Mechanized Operational Semantics: The M1 Story

J Strother Moore
Department of Computer Sciences,
University of Texas at Austin,
Taylor Hall 2.124,
Austin, Texas 78712

DRAFT

1 Abstract

In this paper we explain how to formalize an "operational" or "state-transition" semantics of a von Neumann programming language in a functional programming language. By adopting an "interpretive" style, one can execute the model in the functional language to "run" programs in the von Neumann language. Given the ability to reason about the functional language, one can use the model to reason about programs in the von Neumann language.

In theory at least, such a formal semantics thus has a dual use: as a simulation engine and as an axiomatic basis for code proofs.

The beauty of this approach is that no more logical machinery is needed than to support execution and proof in a functional language: no new program logics and no new meta-logical tools like "verification condition generators" are needed.

In this paper we will illustrate the techniques by formalizing a simple programming language called "M1," for "Machine (or Model) 1." It is loosely based on the Java Virtual Machine but has been simplified for pedagogical purposes. We will demonstrate the executability of M1 models. We will develop several styles of code proofs, including direct (symbolic simulation) proofs based on Boyer-Moore "clock functions" and Floyd-Hoare inductive assertion proofs. We construct proofs only for the the simplest of programs, namely an iterative factorial example. But to illustrate a more realistic use of the model, we discuss the correctness proof for an M1 implementation of the Boyer-Moore fast string searching algorithm.

We also define a compiler for a higher level language called "J1" and show how to do proofs about J1 code without benefit of a formal semantics for that code. Throughout we use the ACL2 logic and theorem proving system.

2 Preface

The most widely accepted meaning of "operational semantics" today is Plotkin's "Structural Operational Semantics" (SOS) [22] in which the semantics is presented as a set of inference rules on syntax and "configurations" (states) defining the valid transitions.

But in these lectures I take an older approach perhaps best called “interpretive semantics,” in which the semantics of a piece of code is given by a recursively defined interpreter on the syntax and a state.

I suspect the older approach came from McCarthy who wrote “the meaning of a program is defined by its effect on the state vector,” in his seminal paper,“Towards a Mathematical Science of Computation” (1962).

The interpretive approach was used with mechanized support in *A Computational Logic* (Boyer and Moore, 1979) to specify and verify an expression compiler. The low level machine was defined as a recursive function on programs (sequence of instructions) against a state consisting of a push down stack and an environment assigning values to variables. We also used the approach to formalize the semantics of the instruction set of the BDX 930 flight control computer as part of the SIFT project at SRI [7] – an exercise that ultimately failed because our theorem prover at the time was not capable of representing states as large and complex as arose in the BDX 930. That motivated us to introduce `quote` and “executable counterparts,” in the late 1970s, but that is another story.

Plotkin rightly states that the interpretive approach tends to produce large and possibly unweildy states. Procedure call and non-determinism make things even worse. But this is mitigated by the presence of a mechanized reasoning system. Interpretive semantics also confer certan advantages I will discuss herein.

The Boyer-Moore community has used operational semantics (in the “interpretive” sense) with great success since the mid-1970s, including, for example, the verification of the CLI Stack [1], the verification of the Berkeley C String Library via compilation to MC68020 machine code with `gcc`, and proofs about JVM bytecode [14].

3 Introduction

In this paper we explain how to formalize the semantics of a von Neumann programming language in a functional programming language.

The entire project will be carried out in a mechanized logic, namely, with the ACL2 theorem proving system. “ACL2” stands for “A Computational Logic for Applicative Common Lisp.” The programming language supported by ACL2 is a very large, functional (or “applicative”) subset of ANSI Standard Common Lisp. The semantics of that programming language is formalized via axioms and definitions within a first-order mathematical logic with induction. A mechanized theorem proving environment supports the discovery of proofs in the theory. We use the name ACL2 for all three of these aspects: the programming language, the logical theory, and the theorem prover.

ACL2 is distributed under the Gnu General Public License and is fully documented online [12]. To learn how to use ACL2, see [10].

The von Neumann language used to illustrate this will be a very simplified version of the Sun Java Virtual Machine we call “M1.” M1 is the first in a series of models of the JVM culminating in M6: an elaborate and accurate model of the JVM supporting multiple threads, class loading, exceptions, etc. The formalization and proof techniques used in M1 were the guiding principles in the construction of the M6 model. But M6 is over 160 pages of ACL2 formalism (not counting another 500 pages representing the M6 translation of the 672 methods and 87 classes in the Sun CLDC API library). In contrast, M1 is less than 3 pages.

This paper is very similar to the author’s “Proving Theorems about Java and the JVM with

ACL2" [20]. But the earlier paper discussed a fairly complicated JVM model, M5, whereas this one uses M1. The differences made by exploring a simpler model are sufficiently startling to bear the creation of this near repetition of the earlier paper.

Unlike the former paper, we also model a compiler and we explore inductive assertion style proofs here.

Associated with this paper is an ACL2 script that contains exactly the definitions and theorems shown here, in the same sequence. With a few minor exceptions the paper is a complete description of what must be presented to ACL2 Version 3.3 to construct the proofs. The exceptions concern the precise definition of the "M1 symbol package" (a Common Lisp construct used to avoid name conflicts) and the correctness proof for the Boyer-Moore fast string searching implementation. The missing definitions and theorems are in the associated ACL2 script (and the ACL2 "books" it loads).

With regard to the string searching proof, it may be broken down into two steps: prove that the M1 code implements the Boyer-Moore algorithm and then prove that the Boyer-Moore algorithm is a correct string searching algorithm. Both parts have been done with ACL2. But in this paper, we only discuss the first part. In particular, we use the direct ("clock function") approach to prove that certain M1 code implements a new and improved variant of the Boyer-Moore fast string searching algorithm. The proof *assumes* that the preprocessing of the pattern has been performed to set up a correctly initialized 2-dimensional array. The code we analyze does not do the preprocesssing. We can combine the M1 code proof with the algorithm correctness proof to show that the M1 code is a correct string searching program (assuming the proper preprocessing has been done).

4 The ACL2 Programming Language

ACL2 is a functional programming language, a mathematical logic, and an automatic theorem prover. For the moment, we will focus entirely on the functional programming language.

4.1 Syntax

The syntax of ACL2 is that of Lisp. Here is how the user introduces a new function, in this case, the factorial function:

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

We write

```
(* n (fact (- n 1)))
```

where more traditionally one would write

$$n * fact(n - 1).$$

The `defun` command above defines `fact` to be a function of one argument, `n`, whose value is determined by the `if`-expression.

The 3-argument function `if` is Lisp's *if-then-else*; `(if` x y z`)` is y if x is non-`nil`, and is z if x is `nil`. The Lisp convention of "testing against `nil`" treats the object `nil` as "false" and treats all other objects as "true." When we say "x is true" we actually mean "x is non-`nil`." When we say "x is false" we mean "x is `nil`." Using this circumlocution, we can say `(if` x y z`)` is y if x is true and z if x is false, acting like x has some conventional Boolean value when in fact it can be any object.

4.2 Data Types

ACL2 supports five data types.

- numbers: The most common numbers in this document will be naturals, written `0`, `1`, `2`, ... ACL2 also supports negative integers, rationals, and complex rationals. We only use the integers here.
- characters: There are 256 character objects, e.g., `#\A`, `#\a`, and `#\Space`. To each character there corresponds an ASCII code, i.e., an integer between 0 and 255.
- strings: A string is a finite sequence of character objects, e.g., `"Hello World!"`.
- symbols: A symbol may be thought of as an object representing a *name*, e.g., of a function, variable, opcode, or theorem. For example, `fact`, `n`, `STORE`, and `associativity--of-append` are all symbols. For our purposes, case is unimportant: `fact`, `Fact`, and `FACT` all denote the same symbol. Symbols containing "unusual" characters like spaces, parentheses, etc., must be written with special delimiters but we do not use such symbols here.
- pairs: A pair is an object containing two arbitrary objects. We call the left component of a pair its *car* and the right component its *cdr*. Typically we use nested pairs to construct lists. For example, the list `(RED BLUE GREEN)` is thought of as a list of three symbols but it is "really" an ordered pair whose car is the symbol `RED` and whose cdr is a list of the two symbols `(BLUE GREEN)`. That list is an ordered pair whose car is the symbol `BLUE` and whose cdr is a list of one symbol `(GREEN)`. That list is an ordered pair whose car is the symbol `GREEN` and whose cdr is the symbol `NIL`. Note that the *symbol* `nil` plays the role of the empty "list!"

Objects of different types are different. Thus, the strings `"HELLO"` and the symbol `HELLO` are different.

Because Lisp tests against `nil`, there is no unique truth value denoting true. For example, `0`, `1`, `"HELLO"`, and `(HELLO WORLD)` are all non-`nil`. If the test of an if-then-else expression returned one of these values, the if-then-else would evaluate to the true branch. By convention when we need to return an object denoting "true" we tend to use the symbol `t`.

When we say a function is *Boolean* we mean it returns `t` or `nil`. When we say a Boolean function *recognizes* a type of object we mean that it returns `t` or `nil` according to whether its argument is in the type.

Lisp terms are written in the same parenthetical notation used to write Lisp objects. Indeed, though it is beyond the scope of this paper, Lisp terms *are* Lisp objects! When we see `(- n 1)` how do we know whether we mean the difference between the values of `n` and

`1` or the list of length three containing the symbol `-`, the symbol `n`, and the integer `1`? The answer is that when we mean to use an expression as a value, we write a "single quote" mark in front of it. Thus, the term `(- n 1)` denotes the difference of `n` and `1` and the term `'(- n 1)` denotes a constant list of length three. The term `n` is a variable; the term `'n` is a constant whose value is the symbol `n`.

The key notion to keep in mind is whether we are displaying an expression as a *term to be evaluated* or as a *value*. For example, we might say "`(car (cdr '(fact (- n 1))))` evaluates to `(- n 1)`." The first parenthesized expression is being used as a term to be evaluated and the second is its value.

4.3 Primitive Functions

All that remains is to list the available primitive function symbols. We use a small subset of ACL2 in this paper and we document only that subset here. For this reason, the set of primitives documented below may seem arbitrary.

`(if` x y z`)` the if-then-else operator; if x is true, then y, else z.

`(and` x_1 x_2 $\ldots$ x_n`)` logical conjunction; if all x_i are true, then the value is the last one, x_n; otherwise the value is `nil`.

`(or` x_1 x_2 $\ldots$ x_n`)` logical disjunction; if there is an i such that x_i is true, then the value is the first such x_i; otherwise `nil`.

`(not` x`)` logical negation; `t` if x is nil; `nil` otherwise.

`(implies` x_1 x_2`)` logical implication; if x_1 is true, then the result is `t` or `nil` depending on whether x_2 is true; if x_1 is `nil`, the result is `t`.

`(cons` x_1 x_2`)` ordered pair constructor; the value is the ordered pair whose left component is x_1 and whose right component is x_2.

`(consp` x`)` Boolean recognizer for ordered pairs.

`(car` x`)` left component of ordered pair x; `nil` if x not a pair.

`(cdr` x`)` right component of ordered pair x; `nil` if x not a pair.

`(endp` x`)` Boolean recognizer for non-pairs (esp. `nil`).

`(atom` x`)` Boolean recognizer for non-pairs.

`(symbolp` x`)` Boolean recognizer for symbols.

`(stringp` x`)` Boolean recognizer for strings.

`(coerce` x `'list)` the list of characters corresponding to the string x

`(characterp` x`)` Boolean recognizer for characters.

`(char-code` c`)` ASCII code for character c.

`(integerp` x`)` Boolean recognizer for integers.

`(natp` x`)` Boolean recognizer for integers such that $0 \leq x$.

`(equal` x_1 x_2`)` `t` or `nil` according to whether x_1 and x_2 are the same object.

`(+` x_1 `...` x_n`)` sum: $x_1 + \ldots + x_n$; non-numeric arguments are treated as 0.

`(-` x_1 x_2`)` difference: $x_1 - x_2$; non-numeric arguments are treated as 0.

`(*` x_1 `...` x_n`)` product: $x_1 \times \ldots \times x_n$; non-numeric arguments are treated as 0.

`(/` x_1 x_2`)` quotient: x_1/x_2; if x_1 is non-numeric, it is treated as

`(<` x_1 x_2`)` less than: `t` or `nil` according to $x_1 < x_2$; non-numeric arguments are treated as 0.

`(<=` x_1 x_2`)` less than or equal: `t` or `nil` according to $x_1 \leq x_2$; non-numeric arguments are treated as 0.

`(>` x_1 x_2`)` greater than: `t` or `nil` according to $x_1 > x_2$; non-numeric arguments are treated as 0.

`(>=` x_1 x_2`)` greater than or equal: `t` or `nil` according to $x_1 \geq x_2$; non-numeric arguments are treated as 0.

`(zp` x`)` if x is `0`, the value is `t`, if x is a positive natural, the value is `nil`; otherwise the value is `t`.

The talk above about non-numeric arguments highlights the fact that ACL2 is an untyped language. It is legal to write `(+ T 3)` and `(<= -2 'MONDAY)`. According to the descriptions above, the first evaluates to `3` and the second evaluates to `nil`, because non-numeric arguments are treated as `0`.

