ACL2 Code Proofs

J Strother Moore
Fall, 2013

Lecture 1

About this Course

Very fast, very simple, focused entirely on ACL2 users

Training: Students taught to use tools they don't understand

Education: Students taught to understand tools they can't use

This course errs on the 'education' side. It is one thing to understand in principle how code proofs are done and another to do them at X86-scale. But I want you to *understand*.

The M1 Directory

The M1 model and many proofs about it are distributed as part of the ACL2 Community Books. See

```
books/models/jvm/m1/
```

on `:dir :system`.

Henceforth, I call this the "M1 directory."

The M1 Package

The M1 symbol package imports all the usual ACL2 symbols except

- `push`, `pop`, `pc`, `program`, `step` – reserved for concepts within the M1 semantics

- `nth`, `update-nth`, `nth-update-nth` – to force JVM students to define some elementary list processing functions

Nth and Update-nth

My m1.lisp file defines these two functions and provides the following rules

```
(defthm nth-add1!
  (implies (natp n)
           (equal (nth (+ 1 n) list)
                  (nth n (cdr list)))))
(defthm nth-update-nth
  (implies (and (natp i) (natp j))
           (equal (nth i (update-nth j v list))
                  (if (equal i j)
                      v
                    (nth i list)))))
```

```
(defthm update-nth-update-nth-2 ; at same index
  (equal (update-nth i v (update-nth i w list))
         (update-nth i v list)))
```

```
(defthm update-nth-update-nth-1 ; at different indices
  (implies
   (and (natp i)
        (natp j)
        (not (equal i j)))
   (equal (update-nth i v (update-nth j w list))
          (update-nth j w (update-nth i v list))))

  :rule-classes
  ((:rewrite :loop-stopper ((i j update-nth)))))
```

Terminology

An *instruction* contains an opcode and some operands or “arguments.”

A *program* is a list of instructions.

An M1 *state* is determined by

- a program counter
- a sequence of local variable values
- a stack
- a program

M1 Instruction Set

Operation	short description
Format	layout of opcode and args
Stack	*stack before* $\Rightarrow$ *stack after*
Description	longer description

Stacks are displayed with the topmost item on the right. Unless otherwise noted, the program counter is always incremented by one.

ILOAD

Operation	push local n
Format	`(ILOAD` n`)`
Stack	$\ldots \Rightarrow \ldots, v$
Description	The value v of local variable n is pushed onto the stack.

ICONST

Operation	push constant
Format	`(ICONST` c`)`
Stack	$\ldots \Rightarrow \ldots, c$
Description	The constant c is pushed onto the stack.

IADD

Operation	add two integers
Format	`(IADD)`
Stack	$\ldots, v_1, v_2 \Rightarrow \ldots, r$
Description	Both v_1 and v_2 must be integers. The values are popped from the stack. Their sum, r, is pushed onto the stack.

ISUB

Operation	subtract two integers
Format	`(ISUB)`
Stack	$\ldots, v_1, v_2 \Rightarrow \ldots, r$
Description	Both v_1 and v_2 must be integers. The values are popped from the stack. The result, r, is $v_1 - v_2$ and is pushed onto the stack.

IMUL

Operation	multiply two integers
Format	`(IMUL)`
Stack	$\ldots, v_1, v_2 \Rightarrow \ldots, r$
Description	Both v_1 and v_2 must be integers. The values are popped from the stack. Their product, r, is pushed onto the stack.

ISTORE

Operation	store into local n
Format	`(ISTORE` n`)`
Stack	$\ldots, v \Rightarrow \ldots$
Description	The value, v, on top of the stack is removed and stored into local n.

GOTO

Operation	jump by n
Format	`(GOTO` n`)`
Stack	$\ldots \Rightarrow \ldots$
Description	Execution proceeds at offset n from this instruction, where n may be positive or negative. The target address must be in the current program.

IFEQ

Operation	conditional jump by n
Format	`(IFEQ` n`)`
Stack	$\ldots, v \Rightarrow \ldots$
Description	Execution proceeds at offset n from this instruction if v is 0 and at the next instruction otherwise. Pop the stack.

Halting

Any instruction other than one of those above halts the machine.

It is convenient to introduce a canonical *halt* instruction with opcode `HALT`

ACL2 Definition of M1

See the community book:

```
(include-book "models/jvm/m1/m1" :dir :system)

(in-package "M1")
```

Demo

An M1 Program

```
((ICONST 0)   ; 0
 (ISTORE 2)   ; 1  a = 0;
 (ILOAD 0)    ; 2  [loop:]
 (IFEQ 10)    ; 3  if x=0 then go to end;
 (ILOAD 0)    ; 4
 (ICONST 1)   ; 5
 (ISUB)       ; 6
 (ISTORE 0)   ; 7  x = x-1;
 (ILOAD 1)    ; 8
 (ILOAD 2)    ; 9
 (IADD)       ;10
 (ISTORE 2)   ;11  a = y+a;
 (GOTO -10)   ;12  go to loop
 (ILOAD 2)    ;13  [end:]
 (HALT))      ;14 ``return'' a
```

