`. Then we disable those functions so that we never see `mod` or other low level arithmetic functions introduced by elementary `int` operations like `IADD`, `IMUL`, etc.

```
(defthm int-lemma0
  (implies (intp x)
           (integerp x)))
(defthm int-lemma1
  (intp (int-fix x)))
(defthm int-lemma4a
  (implies (and (integerp x)
                (integerp y))
           (equal (int-fix (* x (int-fix y)))
                  (int-fix (* x y)))))
```

### *6.2 Structures*

Another important step in managing these proofs is to keep the data abstractions in place during the proof process. For example, if left to its own heuristics, the theorem prover will expand `(make-state tt h c)` to `(cons tt (cons h (cons c nil)))` and will expand `(class-table s)` to `(car (cdr (cdr s)))`. Disabling `make-state` and its accessors like `class-table` will prevent this expansion, but will also prevent the simplification of `(class-table (make-state tt h c))` to `c`. Obviously, the appropriate action is to prove the theorems illuminating the relations between these functions and then to disable the functions.

```
(defthm states
  (and (equal (thread-table (make-state tt h c)) tt)
       (equal (heap         (make-state tt h c)) h)
       (equal (class-table  (make-state tt h c)) c)))
(in-theory
 (disable make-state thread-table heap class-table))
```

We take similar steps for the frame constructor, `make-frame`, and its accessors `pc`, `locals`, `stack`, `program`, `sync-flg`, and `cur-class`, and for the stack constructor `push` and its accessors `top` and `pop`.

Despite these arrangements, we sometimes see `car`s and `cdr`s in our proofs. For example, `binding` could be considered an accessor for alists, but we prefer to see it expand into a `cdr` of an `assoc-equal` because we think of `assoc-equal` as the basic primitive for accessing alists.

The standard ACL2 book `data-structures/structures.lisp` provides a convenient way to define structures and automatically obtain the appropriate rewrite rules to preserve the abstractions. The `structures` book was not used in this work for pedagogical reasons.

### *6.3 Mappings*

Alists are used for a variety of things in our representation. But the two most important are the representation of instance objects and the allocation of instance objects to heap addresses. Many bytecode instructions either access or change these alists.

As noted, we regard `assoc-equal` as the primitive function for accessing alists. It is used by both `binding` and `bound?`, both of which are allowed to expand away. Alists are built by `bind`. We have to relate these functions.

```
(defthm assoc-equal-bind
  (equal (assoc-equal key1 (bind key2 val alist))
         (if (equal key1 key2)
             (cons key1 val)
             (assoc-equal key1 alist))))
(defthm bind-bind
  (equal (bind x v (bind x w a))
         (bind x v a)))
```

These theorems are proved automatically by induction. The first lemma "explains" how to determine the value of a key — field name or heap address —after an assignment to some (other?) key. The second lemma allows us to simplify alist expressions by removing obsolete assignments. There are other algebraic laws for manipulating alists, e.g., `bind` is "commutative" if the keys are bound, but we do not need them here.

### *6.4 Semi-Ground Terms*

ACL2 contains heuristics for expanding recursive functions. Basically, these heuristics expand a function if the recursive calls are simpler in some built-in sense. But the heuristics fail to allow some expansions that we consider "obviously smart." These expansions typically deal with functions in which the "controlling" argument is a constant or "nearly" so.

Here is the first of four theorems generating such rules.

```
(defthm bind-formals-opener
  (implies (and (integerp n)
                (<= 0 n))
           (equal (bind-formals (+ 1 n) stack)
                  (cons (top stack)
                        (bind-formals n (pop stack))))))
```

To explain what the rule generated from this lemma does for us, imagine that we are applying the `INVOKEVIRTUAL` semantic function to a method with `3` formals. Inside the body of the semantic function we will encounter the expression

```
(reverse
  (bind-formals (+ nformals 1)
                (stack (top-frame th s))))
```

representing the value of the `locals` component of the new frame being constructed. The rewriter will simplify this from the inside out. Thus, `(+ nformals 1)` will reduce to `4`. The `(stack (top-frame th s))` expression will typically reduce to some expression $\alpha$ that represents the operand stack of the caller. Typically, $\alpha$ will be something like `(push` $v_3$ `(push` $v_2$ `(push` $v_1$ `(push` $obj$ `...))))` because the caller would have pushed the object $obj$ on which the virtual method is to be invoked, as well as the three actuals, onto the operand stack. In any case, after simplifying the interior subterms, the `locals` expression above becomes `(reverse (bind-formals 4` $\alpha$`))`. The definition of `bind-formals` is such that `(bind-formals 4` $\alpha$`)` could be expanded to `(cons (top` $\alpha$`) (bind-formals 3 (pop` $\alpha$`)))` using the definition. But whether it does this expansion is a heuristic decision, since it involves a recursive call of `bind-formals`. In making that decision, ACL2 rewrites the `(pop` $\alpha$`)` expression and then decides whether the recursive call is "simpler." If $\alpha$ is the typical `push` expression, the `pop` expression will be simpler. But if it is not, it may not be — indeed, it will probably be `(pop` $\alpha$`)` which is more complicated than $\alpha$.

But the `bind-formals-opener` rule, shown above, actually avoids all heuristic matters. It says "if the first argument to bind-formals is a positive integer, expand the function." Given inside-out rewriting that means that the `locals` expression becomes

```
(reverse
 (cons (top α)
       (cons (top (pop α))
             (cons (top (pop (pop α)))
                   (cons (top (pop (pop (pop α))))
                         nil)))))
```

which is further reduced to

```
(cons (top (pop (pop (pop α))))
      (cons (top (pop (pop α)))
            (cons (top (pop α))
                  (cons (top α)
                        nil))))
```

by expanding `reverse`. If $\alpha$ is the typical `push` expression this is actually just

```
(cons obj
      (cons v1
            (cons v2
                  (cons v3
                        nil)))).
```

In any case, the rewritten `locals` expression is now normalized so that as we symbolically execute the bytecodes of the invoked method we can determine the values of the locals.

We similarly force open several other functions that are used this way.

```
(defthm nth-opener
  (implies (and (integerp n)
                (<= 0 n))
           (equal (nth (+ 1 n) lst)
                  (nth n (cdr lst)))))
(defthm popn-opener
  (implies (and (integerp n)
                (<= 0 n))
           (equal (popn (+ 1 n) stack)
                  (popn n (pop stack)))))
(defthm repeat-opener
  (implies (and (integerp n)
                (<= 0 n))
           (equal (repeat th (+ 1 n))
                  (cons th (repeat th n)))))
```

We also prove

```
(defthm run-opener
  (and (equal (run nil s) s)
       (equal (run (cons th sched) s)
              (run sched (step th s))))
  :hints (("Goal" :in-theory (disable step))))
```

There are two unusual things about the `defthm` above: it is a conjunct and hints on how to prove it are provided. The conjunct will generate two rules. The first says "expand `run` when the first argument is `nil`." The second says "expand `run` when the first argument is a `cons` (even though you may introduce the more complicated `step`)." The second of these two rules is of the kind we have seen above. But why did we provide the first rule for `run`? More precisely, why did we not provide a comparable rule for `bind-formals`, `nth`, `popn`, and

repeat saying how they behave when the first argument is 0? We did not have to because we leave those functions enabled and so the definition expansion heuristic takes care of the "base cases." But we do not leave run enabled.

If we did, then the first rule would not be necessary. But the cost would be tremendous because every run expression would be considered for expansion under the definition expansion heuristic. The rewriter would tentatively rewrite the step expression inside each run, at great cost, and then reject the expansion for reasons of complexity. It is faster simply never to consider expanding the definition of run. But if we do that, we need the first rule to take care of the base case.

The second oddity in the defthm is that we provided a hint. The hint says "prove this theorem without expanding step. If left to its own, the theorem prover would expand the step expression — producing 139 cases on the unknown next instruction — during the proof attempt. But there is no need to know anything about step to prove this theorem.