The definition of `zp` is oddly complicated but convenient. Consider the idiom

```
(if (zp n) <base> <recursion>)
```

and inspect the definition of `zp`. If `n` is `0`, the `<base>` case is taken. If `n` is a non-`0` natural number, the `<recursion>` case is taken. But what if `n` is something else, like `-1` or a non-integer rational or even a list? In all such cases, the `<base>` case is taken, i.e., `zp` treats non-naturals as it does `0`.

The following three functions are not used explicitly in the model or theorems shown below. But they play crucial roles behind-the-scenes roles in both the definitional principle and the induction principle, and they appear often in proofs. We discuss ordinals and ϵ_0 further in subsection 8.1 below.

`(o-p` x`)` `t` or `nil` according to whether x is an ordinal below ϵ_0.

`(o<` x_1 x_2`)` `t` or `nil` according to whether x_1 a smaller ordinal than x_2 (provided both are ordinals below ϵ_0).

`(acl2-count` x`)` a natural number measure of the “size” of object x; for objects composed entirely of pairs and natural numbers it, for example, the sum of number of pairs plus the sum of all the natural numbers.

The syntax of ACL2 can be extended with macros. Through macros we can eliminate much repetitive syntax. We avoid macros in what follows for pedagogical purposes. When you see the same patterns of terms emerging again and again think “That could be eliminated, but I’d have to master macros to understand it.”

4.4 A Few Pre-Defined Functions

With these few primitives we can define a wide variety of useful functions. In fact, many such functions come already defined in ACL2. Below we show definitions for the ones we use in this paper. Sometimes are definitions differ from those of the built-in ACL2 functions, but not in ways that are exposed by their usage in this paper.[1]

The following function determines the length of a list.

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

Thus, `(len '(A B C D E))` evaluates to `5`. If applied to a non-list `x`, `len` behaves as though `x` were `nil` (because of the definition of `endp`), e.g., `(len 7)` evaluates to `0`.

This function concatenates two lists.

```
(defun append (x y)
  (if (endp x)
      y
    (cons (car x)
          (append (cdr x) y))))
```

Thus, `(append '(A B C) '(D E))` evaluates to `(A B C D E)`.

Here is the function that returns the n^{th} element of a list.

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

For example, `(nth 3 '(A B C D E))` evaluates to `D`. The function `nth` “inherits” from the primitives the treatment of “inappropriate” arguments. For example, `(nth 3 '(A B C))` evaluates to `nil` because `car` and `cdr` return `nil` on `nil`. `(Nth -3 '(A B C))` evaluates to the symbol `A` because the `-3` is treated as `0` by `zp`. `(Nth 2 'ABC)` evaluates to `nil` by the above properties of `car` and `cdr`.

Two properties of `nth` are shown below. The first says that `nth` returns `nil` when its second argument is `nil` (regardless of its first argument). This follows from the properties of

[1]For example, ACL2 permits functions to have *guards* specifying expected pre-conditions and we omit them. Also, some “functions” here are actually macros in ACL2 permitting optional arguments. But in all cases the expressions used in the paper have exactly the meanings given by the function definitions here.

`car` and `cdr` noted above. The second theorem says that when `x` is an ordered pair, the size of `(nth n x)` is smaller than that of `x`, regardless of `n`. By “size” we mean the previously mentioned `acl2-count`.

```
(defthm nth-nil
  (equal (nth n nil) nil))

(defthm acl2-count-nth
  (implies (consp x)
           (< (acl2-count (nth n x))
              (acl2-count x)))
  :rule-classes :linear)
```

The `defthm` commands above direct ACL2 to prove the indicated formulas and, if successful, store them as theorems with the names `nth-nil` and `acl2-count-nth`. The `rule-classes` argument, when provided, tells ACL2 how to use the theorem in subsequent proofs. ACL2 proves these two lemmas automatically, by induction on `n`.

We do not discuss how ACL2 proves these properties, but their truth should be self-evident. The reason we need them is that we will later define some functions that recur on substructures of `x` obtained by applying `nth` to `x`. These theorems allow ACL2 to prove that such recursion terminates.

The function `char` is like `nth` but takes a string and an index and returns the corresponding character from the string. Logically speaking it is defined in terms of `nth`.

```
(defun char (s n)
  (nth n (coerce s 'list)))
```

Thus, `(char "Hello" 1)` evaluates to `#\e`, the lowercase character ‘e’.

Here is another useful definition.

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

This function “changes” `x` by setting the n^{th} element to `v`. Actually, of course, it copies `x`. `(update-nth 3 'X '(A B C D E))` evaluates to `(A B C X E)`. Interestingly, it extends `x` on the right as necessary to “make room” for an n^{th} element. `(update-nth 4 'Z '(A B C))` evaluates to `(A B C NIL Z)`.

Finally, here is a useful “predicate.”

```
(defun member (e x)
  (if (endp x)
      nil
    (if (equal e (car x))
        x
      (member e (cdr x)))))
```

Note its non-Boolean nature. `(Member 'C '(A B C D E))` evaluates to `(C D E)`, which is non-`nil` and can thus be used as “true” in tests; `(member 'G '(A B C D E))` evaluates to `nil`. `Member` is defined the way it is so that the programmer can discover not just whether `e` occurs in `x` but where.

5 An Operational Semantics for a Toy JVM

We now define the operational or state-transition semantics of a simple programming language. The language (and the machine it runs on) is called M1.[2]

5.1 Basic State Manipulation Functions

We define functions that let us pretend that lists are stacks. `Push` takes an object and a stack and returns the stack with that object on top. `Top` takes a stack and returns the top item. `Pop` takes a stack and returns the stack obtained by removing the top item.

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

Thus, `(push 3 (push 2 (push 1 nil)))` evaluates to the “stack” `(3 2 1)` whose `top` is `3` and which yields the “stack” `(2 1)` when `pop` is applied to it.

We will represent M1 instructions as lists. `Opcode` returns the operation code of the instruction `inst` and `arg1` returns its first operand. For example, `(GOTO -10)` and `(STORE 2)` are instructions. The instruction `(GOTO -10)` has opcode `GOTO` and `arg1` `-10`.

```
(defun opcode (inst) (nth 0 inst))
(defun arg1 (inst) (nth 1 inst))
```

An M1 program is just a list of instructions accessed positionally with `nth`.

The core of our semantics is the notion of a *state*. The states of the M1 machine consist of 4-tuples containing a program counter (pc), a vector of local variable values accessed by 0-based indexing, a stack of intermediate results (the “operand stack”), and a program. Here are the functions to construct a state and return the various components of a state.

```
(defun make-state (pc locals stack program)
     (cons pc
           (cons locals
                 (cons stack
                       (cons program
                             nil)))))
(defun pc (s) (nth 0 s))
(defun locals (s) (nth 1 s))
(defun stack (s) (nth 2 s))
(defun program (s) (nth 3 s))
```

[2]The precise instruction set we attribute to M1 varies with different publications, but the M1 architecture remains as described here. We include here just those instructions needed to do the examples shown. Our code proof techniques are immune to the presence of unused instructions in the instruction set. To actually carry out the definitions shown below it is necessary to be in a symbol package called `"M1"` where certain predefined Common Lisp symbols, e.g., `push` and `step`, are undefined. See the associated script for the package definition.

The function `next-inst` takes a state and returns the instruction indicated by the pc.

```
(defun next-inst (s)
  (nth (pc s) (program s)))
```

5.2 The Semantics of Each Instruction

We define the semantics of an M1 instruction by defining a function that takes an instruction of a given class and a state and returns the next state. Our convention will be that the semantics of the opcode *op* will be given by an ACL2 function with the name `execute-`*op*. All of the instructions defined below are modeled closely on actual JVM instructions. We discuss the relationship between M1 and the JVM in subsection 5.4.

The `(PUSH k)` instruction increments the pc by `1`, pushes `k` on the operand stack, and leaves the locals and program unchanged.

```
(defun execute-PUSH (inst s)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (arg1 inst) (stack s))
              (program s)))
```

The `(LOAD n)` instruction increments the pc by `1` and pushes the value of local variable `n` onto the operand stack.

```
(defun execute-LOAD (inst s)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (nth (arg1 inst)
                         (locals s))
                    (stack s))
              (program s)))
```

The `(ADD)` instruction increments the pc, pops two items off the stack, adds them, and pushes the result.

```
(defun execute-ADD (inst s)
  (declare (ignore inst))
  (make-state (+ 1 (pc s))
              (locals s)
              (push (+ (top (pop (stack s)))
                       (top (stack s)))
                    (pop (pop (stack s))))
              (program s)))
```

The `(STORE n)` instruction increments the pc, pops one item off the operand stack, and deposits it into local variable `n`.

```
(defun execute-STORE (inst s)
  (make-state (+ 1 (pc s))
              (update-nth (arg1 inst)
                          (top (stack s))
                          (locals s))
              (pop (stack s))
              (program s)))
```

The `(SUB)` instruction is like `ADD` but pushes the difference of the top two items.

```
(defun execute-SUB (inst s)
  (declare (ignore inst))
  (make-state (+ 1 (pc s))
              (locals s)
              (push (- (top (pop (stack s)))
                       (top (stack s)))
                    (pop (pop (stack s))))
              (program s)))
```

The `(MUL)` instruction is like `ADD` but pushes the product of the top two items.

```
(defun execute-MUL (inst s)
  (declare (ignore inst))
  (make-state (+ 1 (pc s))
              (locals s)
              (push (* (top (pop (stack s)))
                       (top (stack s)))
                    (pop (pop (stack s))))
              (program s)))
```

The `(GOTO k)` instruction increments the pc by `k` (which may be negative).

```
(defun execute-GOTO (inst s)
  (make-state (+ (arg1 inst) (pc s))
              (locals s)
              (stack s)
              (program s)))
```

The `(IFLE k)` instruction pops one item off the operand stack. If that item is less than or equal to `0`, it increments the pc by `k` (which may be negative); otherwise, it increments the pc by `1`.

```
(defun execute-IFLE (inst s)
  (make-state (if (<= (top (stack s)) 0)
                  (+ (arg1 inst) (pc s))
```

```
                (+ 1 (pc s)))
              (locals s)
              (pop (stack s))
              (program s)))
```

The next two instructions, `(IFLT k)` and `(IFNE k)` are analogous to `IFLE` except test the top of the stack with "less than" or with "not equal" 0 instead. When the test succeeds, they increment the pc by `k`.

```
(defun execute-IFLT (inst s)
  (make-state (if (< (top (stack s)) 0)
                  (+ (arg1 inst) (pc s))
                (+ 1 (pc s)))
              (locals s)
              (pop (stack s))
              (program s)))

(defun execute-IFNE (inst s)
  (make-state (if (not (equal (top (stack s)) 0))
                  (+ (arg1 inst) (pc s))
                (+ 1 (pc s)))
              (locals s)
              (pop (stack s))
              (program s)))
```

The `(IFANE k)` pops two items off the stack and increments the pc by `k` if the two are equal.

```
(defun execute-IFANE (inst s)
  (make-state (if (not (equal (top (pop (stack s)))
                              (top (stack s))))
                  (+ (arg1 inst) (pc s))
                (+ 1 (pc s)))
              (locals s)
              (pop (pop (stack s)))
              (program s)))
```

Finally, the last instruction on M1 is `(ALOAD)`. It finds two items on the stack, an "array" a and index i (with the index on top of the stack). In our model, the array may be either a list or a string. The instruction pops a and i off the stack and pushes either the i^{th} element of a or the ASCII code of that element, depending on whether a is a list or a string.

```
(defun execute-ALOAD (inst s)
  (declare (ignore inst))
  (make-state (+ 1 (pc s))
              (locals s)
              (push
               (if (stringp (top (pop (stack s))))
                   (char-code (char (top (pop (stack s)))
                                    (top (stack s))))
                 (nth (top (stack s))
                      (top (pop (stack s)))))
               (pop (pop (stack s))))
              (program s)))
```

For example, the following sequence of instructions:

```
  (PUSH "HAT")
  (PUSH 1)
  (ALOAD)
```

would leave `65` (the ASCII code for uppercase ‘A’) on the stack. If local `3` contains the “array” of strings `("Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun")` and local `4` contains the integer 5, then

```
  (LOAD 3)
  (LOAD 4)
  (ALOAD)
```

would leave the string `"Sat"` on the stack.

5.3 Putting It All Together

Now we put it all together by defining a “big switch” function that takes an M1 instruction and dispatches on its opcode to invoke the appropriate state-transition function.

```
(defun do-inst (inst s)
  (if (equal (opcode inst) ’PUSH)
      (execute-PUSH  inst s)
    (if (equal (opcode inst) ’LOAD)
        (execute-LOAD  inst s)
      (if (equal (opcode inst) ’STORE)
          (execute-STORE  inst s)
        (if (equal (opcode inst) ’ADD)
            (execute-ADD   inst s)
          (if (equal (opcode inst) ’SUB)
              (execute-SUB   inst s)
            (if (equal (opcode inst) ’MUL)
                (execute-MUL   inst s)
              (if (equal (opcode inst) ’GOTO)
                  (execute-GOTO   inst s)
                (if (equal (opcode inst) ’IFLE)
                    (execute-IFLE   inst s)
                  (if (equal (opcode inst) ’IFLT)
                      (execute-IFLT   inst s)
                    (if (equal (opcode inst) ’IFNE)
                        (execute-IFNE   inst s)
                      (if (equal (opcode inst) 'IFANE)
                          (execute-IFANE  inst s)
                        (if (equal (opcode inst) 'ALOAD)
                            (execute-ALOAD   inst s)
                          s))))))))))))))
```

Note that any opcode other than the dozen tested above is a no-op: applying `do-inst` to an unknown instruction returns the state, `s`, unchanged.