Demo

How Long Does *pi* Run?

```
(defconst *pi*
       '((ICONST 0)    ; 0
         (ISTORE 2)    ; 1  a = 0;
         (ILOAD 0)     ; 2  [loop:]
         (IFEQ 10)     ; 3  if x=0 then go to end;
         (ILOAD 0)     ; 4
         (ICONST 1)    ; 5
         (ISUB)        ; 6
         (ISTORE 0)    ; 7  x = x-1;
         (ILOAD 1)     ; 8
         (ILOAD 2)     ; 9
         (IADD)        ;10
         (ISTORE 2)    ;11  a = y+a;
         (GOTO -10)    ;12  go to loop
         (ILOAD 2)     ;13  [end:]
         (HALT)))      ;14 ``return'' a
```

PC at Loop and X=0

```
(defconst *pi*
        '((ICONST 0)    ; 0
          (ISTORE 2)    ; 1  a = 0;
          (ILOAD 0)     ; 2  [loop:]
          (IFEQ 10)     ; 3  if x=0 then go to end;
          (ILOAD 0)     ; 4
          (ICONST 1)    ; 5
          (ISUB)        ; 6
          (ISTORE 0)    ; 7  x = x-1;
          (ILOAD 1)     ; 8
          (ILOAD 2)     ; 9
          (IADD)        ;10
          (ISTORE 2)    ;11  a = y+a;
          (GOTO -10)    ;12  go to loop
          (ILOAD 2)     ;13  [end:]
          (HALT)))      ;14 ``return'' a
```

PC at Loop and X>0

```
(defconst *pi*
        '((ICONST 0)    ; 0
          (ISTORE 2)    ; 1   a = 0;
          (ILOAD 0)     ; 2   [loop:]
          (IFEQ 10)     ; 3   if x=0 then go to end;
          (ILOAD 0)     ; 4
          (ICONST 1)    ; 5
          (ISUB)        ; 6
          (ISTORE 0)    ; 7   x = x-1;
          (ILOAD 1)     ; 8
          (ILOAD 2)     ; 9
          (IADD)        ;10
          (ISTORE 2)    ;11   a = y+a;
          (GOTO -10)    ;12   go to loop
          (ILOAD 2)     ;13   [end:]
          (HALT)))      ;14 ``return'' a
```

PC at Top

```
(defconst *pi*
        '((ICONST 0)     ; 0
          (ISTORE 2)     ; 1   a = 0;
          (ILOAD 0)      ; 2   [loop:]
          (IFEQ 10)      ; 3   if x=0 then go to end;
          (ILOAD 0)      ; 4
          (ICONST 1)     ; 5
          (ISUB)         ; 6
          (ISTORE 0)     ; 7   x = x-1;
          (ILOAD 1)      ; 8
          (ILOAD 2)      ; 9
          (IADD)         ;10
          (ISTORE 2)     ;11   a = y+a;
          (GOTO -10)     ;12   go to loop
          (ILOAD 2)      ;13   [end:]
          (HALT)))       ;14 ``return'' a
```

The Clock for *pi*

```
(defun loop-clk (x)
  (if (zp x)
      3
      (clk+ 11
            (loop-clk (- x 1)))))

(defun clk (x)
  (clk+ 2
        (loop-clk x)))
```

Demo

Turing Equivalence

M1 is a trivial machine, but it is *Turing Equivalent*: anything a Turing Machine can do, M1 can do.

This can be proved with ACL2. See

`turing-equivalence-talk.pdf`

on the M1 directory. For a guide to the proof files, see the discussion of “The Turing Equivalence of M1” in the `README` file of the M1 directory.

Symbolic Evaluation

(m1 s 11)

$=$

(m1 (step s) 10)

$=$

$\dots$

$=$

(step (step (step ... (step s)...)))