### *6.5 The Model*

This brings us to the biggest problem in dealing with a model the size of the JVM. The expansion of step produces a huge case split in which every possible next instruction is considered. We do not want to expand step unless we know what the next instruction is. We therefore prove the following odd theorem — odd because it is just the definition of step with an unnecessary hypothesis!

```
(defthm step-opener
  (implies (consp (next-inst th s))
           (equal (step th s)
                  (if (equal (call-stack-status th s)
                             'SCHEDULED)
                      (do-inst (next-inst th s) th s)
                    s)))
  :hints (("Goal" :in-theory (disable do-inst))))
```

After proving this, we disable step. Thus, the only time step is expanded is if we can show that the next instruction is a cons. Pragmatically what happens is this. The rewriter encounters an instance of (step th s). This is the only rule that matches it and so the rewriter backchains to the hypothesis. Rewriting inside-out, it rewrites (next-inst th s). If we know enough about th ans s to resolve next-inst to some particular instruction, the consp is rewritten to t and the rule fires. Otherwise, the rule does not fire.

Finally, we need to deal with compositions of schedules. The following beautiful theorem is the key.

```
(defthm run-append
  (equal (run (append sched1 sched2) s)
         (run sched2 (run sched1 s))))
```

This theorem is proved automatically, by induction.

## 7 A Simple Correctness Proof

In this section we discuss the proof of correctness of the fact method in class Demo, presented in Section 5 (page 29).

Clearly, the specification for our `fact` method must be that it computes the mathematical factorial function, in some sense. Here is the familiar mathematical function in ACL2.

```
(defun factorial (n)
  (if (zp n)
      1
    (* n (factorial (- n 1)))))
```

If we view the `fact` method as a list of bytecodes, then it is really the JVM that computes factorial, in some sense, when interpreting that bytecode. We must specify pre- and post-conditions and we must phrase them in terms of JVM states and the state transformation described by `run`.

Our correctness statement for the `fact` method is shown below.

```
(defthm fact-is-correct
  (implies (poised-to-invoke-fact th s n)
           (equal
            (run (fact-sched th n) s)
            (modify th s
             :pc (+ 3 (pc (top-frame th s)))
             :stack (push (int-fix (factorial n))
                          (pop (stack (top-frame th s)))))))
  :hints ...)
```

The theorem says that if the next instruction in thread `th` of state `s` is `(INVOKESTATIC "Demo" "fact" 1)` and an `int` `n` is on top of the operand stack, then the result of running `s` according to `(fact-sched th n)` will modify `s` by advancing the `pc` over the `INVOKESTATIC`, popping `n` off the operand stack, and pushing the `int` representation of `(factorial n)`.

This is a remarkable theorem. It is very similar to a semantic function for the `fact` method. If, in any state, we encounter an `(INVOKESTATIC "Demo" "fact" 1)` instruction with $n$ on the stack, and the schedule is as given above, we can simply advance over the `INVOKESTATIC` and modify the operand stack appropriately. We need never consider the bytecode of `fact` again nor need we think about the new frames it actually pushes on the call stack of thread `th`.

The precondition `poised-to-invoke-fact` is a little more complicated than we have sketched.

```
(defun poised-to-invoke-fact (th s n)
  (and (equal (call-stack-status th s) 'SCHEDULED)
       (equal (next-inst th s)
              '(INVOKESTATIC "Demo" "fact" 1))
       (equal n (top (stack (top-frame th s))))
       (intp n)
       (equal (lookup-method "fact" "Demo" (class-table s))
              *fact-def*)))
```

The five conjuncts assert that the thread `th` is scheduled (otherwise, stepping it would be a no-op), the next instruction is the indicated `INVOKESTATIC`, `n` is on top of the stack, `n` is an `int`, and the resolution of `"fact"` in the `"Demo"` class is the bytecode we showed earlier. The last condition is sometimes forgotten. Note that we do not require that the class-table of `s` be completely specified or even that the `"Demo"` class be the one shown.

Before proving `fact-is-correct` it is helpful to illustrate its use, to drive home the point that the correctness theorem is a powerful abstract tool.

Consider the following program.

```
((ICONST_2)                     ; push constant 2
 (ILOAD_3)                      ; push local var 3
 (ICONST_1)                     ; push constant 1
 (IADD)                          ; add the top two
 (INVOKESTATIC "Demo" "fact" 1) ; invoke fact
 (IMUL)                         ; multiply the top two
 (ISTORE_3))                    ; store into var 3
```

What does this program do? This corresponds to the snippet of Java

```
k = 2 * fact(k+1);
```

where `k` is allocated into local variable 3.

Here is a schedule for executing this program in thread 0.

```
(append (repeat 0 4)
        (fact-sched 0 (+ 1 v3))
        (repeat 0 2))
```

We can easily prove that executing the program with the given schedule sets local variable 3 to

```
(int-fix (* 2 (factorial (+ 1 v3))))
```

provided `(+ 1 k)` is an `int`. Here is the theorem.

```
(defthm symbolic-computation
  (implies
   (intp (+ 1 k))
   (equal
    (nth 3
         (locals
          (top-frame
           0
           (run (append (repeat 0 4)
                        (fact-sched 0 (+ 1 k))
                        (repeat 0 2))
                (make-state
                 (make-tt
                  (push
                   (make-frame
                    0
                    (list v0 v1 v2 k)
                    stk
                    '((ICONST_2)
                      (ILOAD_3)
                      (ICONST_1)
                      (IADD)
                      (INVOKESTATIC "Demo" "fact" 1)
                      (IMUL)
                      (ISTORE_3))
                    'UNLOCKED
                    "Test")
                   nil))
```

```
                *demo-heap*
                *demo-class-table*)))))
    (int-fix (* 2 (factorial (+ 1 k)))))))
```

It is important that we understand how this proof goes. So we will will work through it carefully. To make the presentation more succinct it is convenient to introduce the following definition.

```
(defun alpha (pc locals stk)
  (make-state
   (make-tt
    (push (make-frame pc
                      locals
                      stk
                      '((ICONST_2)
                        (ILOAD_3)
                        (ICONST_1)
                        (IADD)
                        (INVOKESTATIC "Demo" "fact" 1)
                        (IMUL)
                        (ISTORE_3))
                      'UNLOCKED
                      "Test")
          nil))
   *demo-heap*
   *demo-class-table*))
```

Thus, the `run` expression in the theorem `symbolic-computation` is just

```
(run (append (repeat 0 4)     ; [1]
      (append (fact-sched 0 (+ 1 k))
              (repeat 0 2)))
     (alpha 0 (list v0 v1 v2 k) stk)).
```

Note that `(append` $a$ $b$ $c$`)` is just an abbreviation for `(append` $a$ `(append` $b$ $c$`))`. We will rewrite [1] to

```
(alpha 9                      ; [12]
       (list v0 v1 v2 (int-fix (* 2 (factorial (+ 1 k)))))
       stk)
```

using the rules shown earlier. It is easy to show that local `3` of the top-frame of thread `0` in [12] is `(int-fix (* 2 (factorial (+ 1 k))))`.

