```lisp
#|
This file corresponds to the paper

Mechanized Formal Reasoning about Programs and Computing Machines

Robert S. Boyer and J Strother Moore


(defpkg "SMALL-MACHINE"
  (union-eq
   (delete1-eq 'pc
               (delete1-eq 'state *acl2-exports*))
   (append '(true-listp zp nfix fix len quotep defevaluator syntaxp)
           (delete1-eq 'pi
                       (delete1-eq 'step
                                   *common-lisp-symbols-from-main-lisp-package*)))))

|#

(in-package "SMALL-MACHINE")

(defun statep (x)
  (and (true-listp x)
       (equal (len x) 5)))

(defun state (pc stk mem halt code) (list pc stk mem halt code))

(defun pc (s) (nth 0 s))
(defun stk (s) (nth 1 s))
(defun mem (s) (nth 2 s))
(defun halt (s) (nth 3 s))
(defun code (s) (nth 4 s))

(defmacro modify (s &key (pc '0 pcp)
                    (stk 'nil stkp)
                    (mem 'nil memp)
                    (halt 'nil haltp)
                    (code 'nil codep))
  `(state ,(if pcp pc `(pc ,s))
          ,(if stkp stk `(stk ,s))
          ,(if memp mem `(mem ,s))
          ,(if haltp halt `(halt ,s))
          ,(if codep code `(code ,s))))

(defmacro st (&rest args)
  `(modify nil ,@args))

; Utility Functions

(defun put (n v mem)
  (if (zp n)
      (cons v (cdr mem))
      (cons (car mem) (put (1- n) v (cdr mem)))))

(defun fetch (pc code)
  (nth (cdr pc)
       (cdr (assoc (car pc) code))))

(defun current-instruction (s)
  (fetch (pc s) (code s)))

(defun opcode (ins) (nth 0 ins))
(defun a (ins) (nth 1 ins))
(defun b (ins) (nth 2 ins))
(defun pc+1 (pc)
  (cons (car pc) (+ 1 (cdr pc))))

; The Semantics of Individual Instructions

; Move Instructions

(defun move (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a (nth b (mem s)) (mem s))))

(defun movi (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a b (mem s))))

; Arithmetic Instructions

(defun add (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a
                    (+ (nth a (mem s))
                       (nth b (mem s)))
                    (mem s))))

(defun subi (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a
                    (- (nth a (mem s)) b)
                    (mem s))))

; Jump Instructions

(defun jumpz (a b s)
  (modify s
          :pc (if (zp (nth a (mem s)))
                  (cons (car (pc s)) b)
                (pc+1 (pc s)))))

(defun jump (a s)
  (modify s :pc (cons (car (pc s)) a)))

; Subroutine Call and Return

(defun call (a s)
  (modify s
          :pc (cons a 0)
          :stk (cons (pc+1 (pc s)) (stk s))))

(defun ret (s)
  (if (endp (stk s))
      (modify s :halt t)
      (modify s
              :pc (car (stk s))
              :stk (cdr (stk s)))))

; One can imagine adding new instructions.

; The Interpreter

(defun execute (ins s)
  (let ((op (opcode ins))
```

```
        (a (a ins))
        (b (b ins)))
    (case op
          (move  (move a b s))
          (movi  (movi a b s))
          (add   (add a b s))
          (subi  (subi a b s))
          (jumpz (jumpz a b s))
          (jump  (jump a s))
          (call  (call a s))
          (ret   (ret s))
          (otherwise s))))

(defun step (s)
  (if (halt s)
      s
      (execute (current-instruction s) s)))

(defun sm (s n)
  (if (zp n)
      s
      (sm (step s) (+ n -1))))


(defun natp (x)
  (and (integerp x)
       (<= 0 x)))

(defun cplus (i j)
  (if (zp i)
      (nfix j)
      (+ 1 (cplus (1- i) j))))

(defun ctimes (i j)
  (if (zp i) 0 (cplus j (ctimes (1- i) j))))


; Now we move to our first example program.  We will define a program
; that multiplies two naturals by successive addition.  We will then
; prove it correct.

; The program we have in mind is:

; (times (movi 2 0)
;        (jumpz 0 5)
;        (add 2 1)
;        (subi 0 1)
;        (jump 1)
;        (ret))

; Observe that the program multiplies the contents of location 0 by the
; contents of location 1 and leaves the result in location 2.  At the end,
; location 0 is 0 and location 1 is unchanged.  If we start at a (call times)
; this program requires 2+4i+2 instructions, where i is the initial contents of
; location 0.

; We start by defining the constant that is this program:

(defun times-program nil
;         instruction   pc     comment
  '(times (movi 2 0)   ; 0  mem[2] <- 0
          (jumpz 0 5)  ; 1  if mem[0]=0, go to 5
          (add 2 1)    ; 2  mem[2] <- mem[1] + mem[2]
          (subi 0 1)   ; 3  mem[0] <- mem[0] - 1
          (jump 1)     ; 4  go to 1
          (ret)))      ; 5  return to caller

; Here is a state that computes 7*11.