To step an M1 state, we simply fetch the next instruction and execute it against the current state.

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

Note that if the next instruction has an unknown opcode, `step` is a no-op. Hence, if the pc points to an unknown instruction, then it points to that same instruction after a `step` is taken. Repeatedly stepping such a state causes no change. That is, execution on an unknown opcode halts the machine! It is convenient to define the notion of when the machine is halted.

```
(defun haltedp (s)
  (equal s (step s)))
```

To run the M1 machine repeatedly from some state `s`, we step it once for every element in a schedule `sched`.

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

We comment below on why we define `run` to take a list and ignore all aspects of it but its length.

5.4 Comparing M1 to the JVM

To bring home the spiritual similarity of M1 to the JVM, we comment on the differences. Recall the first instruction we defined.

```
(defun execute-PUSH (inst s)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (arg1 inst) (stack s))
              (program s)))
```

The Sun JVM specification [13] describes a close relative of this instruction as follows:

bipush
Operation
Push `int` constant
Format

`bipush` *byte*

Form (actual byte code)

`bipush = 16 (0x10)`

Operand Stack

$\ldots \Rightarrow \ldots, value$

Description

The immediate $byte$ is sign-extended to an `int` $value$. That $value$ is pushed onto the operand stack.

All JVM instructions are described in this format.

Among the differences between M1's `PUSH` and the JVM's `bipush` are the following. First, a program on the JVM is a stream of bytes, so the operand to `bipush` is just the byte after the opcode in the stream; on M1, a program is a list of instructions and each instruction is an object (list) containing its operands. Second, the `bipush` instruction can only push a 32-bit `int`; the M1 `PUSH` instruction can push any ACL2 object. Third, the names of the instructions are different! JVM instructions almost always have names beginning with a one- or two-letter prefix indicating the type of data they operate on; the "`bi`" indicates the instruction converts a byte into an `int`. M1 instructions carry no type information in their names.

The next M1 instruction was `(LOAD n)`. The JVM contains four analogous "load" instructions: `dload`, `fload`, `iload`, and `aload`, to be used according to the type of data in the indicated local variable: double, float, 32-bit integer, or address (reference), respectively. The JVM versions of these instructions each consume two bytes in the instruction stream: one for the opcode and one for the number of the local. Because it is very common to load from locals `0`, `1`, `2`, and `3`, there are one-byte versions of these opcodes with mnemonics like `dload_0`, `dload_1`, `dload_2`, and `dload_3`, etc. M1 avoids this duplication too.

The M1's `ADD`, `SUB`, and `MUL` arithmetic instructions do unbound arithmetic. The JVM's instructions implement bounded arithmetic of various types, e.g., `iadd`, `isub`, and `imul` operate on and return to 32-bit twos-complement ("`int`") representation. The JVM supports doubles and floats as well. In our more sophisticated models of the JVM, we describe bounded arithmetic accurately.

The `(IFANE k)` instruction on M1 is modeled after the JVM's `if_acmpne` instruction. It pops two items off the stack and increments the pc by `k` if the two are equal. On the JVM, the two items are expected to be addresses.

The M1's `(ALOAD)` is based on the JVM family `baload`, `caload`, `daload`, `faload`, `iaload`, `laload`, `aaload`, and `saload`, for pushing onto the stack a datum of type byte, char, double, float, 32-bit integer, long, address, and short (respectively) obtained by indexing into an array. Both the array, a, and the index, i, are found on the stack, popped off, and replaced by $a[i]$ as in M1.

However, our generic `ALOAD` instruction differs in a more fundamental way from those on the JVM. The JVM finds on the stack, in place of a, an address into the JVM heap. The corresponding JVM instructions de-reference that address and obtain the array object into which they index. But M1 does not model a heap. What M1 finds on the stack for a is either an ACL2 string or an ACL2 list.

By omitting the heap and having M1 traffic in ACL2 objects we greatly simplify M1 proofs. But we also make it impossible to model the destructive modification of Java objects. In more sophisticated models we model the heap as a finite map from addresses to ACL2 objects. Bytecode instructions which “operate on Java Objects” actually expect to find addresses on the stack, just as in the JVM. The modification of a Java Object in the heap is then modeled by associating a different ACL2 object with the Java Object’s heap address in the model of the heap.

Aside from the heap, the most glaring omission from M1 is support for procedure call and return, called “method invocation” and return on the JVM. To support it, imagine changing the state so that it is really a stack of M1-like states. Each element of this *call stack* is called a frame and corresponds to the activation of some method. The basic method invocation instruction on the JVM, `invokevirtual` obtains an object from the operand stack, uses it (via *method resolution*) to obtain some bytecode, and constructs a new frame (“M1 state”) to run that code with the actuals loaded into the locals. *Return* instructions of various types pop the frame off the call stack and push the computed value(s) onto the callers operand stack.

As to why we define `run` to take a “schedule” and then ignore all but its length, the reason is that this keeps M1 similar to our more sophisticated JVM models. In those models (e.g., M5 [20] and M6 [14, 15]), we elaborate the state considerably to include a thread table, a heap, and a class table. In these models, the state consists of a thread table, a heap (modeled as described above), and a class table. We model the class table as a finite map from class names to class descriptions, which include the names and types of the fields and methods of the class, including the bytecode for each defined method.

Each thread in the thread table has a thread identifier and a call stack as described above. To step this multi-threaded state, one must specify the thread to be stepped. In these models, the schedule is a list of “thread identifiers” and `run` passes each successive thread identifier to the `step` function so that the appropriate thread state is stepped.

The reader ought to be able at least to imagine growing M1 into a “real” JVM. For pedagogical purposes, we keep M1 very simple.

6 An Example Program

Consider the following simple Java program defining an iterative factorial method, `ifact`.

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

In this section we will show the M1 program corresponding to this and we show how to use the operational semantics model to execute (“simulate”) this program on concrete data.

6.1 The Program

If one compiles the `ifact` program with Sun’s `javac` compiler, one obtains JVM bytecode. Below we define a constant, named `*fact-program*`, whose value is the corresponding program on M1. In Lisp, comments are preceded by a semi-colon and everything to the right of the semi-colon on a line is ignored. Thus, `*ifact-program*` is just the list shown to the left of the column of semi-colons.

The variable n in the Java code is local variable 0 in the bytecode. The variable a is local variable 1. Thus, to run this program on some value n we must first put n into local variable 0.

Consider the display below and note the five columns. The column labeled "M1 code" is the list of instructions on our machine. The column labeled "M1 pc" is the location in the list of each successive instruction; the numbers are sequential starting from 0. The column labeled "JVM pc" is the actual byte address of the corresponding JVM bytecode instruction. The bytecode instructions are shown in the column labeled "bytecode." Finally, the column labeled "Java" contains the Java statements that gave rise to the corresponding bytecode.

```
(defconst *ifact-program*

;    M1               M1      JVM    JVM             Java
;    code             pc       pc   bytecode

 '((PUSH 1)      ;   0         0   bipush_1
   (STORE 1)     ;   1         1   istore_1      a = 1;
   (LOAD 0)      ;   2         2   iload_0       while (n>0){
   (IFLE 10)     ;   3         3   ifle    17
   (LOAD 0)      ;   4         6   iload_0
   (LOAD 1)      ;   5         7   iload_1
   (MUL)         ;   6         8   imul
   (STORE 1)     ;   7         9   istore_1        a = n*a;
   (LOAD 0)      ;   8        10   iload_0
   (PUSH 1)      ;   9        11   bipush_1
   (SUB)         ;  10        12   isub
   (STORE 0)     ;  11        13   istore_0        n = n-1;
   (GOTO -10)    ;  12        14   goto    2        }
   (LOAD 1)      ;  13        17   iload_1
   (RETURN)      ;  14        18   ireturn        return a;
   ))
```

Note that the JVM pcs generally increase by 1 on successive lines. That indicates that the bytecode instruction on that line takes up one byte. But the ifle and goto instructions take 3 bytes each; thus the JVM pc column jumps by three on those lines. On M1, we address the locations in the program by instruction counts, not byte counts.

Also note that the M1 IFLE and GOTO instructions contain operands used as offsets to the current pc. But the numbers shown in the corresponding JVM instructions at JVM pcs 3 and 14 are the absolute byte addresses of the branch targets. This is just an artifact of how bytecode is printed by the Sun javac utility. In fact, the JVM instructions ifle and goto really do operate on offsets.

With the exceptions noted above, and our convention of dropping the type-specify prefixes on the instruction opcodes, the constant *ifact-program* is just what the Sun javac compiler produces. Of course, the semantics of our instructions are actually somewhat simpler than those of the JVM, even for this subset.

Note that the M1 program above uses the undefined instruction `RETURN`. It effectively halts the machine. Given the `(LOAD 1)` at our pc `13`, when the machine halts the value of `a` is on the stack.

6.2 Running the Program

To run the program we must have a schedule. The length of the schedule should be the number of steps to takes to run `*ifact-program*` from the first instruction to the last. To define the schedule we need a utility functions, `repeat`, which returns a list of a given length.

```
(defun repeat (th n)
  (if (zp n)
      nil
    (cons th (repeat th (- n 1)))))
```

For example, `(repeat 0 4)` evaluates to `(0 0 0 0)`.

To define a schedule for `*ifact-program*` we first consider the loop that starts at M1 pc `2`. Inspection of `*ifact-program*` reveals that if n is `0` when the pc is `2`, then it takes four steps to reach the terminating `RETURN` instruction at our pc `14`. In particular, the machine will execute these four instructions: the `LOAD` at 2, the `IFLE` at 3, and then, because local variable `0` is equal to `0`, the `LOAD` at M1 pc `13` (= `3` + `10`), and the `RETURN`. If, on the other hand, n is non-0, the program will execute the 11 instructions between M1 pc `2` and the `GOTO` and be back at pc `2` with $n-1$ in local `0`.

Thus, a suitable schedule to run the program from M1 pc `2` to termination is given as a function of `n` as follows.

```
(defun ifact-loop-sched (n)
  (if (zp n)
      (repeat 0 4)
    (append (repeat 0 11)
            (ifact-loop-sched (- n 1)))))
```

Since it takes 2 instructions to get from the top of the program to the top of the loop, a schedule to run `*ifact-program*` to completion is constructed by the following function.

```
(defun ifact-sched (n)
  (append (repeat 0 2)
          (ifact-loop-sched n)))
```

With this schedule we can, in principle, run `*ifact-program*` on any natural number n. To do it, construct a state, s, with program component being `*ifact-program*` such that the pc is `0`, the locals is a list containing n as its 0^{th} element, and the stack is `nil`. Use `run` to step s according to the schedule `(ifact-sched` n`)`. This produces some other state s'. Finally, get the operand stack from s' and take its top item. We claim the result will be `(fact` n`)`.

For any natural constant n we can phrase this as a theorem. Below is an example run for $n = 5$.

```
(defthm factorial-5-example
  (equal (top
          (stack
           (run
            (ifact-sched 5)
            (make-state
             0
             '(5 0)
             nil
             *ifact-program*))))
         (fact 5))
  :rule-classes nil)
```

This theorem is trivial to prove: we just execute the model. In the case of $n = 5$ the schedule is of length 61, which means `run` takes 61 steps.

We can run the M1 model on larger examples. For example, if we replace `5` above by `1000`, M1 steps 11,006 times and computes $1000!$, a number with 2,568 decimal digits. Our little M1 inherits a lot of power from its parent language, ACL2. We can execute these eleven thousand instructions in less than 0.03 seconds on a 2.4 GHz Intel Core 2 Duo Mac OS X laptop. That is about 360,000 instructions per second.

This supports the claim that an operational model formalized in a functional language can be used as a simulation engine. At AMD, where ACL2 is used to verify floating point hardware designs for all AMD processors from the Athlon onwards, ACL2 models have been run on over 80 million floating point test cases as part of the process of validating the models against more conventional simulators [24, 25]. Researchers at Rockwell Collins, where ACL2 is used to explore security properties of hardware and software artifacts, report that ACL2 models run at 50% to 90% the speed of their C models [8].

While ACL2 models are not as fast as conventional industrial simulators, they are not mere toys.

7 Another Programming Language

To help solidify intuitions about M1, we develop a compiler for a simple language with assignment and while statements. We call the language "J1." The compiler will transform a well-formed J1 program into an M1 program. J1 is not very rich. It is a language of arithmetic, assignment, and while statements, with the only test being a strict "greater than." It does not support arrays or string data. So it does not illustrate use of the `ALOAD` instruction and some of the `IFxx` instructions. But it suits our purposes.