The rewriting shown below is essentially a single-pass of ACL2's inside-out, left-to-right rewriter. We say "essentially" because some steps are commuted or omitted for simplicity but the end result is the same. The justifications below just mention the main rules used.

```
(run (append (repeat 0 4)          ;;; [1]
      (append (fact-sched 0 (+ 1 k))
              (repeat 0 2)))
     (alpha 0 (list v0 v1 v2 k) stk))
=                                  ;;; [2] {repeat-opener}
(run (append '(0 0 0 0)
      (append (fact-sched 0 (+ 1 k))
              '(0 0)))
     (alpha 0 (list v0 v1 v2 k) stk))
=                                  ;;; [3] {run-append}
(run '(0 0)
 (run (fact-sched 0 (+ 1 k))
```

```
  (run '(0 0 0 0)
       (alpha 0 (list v0 v1 v2 k) stk))))
=                                    ;;; [4] {run-opener}
(run '(0 0)
 (run (fact-sched 0 (+ 1 k))
  (step 0
   (step 0
    (step 0
     (step 0
          (alpha 0 (list v0 v1 v2 k) stk)))))))
=                                    ;;; [5] {step-opener}
(run '(0 0)
 (run (fact-sched 0 (+ 1 k))
  (step 0
   (step 0
    (step 0
     (alpha 1 (list v0 v1 v2 k) (push 2 stk)))))))
=                                    ;;; [6] {step-opener}
(run '(0 0)
 (run (fact-sched 0 (+ 1 k))
  (step 0
   (step 0
    (alpha 2 (list v0 v1 v2 k) (push k (push 2 stk)))))))
=                                    ;;; [7] {step-opener}
(run '(0 0)
 (run (fact-sched 0 (+ 1 k))
  (step 0
   (alpha 3
          (list v0 v1 v2 k)
          (push 1 (push k (push 2 stk)))))))
=                                    ;;; [8] {step-opener}
(run '(0 0)
 (run (fact-sched 0 (+ 1 k))
      (alpha 4
             (list v0 v1 v2 k)
             (push (+ 1 k) (push 2 stk)))))
=                                    ;;; [9] {fact-is-correct}
(run '(0 0)
 (alpha 7
        (list v0 v1 v2 k)
        (push (int-fix (factorial (+ 1 k)))
              (push 2 stk))))
=                                    ;;; [10] {run-opener}
(step 0
 (step 0
  (alpha 7
         (list v0 v1 v2 k)
         (push (int-fix (factorial (+ 1 k)))
               (push 2 stk)))))
=                                    ;;; [11] {step-opener}
(step 0
 (alpha 8
        (list v0 v1 v2 k)
        (push (int-fix (* 2 (factorial (+ 1 k))))
              stk)))
=                                    ;;; [12] {step-opener}
(alpha 9
       (list v0 v1 v2 (int-fix (* 2 (factorial (+ 1 k)))))
       stk)
```

There are three key transformations occurring in this proof.

The first is the use of `run-append` (page 39), at equality [3], to decompose long runs into compositions of short ones. The form we choose for the schedule expression determines this decomposition. Schedule expressions can often be written in many equivalent ways. The one above is equal to `(append (repeat 0 6) (fact-sched 0 (+ 1 k)))`, but we chose to write it as we did to decompose the proof into the three separate `run`s on the right hand side of [3].

The second is the repeated use of `step-opener` (page 39) Consider the first use of `step-opener` at equation [5]. On the left hand side of the equation we have

```
(step 0 (alpha 0 (list v0 v1 v2 k) stk)).
```

Applying `step-opener`, we backchain to

```
(consp (next-inst 0 (alpha 0 (list v0 v1 v2 k) stk))).
```

But the `next-inst` expression simplifies to `'(ICONST_2)` because the `pc` of the state is `0` and the $0^{th}$ instruction in the `program` is `'(ICONST_2)`, which is a cons. Since the hypothesis is true, we apply the rule and expand the semantic function for `ICONST_2`. This advances the `pc` to `1` and pushes `2` onto the operand stack, as shown on the right side of the equality at [5]. This *symbolic execution* proceeds as long as we have a `step` expression and can determine what the next instruction is.

The third key transformation is the use of `fact-is-correct` (page 40) at line [9]. The `run` expression on the left side of the equality matches the left-hand side of the `fact-is-correct` rule. Observe also that the `pc` in the state is `4`, which points to the `INVOKESTATIC` instruction. Backchaining leads to the `poised-to-invoke-fact` hypothesis and it rewrites to true in this context. Thus, we apply `fact-is-correct`, advance the `pc` to 7, just past the `INVOKESTATIC`, pop the `(+ 1 k)` off the operand stack and push `(int-fix (factorial (+ 1 k)))`. We can then continue with symbolic execution of the `step` expressions.

One last point is worth making. We proved that [1] is [12]. But we actually derived the state expression at [12] using our rules. That is, rather than prove a theorem, we could use the rewriter to compute the symbolic form of the state created by executing a certain schedule.

### *7.3 Contrasting the Universal Int Program*

Recall the universal `int` program.

```
((ICONST_0)        ;       push 0
 (ICONST_1)        ; loop: push 1
 (IADD)            ;       pop two and push sum
 (GOTO -2))        ;       goto loop
```

Let `(poised-to-invoke-universal th s i)` be defined to check that thread `th` in `s` is about to invoke this program (on no arguments) and that `i` is an arbitrary natural number.

We can prove the following theorem.

```
(defthm universal-is-correct
  (implies (poised-to-invoke-universal th s i)
           (equal (top
                   (stack
                    (top-frame th
                               (run (universal-sched th i) s))))
                  (int-fix i))))
```

This theorem states that if the state `s` is run a certain number of steps, then `(int-fix i)` is left on top of the stack.

It is possible also to prove that any `int` can be produced by supplying `int-fix` with a suitable natural number.

Thus, we can define a universal state, here called `*universal-state*`, poised to invoke the universal `int` program in thread `0` and then prove that if `f` is any `int`-valued function, then there exists a time at which we will find `(f x)` on top of the stack.

```
(defthm universal-computes-f
  (equal (top
          (stack
           (top-frame 0
                      (run (universal-schedule x)
                           *universal-state*))))
         (f x)))
```

Note that in ACL2 it is possible, using the `encapsulate` mechanism, to constrain `f` to be a one argument function satisfying the axiom `(intp (f x))`.

We can use this theorem to prove that the universal `int` program computes factorial, in a suitably twisted sense. To be precise, we can prove

```
(defthm universal-computes-factorial
  (equal (top
          (stack
           (top-frame 0
                      (run (universal-factorial-schedule n)
                           *universal-state*))))
         (int-fix (factorial n)))).
```

Of course, we could prove analogous results establishing that the universal `int` program sums the natural numbers below `n` and that it computes the `int-fix` of the `n`th prime.

What is wrong? In particular, what is wrong with the use of the universal `int` program to compute factorial?

Somewhat troubling is the fact that careful inspection of `universal-factorial-schedule` would reveal that the value of `(factorial n)` is used to determine "when to look for the answer." However, it is often the case that the performance of a program is closely related to the value delivered by the program.

But the theorem above does not tell us that invoking the universal `int` program eventually returns us to the caller. Our `fact-is-correct` does tell us that, though it does it in a subtle way.

```
(defthm fact-is-correct
  (implies (poised-to-invoke-fact th s n)
           (equal
            (run (fact-sched th n) s)
            (modify th s
             :pc (+ 3 (pc (top-frame th s)))
             :stack (push (int-fix (factorial n))
                          (pop (stack (top-frame th s)))))))
  :hints ...)
```

It says that the `pc` is advanced, the answer is on the stack, and nothing else (including the rest of the caller's frame, its caller's frame, etc) has changed. In particular, we can use `fact-is-correct` to reason about a method that calls `fact`. But we cannot use `universal-computes-factorial` to reason about such a method.

The key to proving `fact-is-correct` is to do the right induction. Everything else is "automatic" in the sense that the rules we have discussed suffice to complete the proof.

Here is the theorem we wish to prove.

```
(defthm fact-is-correct
  (implies (poised-to-invoke-fact th s n)
           (equal
            (run (fact-sched th n) s)
            (modify th s
             :pc (+ 3 (pc (top-frame th s)))
             :stack (push (int-fix (factorial n))
                          (pop (stack (top-frame th s)))))))
  :hints ...)
```

Let `(p th s n)` be defined to be the formula above. It is clear that an inductive proof is necessary (the `fact` method and the specification function, `factorial`, are both recursive) and the induction is on `n` (the method and the specification recur on `n` by subtracting `1`). Here is a sketch of an appropriate induction scheme.

```
(and (implies (zp n) (p th s n))          ;;; Base Case
     (implies (and (not (zp n))           ;;; Induction Step
                   (p th s (- n 1)))      ;;;  Induction hyp
              (p th s n)))                ;;;  Induction concl
```

The induction step is conditioned on the test `(not (zp n))`, `n` is not `0`. The induction hypothesis is the conjecture we are trying to prove, with `n` replaced by `(- n 1)`. But in the display above we write $th$ and $s$ for the occurrences of `th` and `s` in the induction hypothesis. Why? The answer is that ACL2's induction principle we are permitted to assume the induction hypothesis for arbitrary values of the variables other than the one(s) we are inducting upon.[3]

What values of $th$ and $s$ do we wish to use? What is the induction hypothesis supposed to tell us? It is supposed to tell us that the recursive invocation of `fact` "works" when it is applied to `(- n 1)`. The choice for $th$ is clear: we should assume the recursive call "works" when it is running in the same thread as the conclusion, `th`. What is the state $s$ the machine will be in when the recursive call of fact is invoked? The answer can be gotten simply by running `s` to the recursive call! That is we produce $s$ by symbolically simplifying `(run (repeat th 7) s)`, under the hypotheses that thread `th` in `s` is poised to invoke `fact` on `n` and that `n` is non-0.