(defun demo-state nil
  (st :pc   '(times . 0)
      :stk   nil
      :mem  '(7 11 3 4 5)
      :halt nil
      :code (list (times-program))))

; And a trivial theorem to prove it:

(defthm demo-theorem
  (equal (sm (demo-state) 31)
         (st :pc   '(times . 5)
             :stk  nil
             :mem  '(0 11 77 4 5)
             :halt t
             :code (list (times-program)))))

; The clock function for times:
(defun times-clock (i)
  (cplus 2 (cplus (ctimes i 4) 2)))


; And a trivial theorem to prove it:
(thm (equal
      (sm (st :pc   '(times . 0)
              :stk   nil
              :mem  '(500 11 3 4 5)
              :halt nil
              :code (list (times-program)))
          (times-clock 500))
      (sm (st :pc   '(times . 0)
              :stk   nil
              :mem  '(500 11 3 4 5)
              :code (list (times-program)))
          (times-clock 500))))

; Takes about 21 seconds.

(comp t)

; Now, the above takes .05 seconds

(defthm times-correct
  (implies (and (statep s0)
                (< 2 (len (mem s0)))
                (equal i (nth 0 (mem s0)))
                (equal j (nth 1 (mem s0)))
                (natp i)
                (natp j)
                (equal (current-instruction s0) '(call times))
                (equal (assoc 'times (code s0)) (times-program))
                (not (halt s0)))
           (equal (sm s0 (times-clock i))
                  (modify s0
                          :pc (pc+1 (pc s0))
                          :mem (put 0 0
                                (put 2 (* i j) (mem s0)))))))

; We now consider the role of subroutine call and return in this
; language.  To illustrate it we'll implement exponentiation, which
; will CALL our TIMES program.  The proof of the correctness of the
```

```
; exponentiation program will rely on the correctness of TIMES, not on
; re-analysis of the code for TIMES.

; The mathematical function we wish to implement is (expt i j), where
; i and j are naturals.

; The program we have in mind is:

(defun expt-program nil
  '(expt (move 3 0)   ; 0  mem[3] <- mem[0] (save args)
         (move 4 1)   ; 1  mem[4] <- mem[1]
         (movi 1 1)   ; 2  mem[1] <- 1      (initialize ans)
         (jumpz 4 9)  ; 3  if mem[4]=0, go to 9
         (move 0 3)   ; 4  mem[0] <- mem[3] (prepare for times)
         (call times) ; 5  mem[2] <- mem[0] * mem[1]
         (move 1 2)   ; 6  mem[1] <- mem[2]
         (subi 4 1)   ; 7  mem[4] <- mem[4]-1
         (jump 3)     ; 8  go to 3
         (ret)))      ; 9  return

; This program computes (expt mem[0] mem[1]) and leaves the result in mem[1].
; Because we use times (which requires repeatedly loading mem[0] and mem[1] to
; pass in its parameters) and because times smashes mem[2] with its result, we
; will use mem[3] and mem[4] as our "locals."  We will use mem[1] as our
; running answer, which starts at 1.  After moving mem[0] and mem[1] to mem[3]
; and mem[4] respectively and initializing our running answer to 1, we just
; multiply mem[3] by mem[1] (mem[4] times), moving the product back into mem[1]
; after each multiplication.

; Here is the clock function for expt.  Again we use an algebraically
; odd form simply to gain instant access to the desired sm-plus
; decomposition.  The "4" nths us past the CALL and the first 3
; initialization instructions; the times exptression takes us around
; the expt loop j times, and the final "2" nths us out through the RET.
; Note that as we go around the loop we make explicit reference to
; TIMES-CLOCK to explain the CALL of TIMES.

(defun expt-clock (i j)
  (cplus 4
         (cplus (ctimes j (cplus 2 (cplus (times-clock i) 3)))
                2)))

(defthm expt-correct
  (implies (and (statep s0)
                (< 4 (len (mem s0)))
                (equal i (nth 0 (mem s0)))
                (equal j (nth 1 (mem s0)))
                (natp i)
                (natp j)
                (equal (current-instruction s0) '(call expt))
                (equal (assoc 'expt (code s0)) (expt-program))
                (equal (assoc 'times (code s0)) (times-program))
                (not (halt s0)))
           (equal (sm s0 (expt-clock i j))
                  (modify s0
                          :pc (pc+1 (pc s0))
                          :mem
                          (if (zp j)
                              (put 1 (expt i j)
                               (put 3 i
                                (put 4 0 (mem s0))))
                              (put 0 0
                               (put 1 (expt i j)
                                (put 2 (expt i j)
                                 (put 3 i
                                  (put 4 0 (mem s0))))))))))))
```