7.1 Grammar of J1

```
<expr>         := <var>|<int-constant>|( <expr> <op> <expr> )
<op>           := + | - | *
<test>         := ( <expr> > <expr>)
<stmt>         := ( <var> = <expr> ) |
                  ( while <test> <stmt*>) |
                   ( return <expr> )
<stmt*>         := <stmt> | <stmt> <stmt*>
<program>       := ( <stmt*> )
<var>           := any ACL2 symbol
<int-constant> := any ACL2 integer
```

Thus, an example J1 program is:

```
 ((a = 1)
  (while (n > 0)
    (a = (n * a))
    (n = (n - 1)))
  (return a))
```

For the purposes of this exercise, we will assume that every program we wish to compile is syntactically well-formed. We also assume every J1 program is implicitly paired with a list of *formal parameters*. In the J1 program above, the only formal parameter is `n`. In a more sophisticated language, programs are given names and their locals are explicitly declared, as in the first line of a Java method.

We show the complete compiler in Appendix A. It takes six pages to define, explain, and illustrate the compiler. But if one evaluates

```
(compile '(n)
         '((a = 1)
           (while (n > 0)
             (a = (n * a))
             (n = (n - 1)))
           (return a)))
```

the result is the M1 program shown above.

At this point, we have a choice. One “natural” activity would be to prove that the J1 to M1 compiler is correct. The other is to prove some M1 programs correct, using the compiler merely as a means to obtain some interesting programs.

To prove the compiler correct we would need to define the semantics of J1. Semantics of higher level languages have been repeatedly defined in systems like ACL2 and compilers for them have been proved correct [18, 26, 4]. As with M1, we would use an operational semantics for J1 and the state would be the bindings of the variables mentioned in the program.

But we prefer not to formalize the semantics of J1. The reason is that this better reflects the situation we find today in the software industry. The semantics of widely used programming languages are most often given only by their compilers! For example, C and C++ actually have platform-specific semantics. Java’s semantics is made precise and nearly platform-independent by virtue of a very well-designed and specified virtual machine.

So we proceed now to prove J1 programs correct – by proving their M1 counterparts correct. This work, too, has been carried out with very large models. For example, Yuan Yu used the earlier Boyer-Moore theorem prover, NQTHM, to verify the Berkeley C String Library by compiling it with `gcc -o` to obtain Motorola MC68020 binary and then verified the binary with respect to a formal model of the MC68020 [3]. Sandip Ray [private communication] used the M5 model of the JVM to verify the invertibility of AES-128 encryption/decryption.

8 Proving Theorems with ACL2

8.1 The ACL2 Logic and Theorem Prover

ACL2 is a first-order mathematical logic that includes a principle of recursive definition and a principle of mathematical induction. Key to both principles are the ordinals below $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$ and the well-founded relation `o<` on them. The ordinals, recognized by `o-p`, are constructed from ordered pairs and natural numbers along the lines of the concrete representation proposed by Gentzen [6]. See [17].

The principle of definition requires that every recursive function definition be "proved to terminate" by exhibiting an ordinal measure of the arguments that is proved to decrease in every recursive call. The principle of induction allows one to assume inductive instances of the conjecture being proved, provided there is an ordinal measure of the variables being instantiated that can be proved to decrease under the instantiating substitutions.

To this logic we add axioms characterizing the primitive function symbols. For example, one of those axioms is

```
(defaxiom car-cons
  (equal (car (cons x y)) x))
```

We leave the other axioms to the reader's imagination [11].

The ACL2 theorem prover attempts to prove conjectures submitted by the user. To a first approximation, the theorem prover works as follows: Each subgoal is attacked first by exhaustive simplification, applying axioms, definitions, and previously proved lemmas a *rewrite rules*. When `if`-expressions are introduced by rewriting, the resulting subgoals are split into cases and exhaustive simplification continues on each case. Any subgoal that fails to be proved this way is attacked with induction. Inductive arguments are formulated based on the recursions of the functions involved. Of course, there are many more heuristics, dozens of uses for previously proved lemmas besides rewriting, integrated decision procedures, accommodations for user-supplied hints, and many other features [10].

But the salient feature of the above description is that the theorem prover's behavior is determined by the theorems previously proved. Thus, to make the system capable of automatically proving many theorems in some particular problem domain the user's task is to formula lemmas about the function symbols in that domain, typically with the aim of making the simplifier canonicalize expressions over those function symbols.

8.2 The M1 Problem Domain

With that in mind, we now present the theorems about M1 that are involved in virtually every M1 code proof.

Since M1 involves arithmetic, we include a standard set of arithmetic theorems, called `"arithmetic/top-with-meta"` that is distributed with ACL2. Such sets of definitions and theorems in ACL2 are called *books*. The standard ACL2 distribution comes with over 200 books supporting formal reasoning in various domains. We would use other arithmetic books, for example, if M1 supported 32-bit `int` arithmetic.

To support our pretense that certain ordered pairs are stacks, we have the following rewrite rules.

```
(defthm stacks
  (and (equal (top (push x s)) x)
       (equal (pop (push x s)) s)

       (equal (top (cons x s)) x)
       (equal (pop (cons x s)) s)))
```

We then instruct the rewriter *not* to expand the definitions of `push`, `top`, and `pop`.

```
(in-theory (disable push top pop))
```

This way, we do not see `cons`, `car`, and `cdr` in our proofs where we expect to see `push`, `top`, and `pop`.

One might wonder why we need the odd rules about `(top (cons x s))` and `(pop (cons x s))`, given that `push` is disabled. The reason is that even when `push` is disabled ACL2 will compute `(push 3 (push 2 (push 1 nil)))` to the constant expression `'(3 2 1)`, which unifies with `(cons x s)`; so we need rules for taking the `top` and `pop` of explicit “stack” constants.

We treat M1 states analogously

```
(defthm states
  (and (equal (pc (make-state pc locals stack program)) pc)
       (equal (locals
                (make-state pc locals stack program))
               locals)
       (equal (stack
                (make-state pc locals stack program))
               stack)
       (equal (program
                (make-state pc locals stack program))
               program)

       (equal (pc (cons pc x)) pc)
       (equal (locals (cons pc (cons locals x))) locals)
       (equal (stack
                (cons pc (cons locals (cons stack x))))
               stack)
       (equal (program
                (cons pc
                 (cons locals (cons stack (cons program x)))))
               program)))

(in-theory (disable make-state pc locals stack program))
```

We next arrange for the M1 state transition function, `step`, to expand only if we can determine that the next instruction is an ordered pair. Note that the right-hand side of the concluding equality below is just the body of the definition of `step`. The theorem is trivial to prove. But after proving it, we disable `step`.

```
(defthm step-opener
  (implies (consp (next-inst s))
           (equal (step s)
                  (do-inst (next-inst s) s))))
(in-theory (disable step))
```

The effect is that if we have deeply nested `step` expression, e.g., `(step (step (step (step` s`))))` then we will not expand any of the `step`s unless we know the corresponding instruction to be executed. Typically, that will force only the innermost `step` to expand – and only if enough is known about s to resolve the instruction, e.g., the specific pc and program. Allowing the theorem prover to expand `step` prematurely is disastrous: each `step` above would expand to a case analysis on all possible M1 instructions.

We also control the expansion of `run`. First we prove

```
(defthm run-opener
  (and (equal (run nil s) s)
       (equal (run (cons th sched) s)
              (run sched (step s)))))
```

to force `run` open only when the schedule is an explicit pair or `nil`. Our goal is to disable `run`.

But first we prove a most important and beautiful theorem about M1.

```
(defthm run-append
  (equal (run (append a b) s)
         (run b (run a s))))
(in-theory (disable run))
```

Recall that schedules are lists. This theorem considers a schedule created by concatenating two schedules. It tells us that we can determine the final state by running the second part of the schedule on the state produced by running the first part. This is “sequential composition” and is trivially proved by induction on schedule `a`.

Rewriting with this theorem is a critical move in code proofs in which explicit schedules are constructed. For example, recall our top-level schedule for `ifact`.

```
(defun ifact-sched (n)
  (append (repeat 0 2)
          (ifact-loop-sched n)))
```

A typical sequence of simplifications is

```
(run (ifact-sched n) s)
=                                {by def ifact-sched}
(run (append (repeat 0 2)
             (ifact-loop-sched n))
     s)
=                                {by def repeat}
(run (append ’(0 0)
             (ifact-loop-sched n))
     s)
=                                {by run-append}
```

```
(run (ifact-loop-sched n)
     (run '(0 0) s))
=                              {by run-opener}
(run (ifact-loop-sched n)
     (step (step s))).
```

Note that if we know enough about state `s` above to determine the first two instructions, we can then expand the `steps` (using `step-opener`) to calculate the symbolic state upon which we run the schedule `(ifact-loop-sched n)`.

Similarly, to prove something about `(run (ifact-loop-sched n) s)`, we might inductively assume the conjecture for `(run (ifact-loop-sched (- n 1))` s'`)`, for non-zero `n` and some state s', and prove the conjecture for `(run (ifact-loop-sched n) s)`. But

```
(run (ifact-loop-sched n) s)
=                              {by ifact-loop-sched}
(run (append (repeat 0 11)
             (ifact-loop-sched (- n 1)))
     s)
```

Then following the same pattern seen earlier, we use `run-append`, `run-opener`, and `step-opener` to run `s` symbolically for eleven steps to get s', reducing the term above to

`(run (ifact-loop-sched (- n 1))` s'`)`

which matches the `run` in our induction hypothesis (if we chose the s' in our induction hypothesis wisely).

Resuming our discussion of useful lemmas for M1 code proofs, we also must deal with how the local variables are accessed and updated. The following two lemmas are sufficient to force ACL2 to eliminate any `nth` or `update-nth` expression in which the first argument is a specific natural number. standard lemmas.

```
(defthm nth-add1!
  (implies (natp n)
           (equal (nth (+ 1 n) list)
                  (nth n (cdr list)))))

(defthm update-nth-add1!
  (implies (natp n)
           (equal (update-nth (+ 1 n) v x)
                  (cons (car x) (update-nth n v (cdr x))))))
```

For example, suppose the expression `(nth 2 x)` arises. Because ACL2 can match `2` with `(+ 1 n)` (by binding `n` to `1`) and because `1` is a natural, `(nth 2 x)` is rewritten to `(nth 1 (cdr x))`. This process repeats and we get `(nth 0 (cdr (cdr x)))` which then becomes `(car (cdr (cdr x)))` because `0` is the base case for `nth`. While this same expansion might be done by ACL2's heuristic use of the recursive definitions of `nth` and `update-nth`, these rules ensure faster elimination of the functions.

With these basic lemmas we can begin automatic code proofs about M1 code.

9 Proving Theorems about M1 Code

9.1 General Strategy and Terminology

In our first approach to code proofs we illustrate a method pioneered by the Boyer-Moore community and first used extensively in [1]. It is sometimes called the *clock function* approach or simply the *direct* approach.

A *total correctness theorem* about a program in some state s says that there exists some schedule or number of steps k such that, from an s satisfying the *pre-condition*, ψ, the program terminates in k steps and produces a state satisfying the *post-condition*, ϕ, relating the initial and final states. Given an operational semantics embodied in a function run, the direct expression of a total correctness theorem is something like:

$$\exists k(\psi(s) \rightarrow (haltedp(run(k,s)) \wedge \phi(s, run(k,s))))$$

.

Often, to prove such a formula, one exhibits the construction of a suitable k. That is, one proves instead

$$(\psi(s) \rightarrow (haltedp(run(k(s),s)) \wedge \phi(s, run(k(s),s))))$$

where the k above is a *Skolem function* of s witnessing the existence of a suitable schedule or clock to drive s to termination. Since ACL2 does not have an existential quantifier, our direct total correctness theorems take this latter form and the Skolem function k is often called a *clock function* because it tells us how many steps it takes to run s to completion. It is defined by the user.

How do we prove such theorems? The first step is to define a mathematical function, independent of the programming language, that expresses the algorithm used. We call this function the *algorithm*. Often the programmer had this function in mind before the program was coded.

By making the algorithm explicit one can decompose the correctness proof into two big steps: (a) prove that the code implements the algorithm, and (b) prove that the algorithm satisfies the correctness condition. Good programmers often carry out step (b) in some informal form first, although there are situations where it is advantageous to be able to experiment with code's performance and behavior before investing any time in steps (a) or (b).

Typically every loop in the program corresponds to a recursive function in the algorithm. So step (a) breaks down into proving that each loop is correct and then composing the results.

The beauty of this approach is that step (a) is generally straightforward because the code and the algorithm operate in lock-step *by construction*. But step (b) does not involve the code or the operational semantics – they have been factored out and one is left with a mathematical problem.

Of course, the devil is in the details. So we use the technique just described to prove that `*ifact-program*` is correct.