This construction of $s$ can be seen quite clearly if we consider the induction conclusion, `(p th s n)`. This contains `(run s (fact-sched th n))`. If `n` is non-0, then by the definition of `fact-sched` this is equal to

```
(run s (append (repeat th 7)              ;;; [1]
               (fact-sched th (- n 1))
               (repeat th 2)))
```

which is, by `run-append`, just

```
(run (repeat th 2)                        ;;; [1a]
     (run (fact-sched th (- n 1))         ;;; [1b]
          (run (repeat th 7) s))).        ;;; [1c]
```

Note we have three `run` expressions, [1a]-[1c]. The middle `run` expression, [1b], is running

[3]The legitimacy of this is obvious from the standard induction principle and the observation that all our variables are universally quantified.

with the recursively obtained schedule for `(- n 1)`. Notice the state, [1c], in which that `run` starts. That is $s$ and it is determined entirely by the schedule generator we wrote for `fact`.

Here then is our choice of $s$, obtained by simplifying [1c] under the hypotheses given.

```
(make-state                  ;;; [1c'], aka s
 (modify-tt
  th
  (push
   (make-frame 8
               (list (top (stack (top-frame th s))))
               (push (- (top (stack (top-frame th s))) 1)
                     (push (top (stack (top-frame th s)))
                           nil))
               (method-program *fact-def*)
               'UNLOCKED
               "Demo")
   (push (make-frame (+ 3 (pc (top-frame th s)))
                     (locals (top-frame th s))
                     (pop (stack (top-frame th s)))
                     (program (top-frame th s))
                     (sync-flg (top-frame th s))
                     (cur-class (top-frame th s)))
         (pop (call-stack th s))))
  'scheduled
  (thread-table s))
 (heap s)
 (class-table s))
```

Observe that `(poised-to-invoke-fact th` $s$ `(- n 1))` is true. In particular, `pc 8` in the top frame in $s$ points to an `INVOKESTATIC` of `fact` and `(- n 1)` is on top of the operand stack in that top frame. (Many people are surprised by the fact that the top frame of $s$ is not the top frame of `s`. The former is running the `fact` bytecode, while the latter is running some arbitrary caller.)

Hence, the induction hypothesis tells us that [1b] `(run (fact-sched th (- n 1))` $s$`)`, is just $s$ with the `pc` advanced to `11`, the `(- n 1)` popped, and `(int-fix (factorial (- n 1)))` pushed. That is, the induction hypothesis tells us that [1b] is

```
(make-state                  ;;; [1b']
 (modify-tt
  th
  (push
   (make-frame 11
               (list (top (stack (top-frame th s))))
               (push (int-fix
                      (factorial
                       (- (top (stack (top-frame th s)))
                          1)))
                     (push (top (stack (top-frame th s)))
                           nil))
               (method-program *fact-def*)
               'UNLOCKED
               "Demo")
   (push (make-frame (+ 3 (pc (top-frame th s)))
                     (locals (top-frame th s))
                     (pop (stack (top-frame th s)))
                     (program (top-frame th s))
                     (sync-flg (top-frame th s))
                     (cur-class (top-frame th s)))
         (pop (call-stack th s))))
```

```
  'scheduled
  (thread-table s))
 (heap s)
 (class-table s))
```

If we use the induction hypothesis by substituting [1b'] for [1b] in [1a], we reduce [1a] to `(run (repeat th 2)` [1b']). We therefore take two symbolic execution steps. The first executes the `IMUL` at `pc 11` in [1b'], which pops the two items off the operand stack and pushes their product. Given the definition of `factorial` and properties of `int-fix`, we see that the top item on the operand stack after the `IMUL` is `(int-fix (factorial n))`. The second step is the `IRETURN` at `pc 12`. This throws away the top frame of the call stack and pushes the top item of that discarded frame onto the operand stack of the newly exposed frame. The result is

```
(make-state                  ;;; [1c']
 (modify-tt
  th
  (push
   (make-frame (+ 3 (pc (top-frame th s)))
               (locals (top-frame th s))
               (push (int-fix
                      (factorial
                       (top (stack (top-frame th s)))))
                     (pop (stack (top-frame th s))))
               (program (top-frame th s))
               (sync-flg (top-frame th s))
               (cur-class (top-frame th s)))
   (pop (call-stack th s)))
  'scheduled
  (thread-table s))
 (heap s)
 (class-table s))
```

But this is just

```
(modify th s
  :pc (+ 3 (pc (top-frame th s)))
  :stack (push (int-fix (factorial n))
               (pop (stack (top-frame th s)))))
```

as required by `(p th s n)`.

We have just done the proof of the induction step of `fact-is-correct`. The base case is just five steps of symbolic execution.

ACL2 carries out the proof of `fact-is-correct` in 20.93 seconds on a 728 MHz Pentium III running GNU GCL under Redhat Linux.

The most important lesson here is that even though we described the proof operationally, all the symbolic manipulation was done strictly with the definitions of the semantics, the specifications, and our previously proved lemmas. No special-purposes machinery is being used here, other than a powerful theorem prover.

We now turn to a technical matter: how to tell ACL2 to do the induction we just described. Is that required? Yes. ACL2's heuristics use the recursive functions in the conjecture to suggest inductions. But the heuristics do not lead to the induction above. (They lead to a simple induction on `n` without instantiation of `th` or `s`.) The user has to tell ACL2 what induction to do in this case. That is done with the `:hints` argument to the `defthm` event, which we have previously elided away. Here is the full command used to prove `fact-is-correct`.

```
(defthm fact-is-correct
  (implies (poised-to-invoke-fact th s n)
           (equal
            (run (fact-sched th n) s)
            (modify th s
             :pc (+ 3 (pc (top-frame th s)))
             :stack (push (int-fix (factorial n))
                          (pop (stack (top-frame th s)))))))
  :hints (("Goal"
           :induct (induction-hint th s n))))
```

`:Induct` hints are just terms that suggest the induction the user wants. The function `induction-hint` is a recursively defined function that is introduced by the user for the sole purpose of suggesting this induction in this hint. The value of the function is unimportant; all that matters is how it breaks down the cases and how it recurses. The well-foundedness arguments made when the function is admitted under the definitional principle are sufficient to justify the induction it suggests.

Here is the `induction-hint` function, where we have written $s$ in place of the `make-state` expression [1c'] above.

```
(defun induction-hint (th s n)
  (if (zp n)
      s
      (induction-hint th s (- n 1))))
```

We conclude this section with a minor observation. Many users would specify `fact` with this theorem.

```
(defthm weak-version-of-fact-is-correct
  (implies (poised-to-invoke-fact th s n)
           (equal (top
                   (stack
                    (top-frame
                     th
                     (run (fact-sched th n) s))))
                  (int-fix (factorial n)))))
```

That is, all that is required is that it leave the appropriate `int` on top of the stack. By itself, this specification is not only too weak to be proved by induction but it is too weak to permit `fact` to be used as a subroutine by some other method. Consider the possibility that `fact` cleared the operand stack or the local variables. As a matter of fact, those kinds of side-effects on the caller's frame cannot be achieved on the JVM by `INVOKESTATIC`. One could prove general theorems establishing the preservation of the rest of the frame and then cope with this weaker correctness statement. However, our approach is to prove the stronger theorem because the proof is so elegant, and then we can derive the weaker version, if desired, immediately.

## 8 More Complicated Examples

In this section we briefly describe two other Java-related proofs. The first is the correctness of an iterative version of factorial and the second is the correctness of an applicative method for sorting a linked list with insertion sort.