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(m1                                    ; 2. (ILOAD 0)
 (make-state 2                         ; 3. (IFEQ 10)
             (list x y a)              ; 4. (ILOAD 0)
             nil                       ; 5. (ICONST 1)
             *pi*)                     ; 6. (ISUB)
 11)                                   ; 7. (ISTORE 0)
                                       ; 8. (ILOAD 1)
                                       ; 9. (ILOAD 2)
                                       ;10. (IADD)
                                       ;11. (ISTORE 2)
                                       ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(m1                                    ; 2. (ILOAD 0)
 (make-state 3                         ; 3. (IFEQ 10)
             (list x y a)              ; 4. (ILOAD 0)
             (push x nil)              ; 5. (ICONST 1)
             *pi*)                     ; 6. (ISUB)
 10)                                   ; 7. (ISTORE 0)
                                       ; 8. (ILOAD 1)
                                       ; 9. (ILOAD 2)
                                       ;10. (IADD)
                                       ;11. (ISTORE 2)
                                       ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(m1                                       ; 2. (ILOAD 0)
 (make-state 4                            ; 3. (IFEQ 10)
             (list x y a)                 ; 4. (ILOAD 0)
             nil                          ; 5. (ICONST 1)
             *pi*)                        ; 6. (ISUB)
 9)                                       ; 7. (ISTORE 0)
                                          ; 8. (ILOAD 1)
                                          ; 9. (ILOAD 2)
                                          ;10. (IADD)
                                          ;11. (ISTORE 2)
                                          ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(m1                                        ; 2. (ILOAD 0)
 (make-state 5                             ; 3. (IFEQ 10)
             (list x y a)                  ; 4. (ILOAD 0)
             (push x nil)                  ; 5. (ICONST 1)
             *pi*)                         ; 6. (ISUB)
 8)                                        ; 7. (ISTORE 0)
                                           ; 8. (ILOAD 1)
                                           ; 9. (ILOAD 2)
                                           ;10. (IADD)
                                           ;11. (ISTORE 2)
                                           ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(m1                                          ; 2. (ILOAD 0)
 (make-state 6                               ; 3. (IFEQ 10)
             (list x y a)                    ; 4. (ILOAD 0)
             (push 1 (push x nil))           ; 5. (ICONST 1)
             *pi*)                           ; 6. (ISUB)
 7)                                          ; 7. (ISTORE 0)
                                             ; 8. (ILOAD 1)
                                             ; 9. (ILOAD 2)
                                             ;10. (IADD)
                                             ;11. (ISTORE 2)
                                             ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and ¬`(zp x)`.

```
(m1                                      ; 2. (ILOAD 0)
 (make-state 7                           ; 3. (IFEQ 10)
             (list x y a)                ; 4. (ILOAD 0)
             (push (- x 1) nil)          ; 5. (ICONST 1)
             *pi*)                       ; 6. (ISUB)
 6)                                      ; 7. (ISTORE 0)
                                         ; 8. (ILOAD 1)
                                         ; 9. (ILOAD 2)
                                         ;10. (IADD)
                                         ;11. (ISTORE 2)
                                         ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(m1                                      ; 2. (ILOAD 0)
 (make-state 8                           ; 3. (IFEQ 10)
             (list (- x 1) y a)          ; 4. (ILOAD 0)
             nil                         ; 5. (ICONST 1)
             *pi*)                       ; 6. (ISUB)
 5)                                      ; 7. (ISTORE 0)
                                         ; 8. (ILOAD 1)
                                         ; 9. (ILOAD 2)
                                         ;10. (IADD)
                                         ;11. (ISTORE 2)
                                         ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(m1                                         ; 2. (ILOAD 0)
 (make-state 9                              ; 3. (IFEQ 10)
             (list (- x 1) y a)             ; 4. (ILOAD 0)
             (push y nil)                   ; 5. (ICONST 1)
             *pi*)                          ; 6. (ISUB)
 4)                                         ; 7. (ISTORE 0)
                                            ; 8. (ILOAD 1)
                                            ; 9. (ILOAD 2)
                                            ;10. (IADD)
                                            ;11. (ISTORE 2)
                                            ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x,y,a` and ¬`(zp x)`.

```
(m1                                          ; 2. (ILOAD 0)
 (make-state 10                              ; 3. (IFEQ 10)
             (list (- x 1) y a)              ; 4. (ILOAD 0)
             (push a (push y nil))           ; 5. (ICONST 1)
             *pi*)                           ; 6. (ISUB)
 3)                                          ; 7. (ISTORE 0)
                                             ; 8. (ILOAD 1)
                                             ; 9. (ILOAD 2)
                                             ;10. (IADD)
                                             ;11. (ISTORE 2)
                                             ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and ¬`(zp x)`.

```
(m1                                          ; 2. (ILOAD 0)
 (make-state 11                              ; 3. (IFEQ 10)
             (list (- x 1) y a)              ; 4. (ILOAD 0)
             (push (+ y a) nil)              ; 5. (ICONST 1)
             *pi*)                           ; 6. (ISUB)
 2)                                          ; 7. (ISTORE 0)
                                             ; 8. (ILOAD 1)
                                             ; 9. (ILOAD 2)
                                             ;10. (IADD)
                                             ;11. (ISTORE 2)
                                             ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(m1                                          ; 2. (ILOAD 0)
 (make-state 12                              ; 3. (IFEQ 10)
             (list (- x 1) y (+ y a))        ; 4. (ILOAD 0)
             nil                             ; 5. (ICONST 1)
             *pi*)                           ; 6. (ISUB)
 1)                                          ; 7. (ISTORE 0)
                                             ; 8. (ILOAD 1)
                                             ; 9. (ILOAD 2)
                                             ;10. (IADD)
                                             ;11. (ISTORE 2)
                                             ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(m1                                      ; 2. (ILOAD 0)
 (make-state 2                           ; 3. (IFEQ 10)
             (list (- x 1) y (+ y a))    ; 4. (ILOAD 0)
             nil                         ; 5. (ICONST 1)
             *pi*)                       ; 6. (ISUB)
 0)                                      ; 7. (ISTORE 0)
                                         ; 8. (ILOAD 1)
                                         ; 9. (ILOAD 2)
                                         ;10. (IADD)
                                         ;11. (ISTORE 2)
                                         ;12. (GOTO -10)
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and ¬`(zp x)`.