9.2 Direct Total Correctness

Recall

```
(defconst *ifact-program*

;    M1              M1
;    code            pc

 '((PUSH 1)      ;   0
   (STORE 1)     ;   1
   (LOAD 0)      ;   2
   (IFLE 10)     ;   3
   (LOAD 0)      ;   4
   (LOAD 1)      ;   5
   (MUL)         ;   6
   (STORE 1)     ;   7
   (LOAD 0)      ;   8
   (PUSH 1)      ;   9
   (SUB)         ;  10
   (STORE 0)     ;  11
   (GOTO -10)    ;  12
   (LOAD 1)      ;  13
   (RETURN)      ;  14
   ))
```

Our goal is to prove

```
(defthm ifact-correct
  (implies (natp n)
           (equal (run (ifact-sched n)
                       (make-state 0
                                   (cons n (cons a nil))
                                   stack
                                   *ifact-program*))
                  (make-state 14
                              (cons 0 (cons (fact n) nil))
                              (push (fact n) stack)
                              *ifact-program*))))
```

This can be seen as being in the schematic form of a total correctness theorem where the initial state s is the first `make-state` above. Thus, pre-condition ψ checks that the pc of s is `0`, the locals are n and a, the program is `*ifact-program*` (all of which are true by construction of s) and n is a natural number (which is the only part of ψ checked explicitly). The post-condition ϕ above is that final state is some quite particular state, namely the second `make-state` above. In that state, the pc is `14` (pointing to the `RETURN`) in `*ifact-program*`, local `0` has been cleared and local `1` has the value `(fact n)`, and the stack of the initial state has `(fact n)` pushed upon it. We do not state the `haltedp` condition explicitly here because our ϕ makes it obvious (the instruction at 14 is `RETURN`).

What is the algorithm behind `*ifact-program*`? Perhaps surprisingly, it is *not* the function `fact`! The algorithm behind `*ifact-program*` is

```
(defun ifact (n a)
  (if (zp n)
      a
    (ifact (- n 1) (* n a))))
```

Technically, this recursive function corresponds to the loop from pcs `2` – `12`. The program itself corresponds to the expression `(ifact n 0)`. We are using a loose notion of "corresponds." The algorithm just describes the result of the code, not the entire transformation on the state.

To prove the loop correct, we prove

```
(defthm ifact-loop-lemma
  (implies (and (natp n)
                (natp a))
           (equal (run (ifact-loop-sched n)
                       (make-state 2
                                   (cons n (cons a nil))
                                   stack
                                   *ifact-program*))
                  (make-state 14
                              (cons 0 (cons (ifact n a) nil))
                              (push (ifact n a) stack)
                              *ifact-program*))))
```

This lemma states the total "correctness" of the loop in exactly the same way the main theorem states the total correctness of the entire program – except here it phrased in terms of the algorithm `ifact` instead of the "more abstract" specification function `fact`. This lemma is proved automatically by ACL2. The key to that automation is that the term `(ifact n a)` suggests the right induction.

To move up from the loop to the whole program, we prove

```
(defthm ifact-lemma
  (implies (natp n)
           (equal (run (ifact-sched n)
                       (make-state 0
                                   (cons n (cons a nil))
                                   stack
                                   *ifact-program*))
                  (make-state 14
                              (cons 0 (cons (ifact n 1) nil))
                              (push (ifact n 1) stack)
                              *ifact-program*))))
```

This is trivial because `(ifact-sched n)` opens into an `append`, we use `run-append` as previously discussed, `run` the program on the first two steps, initializing `a` to `1`, and find ourselves at pc `2` prepared to `run` the schedule `(ifact-loop-sched n)`. But that `run` is correct by `ifact-loop-lemma`.

We can now disable `ifact-sched` so that we never run the bytecode again in proofs.

```
(in-theory (disable ifact-sched))
```

We have completed step (a) of our proof methodology. We would be done had our original specification been in terms of `(ifact n 1)` instead of `(fact n)`. Step (b) is to establish the relation between these two different mathematical descriptions of the same idea.

```
(defthm ifact-is-factorial
  (implies (and (natp n)
                (natp a))
           (equal (ifact n a)
                  (* (fact n) a))))
```

Note that to prove that `(ifact n 1)` is `(fact n)` we have to prove a more general mathematical relation. This is just a fact of life when dealing with induction.

Given this theorem and the arithmetic theorem that `1` is the multiplicative identity, our main theorem follows.

```
(defthm ifact-correct
  (implies (natp n)
           (equal (run (ifact-sched n)
                       (make-state 0
                                   (cons n (cons a nil))
                                   stack
                                   *ifact-program*))
                  (make-state 14
                              (cons 0 (cons (fact n) nil))
                              (push (fact n) stack)
                              *ifact-program*))))
```

The beautiful thing about this theorem is that if one is ever asked to `run` the particular initial state with the `ifact-sched` one need not actually do the step-by-step computation. One may simply set the pc to `14`, adjust the locals, and push `(fact n)` on the stack.

This makes this style of theorem compositional. Given a large program we can prove the correctness of pieces of it, each with their own schedule, and then derive the correctness of the whole run (with a schedule obtained by concatenating the individual schedules) – without re-inspecting the pieces. Similar comments apply if we extend the computational model to include procedure call and return.

From `ifact-correct` we can also prove two simple, weaker, corollaries. Often these two facts are enough to satisfy the “user” of the program in question. First, the program terminates.

```
(defthm ifact-correct-corollary-1
  (implies (natp n)
           (haltedp (run (ifact-sched n)
                         (make-state 0
                                     (cons n (cons a nil))
                                     stack
                                     *ifact-program*)))))
```

It is important to prove this theorem. To see why, look at Appendix B in which we show a *universal M1 program* capable of computing *any* function into the naturals – if we ignore the `haltedp` condition!

Second, the program leaves `(fact n)` on top of the stack.

```
(defthm ifact-correct-corollary-2
  (implies (natp n)
           (equal (top
                   (stack
                    (run (ifact-sched n)
                         (make-state 0
                                     (cons n (cons a nil))
                                     stack
                                     *ifact-program*))))
                  (fact n))))
```

These weaker corollaries do not say *what else the program does*! For all we know from these lemmas, execution of the program might do arbitrary damage to our machine's state. `Ifact-correct`, above, is truly a *total* specification of the program and in many situations, especially regarding security, such specifications are highly desirable.

We could re-phrase all our lemmas by replacing `*ifact-program*` by a call of the compiler, e.g., we could state the previous corollary in terms of a J1 program:

```
(defthm ifact-correct-corollary-3
  (implies (natp n)
           (equal (top
                   (stack
                    (run (ifact-sched n)
                         (make-state 0
                                     (cons n (cons a nil))
                                     stack
                                     (compile
                                      '(n)
                                      '((a = 1)
                                        (while (n > 0)
                                          (a = (n * a))
                                          (n = (n - 1)))
                                        (return a)))))))
                  (fact n))))
```

This is trivial to prove because the `compile` expression evaluates to the same constant as `*ifact-program*`.

Note that our theorem is still phrased in terms of the states of the lower level machine, M1, because we do not have a J1 machine. We can imagine defining a J1 machine in terms of M1 and the compiler. But we would then have to decide which resources of the M1 machine matter to the J1 programmer. Is it just the bindings of the variables? Just the return value? Both? And if we hide resources can we still state the key loop invariants in terms of visible resources? Finally, what do we do about the clock? Could the compiler be modified to produce a proposed clock function?

These questions cast doubt on the viability of the idea of using a compiler to give meaning to programs. And yet, that is how the world's programmers operate today! Indeed, without admitting the intimate link between high-level language programs and their compiled counterparts it is virtually impossible to explain or reason effectively about many systems because they mix languages. In many situations, a program in a high level programming language should be viewed simply as an abbreviation for its machine code, as we are doing here.

9.3 Inductive Assertion Style Proofs

A more common way to argue the correctness of code is the *inductive assertion* method introduced by Floyd [5] and formalized via *Hoare logic* or *program logic* by Hoare in [9]. The basic idea is to annotate the code with assertions, including an assertion at the top of the code characterizing the pre-condition. Then one derives formulas stating that if the assertion at some program location holds then the next assertion reached along any program path from that location also holds. These formulas are called *verification conditions (VCs)* or simply *proof obligations* and the software that generates them is called a *verification condition generator (VCG)*. Finally, these VCs are proved with a mechanical theorem prover. If they are all theorems, then it can be concluded that if the pre-condition is true, then every assertion is true every time it is reached in program execution. In particular, any post-condition assertions at the program exits are true when (and if) reached.

Generally, this method of code proof requires either the implementation of a VCG to give semantics to the programming language or else a Hoare semantics and a mechanization of it that derives proof obligations to submit to a theorem prover. In either case, one also needs a theorem prover.

Though it is not widely appreciated – and had apparently never been published until 2003 [19] – it is possible to do mechanized inductive assertion style proofs without a VCG or Hoare logic. All that is needed is an operational semantics and a theorem prover.

We briefly illustrate the technique on `*ifact-program*`. It is convenient to define functions `n` and `a` to return the current values (in some state `s`) of locals `0` and `1`.

```
(defun n (s) (nth 0 (locals s)))
(defun a (s) (nth 1 (locals s)))
```

Recall, again,

```
(defconst *ifact-program*

;    M1               M1
;    code             pc

 '((PUSH 1)      ;   0
   (STORE 1)     ;   1
   (LOAD 0)      ;   2
   (IFLE 10)     ;   3
   (LOAD 0)      ;   4
   (LOAD 1)       ;   5
   (MUL)          ;   6
   (STORE 1)      ;   7
   (LOAD 0)       ;   8
   (PUSH 1)       ;   9
   (SUB)          ;  10
   (STORE 0)      ;  11
   (GOTO -10)     ;  12
   (LOAD 1)       ;  13
   (RETURN)       ;  14
   ))
```

and the fact that there is a loop from pcs `2` – `12`.

Now consider the following command.

```
(defspec ifact *ifact-program* (n0 a0) 0 14
  ((0 (and (equal n0 (n s))
           (natp n0)))
   (2 (and (natp n0)
           (natp (n s))
           (natp (a s))
           (<= (n s) n0)
           (equal (fact n0) (* (fact (n s)) (a s)))))
   (14 (equal (top (stack s)) (fact n0)))))
```

This is actually just an abbreviation for a sequence of `defun`, `defthm`, and other events. Surprisingly perhaps, `defspec` is not part of the ACL2 system. It is just a macro, defined by the author of M1, to hide a standard sequence of definitions and lemmas generated from the expressions in the `defspec` command above.

The successive "arguments" in the `defspec` expression above are: a symbol, `ifact`, used to generate names of functions and lemmas, the program code, `*ifact-program*`, to be verified, the names to use for the initial values of the variables, the initial and final pcs, and an annotation that associates assertions with certain pcs. Look at those assertions.

- `0` – *the pre-condition*: the current value of `n` is `n0`, which is some natural number
- `2` – *loop invariant*: both `n0` and the current value of `n` are naturals, `n` is smaller than `n0` and `(fact n0)` is the product of `(fact n)` and the current value of `a`
- `14` – *post-condition*: the top of the stack is `(fact n0)`

If we were to verify the VCs generated from this annotation we could conclude that if the program were called on a natural `n0` and ever reaches pc `14` then the top of the stack contains the correct answer. But how do we do this without a VCG or Hoare semantics?

The trick is that the `defspec` command above defines the partial function `ifact-inv`.

```
(defp ifact-inv (n0 a0 s)
  (if (member (pc s) '(0 2 14))
      (and (equal (program s) *ifact-program*)
           (if (equal (pc s) 0)
               (and (equal n0 (n s)) (natp n0))
             (if (equal (pc s) 2)
                 (and (natp n0)
                      (natp (n s))
                      (natp (a s))
                      (<= (n s) n0)
                      (equal (fact n0)
                             (* (fact (n s)) (a s))))
               (if (equal (pc s) 14)
                   (equal (top (stack s)) (fact n0))
                 nil))))
    (ifact-inv n0 a0 (step s))))
```

The suffix "`-inv`" indicates that this predicate is allegedly an invariant. "`Defp`" stands for "define partial function." See [16] for the details of how ACL2 allows the sound axiomatization of some possibly non-terminating tail-recursive functions. `Defp` is an extension by Matt Kaufmann that allows for multiple tail-recursive calls and is supported by the standard ACL2 book `"misc/defp.lisp"`.

`Defspec` automatically generates a variety of lemmas about `ifact-inv`, including the following key lemma.

```
(defthm ifact-inv-step
  (implies (ifact-inv n0 a0 s)
           (ifact-inv n0 a0 (step s))))
```

The reader may confirm that the *proof of* `ifact-inv-step` *generates and proves the VCs*!

The basic idea of the proof is that the case analysis on the pc caused by expanding `ifact-inv` in the hypothesis splits the proof into four cases according to whether the pc is `0`, `2`, `14`, or otherwise. Given one of these initial pcs and the definition of `ifact-inv` the expansion of `(ifact-inv n0 a0 (step s))` simply forces the system to symbolically execute forward from the given pc, building up a symbolic state, until it encounters one of the annotated locations, at which point `ifact-inv` simplifies to the properly instantiated assertion for that symbolic state. The system, of course, simplifies the evolving VCs as they are produced. See [19] for details. `Ifact-inv` is partial since the user may write a `defspec` in which some loop is not "cut" with an assertion. In that case, the attempt to prove `ifact-inv-step` will run indefinitely.

This establishes that the assertions in `ifact-inv` hold for every state reachable by `run` from a state satisfying the starting state. That in turn establishes that if the pre-condition holds the post-condition holds when (and if) pc `14` is ever reached. The lemmas generated by the `defspec` command allow all these theorems to be proved automatically, if ACL2 can prove the VCs.