## *8.1 Iterative Factorial*

Here is an iterative factorial program in Java.

```
public static int ifact(int n){
  int temp = 1;
  while (0<n) {
      temp = n*temp;
      n = n-1;
  }
  return temp;
}
```

The associated bytecode produced by `javac` as displayed by `javap -c` is shown below with the ACL2 version printed to the right.

```
Method int ifact(int)
   0 iconst_1             ; (ICONST_1)
   1 istore_1             ; (ISTORE_1)
   2 goto 13              ; (GOTO 11)
   5 iload_0              ; (ILOAD_0)
   6 iload_1              ; (ILOAD_1)
   7 imul                 ; (IMUL)
   8 istore_1             ; (ISTORE_1)
   9 iload_0              ; (ILOAD_0)
  10 iconst_1             ; (ICONST_1)
  11 isub                 ; (ISUB)
  12 istore_0             ; (ISTORE_0)
  13 iload_0              ; (ILOAD_0)
  14 ifgt 5               ; (IFGT -9)
  17 iload_1              ; (ILOAD_1)
  18 ireturn              ; (IRETURN)
```

Observe that the `while` loop starts at `pc 13`, where the `0` local is loaded onto the stack, and proceeds around through `pc 5` and back to `13`. Local variable `1` is being used to accumulate the final product. It is initialized to `1` at `pc 1`, before the loop is entered, and is put on the stack at `pc 17` to be returned by the instruction at `pc 18`.

We can prove the following about this `ifact` method,

```
(defthm ifact-main-result
  (implies (poised-to-invoke-ifact th s n)
           (equal
            (run (ifact-sched th n) s)
            (modify th s
             :pc (+ 3 (pc (top-frame th s)))
             :stack
             (push (int-fix (factorial n))
                   (pop (stack (top-frame th s))))))))
```

where `poised-to-invoke-ifact` is as expected and shown below.

```
(defun poised-to-invoke-ifact (th s n)
  (and (equal (call-stack-status th s) 'SCHEDULED)
       (equal (next-inst th s)
              '(invokestatic "IterativeDemo" "ifact" 1))
       (equal n (top (stack (top-frame th s))))
       (intp n)
       (equal (lookup-method "ifact"
                             "IterativeDemo"
                             (class-table s))
```

```
            *ifact-def*)))
```

There are two interesting things about this proof. The first is that we have to handle a loop. We describe that below.

The second is that instead of proving that it computes `factorial` we prove that it computes another function, `ifactorial`, which is defined in a way to mimic the computation above.

```
(defun ifactorial (n temp)
  (if (zp n)
      temp
    (ifactorial (- n 1) (int-fix (* n temp)))))
```

The use of `ifactorial` allows us to factor the proof into two parts: showing that the bytecode computes `ifactorial` and showing that `ifactorial` is "the same as" `factorial` in a suitable sense.

The analysis of the loop is done, of course, with induction. As usual, the schedule we define identifies the loop.

```
(defun ifact-loop-sched (th n)
  (if (zp n)
      (repeat th 3)
    (append (repeat th 10)
            (ifact-loop-sched th (- n 1)))))
(defun ifact-sched (th n)
  (append (repeat th 4)
          (ifact-loop-sched th n)
          (repeat th 1)))
```

The key idea is that the schedule generated by `ifact-loop-sched` is designed to start when control reaches the top of the loop (`pc 13`) and to drive the machine right through the termination of the loop, leaving the `pc` at the `IRETURN` at `pc 18`.

`Ifact-sched` then takes care of the invocation (building another frame), initializing the `temp` and getting control to the top of the loop. Then it uses `ifact-loop-sched` to finish the loop, and has one more `step` to execute the `IRETURN`, popping the frame it built and returning the value. So the key is handling the loop.

There are two interesting things about this. One is that the "poised" predicate now talks about where the `pc` is, rather than the next instruction. The other is that the induction hypothesis must exhibit the new values of `n` and `temp` one iteration later. The state (called "$s$" earlier) is obtained the same way: by the symbolic execution of the program once around the loop.

Here is the loop property we proved.

```
(defthm ifact-loop-is-correct
  (implies
   (poised-at-ifact-loop th s n)
   (equal
    (run (ifact-loop-sched th n) s)
    (modify
     th s
     :pc 18
     :locals
     (if (zp n)
         (locals (top-frame th s))
       (update-nth 0 0
        (update-nth 1
```

```
                    (int-fix
                     (ifactorial
                      n
                      (nth 1 (locals (top-frame th s)))))
                    (locals (top-frame th s)))))
             :stack
             (push (int-fix
                    (ifactorial
                     n
                     (nth 1 (locals (top-frame th s)))))
                   (stack (top-frame th s))))))
  :hints (("Goal"
           :induct (ifact-loop-induction-hint th s n))))
```

The notion of being poised at the top of the loop is defined as follows.

```
(defun poised-at-ifact-loop (th s n)
  (and (equal (call-stack-status th s) 'SCHEDULED)
       (equal (pc (top-frame th s)) 13)
       (equal (program (top-frame th s))
              (method-program *ifact-def*))
       (equal n (nth 0 (locals (top-frame th s))))
       (intp n)
       (intp (nth 1 (locals (top-frame th s))))))
```

### *8.2 Insertion Sort*

So far we have not dealt with methods that modify the heap or virtual methods. See [33] for the basic work on these topics, in the context of the simpler M3 machine.

Our next M5 proof concerns an applicative implementation of insertion sort in Java. We will represent lists as linked objects, created by the static method `cons`. Each such object has a `car` and a `cdr` field. The former will always be an `int`; the latter will be the null reference or a `cons`. `Conses` are implemented in the `Cons` class.

Then we define the `ListProc` (list processing) class. In this example it only includes two methods, one for inserting an `int` into a linked list by copying the list down to the first element greater than the `int`, and one that uses the insertion method to implement sorting.

Here are the two classes.

```
class Cons {
    int car;
    Object cdr;
    public static Cons cons(int x, Object y){
        Cons c = new Cons();
        c.car = x;
        c.cdr = y;
        return c;
    }
}
class ListProc extends Cons {
    public static Cons insert(int e,Object x){
        if (x==null)
            {return cons(e,x);}
        else if (e <= ((Cons)x).car)
            {return cons(e,x);}
        else return cons(((Cons)x).car,
                         insert(e,((Cons)x).cdr));
```

```
    }
    public static Object isort(Object x){
        if (x==null)
            {return x;}
        else return insert(((Cons)x).car,
                           isort(((Cons)x).cdr));
    }
}
```

Here is our bytecode for `cons`. It is exactly as created by `javac`.

```
(defconst *cons-def*
  '("cons" (int java.lang.Object) nil
    (NEW "Cons")
    (DUP)
    (INVOKESPECIAL "Cons" "<init>" 0)
    (ASTORE_2)
    (ALOAD_2)
    (ILOAD_0)
    (PUTFIELD "Cons" "car")
    (ALOAD_2)
    (ALOAD_1)
    (PUTFIELD "Cons" "cdr")
    (ALOAD_2)
    (ARETURN)))
```

This is the first method we have dealt with that writes to fields in the heap.

Here is our bytecode for `insert`. It is as created by `javac` with one exception: we have deleted three `checkcast` instructions because they are not supported by our model at the moment (and, in this method, never throws an exception).

```
(defconst *insert-def*
  '("insert" (int java.lang.Object) nil
    (ALOAD_1)
    (IFNONNULL 9)             ;;; if nonnull goto label1
    (ILOAD_0)
    (ALOAD_1)
    (INVOKESTATIC "Cons" "cons" 2)
    (ARETURN)
                          ;;; label1
    (ILOAD_0)
    (ALOAD_1)
                          ;;; checkcast "Cons" omitted
    (GETFIELD "Cons" "car")
    (IF_ICMPGT 9)            ;;; if gt goto label2
    (ILOAD_0)
    (ALOAD_1)
    (INVOKESTATIC "Cons" "cons" 2)
    (ARETURN)
                          ;;; label2
    (ALOAD_1)
                          ;;; checkcast "Cons" omitted
    (GETFIELD "Cons" "car")
    (ILOAD_0)
    (ALOAD_1)
                          ;;; checkcast "Cons" omitted
    (GETFIELD "Cons" "cdr")
    (INVOKESTATIC "ListProc" "insert" 2)
    (INVOKESTATIC "Cons" "cons" 2)
```

```
    (ARETURN)))
```

Here is the bytecode for `isort`.

```
(defconst *isort-def*
  '("isort" (java.lang.Object) nil
    (ALOAD_0)
    (IFNONNULL 5)          ;;; if nonnull goto label
    (ALOAD_0)
    (ARETURN)
                           ;;; label
    (ALOAD_0)
                           ;;; checkcast "Cons" omitted
    (GETFIELD "Cons" "car")
    (ALOAD_0)
                           ;;; checkcast "Cons" omitted
    (GETFIELD "Cons" "cdr")
    (INVOKESTATIC "ListProc" "isort" 1)
    (INVOKESTATIC "ListProc" "insert" 2)
    (ARETURN)))
```

Again, we have omitted two `checkcast` instructions.