```
(make-state 2                              ; 2. (ILOAD 0)
            (list (- x 1) y (+ y a))       ; 3. (IFEQ 10)
            nil                            ; 4. (ILOAD 0)
            *pi*)                          ; 5. (ICONST 1)
                                           ; 6. (ISUB)
                                           ; 7. (ISTORE 0)
                                           ; 8. (ILOAD 1)
                                           ; 9. (ILOAD 2)
                                           ;10. (IADD)
                                           ;11. (ISTORE 2)
                                           ;12. (GOTO -10)
```

Symbolic Evaluation

Possible when `next-inst` and clock are fixed.

But case explosion is likely

(step (step ... (step (step s))...))

if the outer steps are expanded before it is possible to determine the `next-inst` produced by the inner step.

How do we make ACL2 do symbolic evaluation?

Controlling Symbolic Evaluation

```
; Abstract Data Type Stuff

(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)))

(in-theory (disable push top pop
                    (:executable-counterpart push)))
```

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

  (equal (pc (cons pc x))                       pc)
  (equal (locals (cons pc (conslocs x)))        locs)
  ...))

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

```
; Step Stuff

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

(defthm m1-opener
  (and (equal (m1 s 0) s)
       (implies (natp i)
                (equal (m1 s (+ 1 i))
                       (m1 (step s) i)))))
(in-theory (disable m1))
```

Demo

Stating Correctness

Partial Correctness:
If computation on ok inputs halts, then the answer is as expected.

Total Correctness:
Computation from ok inputs halts and the answer is as expected.

State-Based Formalization

The most general way to formulate these conditions is in terms of entire M1 states.

`(pre` s_i`)` – checks that s_i is an acceptable initial state for your program

`(post` s_i s_f`)` – checks that the final state, s_f, is in the expected relation with the initial state s_i

Most-General State Based Partial Correctness

$\forall\ s_i, \kappa$:

((pre s_i)

 $\wedge$

 (haltedp (m1 s_i κ))

$\rightarrow$

 (post s_i (m1 s_i κ))

Most-General State Based Total Correctness

$\forall\ s_i\ \exists\ \kappa$:

(pre s_i)

$\rightarrow$

((haltedp (m1 s_i κ))

$\wedge$

(post s_i (m1 s_i κ)))

However...

It is generally more convenient to relate program inputs and outputs rather than whole states.

Partial Correctness

"If computation on ok inputs halts, then the answer is as expected."

$$
\begin{array}{l}
(\quad s_i = \texttt{(make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)} \\
\quad \wedge\ \texttt{(ok-inputs } v_1 \ldots\texttt{)} \\
\quad \wedge\ s_f = \texttt{(m1 } s_i\ \kappa\texttt{)} \\
\quad \wedge\ \texttt{(haltedp } s_f\texttt{))} \\
\rightarrow \\
\texttt{(top (stack } s_f\texttt{))} = \theta\texttt{.}
\end{array}
$$

Partial Correctness (in ACL2)

```
(implies
 (and
  (equal $s_i$ (make-state 0 (list $v_1 \ldots$) nil $\pi$))
  (ok-inputs $v_1 \ldots$)
  (equal $s_f$ (m1 $s_i$ $\kappa$))
  (haltedp $s_f$))
 (equal (top (stack $s_f$)) $\theta$))
```

You may wish to re-state this as a more effective rule.

Total Correctness

“Computation from ok inputs halts and the answer is as expected.”

$\exists\ \kappa$:

 (s_i = (make-state 0 (list $v_1 \ldots$) nil π)

  $\wedge$ (ok-inputs $v_1 \ldots$)

  $\wedge$ s_f = (m1 s_i κ))

$\rightarrow$

 ((haltedp s_f)

  $\wedge$

   (top (stack s_f)) = θ).

Total Correctness (in ACL2)

“Computation from ok inputs halts and the answer is as expected.”

$\exists\ \kappa$:

```
(implies
 (and
  (equal $s_i$ (make-state 0 (list $v_1 \ldots$) nil $\pi$))
  (ok-inputs $v_1 \ldots$)
  (equal $s_f$ (m1 $s_i$ $\kappa$)))
 (and (haltedp $s_f$)
      (equal (top (stack $s_f$)) $\theta$)))
```

Total Correctness (in ACL2)

"Computation from ok inputs halts and the answer is as expected."

$\exists\ \kappa$:

```
(implies
 (and
  (equal $s_i$ (make-state 0 (list $v_1 \ldots$) nil $\pi$))
  (ok-inputs $v_1 \ldots$)
  (equal $s_f$ (m1 $s_i$ $\kappa$)))
 (and (haltedp $s_f$)
      (equal (top (stack $s_f$)) $\theta$)))
```

Total Correctness (in ACL2)

"Computation from ok inputs halts and the answer is as expected."

```
(implies
 (and
  (equal $s_i$ (make-state 0 (list $v_1\ldots$) nil $\pi$))
  (ok-inputs $v_1\ldots$)
  (equal $s_f$ (m1 $s_i$ (clk $v_1\ldots$))))
 (and (haltedp $s_f$)
      (equal (top (stack $s_f$)) $\theta$)))
```

Bogus Total Correctness (in ACL2)

"Computation from ok inputs eventually produces the expected answer"

(implies
 (and
  (equal s_i (make-state 0 (list $v_1 \ldots$) nil π))
  (ok-inputs $v_1 \ldots$)
  (equal s_f (m1 s_i (clk $v_1 \ldots$))))
 (and (haltedp s_f)
      (equal (top (stack s_f)) θ)))

Bogus Total Correctness (in ACL2)

"Computation from ok inputs eventually produces the expected answer"