From the final theorem generated by the `defspec` it is trivial to prove:

```
(defthm partial-correctness-of-program-ifact-corollary
  (implies (and (natp n0)
                (equal (pc s0) 0)
                (equal (locals s0)
                       (cons n0 (cons a0 nil)))
                (equal (program s0) *ifact-program*)
                (equal sk (run sched s0))
                (equal (pc sk) 14))
           (equal (top (stack sk))
                  (fact n0)))
  :hints ...)
```

The first four hypotheses require that `n0` is a natural number and `s0` is an initial state with pc `0`, locals, `n0` and `a0`, and program `*ifact-program*`. The fifth hypothesis says `sk` is the state `(run sched s0)`, that is, `sk` is an *arbitrary* reachable state because `sched` is unconstrained. The sixth hypothesis supposes that the pc of `sk` is `14`, the terminal state.

Then the conclusion states that the top of the stack of `sk` is `(fact n0)`.

This theorem states the partial correctness of `ifact`. It does not establish that `ifact` halts. It just establishes that if it halts, the right answer is computed.

We could, as usual, replace `*ifact-program*` with a call of the compiler. But the earlier problem rears its head: our theorems – in particular the locations of the assertions in the code – are stated in terms of the lower level language. A nice project would be to augment the compiler to allow the assertions to be embedded in the source code and have the compiler generate the `defspec` automatically.

9.4 On Alternative Proof Styles

The inductive assertion method as implemented here may also be used to prove total correctness, by incorporating an ordinal measure into the assertions and proving that the measure decreases between cut points.

It may appear that using inductive assertions with measures to establish total correctness involves less work than the clock function approach. But it turns out that the clock function approach and the inductive assertion approach to total correctness are equally powerful. If you can prove a program correct by one method you can prove it by the other, entirely automatically.

Indeed, there is an ACL2 book that allows the user to switch between proof styles, trading theorems proved in one style for those in another and combining them at will.

For details of these and other meta-logical results about alternative proof styles, see [23].

10 Boyer-Moore Fast String Searching

We conclude our demonstration of code proofs by considering the Boyer-Moore fast string searching algorithm [2].

Recall our description of how to prove code correct by the direct method: Step (a) is to prove the code implements a certain algorithm and step (b) is to prove the algorithm satisfies the specification. In this section we address step (a) only. Step (b) has been carried out independently of this work. See [21].

To apply our proof methodology to the Boyer-Moore algorithm we must have a formal expression of the algorithm. That was actually developed independently when step (b) was carried out. Oddly, it is not actually necessary for the reader to understand what an algorithm is intended to do while carrying out step (a)! It is sufficient to prove merely that the code does whatever the algorithm does. But for the reader's edification (and sanity) we explain the Boyer-Moore algorithm informally and then exhibit its formal definition in ACL2.

The next steps in our methodology are then: write the M1 code for the algorithm, define the schedule or "clock" function for that code (together, in this case, with another function used in the specification), state and prove a general theorem about the loop in the code, and state and prove that the top-level entry to the code computes the same answer as the algorithm and terminates. By combining this work with step (b) we get the final theorem that the computed answer is correct. We follow his methodology in the subsequent subsections.

The main lesson of this entire section is that the methodology already presented allows us to prove interesting code correct.

In the presentation we use the following additional ACL2 forms:

`(declare (xargs ...))` pragmatic advice associated with the admission of a function, e.g., the measure to use to justify its termination, the expected types of its arguments, etc.

`(cond (`p_1 x_1`) (`p_2 x_2`) ... (t` x_k`))` an abbreviation for `(if` p_1 x_1 `(if` p_2 x_2 `(if ...` x_k`)))`.

`(length` x`)` if x is a string, then the numbers of characters in it, i.e., `(len (coerce` x `'list))`; else, `(len` x`)`.

`(list` x_1 x_2 ... x_k`)` an abbreviation for `(cons` x_1 `(cons` x_2 `(cons ... (cons` x_k `nil))))`.

10.1 The Algorithm

The Boyer-Moore fast string algorithm looks for the first exact match of one string, called the *pattern* in another, called the *text*. Given a proposed alignment of the two strings, the algorithm compares them character by character *starting at the right-hand end* of the pattern. Consider two corresponding characters, say u from the pattern at index j and v from the text at the corresponding index i. If $u = v$, the algorithm backs up, decrementing j and i. If $u \neq v$, the algorithm has "discovered" a substring in the text. This substring is *almost* a terminal substring of the pattern starting at j, except the discovered string starts with v instead of u. The pattern can be realigned with the text by shifting to the right. The next possible exact match of the pattern and the text must align the discovered substring with its rightmost occurrence in the pattern. But there are only a finite number of such substrings: one for each choice of v and terminal substring of the pattern. Therefore, we can preprocess the pattern to compute a 2-dimensional array indexed by v and the position, j, of the unequal character of the pattern. We store in this array the distance, δ, we advance i upon discovering that v fails to match the character at position j in the pattern.

Here is an example. Find the first occurrence of the indicated pattern (*pat*) in the text (*txt*) below. We show a trace of the algorithm below and then we explain each step.

```
1. pat: aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
                ⇑                                    i=5

2. pat: aBCdBC
```

```
    txt: xxxaBCxxGxaBCdBCxxxx
             ⇑                                         i=4

 3. pat: aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
            ⇑                                          i=3

 4. pat:    aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                 ⇑                                     i=8

 5. pat:          aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                       ⇑                               i=14

 6. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                        ⇑                              i=15

 7. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                       ⇑                               i=14

 8. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                      ⇑                                i=13

 9. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                     ⇑                                 i=12

10. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                    ⇑                                  i=11

11. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                   ⇑                                   i=10

12. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                  ⇑                                    i=9
```

Note on line `1`, we start by reading and matching the ‘`C`’ at text position `i=5` with the ‘`C`’ at the end of the pattern. We back up. We match the ‘`B`’s. We back up. On line `3` we read the ‘`a`’ at `i=3` and it fails to match the ‘`d`’ (at position `j=3`)’ in the pattern. So we skip ahead by some precomputed amount as a function of the character v just read from the text, ‘`a`’, and the index of the matched terminal substring (`"BC"`) of the pattern (`j=4`). It turns out that the precomputed amount δ is 5. So we add δ to `i` and get the new `i=8` of line `4`. What is special about 5? Note that on line `4`, after adding δ to i and shifting the pattern rightwards to that position, the discovered `"aBC"` of the text aligns with its last occurrence in the pattern.

On line `4`, we read ‘`G`’. It does not match its counterpart in the pattern. The precomputed table (for ‘`G`’ and the empty substring) says we can increment `i` by δ = `6`. This is because *there is no* ‘`G`’ *in the pattern*! So we can slide the pattern forward by its length to get entirely past the ‘`G`’,

On `line` 5, we read ‘`B`’. Following the same routine, we use the precomputed table to shift the pattern to align the last ‘`B`’ in the pattern with the discovered `B`.

On lines `6` through `11` we just back up confirming each character.

On line `12`, we have “walked off the left end of the pattern.” That means we matched all the characters. The match starts at `i+1`, or position `10` in the text.

Here is a more realistic example.

```
pat: pattern
txt: we can preprocess the pattern to
           ⇑                                       i=6

pat:        pattern
txt: we can preprocess the pattern to
                  ⇑                                i=13

pat:               pattern
txt: we can preprocess the pattern to
                         ⇑                         i=20

pat:                 pattern
txt: we can preprocess the pattern to
                           ⇑                       i=22

pat:                       pattern
txt: we can preprocess the pattern to
                                 ⇑                 i=28

pat:                       pattern
txt: we can preprocess the pattern to
                                ⇑                  i=27

etc.
```

Here we see the algorithm skipping through the text in steps proportional to the length of the pattern. This illustrates the key advantage of the Boyer-Moore fast string searching algorithm: *it advances through the text without reading all the characters and in steps that are often nearly as big as the pattern is long*.

In this paper we do not discuss the preprocessing. However, in [21], we define `(preprocess pat)` to produce a 2-dimensional array, i.e., a list of lists, with the following property.

```
(defthm preprocess-correct
  (implies (and (stringp pat)
                (characterp v)
                (natp j)
                (< j (length pat)))
           (equal (index2 (preprocess pat) (char-code v) j)
                  (delta v j pat))))
```

where

```
(defun index2 (array c j)
  (nth j (nth c array)))
```

Here `(delta v j pat)` is the amount by which we are to increment `i` upon reading character `v` from text and finding it is unequal to the corresponding character at position `j` in `pat`.

We then define the Boyer-Moore algorithm as shown below.

```
(defun fast-loop (pat j txt i)
  (declare (xargs ...))
  (cond
   ((not (and (stringp pat) (integerp j)
              (stringp txt) (integerp i)
              (<= -1 j) (< j (length pat))
              (<= j i)))
    nil)
   ((< j 0)
    (+ 1 i))
   ((<= (length txt) i)
    nil)
   ((equal (char pat j) (char txt i))
    (fast-loop pat (- j 1) txt (- i 1)))
   (t (fast-loop pat
                 (- (length pat) 1)
                 txt
                 (+ i (delta (char txt i)
                             j
                             pat))))))

(defun fast (pat txt)
  (declare (xargs ...))
  (if (equal pat "")
      (if (equal txt "")
          nil
        0)
    (fast-loop pat
               (- (length pat) 1)
               txt
               (- (length pat) 1))))
```

As noted, we have already carried out step (b): an ACL2 proof of the correctness of this algorithm [21]. In particular, we prove that `(fast pat txt)` is equivalent to the naive, obviously correct algorithm which tests, successively, each location in `txt` to see whether the pattern matches the text at that location. We call the obviously correct algorithm `(correct pat txt)`.

But in the course of completing step (b) we proved that `fast-loop` terminates (using a measure not shown in the `(declare (xargs ...))` above), we defined the (inefficient but correct) preprocessing algorithm, `(preprocess pat)` to compute a list of lists, and we proved that indexing into that list of lists with two successive `nth`s produces `(delta v j pat)` as indicated above. It turns out this is all we need to carry out step (a) given the methodology already described here.

10.2 The Code

In the M1 code for the algorithm we use eight local variables.

local	*symbolic name*	*general use*	*initial value*
`0`	`pat`	the pattern string	`pat`
`1`	`j`	current index into pattern	`(- (length pat) 1)`
`2`	`txt`	the text string	`txt`
`3`	`i`	current index into text	`(- (length pat) 1)`
`4`	`pmax`	length of pattern	`(length pat)`
`5`	`tmax`	length of text	`(length txt)`
`6`	`array`	2-dimensional array	`(preprocess pat)`
`7`	`v`	last character read from text	*initial value irrelevant*

Here is the M1 code for the Boyer-Moore fast string searching algorithm. In the left column, with capitalized opcodes, we exhibit the M1 code. In the comment columns we exhibit the code with symbolic variable names and some pseudo-code to explain the various code fragments.

```
(defconst *m1-boyer-moore-program*
  '(
    (LOAD 0)   ;  0 (load pat)
    (PUSH "")  ;  1 (push "")
    (IFANE 5)  ;  2 (ifane loop) ; if pat/="", goto loop
    (LOAD 2)   ;  3 (load txt)
    (PUSH "")  ;  4 (push "")
    (IFANE 40) ;  5 (ifane win)  ; if txt/="", goto win
    (GOTO 43)  ;  6 (goto lose)
; loop:
    (LOAD 1)   ;  7 (load j)
    (IFLT 37)  ;  8 (iflt win))  ; if j<0, goto win
    (LOAD 5)   ;  9 (load tmax)
    (LOAD 3)   ; 10 (load i)
    (SUB)      ; 11 (sub)
    (IFLE 37)  ; 12 (ifle lose)  ; if |txt|-i<=0, goto lose
    (LOAD 0)   ; 13 (load pat)
    (LOAD 1)   ; 14 (load j)
    (ALOAD)    ; 15 (aload)      ; pat[j]
    (LOAD 2)   ; 16 (load txt)
    (LOAD 3)   ; 17 (load i)
    (ALOAD)    ; 18 (aload)      ; txt[i]
    (STORE 7)  ; 19 (store v)    ; (store into v)
    (LOAD 7)   ; 20 (load v)
    (SUB)      ; 21 (sub)
    (IFNE 10)  ; 22 (ifne skip)  ; if pat[j]≠txt[i],goto skip
    (LOAD 1)   ; 23 (load j)
    (PUSH 1)   ; 24 (push 1)
    (SUB)      ; 25 (sub)
    (STORE 1)  ; 26 (store j)    ; j=j-1
    (LOAD 3)   ; 27 (load i)
    (PUSH 1)   ; 28 (push 1)
    (SUB)      ; 29 (sub)
    (STORE 3)  ; 30 (store i)    ; i=i-1
    (GOTO -24) ; 31 (goto loop)  ; goto loop
; skip:
    (LOAD 3)   ; 32 (load i)
    (LOAD 6)   ; 33 (load array)
    (LOAD 7)   ; 34 (load v)
    (ALOAD)    ; 35 (aload)
    (LOAD 1)   ; 36 (load j)
    (ALOAD)    ; 37 (aload)
    (ADD)      ; 38 (add)
    (STORE 3)  ; 39 (store i)    ; i := i+array[c][j]
    (LOAD 4)   ; 40 (load pmax)
    (PUSH 1)   ; 41 (push 1)
    (SUB)      ; 42 (sub)
    (STORE 1)  ; 43 (store j)    ; j := |pat|-1
    (GOTO -37) ; 44 (goto loop)
; win:
    (LOAD 3)   ; 45 (load i)
    (PUSH 1)   ; 46 (push 1)
    (ADD)      ; 47 (add)
    (RETURN)   ; 48 (return)
; lose:
    (PUSH nil) ; 49 (push nil)
    (RETURN)   ; 50 (return))
  )
```

A few highlights of this code are worth noting. The loop from pc `7` through pc `44` does not necessarily terminate if an arbitrary array is stored in local variable `6`. The `aload` at pc `35` indexes into the array and pulls out another array, representing the row for character code

v. The aload at pc 37 indexes into that row at j and pulls out (delta v j pat), by the theorem about preprocess. Finally, note that at pcs 32 – 39 we first increment i by (delta v j pat) and then from pcs 40 – 43 we reset j to (- (length pat) 1). It would have been "natural" to do this in the other order (j first and then i) but that would be incorrect since the old value of j is to be used in the computation of i.