A crucial concept in dealing with the heap is the idea of "chasing pointers" through it. In particular, we define `deref*` so that if given a reference to a `cons` (or `null`) and a heap it returns the ACL2 list obtained by consing together the successive `car`s of the `cons`es encountered by `deref` and recursively `deref`ing the `cdr`s. We call this *recursive dereferencing*.

In order for the operation just described to be well-defined, `deref*` must not encounter circular pointer references. We define an invariant on the heap that precludes this possibility and we prove that the heap generated by `cons`, `insert`, and `isort` preserves the invariant. In particular, this tells us that every reference in the `cdr` of a `cons` is to an address smaller than the address of the `cons`. The definition of `deref*` is shown below.

```
(defun deref* (xref heap)
  (declare (xargs :measure (ref-measure xref)))
  (cond
   ((nullrefp xref) nil)
   ((not (and (heap-invariantp heap)
              (ok-refp xref heap)))
    nil)
   (t (cons (car-heap xref heap)
            (deref* (cdr-heap xref heap) heap)))))
```

`Car-heap` and `cdr-heap` fetch the corresponding fields of the `cons` object indicated by the given reference.

Here is an execution of `isort`, demonstrating it sorting the list `(3 2 1)` into the list `(1 2 3)`.

```
(defthm isort-3-2-1
  (let* ((s0
          (make-state
           (list
            (cons 0
                  (make-thread
                   (push
                    (make-frame
                     0
                     '((REF -1))
                     nil
                     '((BIPUSH 3)
```

```
                    (BIPUSH 2)
                    (BIPUSH 1)
                    (ALOAD_0)
                    (INVOKESTATIC "Cons" "cons" 2)
                    (INVOKESTATIC "Cons" "cons" 2)
                    (INVOKESTATIC "Cons" "cons" 2)
                    (INVOKESTATIC "ListProc" "isort" 1)
                    (HALT))
                  'UNLOCKED
                  "ListProc")
                 nil)
                'SCHEDULED
                nil)))
          *isort-heap0*
          *isort-class-table*))
        (s1
         (run (append (repeat 0 4)
                      (cons-sched 0)
                      (cons-sched 0)
                      (cons-sched 0))
              s0))
        (sched (isort-sched 0
                            (top (stack (top-frame 0 s1)))
                            (heap s1)))
        (s2 (run sched s1)))
   (and (equal (deref* (top (stack (top-frame 0 s1)))
                       (heap s1))
               '(3 2 1))
        (equal (len sched) 188)
        (heap-invariantp (heap s2))
        (equal (deref* (top (stack (top-frame 0 s2)))
                       (heap s2))
               '(1 2 3))
        (equal (next-inst 0 s2) '(HALT))))
 :rule-classes nil)
```

The theorem, which is proved by execution, constructs an initial state `s1` that has the list `(3 2 1)` built in the heap and is poised to invoke `isort` on a reference to that list. Then we build a schedule, `sched`, suitable for evaluating the call of `isort`. We then create `s2` by running `s1`. The conclusion of the theorem makes certain observations. The `deref*` of the input is `(3 2 1)`. The schedule has length `188`. The heap invariant holds of the final heap in `s2`. The `deref*` of the output of `isort` is `(1 2 3)`. The next instruction is the `(HALT)` instruction.

Of course, the theorem above is not very interesting except to illustrate the execution of a formal specification.

But here is a much more interesting theorem. Roughly speaking, it says that `isort` produces an ordered permutation of its input.

```
(defthm main-isort-theorem
  (let ((x0 (top (stack (top-frame th s))))
        (heap0 (heap s)))
    (implies (poised-to-invoke-isort th s x0 heap0)
             (let* ((sched (isort-sched th x0 heap0))
                    (s1 (run sched s))
                    (x1 (top (stack (top-frame th s1))))
                    (heap1 (heap s1)))
               (let ((list0 (deref* x0 heap0))
                     (list1 (deref* x1 heap1)))
                 (and (ordered list1)
```

```
                    (perm list1 list0))))))
  :rule-classes nil)
```

Suppose `x0` is the object on top of the stack in state `s` and `heap0` is the heap of state `s`. Suppose thread `th` is poised to invoke `isort` on `x0`. Let `sched` be a suitable schedule. (Note that the schedule is a function of the reference and the heap, since we have to chase pointers to determine how long `isort` will run.) Let `s1` be the result of running `s`. Let `x1` be the item left on top of the stack at the end of the run and let `heap1` be the final heap. Let `list0` be the result of dereferencing `x0` recursively with respect to its heap, `heap0`. Let `list1` be the result of dereferencing `x1` recursively with respect to its heap `heap1`. Then `list1` is an ordered permutation of `list0`.

We proved this theorem by proving first that the execution of `isort` bytecode produced the heap created by the following function.

```
(defun isort-heap (xref heap)
  (declare (xargs :measure (ref-measure xref)))
  (cond
   ((nullrefp xref)
    heap)
   ((not (and (heap-invariantp heap)
              (ok-refp xref heap)))
    heap)
   (t
    (insert-heap (car-heap xref heap)
                 (if (nullrefp (cdr-heap xref heap))
                     '(ref -1)
                   (list 'ref
                         (- (len (isort-heap
                                  (cdr-heap xref heap)
                                  heap))
                            1)))
                 (isort-heap (cdr-heap xref heap)
                             heap)))))
```

This function is to the `isort` method what `ifactorial` is to `ifact`: an expression of an algorithm.

And then we proved the following theorem relating this heap to a simple insertion sort function in ACL2.

```
(defthm deref*-isort-heap
  (implies (and (heap-invariantp heap)
                (ok-refp xref heap)
                (not (nullrefp xref)))
           (equal (deref*
                   (list 'ref
                         (- (len (isort-heap xref heap)) 1))
                   (isort-heap xref heap))
                  (isort (deref* xref heap)))))
```

Consider the heap created by `(isort-heap xref heap)` and the reference to its biggest address. We call these two quantities the output heap and the output reference. The theorem above says that the result of recursively dereferencing the output reference in the output heap is the same list produced by sorting the result of recursively dereferencing input reference in the input heap. Here, the function `isort`, is defined simply as shown below.

```
(defun isort (x)
  (if (endp x)
      nil
      (insert (car x)
```

```
            (isort (cdr x)))))
```

It is then a simple matter to prove that `isort` returns an ordered permutation of its input.

The two-step methodology mentioned here is crucial. To verify code, first prove that implements some algorithm that simply abstracts away from the control (and possibly data) of the particular computational paradigm or programming language. Then verify that the algorithm has the desired properties.

Despite the brevity with which we presented it, the `isort` proof is an interesting challenge, because the schedules are a function of the heap. It should be noted that the majority of the work, however, was in creating a useful set of lemmas making it easy to establish that methods that modify the heap only with our `cons` method preserve the invariant and that once the invariant is known, it is relatively easy to maintain the isomorphism between methods on the heap and functions manipulating the data represented.

## 9 Multi-Threading

```
class Container {
    public int counter;
}
class Job extends Thread {
    Container objref;
    public Job incr () {
        synchronized(objref) {
            objref.counter = objref.counter + 1;
        }
        return this;
    }
    public void setref(Container o) {
        objref = o;
    }
    public void run() {
        for (;;) {
            incr();
        }
    }
}
class Apprentice {
    public static void main(String[] args) {
        Container container = new Container();
        Container bogus = new Container();
        for (;;) {
            Job job = new Job();
            Job.setref(bogus);
            job.start();
            job.setref(container);
        }
    }
}
```

Figure 3: A Bad Apprentice

In this section we will deal briefly with a multi-threaded application and we will describe the proof of a safety progress involving mutual exclusion. The proof involves reasoning about arithmetic, infinite loops, the creation and modification of instance objects in the heap, including threads, the inheritance of fields from superclasses, pointer chasing and smashing,

the invocation of instance (virtual as opposed to static) methods (and the concomitant dynamic method resolution), use of the `start` method on thread objects, the use of monitors to attain synchronization between threads, and consideration of all possible interleavings (at the bytecode level) over an unbounded number of threads. The proof is described in detail in [35].