```
(implies
 (and
  (equal $s_i$ (make-state 0 (list $v_1 \ldots$) nil $\pi$))
  (ok-inputs $v_1 \ldots$)
  (equal $s_f$ (m1 $s_i$ (clk $v_1 \ldots$))))
 (equal (top (stack $s_f$)) $\theta$))
```

Bogus Correctness

It is possible to prove that the program:

```
((ICONST 0)    ; 0   tos = 0;
 (ICONST 1)    ; 1 loop:
 (IADD)        ; 2   tos = tos+1;
 (GOTO -2))    ; 3   goto loop;
```

satisfies the Bogus Correctness Theorem for any natural number producing function, i.e., this Π is a "correct" implementation of every natural number-producing function!

Magic.lisp on M1 Directory

```
(defthm pi-bogusly-computes-fact
 (implies
  (and (natp n)
       (equal sf (m1 (make-state 0 (list n) nil Π)
                     (fact-clk n))))
  (equal (top (stack sf)) (fact n))))

(defthm pi-bogusly-computes-fib
  (implies
   (and (natp n)
        (equal sf (m1 (make-state 0 (list n) nil Π)
                      (fib-clk n))))
   (equal (top (stack sf)) (fib n))))
```

Now Back to Total Correctness

How do we get ACL2 to do total correctness proofs?

Moving Parts of M1 Proof Engine

- clock functions $\surd$
- symbolic evaluation $\surd$
- sequential composition
- how they mesh
- behavior v intention
- dealing with loops
- separation of concerns

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

a.k.a.

```
(defthm m1-clk+
  (equal (m1 s (clk+ i j))
         (m1 (m1 s i) j)))
```

where, for reasons explained later

`(clk+ i j)` $=$ `(+ i j)`

and `clk+` is disabled.

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. `By induction on i.`

Aside on Induction

Thm. $\forall$ `i,j,s : (`ϕ` i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow$ $\forall$ `j,s : (`ϕ` i j s)`

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (ϕ (- i 1) j s))
→
 ∀ j,s :  (ϕ i j s).
```

Aside on Induction

Thm. $\forall$ `i,j,s :` `(`ϕ `i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow$ $\forall$ `j,s :` `(`ϕ `i j s)`

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (φ (- i 1) j s))
→
 ∀ j,s :  (φ i j s).
```

Aside on Induction

Thm. `(`ϕ` i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow \forall$ `j,s : (`ϕ` i j s)`

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (φ (- i 1) j s))
→
 ∀ j,s :  (φ i j s).
```

Aside on Induction

Thm. `(`ϕ` i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow$ $\forall$ `j,s :` `(`ϕ` i j s)`

Induction Step:

```
( ¬(zp i)
   ∧
    ∀ j,s : (φ (- i 1) j s))
→
 ∀ j,s :  (φ i j s).
```

Aside on Induction

Thm. (ϕ i j s)

Proof. By induction on i.

Base:

(zp i) $\rightarrow$ (ϕ i j s)

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (ϕ (- i 1) j s))
→
 ∀ j,s :  (ϕ i j s).
```

Aside on Induction

Thm. `(`ϕ` i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow$ `(`ϕ` i j s)`

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (φ (- i 1) j s))
→
 ∀ j,s :  (φ i j s).
```

Aside on Induction

Thm. `(`ϕ` i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow$ `(`ϕ` i j s)`

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (φ (- i 1) j s))
→
 (φ i j s).
```

Aside on Induction

Thm. (ϕ i j s)

Proof. By induction on i.

Base:

(zp i) $\rightarrow$ (ϕ i j s)

Induction Step:

```
 ( ¬(zp i)
   ∧
    ∀ j,s : (φ (- i 1) j s))
→
 (φ i j s).
```

Aside on Induction

Thm. $(\phi$ i j s)

Proof. By induction on i.

Base:

(zp i) $\rightarrow$ $(\phi$ i j s)

Induction Step:

($\neg$(zp i)

$\wedge$ $(\phi$ (- i 1) α_1 β_1)

$\wedge$ $\forall$ j,s : $(\phi$ (- i 1) j s))

$\rightarrow$

$(\phi$ i j s).

Aside on Induction

Thm. (ϕ i j s)

Proof. By induction on i.

Base:

(zp i) $\rightarrow$ (ϕ i j s)

Induction Step:

($\neg$(zp i)

$\wedge$ (ϕ (- i 1) α_1 β_1) $\wedge$ (ϕ (- i 1) α_2 β_2)

$\wedge$ $\forall$ j,s : (ϕ (- i 1) j s))

$\rightarrow$

(ϕ i j s).

Aside on Induction

Thm. (ϕ i j s)

Proof. By induction on i.

Base:

(zp i) $\rightarrow$ (ϕ i j s)

Induction Step:

($\neg$(zp i)

$\wedge$ (ϕ (- i 1) α_1 β_1) ... $\wedge$ (ϕ (- i 1) α_k β_k)

$\wedge$ $\forall$ j,s : (ϕ (- i 1) j s))

$\rightarrow$

(ϕ i j s).

Aside on Induction

Thm. `(`ϕ` i j s)`

Proof. `By induction on i.`

Base:

`(zp i)` $\rightarrow$ `(`ϕ` i j s)`

Induction Step:

```
 ( ¬(zp i)
   ∧ (φ (- i 1) α_1 β_1) ... ∧ (φ (- i 1) α_k β_k)
                                  )
→
 (φ i j s).
```

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

Base:

```
 (zp i)
```

$\rightarrow$

```
 (m1 s (+ i j)) = (m1 (m1 s i) j)
```

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. `By induction on i.`

Base:

```
 (zp i)
```

$\rightarrow$

```
 (m1 s (+ i j)) = (m1 (m1 s i) j)
```

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. By induction on i.

Base:

```
 (zp i)
→
 (m1 s j) = (m1 (m1 s i) j)
```

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. `By induction on i.`

Base:

`(zp i)`

$\rightarrow$

`(m1 s j) = (m1 (m1 s i) j)`

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. `By induction on i.`

Base:

```
 (zp i)
→
 (m1 s j) = (m1 s j)
```

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. `By induction on i.`

Base:

```
 (zp i)
→
 (m1 s j) = (m1 s j)
```
$\surd$

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. By induction on i.

Base: $\checkmark$

Induction Step:

$$
\begin{array}{l}
(\ \neg\texttt{(zp i)} \\
\quad \wedge \\
\qquad \texttt{(m1 } \alpha \texttt{ (+ (- i 1) } \beta\texttt{))} \\
\qquad\ = \texttt{(m1 (m1 } \alpha \texttt{ (- i 1)) } \beta\texttt{))} \\
\rightarrow \\
\ \texttt{(m1 s (+ i j))} \\
\qquad = \texttt{(m1 (m1 s i) j)}
\end{array}
$$

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. By induction on `i`.

Base: $\checkmark$

Induction Step:

```
( ¬(zp i)
   ∧
    (m1 α (+ (- i 1) β))
     = (m1 (m1 α (- i 1)) β))
→
 (m1 s (+ i j))
    = (m1 (m1 s i) j)
```

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. By induction on i.

Base: ✓

Induction Step:

```
 ( ¬(zp i)
   ∧
    (m1 α (+ (- i 1) β))
     = (m1 (m1 α (- i 1)) β))
→
 (m1 s (+ i j))
    = (m1 (m1 s i) j)
```

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

Base: $\surd$

Induction Step:

($\neg$`(zp i)`

$\wedge$

`(m1` α `(+ (- i 1)` β`))`

$=$ `(m1 (m1` α `(- i 1))` β`))`

$\rightarrow$

`(m1 (step s) (+ (- i 1) j))`

$=$ `(m1 (m1 s i) j)`

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

Base: $\surd$

Induction Step:

```
 ( ¬(zp i)
   ∧
    (m1 α (+ (- i 1) β))
     = (m1 (m1 α (- i 1)) β))
→
 (m1 (step s) (+ (- i 1) j))
    = (m1 (m1 s i) j)
```

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

Base: $\surd$

Induction Step:

($\neg$`(zp i)`

$\wedge$

`(m1` α `(+ (- i 1)` β`))`

$=$ `(m1 (m1` α `(- i 1))` β`))`

$\rightarrow$

`(m1 (step s) (+ (- i 1) j))`

$=$ `(m1 (m1 (step s) (- i 1)) j)`

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

Base: $\surd$

Induction Step:

```
( ¬(zp i)
   ∧
    (m1 α (+ (- i 1) β))
     = (m1 (m1 α (- i 1)) β))
→
 (m1 (step s) (+ (- i 1) j))
    = (m1 (m1 (step s) (- i 1)) j)
```

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

Base: $\surd$

Induction Step:

```
 ( ¬(zp i)
   ∧
    (m1 (step s) (+ (- i 1) j))
     = (m1 (m1 (step s) (- i 1)) j))
→
 (m1 (step s) (+ (- i 1) j))
    = (m1 (m1 (step s) (- i 1)) j)
```

Sequential Composition

Thm. `(m1 s (+ i j)) = (m1 (m1 s i) j)`

Proof. By induction on i.

Base: $\surd$

Induction Step:

```
( ¬(zp i)
  ∧
   (m1 (step s) (+ (- i 1) j))
    = (m1 (m1 (step s) (- i 1)) j))
→
 (m1 (step s) (+ (- i 1) j))
   = (m1 (m1 (step s) (- i 1)) j)√
```

Sequential Composition

Thm. `(m1 s (+ i j))` $=$ `(m1 (m1 s i) j)`

Proof. `By induction on i.`

`Base:` $\surd$

`Induction Step:` $\surd$

`Q.E.D.`

Note that we did not even need to expand step!

Demo

Moving Parts of M1 Proof Engine

- clock functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- how they mesh
- behavior v intention
- dealing with loops
- separation of concerns

How They Mesh

Thm. `(`ϕ` (m1 s (clk x))) ; `*`s must be ``semi-explicit''`*

By “semi-explicit” I mean that `(next-inst s)` must simplify to a constant under the available hypotheses.

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

(ϕ (m1 s (clk x)))

Induction Step:

(¬(zp x) ∧ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

(ϕ (m1 s (clk x)))

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

 (zp x)

$\rightarrow$

 (ϕ (m1 s (clk x))) ; $\{$*Clock!*$\}$

Induction Step:

 ($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

 (ϕ (m1 s (clk x)))

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

(ϕ (m1 s 3))

Induction Step:

($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

(ϕ (m1 s (clk x)))

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

(ϕ (m1 s 3)) ; {*Symb Eval!*}

Induction Step:

($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

(ϕ (m1 s (clk x)))

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

(ϕ s')

Induction Step:

($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

(ϕ (m1 s (clk x)))

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

 (zp x)

$\rightarrow$

 (ϕ s')

Induction Step:

 ($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

 (ϕ (m1 s (clk x))) ; {*Clock!*}

How They Mesh

Thm. `(`ϕ` (m1 s (clk x)))`

Base:

`(zp x)`

$\rightarrow$

`(`ϕ s'`)`

Induction Step:

`(`$\neg$`(zp x)` $\wedge$ `(`ϕ` (m1 `α` (clk (- x 1)))))`

$\rightarrow$

`(`ϕ` (m1 s (clk+ 11 (clk (- x 1)))))`

Note: We don't want `clk+` expressions rearranged!

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

(ϕ s')

Induction Step:

($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

(ϕ (m1 s (clk+ 11 (clk (- x 1)))));{*Seq Comp!*}

How They Mesh

Thm. $(\phi$ `(m1 s (clk x)))`

Base:

`(zp x)`

$\rightarrow$

$(\phi\ s')$

Induction Step:

$(\neg$`(zp x)` $\wedge$ $(\phi$ `(m1` α `(clk (- x 1)))))`

$\rightarrow$

$(\phi$ `(m1 (m1 s 11) (clk (- x 1))))`

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

 (zp x)

$\rightarrow$

 (ϕ s')

Induction Step:

 ($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

 (ϕ (m1 (m1 s 11) (clk (- x 1)))) ; {*Symb Eval!*}

How They Mesh

Thm. (ϕ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

(ϕ s')

Induction Step:

($\neg$(zp x) $\wedge$ (ϕ (m1 α (clk (- x 1)))))

$\rightarrow$

(ϕ (m1 s'' (clk (- x 1))))

How They Mesh

Thm. $(\phi$ (m1 s (clk x)))

Base:

(zp x)

$\rightarrow$

$(\phi\ s')$

Induction Step:

$(\neg$(zp x) $\wedge$ $(\phi$ (m1 s'' (clk (- x 1)))))

$\rightarrow$

$(\phi$ (m1 s'' (clk (- x 1)))) ; $\{$*Ind Hyp* α*!*$\}$

Moving Parts of M1 Proof Engine

- clock functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- how they mesh $\surd$
- behavior v intention
- dealing with loops
- separation of concerns

Behavior v Intention

Decompose code proofs into two big stages:

- behavior: the code π implements some algorithm
- intention: the algorithm satisfies (or is equal to) the specification

In the case of `*pi*`:

- *algorithm*: `(g x y 0)`, where

```
(defun g (x y a)
  (if (zp x)
      a
      (g (- x 1) y (+ y a))))
```

- *spec*: `(* x y)`

Behavior v Intention

Decompose code proofs into two big stages:

- behavior: the code π implements `(g x y 0)`. *This is the only part of the proof involving the semantics of the machine.*
- intention: `(g x y 0)` is `(* x y)`. *This is typically the harder of the two proofs because it lifts the algorithm to some more abstract spec, but it does not involve the semantics of the machine.*

This decomposition is often crucial to managing the complexity of a hard code proof project.

Furthermore, the algorithm often tells ACL2 what it needs to know to do the right induction.

Finish your dealings with the behavior (code v algorithm) before you turn to intention (algorithm v specs).

More precisely, do not let your work on ‘intention’ interfere with your work on ‘behavior.’

Moving Parts of M1 Proof Engine

- clock functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- how they mesh $\surd$
- behavior v intention $\surd$
- dealing with loops
- separation of concerns

Dealing with Loops

Consider the ‘behavior’ step – proving that the code implements the algorithm.

If the code loops (i.e., the algorithm is recursive) the proof will be inductive.

Two lessons are drawn from this:

- verify the loop code first – specify the loop as though it were a separate module/subroutine
- make sure your specification characterizes the whole state

Remember: you must be able to symbolically evaluate from where ever the loop (the inductive hypothesis) leaves you!

It is insufficient just to know the *output*.

To the extent that subsequent execution depends on them, you must specify the pc, the locals, the stack, and the program so you can continue executing from the state ‘produced’ by your inductive hypothesis. [Note: Our running example does not illustrate this lesson.]

Recall Program *pi*

```
(defconst *pi*
        '((ICONST 0)    ; 0
          (ISTORE 2)    ; 1   a = 0;
          (ILOAD 0)     ; 2   [loop:]
          (IFEQ 10)     ; 3   if x=0 then go to end;
          (ILOAD 0)     ; 4
          (ICONST 1)    ; 5
          (ISUB)        ; 6
          (ISTORE 0)    ; 7   x = x-1;
          (ILOAD 1)     ; 8
          (ILOAD 2)     ; 9
          (IADD)        ;10
          (ISTORE 2)    ;11   a = y+a;
          (GOTO -10)    ;12   go to loop
          (ILOAD 2)     ;13   [end:]
          (HALT)        ;14 ``return'' a
          ))
```

Entry-Level Code Correctness