10.3 The Schedule

Following the methodology previously sketched for ifact we next define the function that determines how long the algorithm runs for a given pat and txt. But recall that we are not looking for a closed form or a general worse case analysis, we just want to know how many steps the algorithm takes for specific inputs. We can exploit the fact that the algorithm is supposed to compute fast and simply define a version of fast (and its sub-function fast-loop) that *counts* the steps!

We start by counting the steps from the top of the loop at pc 7 to the exits. That is done by m1-boyer-moore-loop-sched. Inspection of the following definition reveals that it has exactly the same case analysis and recursive structure as fast-loop. However, on each recursive call we append the schedules for the corresponding paths through the code. The path lengths, namely 6, 8, 25 and 29, are just obtained by counting the instructions on the four paths starting at the top of the loop at pc 7, namely (a) the path leading to win, (b) the path leading to lose, (c) the path that discovers equal characters, backs up and returns to pc 7, and (d) the path that discovers unequal characters, increments by delta, and returns to pc 7. The termination argument for this schedule function is *exactly the same* as used to admit fast-loop. No work need be done to admit this function. We then define the function that counts the steps from the top-level entry at pc 0, using the previously defined function to count the passage through the loop.

```
(defun m1-boyer-moore-loop-sched (pat j txt i)
  (declare (xargs ...))
  (cond
   ((not (and (stringp pat) (integerp j)
              (stringp txt) (integerp i)
              (<= -1 j) (< j (length pat))
              (<= j i)))
    nil)
   ((< j 0)
    (repeat 0 6))                        ; path (a)
   ((<= (length txt) i)
    (repeat 0 8))                        ; path (b)
   ((equal (char-code (char pat j))
           (char-code (char txt i)))
    (append (repeat 0 25)                ; path (c)
            (m1-boyer-moore-loop-sched
             pat (- j 1) txt (- i 1))))
   (t (append (repeat 0 29)              ; path (d)
              (m1-boyer-moore-loop-sched
               pat
               (- (length pat) 1)
               txt
               (+ i (delta (char txt i)
                           j
                           pat)))))))

(defun m1-boyer-moore-sched (pat txt)
  (if (equal pat "")
      (if (equal txt "")
          (repeat 0 9)
        (repeat 0 10))
    (append (repeat 0 3)
            (m1-boyer-moore-loop-sched
             pat
             (- (length pat) 1)
             txt
             (- (length pat) 1)))))
```

The definition of these schedules may look complex. But we urge the reader to compare them to their algorithmic counterparts, `fast-loop` and `fast`, to understand how simple and straightforward they are. A mistake people often make when reviewing ACL2 input is to confuse the *expression* of an idea with the complexity of *creating* it. In the case of these schedule functions, it probably took the author a minute to "annotate" (edit with Emacs) the definitions of two pre-existing functions to produce these definitions.

We are almost done. But this problem introduces a complication with our methodology not exposed by `ifact`. We wish to prove the lemma that executing loop from pc `7` returns a particular state as a function of `fast-loop`. That state will have the value of `(fast-loop pat j txt i)` on top of its stack and will terminate at either pc `48` or `50` accordingly. But what are the final values of the local variables?

In the `ifact` example, the final value of the local `n` was `0`. But what is the final value of `j` (the index into `pat`), `i` (the index into `txt`) and `v` (the last character read from `txt`)?

Rather than try to figure out the answers is some closed form, we just define a function that returns the final values of the local variables, repeating the same computation done by `fast-loop`. That function is defined below and returns a list of three elements. The 0^{th} element of the answer is the final value of `j`, the 1^{st} is the final value of `i` and the 2^{nd} is the final value of `v`. Again, no work has to be done to create or admit this definition. This is probably another minute's work. Note that unlike the definition of `fast-loop`, the function below tracks the value of `v` with added formal. This might be called a *ghost variable* and the definition below explains it. Again, we see the definition of `fast-loop`, except at the exits we see `(list j i v)`.

```
(defun m1-boyer-moore-loop-vars (pat j txt i v)
  (declare (xargs ...))
  (cond
   ((not (and (stringp pat) (integerp j)
              (stringp txt) (integerp i)
              (<= -1 j) (< j (length pat))
              (<= j i)))
    (list j i v))
   ((< j 0)
    (list j i v))
   ((<= (length txt) i)
    (list j i v))
   ((equal (char-code (char pat j))
          (char-code (char txt i)))
    (m1-boyer-moore-loop-vars
     pat
     (- j 1)
     txt
     (- i 1)
     (char-code (char txt i))))
   (t (m1-boyer-moore-loop-vars
       pat
       (- (length pat) 1)
       txt
       (+ i (delta (char txt i)
                   j
                   pat))
       (char-code (char txt i))))))
```

10.4 The Theorems

Following the methodology, we now state the correctness of the loop. This formula is long but should be self-explanatory. It says that if we start the loop at pc `7` – with the appropriate values in local variables `0` – `7` – and run according to the loop schedule, we either end up at pc `48` or `50` with the appropriate values in the locals and on the stack.

```
(defthm m1-boyer-moore-loop-is-fast-loop
  (implies
   (and (stringp pat) (integerp j)
        (stringp txt) (integerp i)
        (<= -1 j) (< j (length pat))
        (<= j i))
   (equal
    (run (m1-boyer-moore-loop-sched pat j txt i)
         (make-state 7
                     (list pat
                           j
                           txt
                           i
                           (length pat)
                           (length txt)
                           (preprocess pat)
```

```
                             v)
                       nil
                       *m1-boyer-moore-program*))
    (if (fast-loop pat j txt i)
        (make-state
         48
         (list
          pat
          (nth 0 (m1-boyer-moore-loop-vars pat j txt i v))
          txt
          (nth 1 (m1-boyer-moore-loop-vars pat j txt i v))
          (length pat)
          (length txt)
          (preprocess pat)
          (nth 2 (m1-boyer-moore-loop-vars pat j txt i v)))
         (push (fast-loop pat j txt i) nil)
         *m1-boyer-moore-program*)
      (make-state
       50
       (list
        pat
        (nth 0 (m1-boyer-moore-loop-vars pat j txt i v))
        txt
        (nth 1 (m1-boyer-moore-loop-vars pat j txt i v))
        (length pat)
        (length txt)
        (preprocess pat)
        (nth 2 (m1-boyer-moore-loop-vars pat j txt i v)))
       (push nil nil)
       *m1-boyer-moore-program*))))
  :hints (("Goal" :in-theory (enable preprocess))))
```

This is proved entirely automatically, using just the lemmas we introduced to prove `ifact` and the lemmas used to establish step (b). [3]

Following the methodology, we conclude by proving that if we enter the code at pc `0` – with appropriate values in the local variables – and run according to the top-level schedule, we get a state in which `(fast pat txt)` is pushed on the stack and the computation has terminated.

```
(defthm m1-boyer-moore-is-fast
  (implies (and (stringp pat)
                (stringp txt))
```

[3]We had to enable `preprocess`; it was left disabled by our step (b) work because it was always accessed via the `index2` abstraction, where as in our code it is accessed by two successive `nth`s. For a similar reason in our actual script at this point we disable `length`, because the lemma just proved uses `length` in the `(make-state 7 ...)` expression which is part of the left-hand side of the rewrite rule generated by the lemma. The ACL2 user must realize that these `length` expressions will open up to `(len (coerce ... 'list))` and either rephrase the lemma in those canonical terms or disable `length`.

```
           (equal
            (top
             (stack
              (run (m1-boyer-moore-sched pat txt)
                   (make-state
                    0
                    (list pat
                          (- (length pat) 1)
                          txt
                          (- (length pat) 1)
                          (length pat)
                          (length txt)
                          (preprocess pat)
                          0)
                    nil
                    *m1-boyer-moore-program*))))
                  (fast pat txt))))

(defthm m1-boyer-moore-halts
  (implies (and (stringp pat)
                (stringp txt))
           (haltedp
            (run (m1-boyer-moore-sched pat txt)
                 (make-state
                  0
                  (list pat
                        (- (length pat) 1)
                        txt
                        (- (length pat) 1)
                        (length pat)
                        (length txt)
                        (preprocess pat)
                        0)
                  nil
                  *m1-boyer-moore-program*)))))
```

Since we have carried out step (b) elsewhere and know that `(fast pat txt)` is equal to `(correct pat txt)`, we can now trivially establish that the M1 code shown above is a correct string searching algorithm.

```
(defthm m1-boyer-moore-is-correct
  (implies (and (stringp pat)
                (stringp txt))
           (equal
            (top
             (stack
              (run (m1-boyer-moore-sched pat txt)
                   (make-state
                    0
                    (list pat
                          (- (length pat) 1)
                          txt
                          (- (length pat) 1)
                          (length pat)
                          (length txt)
                          (preprocess pat)
                          0)
                    nil
                    *m1-boyer-moore-program*))))
                  (correct pat txt))))
```

10.5 Summary

We can summarize this section by saying that the methodology used to prove a trivial program like `ifact` scales up to interesting programs. The clock function may be complicated looking, but it is easy to generate if one has formally defined the algorithm being computed: just modify the algorithm to assemble the counts of each path as it executes. The only novelty above was the introduction of the function `m1-boyer-moore-loop-vars` to characterize the final values of the various machine resources. Again, the function looks hard to define but is not: just modify the algorithm to return a vector of the variables it changes.

It may seem awkward that interesting programs have “irrelevant” side-effects that must be characterized to apply this method of proof, but it is sometimes useful to specify all the effects and it is not difficult given the need to formalize the algorithm used. Furthermore, for many applications, especially where security is at risk, it is advantageous to specify *completely* the transformation caused by execution of a piece of code.

11 Conclusion

We have shown some of the techniques involved in using a mechanized operational semantics. Why might one want to use an operational semantics?

- It has dual use: one can use it as a simulation engine for a language and as the basis of a code proof methodology.
- We have hinted that it may be practical to reason about higher level languages by reasoning operationally about the object code produced by an unverified compiler.
- By casting the entire problem into a traditional mathematical logic, it is easier to understand the semantics and to relate it to an actual implementation.
- In addition, it is possible to support alternative proof styles and investigate their logical connections [23].

- It is possible to use the theorem prover to prove theorems about the semantics instead of just about code. Consider the `run-append` theorem, an interesting fact about M1 that has nothing to do with any particular code.
- One can move up and down the abstraction stack. We have stayed at the M1 level, with a hint of how to move up to J1. We could also move down and prove that an implementation of M1 on a more conventional machine is correct. See [1, 18].

12 Acknowledgements

This work could not have been done without the support of the entire Boyer-Moore community. It has taken years to develop the basic techniques shown here and to engineeer a theorem prover capable of carrying them out at scale. I would particularly like to thank Bob Boyer, Warren Hunt, Matt Kaufmann, and Sandip Ray. But the list goes on and on and includes the people who did the CLI Stack in the 1980s and the many machine models and code proofs since then.

This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591, by the National Science Foundation under Grant No. ISS-0417413, and by DARPA under Contract No. NBCH30390004.

A Appendix: The J1 Compiler

In this six page Appendix we define the J1 compiler to M1, explain every part of it, and illustrate it on our factorial example.

A.1 Allocating Variables to Locals

We first define a function, `collect-vars-in-stmt*`, that sweeps over a list of statements and collects all the variables it finds. The position of a variable in the resulting list will determine which local we allocate to the variable.

The sweep function adds the variables to the end of a running accumulator. That accumulator will be initialized to the list of formals of the method we are compiling. Thus, declared formals will be allocated the lowest indices among the locals. Variables in the program that are not declared among the formals will be allocated to higher indices, as they are encountered in the sweep.

The next function adds `e` as an element to the right end of `x` if `e` is not already a member of `x`.

```
(defun collect-at-end (x e)
  (if (member e x)
      x
    (append x (cons e nil))))
```

We use `collect-at-end` in the next function, which collects all the variable symbols used in a J1 expression, in “print order,” modulo the predetermined formals.

```
(defun collect-vars-in-expr (vars expr)
  (if (atom expr)
      (if (symbolp expr)
          (collect-at-end vars expr)
        vars)
    (collect-vars-in-expr
     (collect-vars-in-expr vars
                           (nth 0 expr))
     (nth 2 expr))))
```

Note that if `expr` is not an atom, it is of the form

`( <expr> + <expr> )` or
`( <expr> - <expr> )` or
`( <expr> * <expr> )`.