Readers familiar with monitor-based proofs of mutual exclusion will recognize our proof as fairly classical. The novelty here comes from (i) the complexity of the individual operations on the abstract machine, (ii) the dependencies between Java threads, heap objects, and synchronization, (iii) the bytecode-level interleaving, (iv) the unbounded number of threads, (v) the presence in the heap of incompletely initialized threads and other objects, and (vi) the proof engineering permitting automatic mechanical verification of code-level theorems.

Figure 3 shows little system of Java classes. The `main` program in the `Apprentice` class spawns an unbounded number of threads, each of which is running a `Job`. Each `Job` is in an infinite loop applying the `incr` method to the self object (the heap manifestation of the `Job`). The `incr` method locks the object in the `objref` field of the self object and then proceeds to increment a `counter` field within that locked object.

As shown in Figure 3, the `Apprentice main` program puts the same object in the `objref` field of every `Job`. But it does it in a strange way. It sets the `objref` field to a "bogus" container, then it starts the `Job`, and then it sets the field to the "good" container. So we have an unbounded number of threads in eternal contention for a couple of objects and each is attempting to lock the object before modifying it.

Does the `counter` of the object go up? Obviously, the counter wraps around, because it is a 32-bit `int`. But, with the caveat about wrapping around, one might think the counters in both containers increase monotonically since the object is locked before it is modified in `incr`. In particular, the line

```
synchronized(objref) {
    objref.counter = objref.counter + 1;
}
```

is pretty reassuring.

But the bytecode for the `incr` method is shown below.

```
("incr"                          ;  incr method
 nil                             ;  parameters (none)
 nil                             ;  synchronization flag
 (ALOAD_0)                                          ;  0
 (GETFIELD "Job" "objref")                          ;  1
 (ASTORE_1)                                         ;  4
 (ALOAD_1)                                          ;  5
 (MONITORENTER)                                     ;  6
 (ALOAD_0)                                          ;  7 *
 (GETFIELD "Job" "objref")                          ;  8 *
 (ALOAD_0)                                          ; 11 *
 (GETFIELD "Job" "objref")                          ; 12 *
 (GETFIELD "Container" "counter")                   ; 15 *
 (ICONST_1)                                         ; 18 *
 (IADD)                                             ; 19 *
 (PUTFIELD "Container" "counter")                   ; 20 *
 (ALOAD_1)                                          ; 23 *
 (MONITOREXIT)                                      ; 24 *
 (GOTO 8)                                           ; 25
 (ASTORE_2)                                         ; 28
 (ALOAD_1)                                          ; 29
 (MONITOREXIT)                                      ; 30
 (ALOAD_2)                                          ; 31
```

```
(ATHROW)                                        ; 32
(ALOAD_0)                                       ; 33
(ARETURN))
```

When `incr` is invoked, the heap manifestation of the `Job` instance is passed into local variable `0`. So the first three instructions store the current contents of the `objref` field into local `1`. This will either be the bogus container or the good container, depending on whether `main` method of `Apprentice` has reset the `objref` field yet. Assume it is the bogus container.

In the next two instructions (`pc 5` and `pc 6`), `incr` gets the lock on the value of local `1`, the bogus container. It then enters its critical section, the body of the `synchronized` block, marked with * above.

Note which container it increments: the one in the `objref` field of local `0`, not necessarily the one on which it is holding the lock. Local `1` is only used when it releases the lock.

The reassurance gained by looking casually at

```
synchronized(objref) {
    objref.counter = objref.counter + 1;
}
```

is completely bogus.

The `counter` of the good container can decrease under some schedules. For example, run thread `0` just enough to create and start the first `Job` (in thread `1`). Then run thread `1` enough to lock the bogus container. Now run thread `0` again so that it resets the `objref` of the first `Job` to the good container. Now run thread `1` again to execute instructions `7` through `19`. This will leave it ready to write a `1` into the `counter` field of the good container. But the `Job` is holding a lock on the bogus container. So, next, schedule thread `0` to create a second `Job`, set its `objref` to the bogus container, start the `Job`, and set its `objref` to the good container. Then schedule thread `2`, the second `Job` to run thousands of steps, incrementing the `counter` field of the good container to a large positive integer. Finally, schedule thread `1` to execute its next instruction, the `PUTFIELD` at `19`. It will write a `1` where just before there was a large positive integer: the `counter` field of the good container can decrease.

The sense in which the reassurance of the `synchronized` block is bogus is that we must inspect the entire system and investigate how the `objref` fields of `Job`s are being manipulated. That is, the mutual exclusion we might have thought we were getting from the `synchronized` block actually depends on the disciplined but un-enforced handling of the `Job` objects themselves. The synchronization is occurring on objects in the heap, not static variables, and those objects can be changed.

If we change `Apprentice` so that it reads

```
class Apprentice {
    public static void main(String[] args) {
        Container container = new Container();
        for (;;) {
            Job job = new Job();
            job.setref(container);
            job.start();
        }
    }
```

then we can prove that the `counter` field of `container` increases monotonically (modulo wraparound). This proof is described in detail in [35].

The theorem we prove is shown below.

```
(defthm Monotonicity
```

```
  (let* ((s1 (run sched *a0*))
         (s2 (step th s1)))
    (implies (not (equal (counter s1) nil))
             (or (equal (counter s1)
                        (counter s2))
                 (equal (int-fix (+ 1 (counter s1)))
                        (counter s2)))))
  ...)
```

In this theorem, `*a0*` is the JVM state constant obtained by mechanically translating the Java classes `Container`, `Job` and (the corrected) `Apprentice`, into our formalism. The `*a0*` state contains only one thread, `0`, poised to begin execution of the `main` program of `Apprentice`. Its initial heap contains only the heap manifestations of the built in and declared classes.

The theorem lets `s1` and `s2` be two successive states and makes a claim about them. The first state, `s1`, is an arbitrary reachable state, that is, is the state reached from `*a0*` by executing an arbitrary schedule `sched`. The second state, `s2`, is the successor to `s1` obtained by stepping any thread `th`.

The claim is that if the `counter` value in `s1` is non-`nil` — i.e., `sched` has executed thread `0` enough to have created and initialized the `Container` — then either the `counter`s in `s1` and `s2` are the same or else the `counter` in `s2` is one greater than the `counter` in `s1` (modulo `int` arithmetic).

This is not an easy theorem to prove. It requires the definition of an invariant on states characterizing the reachable states. We call the invariant `good-statep`. The basic idea is that the `objref` field of every scheduled `Job` points to the container and that if any thread is in the critical section of its `incr` thread then that thread holds the lock on the container. These obvious requirements must then be propagated so that they are preserved. For example, just before a thread executes the `MONITORENTER`, the object on top of the stack must be the container.

Once we have defined `good-state` we prove four lemmas.

```
(defthm [1]
  (good-state *a0*)
  ...)
(defthm [2]
  (implies (good-state s)
           (good-state (step th s)))
  ...)
(defthm [3]
  (implies (good-state s)
           (implies (not (equal (counter s) nil))
                    (or (equal (counter s)
                               (counter (step th s)))
                        (equal (int-fix (+ 1 (counter s)))
                               (counter (step th s)))))))
  ...)
(defthm [4]
  (good-state (run sched *a0*)))
```

`[1]` and `[2]` (and an easy inductive lemma) are used to prove `[4]`. `[3]` and `[4]` can be combined to get `Monotonicity`.

The crux of the proof is, of course, `[2]`, which states that `good-state` is invariant under `step`s by an arbitrary thread `th`. The proof of `[2]` is divided up into three lemmas depending on `th`. The first lemma, `[2a]`, deals with the case where `th` is 0 (the `main` thread). The second

lemma, [2b], deals with the case that th is the thread of a scheduled Job. The third lemma, [2c], deals with all the other cases (i.e., when th is unscheduled or is a non-existent thread).