Hence, `(nth 0 expr)` is the first subexpression and `(nth 2 expr)` is the second. `Collect-vars-in-expr` can be proved to terminate because of the previously mentioned properties of `(acl2-count (nth n x))`.

Now we collect the variables in a J1 statement. This is defined mutually recursively with the variables in a list of statements. In ACL2, we must declare our intention to define a clique of mutually recursive functions by wrapping their `defun` commands in a `mutual-recursion` form.

```
(mutual-recursion

 (defun collect-vars-in-stmt* (vars stmt-list)
   (if (endp stmt-list)
       vars
     (collect-vars-in-stmt*
      (collect-vars-in-stmt vars (car stmt-list))
      (cdr stmt-list))))

 (defun collect-vars-in-stmt (vars stmt)
   (if (equal (nth 1 stmt) '=)
       (collect-vars-in-expr
        (collect-at-end vars (nth 0 stmt))
        (nth 2 stmt))
     (if (equal (nth 0 stmt) 'WHILE)
         (collect-vars-in-stmt*
          (collect-vars-in-expr vars (nth 1 stmt))
          (cdr (cdr stmt)))
       (if (equal (nth 0 stmt) 'RETURN)
           (collect-vars-in-expr vars (nth 1 stmt))
         vars))))
)
```

For example, `(collect-vars-in-stmt '(n) '(a = (a * (b + n))))` evaluates to `(N A B)`.

Given a list, `vars`, of all the variables (in order of their allocation among the locals), we compute the index of a given variable as follows.

```
(defun index (vars var)
  (if (endp vars)
      0
    (if (equal var (car vars))
        0
      (+ 1 (index (cdr vars) var)))))
```

A.2 Code Generation

To compile an expression we will sweep through it recursively and concatenate the code generated for each subexpression, with suitable "glue" instructions between the various sections. For example, to compile an expression such as `(x + y)` we will generate code that pushes the values of `x` onto the operand stack, concatenate that with the code that pushes the value of `y` onto the operand stack and then append an `ADD` instruction to the list of instructions.

This function generates the appropriate M1 arithmetic "glue" given a J1 arithmetic operator. (All of our code generation functions have names that end with an exclamation point.)

```
(defun OP! (op)
  (if (equal op '+)
      '((ADD))
    (if (equal op '-)
        '((SUB))
      (if (equal op '*)
          '((MUL))
        '((ILLEGAL))))))
```

Note that the output above is an M1 program, i.e., a list of M1 instructions (in this case, always a trivial list of length 1). All our functions for generating code in fact generate programs so we can combine them with concatenation.

Here is the generator for the bytecode program to put the value of a J1 variable `var` on the stack, given the list of variables `vars` determining allocation of the locals.

```
(defun LOAD! (vars var)
  (cons (cons 'LOAD (cons (index vars var) nil))
        nil))
```

For example, `(LOAD! '(n a) 'a)` evaluates to `((LOAD 1))`.

Here is the generator for the bytecode program to put the value of a J1 constant on the stack.

```
(defun PUSH! (n)
  (cons (cons 'PUSH (cons n nil))
        nil))
```

Using the above three functions, we now define the compiler for a J1 expression. Execution of the generated code leaves the value of the expression on the stack.

```
(defun expr! (vars expr)
  (if (atom expr)
      (if (symbolp expr)
          (LOAD! vars expr)
        (PUSH! expr))
    (append (expr! vars (nth 0 expr))
            (append (expr! vars (nth 2 expr))
                    (OP! (nth 1 expr))))))
```

For example, `(expr! ’(a b c) ’((a + (b + 3)) * c))` produces

```
((LOAD 0)
 (LOAD 1)
 (PUSH 3)
 (ADD)
 (ADD)
 (LOAD 2)
 (MUL))
```

Next we deal with branches. The generator for the bytecode program to test the top of the stack and branch by `offset` if it is less than or equal to `0` is defined as follows.

```
(defun IFLE! (offset)
  (cons (cons ’IFLE (cons offset nil))
        nil))
```

Here is the generator for the bytecode program to jump by `offset`.

```
(defun GOTO! (offset)
  (cons (cons ’GOTO (cons offset nil))
        nil))
```

To compile `(while` $p \ \ s_1 \ldots s_n$`)` we will first compile code that leaves a positive on the stack if the test p is true and a non-positive on the stack if p is false. Let $a_1 \ldots a_k$ be the code for p. Then we compile the statements s_i in the body. Let $b_1 \ldots b_n$ be the code for the body. Note that the length of the test code is k and the length of the body code is n. We use those offsets in the `IFLE` and `GOTO` instructions below. The compiled code for the `while` statement above is:

```
 (
  $a_1$                  ; top of WHILE
  ...
  $a_k$                  ; value of test is on the stack
  (IFLE $2+n$)           ; if test false, jump past body code
  $b_1$
  ...
```

```
   $b_n$
   (GOTO $-(n+1+k)$)  ; go back to top of WHILE
   )                  ; we're done with the WHILE
```

And so now we can define the generator for a while statement, given the M1 programs for the test expression and the list of statements in the body.

```
(defun while! (test-code body-code)
  (append test-code
          (append (IFLE! (+ 2 (len body-code)))
                  (append body-code
                          (GOTO! (- (+ (len test-code)
                                       1
                                       (len body-code))))))))
```

The bytecode program to leave a positive on the stack if test is true and a non-positive otherwise is generated by `test!`. The argument `test` must be of the form `(`x `>` y`)` where x and y are expressions.

```
(defun test! (vars test)
  (if (equal (nth 1 test) '>)
      (if (equal (nth 2 test) 0)
          (expr! vars (nth 0 test))
        (append (expr! vars (nth 0 test))
                (append (expr! vars (nth 2 test))
                        '((SUB)))))
    '((ILLEGAL))))
```

To generate the bytecode program to pop the stack into the local allocated for var we use `STORE!`.

```
(defun STORE! (vars var)
  (cons (cons 'STORE (cons (index vars var) nil))
        nil))
```

We use mutual recursion again to define how to compile a list of statements and how to compile a single statement.

```
(mutual-recursion

 (defun stmt*! (vars stmt-list)
   (if (endp stmt-list)
       nil
     (append (stmt! vars (car stmt-list))
             (stmt*! vars (cdr stmt-list)))))

 (defun stmt! (vars stmt)
   (if (equal (nth 1 stmt) ’=)
       (append (expr! vars (nth 2 stmt))
               (STORE! vars (nth 0 stmt)))
     (if (equal (nth 0 stmt) ’WHILE)
         (while!
          (test! vars (nth 1 stmt))
          (stmt*! vars (cdr (cdr stmt))))
       (if (equal (nth 0 stmt) ’RETURN)
           (append (expr! vars (nth 1 stmt))
                   ’((RETURN)))
         ’((ILLEGAL))))))
 )
```

Finally, we can define the compiler to take a list of formal parameters and a list of statements and return the M1 code.

```
(defun compile (formals stmt-list)
  (stmt*! (collect-vars-in-stmt* formals stmt-list)
          stmt-list))
```

Here is an example call of `compile`, on our `fact` program. We exhibit this as a theorem; its proof is trivial by computation.

```
(defthm example-compilation-1
  (equal (compile ’(n)
                  ’((a = 1)
                    (while (n > 0)
                      (a = (n * a))
                      (n = (n - 1)))
                    (return a)))
         ’((PUSH 1)
           (STORE 1)
           (LOAD 0)
           (IFLE 10)
           (LOAD 0)
           (LOAD 1)
           (MUL)
           (STORE 1)
           (LOAD 0)
           (PUSH 1)
           (SUB)
           (STORE 0)
           (GOTO -10)
           (LOAD 1)
           (RETURN)))
  :rule-classes nil)
```

B Appendix: The Universal M1 Program

In this appendix we illustrate the importance of addressing the question of whether the program has halted when doing clock-style total correctness proofs. We also show that it is possible to prove that M1 programs do not halt. This fact sometimes surprises newcomers to ACL2, since ACL2 requires that all `defun`s terminate.

Here is an amazing (and ambiguous) claim:

> It is possible to define a single, universal, M1 program that can be used to compute every numeric function.

Consider the following M1 program:

```
(defconst *universal-program*
  '((PUSH 0)
    (PUSH 1)
    (ADD)
    (GOTO -2)))
```

Notice that on successive arrivals at pc `1`, the top of the stack is successively each of the naturals. That is, with the appropriate schedule this program can be made to compute any natural – provided it does not have to terminate upon the production of that natural!

Here is a suitable schedule function.

```
(defun universal-sched-loop (k)
  (if (zp k)
      nil
    (append (repeat 0 3)
            (universal-sched-loop (- k 1)))))

(defun universal-sched (k)
  (append (repeat 0 1)
          (universal-sched-loop k)))
```

We will prove that running the program according to `(universal-sched n)` will leave `n` on top of the stack.

We have to prove the loop behaves as expected. We use our standard clock proof methodology, by defining the algorithm and proving that the loop and then the top-level program compute according to the algorithm.

```
(defun universal-algorithm (k n)
  (if (zp k)
      n
    (universal-algorithm (- k 1) (+ 1 n))))

(defthm step-a-run-universal-loop
  (implies (and (natp k)
                (natp n))
           (equal (run (universal-sched-loop k)
                       (make-state 1
                                  locals
                                  (push n stack)
                                  *universal-program*))
                  (make-state 1
                             locals
                             (push (universal-algorithm k n)
                                   stack)
                             *universal-program*))))

(defthm step-a-run-universal
  (implies (natp k)
           (equal (run (universal-sched k)
                       (make-state 0
                                   locals
                                   stack
                                   *universal-program*))
                  (make-state 1
                              locals
                              (push (universal-algorithm k 0)
                                    stack)
                              *universal-program*))))
```

Then, in step (b), we prove the “universal algorithm” is just addition.

```
(defthm step-b
  (implies (and (natp k)
                (natp n))
           (equal (universal-algorithm k n)
                  (+ k n))))
```

So now we know that if the `*universal-program*` is run according to `(univer-sal-sched k)` it pushes `k` on the stack.

From this it is trivial to prove that there exists a schedule that causes `*universal--program*` to push `(fact n)`! What is the appropriate schedule? Obviously it is:

```
(defun new-fact-sched (n)
  (universal-sched (fact n)))
```

And with that we can prove a theorem that looks very much like `ifact-correct--corollary-2`, except that it seems to say that `*universal-program*` is a correct factorial program!

```
(defthm universal-computes-fact
  (equal (top
          (stack
           (run (new-fact-sched n)
                (make-state 0
                            locals
                            stack
```

```
                              *universal-program*))))
         (fact n)))
```

The only thing we have not proved about the universal program is that it halts. In the case of the factorial proof, we do not have `ifact-correct-corollary-1`. Of course, we are not able to prove that because the universal program does not halt.

Indeed, we can prove that `*universal-program*` never halts. Here is the key lemma.

```
(defthm universal-never-halts-lemma
  (implies (and (member (pc s) '(0 1 2 3))
                (equal (program s) *universal-program*))
           (not (haltedp (run sched s)))))
```

from which it is trivial to prove:

```
(defthm universal-never-halts
  (not
   (haltedp
     (run sched
          (make-state 0
                      locals
                      stack
                      *universal-program*)))))
```

References

[1] 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.

[2] R. S. Boyer and J S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[3] 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.

[4] A. D. Flatau. A verified implementation of an applicative language with dynamic storage allocation. Ph.d. thesis, University of Texas at Austin, 1992.

[5] R. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, volume XIX, pages 19–32. American Mathematical Society, Providence, Rhode Island, 1967.

[6] 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.

[7] J. Goldberg, W. Kautz, P. M. Mellear-Smith, M. Green, K. Levitt, R. Schwartz, and C. Weinstock. Development and analysis of the software implemented fault-tolerance (sift) computer. Technical Report NASA Contractor Report 172146, NASA Langley Research Center, Hampton, VA, 1984.

[8] 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.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

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

[11] M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. In *http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz*. Dept. of Computer Sciences, University of Texas at Austin, 1997.

[12] M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2008.

[13] T. Lindholdm and F. Yellin. *The Java Virtual Machine Specification, 2nd edition*. Prentice Hall, 1999.

[14] H. Liu and J S. Moore. Java program verification via a jvm deep embedding in acl2. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *17th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 2004.

[15] Hanbing Liu. *Formal Specification and Verification of a JVM and its Bytecode Verifier*. PhD thesis, University of Texas at Austin, 2006.

[16] P. Manolios and J S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[17] P. Manolios and D. Vroon. Ordinal arithmetic in acl2. In *ACL2 Workshop 2003*, Boulder, Colorado, July 2003. http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/.

[18] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series, Kluwer Academic Publishers, 1996.

[19] J S. Moore. Inductive assertions and operational semantics. In D. Geist, editor, *Proceedings of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 289–303. Springer Verlag, 2003.

[20] J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.

[21] J S. Moore and M. Martinez. An acl2 proof of the correctness of the boyer-moore string searching algorithm. Technical report, Department of Computer Sciences, University of Texas at Austin, 2008.

[22] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[23] Sandip Ray, Warren A. Hunt Jr., John Matthews, and J Strother Moore. A mechanical analysis of program verification strategies. *Journal of Automated Reasoning*, 40(4):245–269, May 2008.

[24] 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.

[25] D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA., 2000. Kluwer Academic Press.

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