The proofs of [2a] and [2b] are very similar. Symbolic execution is used: start from any possible good-state (consider the cases), symbolically execute the next instruction, and check that good-state approves.

In all, 39 defuns are made to formalize the problem (including good-state and its sub-functions) and 75 theorems are proved leading to Monotonicity.

## 10 Ongoing ACL2 Work

Space does not permit the discussion of much ongoing ACL2 work related to the JVM. However, we will briefly note some of it. All of the people mentioned below are graduate students at the University of Texas at Austin. Except where noted, all are in the Computer Sciences department.

Hanbing Liu is working on verifying the bytecode verifier with respect to a model like this one but with runtime error checks. This work is also driving the creation of a still-more authentic model of the JVM, including support for exceptions and other features.

Jeff Golden is working on tools to help create schedule functions and invariants to make code verification easier.

Jeff Golden and Sandip Ray are working on the correctness proof for the *in situ* Java quicksort method provided in the standard Java API. They are essentially lifting a proof done earlier by Rob Sumners (ECE Department and AMD) and Ray on the *in situ* quicksort algorithm [42].

Rob Sumners is also using methods similar to those described in [29] to obtain a progress property for the Apprentice system based on the invariant proved.

In [28] we describe a method for introducing tail-recursive functions that do not necessarily terminate. Such functions can be used to model single-threaded machines without introducing the notion of a "clock" or "schedule" and without having to count the number of instructions executed. But our schedules in M5 do more than just provide instruction counts. They allow us to deal with the nondeterminism inherent in multi-threading by specifying which thread steps next. It is not clear that the methods of [28] will aid the formal analysis of multi-threaded code. In addition, our schedules (and the analogous "clock functions" we use in single-threaded models) allow us to structure inductive proofs.

Finally, work reported in [32] suggests a way we might approach the verification of multi-threaded code by lifting the view of the system to a single-threaded machine in which "spontaneous" changes are visited upon the shared resources visible to the thread in question. We have not yet tried to apply that work to M5.

## 11 Conclusion

We have presented a detailed operational model of a significant subset of the Java Virtual Machine. We have used the model to execute certain Java programs by compiling them into bytecode. We have described how the ACL2 theorem prover can be configured to help prove theorems about bytecode programs. We have illustrated correctness proofs of several Java methods, including recursive and iterative factorial (over the ints), the implementation of cons and linked lists in Java and the use of that class to implement insertion sort, and a multi-threaded application that spawns an unbounded number of threads in contention for a shared object in the heap.

Our correctness theorems generally provide a constructive measure of how long the programs run, as a function of the input data. We have shown how these schedule functions can be composed to give constructive schedules for larger methods. In some instances the constructive schedules can be eliminated entirely.

Our model and our theorems probably appear "too concrete" to many readers. This is in part due to the constructive logic in which we work and perhaps to the unfamiliar notation we use. But find the detail and authenticity appealing.

Most importantly, we believe that work like this — and the improved theorem proving capacity it demands — is crucial to the adoption of formal mechanized tools for facilitating code proofs.

We see two reasons. First, programming languages come and go. The work involved in building a verification system for one of them is enormous and that work is error-prone and obscure. Techniques such as those described here will eventually enable general purpose theorem provers such as ACL2, Coq [12], HOL [16], or PVS [37], to be used as special-purpose reasoning engines for particular programming languages. In principle, it should be possible to develop a code verification system for a given language merely by formalizing the programming language and then reasoning with a general-purpose tool. This is not only safer — avoiding as it does the possibility of bugs in the special-purpose verification engine — but should make it easier to produce code-verification tools for new or niche-market languages.

Second, these methods will allow to "verify the verifiers" if special-purpose verifiers are built for a given language. This is already happening with the investigations into the Java bytecode verifier.

Third, the direct application of a general-purpose theorem prover to problems of this scale and complexity is an excellent way to drive theorem-proving research and increase our capabilities and capacities for mechanically checked formal reasoning.

## References


[1] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the JavaCard platform. In D. Sands, editor, *ESOP 2001*, volume LNCS 2028, pages 302–319, Heidelberg, 2001. Springer-Verlag.

[2] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques (WADT'99)*, number 1827 in LNCS, pages 1–21, Heidelberg, 2000. Springer.

[3] W. R. Bevier. A verified operating system kernel. Ph.d. dissertation, University of Texas at Austin, 1987.

[4] W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[5] R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176, Cambridge, MA, 1996. MIT Press.

[6] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.

[7] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[8] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - a second generation of a Java model checker. In *post-CAV 2000 Workshop on Advances in Verification, Chicago, IL.*, Moffett Field, CA., July 2000. `http://ase.arc.nasa.gov/jpf/wave00.ps.gz`.

[9] B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.

[10] R. M. Cohen. The defensive Java Virtual Machine specification, Version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[11] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report TR 159, Compaq Systems Research Center, December 1998.

[12] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. The Coq proof assistant user's guide, Version 5.6. Technical Report TR 134, INRIA, December 1991.

[13] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, Los Alamitos, CA., May 2001. IEEE Computer Society Press.

[14] R. Gamboa and B. Middleton. Taylor's theorem with remainder. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002`, Grenoble, April 2002.

[15] G. Gentzen. New version of the consistency proof for elementary number theory. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 132–213. North-Holland Publishing Company, Amsterdam, 1969.

[16] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[17] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[18] D. Greve and M. Wilding. Evaluatable, high-assurance microprocessors. In *NSA High-Confidence Systems and Software Conference (HCSS)*, Linthicum, MD, March 2002. `http://hokiepokie.org/docs/hcss02/proceedings.pdf`.

[19] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In Kaufmann et al. [24], pages 113–136.

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

[21] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design – FMCAD*, LNCS 1522, Heidelberg, 1998. Springer-Verlag.

[22] W. A. Hunt. *FM8501: A Verified Microprocessor*. Springer-Verlag LNAI 795, Heidelberg, 1994.

[23] W.A. Hunt and B. Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, April 1992.

[24] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

[25] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[26] K. R. M. Leino, G. Nelson, and J. B. Saxe. Esc/java user's manual. Technical Report Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[27] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, Boston, MA., 1999.

[28] P. Manolios and J S. Moore. Partial functions in acl2. Technical Report `http://-www.cs.utexas.edu/users/moore/publications/defpun/% -index.html`, Computer Sciences, University of Texas at Austin, 2001.

[29] P. Manolios, K. Namjoshi, and R. Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In *Computed Aided Verification, CAV '99*, pages 369–379, Heidelberg, 1999. Springer-Verlag LNCS 1633.

[30] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical Report TR 4215, Computer Science Department, University of Maryland, May 2001. `http://www.cs.umd.edu/~pugh/java/-memoryModel/semantics.pdf`.

[31] John McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.

[32] J S. Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design*, 14(2):213–228, March 1999.

[33] J S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, pages 139–162, Heidelberg, 1999. LNCS 1710.

[34] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[35] J S. Moore and G. Porter. The apprentice challenge. *ACM TOPLAS*, 24(3):1–24, May 2002.

[36] T. Nipkow and L. C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, Heidelberg, 1992. Springer-Verlag LNAI 607. System abstract.

[37] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Heidelberg, June 1992. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag.

[38] C. Pusch. Formalizing the Java virtual machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universiät München, 1998. See URL `http://www.in.tum.de/~pusch/`.

[39] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://www.onr.com/user/-russ/david/k7-div-sqrt.html.

[40] D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In Kaufmann et al. [24], pages 201–232.

[41] J. Sawada. Formal verification of divide and square root algorithms using series calculation. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002`, Grenoble, April 2002.

[42] R. Sumners and S. Ray. Verification of an in-place quicksort in ACL2. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002`, Grenoble, April 2002.

[43] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS 2001*, volume LNCS 2031, pages 299–313, Heidelberg, 2001. Springer.

[44] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE)*, pages 3–12, Grenoble, France, September 2000. IEEE Computer Society.

[45] M. Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Heidelberg, 1993. Springer-Verlag. See URL ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/wilding-cav93.ps.

[46] W. D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Comp. Logic. Inc., Austin, Texas, 1988.