```
(   $s_i$ = (make-state 0
                         (list x y)
                         nil
                         $\pi$)
  $\wedge$ (natp x) $\wedge$ (natp y)
  $\wedge$ $s_f$ = (m1 $s_i$ (clk x)))
$\rightarrow$
( (haltedp $s_f$)
  $\wedge$
   (top (stack $s_f$)) = (g x y 0))
```

Loop Correctness: Proof by induction on x.

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (loop-clk x))
 =
 (make-state
  14                     ; final HALTed pc
  (list 0 y (g x y a))   ; final locals
  (push (g x y a) nil)   ; final stack
  *pi*)                  ; final program
```

Base: `(zp x)`

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Base: (zp x)

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Base: `(zp x)`

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     3)
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Base: `(zp x)`

$((\texttt{natp x}) \wedge (\texttt{natp y}) \wedge (\texttt{natp a}))$

$\rightarrow$

```
 (m1 (make-state 2 (list x y a) nil *pi*)
     3)
```

$=$ {*by symb eval*}

```
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

✓

Step: $\neg$`(zp x)`

[Ind Hyp]:

```
((natp (- x 1)) ∧ (natp α) ∧ (natp β))
→
 (m1 (make-state 2 (list (- x 1) α     β     ) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 α (g (- x 1) α β))
  (push (g (- x 1) α β) nil)
  *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$`(zp x)`

[Ind Concl]:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (clk+ 11 (loop-clk (- x 1))))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list x y a) nil *pi*)
     (clk+ 11 (loop-clk (- x 1)))) {seq comp}
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$`(zp x)`

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (m1 (make-state 2 (list x y a) nil *pi*)
         11)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: ¬`(zp x)`

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (m1 (make-state 2 (list x y a) nil *pi*)
         11) {symb eval}
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list (- x 1) y (+ y a)) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: ¬`(zp x)`

[Ind Hyp]:

```
((natp (- x 1)) ∧ (natp α) ∧ (natp β))
→
 (m1 (make-state 2 (list (- x 1) α    β    ) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 α (g (- x 1) α β))
  (push (g (- x 1) α β) nil)
  *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list (- x 1) y (+ y a)) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$(zp x)

[Ind Hyp]:

((natp (- x 1)) $\wedge$ (natp α) $\wedge$ (natp β))

$\rightarrow$

```
 (m1 (make-state 2 (list (- x 1) α    β    ) nil *pi*)
     (loop-clk (- x 1)))
```

=

```
 (make-state
  14
  (list 0 α (g (- x 1) α β))
  (push (g (- x 1) α β) nil)
  *pi*)
```

Step: ¬`(zp x)`

[Ind Hyp]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp (+ y a)))
→
 (m1 (make-state 2 (list (- x 1) y (+ y a)) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g (- x 1) y (+ y a)))
  (push (g (- x 1) y (+ y a)) nil)
  *pi*)
```

Induct as suggested by `(g x y a)`!

Step: $\neg$`(zp x)`

[Ind Hyp]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp (+ y a)))
→
 (m1 (make-state 2 (list (- x 1) y (+ y a)) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g (- x 1) y (+ y a)))
  (push (g (- x 1) y (+ y a)) nil)
  *pi*)
```

Step: $\neg$`(zp x)`

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (m1 (make-state 2 (list (- x 1) y (+ y a)) nil *pi*)
     (loop-clk (- x 1)))
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *pi*)
```

Step: $\neg$`(zp x)`

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *pi*)  {ind hyp}
  =
  (make-state 14
              (list 0 y (g x y a))
              (push (g x y a) nil)
              *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *pi*)
 =
 (make-state 14
             (list 0 y (g x y a))
             (push (g x y a) nil)
             *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *pi*)
 =
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *pi*)
```

Step: $\neg$(zp x)

[Ind Concl]:

```
((natp (- x 1)) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *pi*)
 =
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *pi*)  √
```

Loop Correctness – :REWRITE Rule

$((\texttt{natp x}) \wedge (\texttt{natp y}) \wedge (\texttt{natp a}))$

$\rightarrow$

```
 (m1 (make-state 2 (list x y a) nil *pi*)
     (loop-clk x))
```

$=$

```
 (make-state
  14                  ; final HALTed pc
  (list 0 y (g x y a)) ; final locals
  (push (g x y a) nil) ; final stack
  *pi*)               ; final program
```

When done with a code block, disable its clock!

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (make-state 0 (list x y) nil *pi*)
     (clk x))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

$((\texttt{natp x}) \wedge (\texttt{natp y}))$

$\rightarrow$

```
 (m1 (make-state 0 (list x y) nil *pi*)
     (clk x))
```

$=$

```
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (make-state 0 (list x y) nil *pi*)
     (clk+ 2 (loop-clk x)))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (make-state 0 (list x y) nil *pi*)
     (clk+ 2 (loop-clk x)))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (m1 (make-state 0 (list x y) nil *pi*) 2)
     (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (m1 (make-state 0 (list x y) nil *pi*) 2)
         (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (make-state 2 (list x y 0) nil *pi*)
     (loop-clk x))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (m1 (make-state 2 (list x y 0) nil *pi*)
     (loop-clk x)) {Loop Correctness}
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Entry Correctness

```
((natp x) ∧ (natp y))
→
 (make-state 14 (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
 =
 (make-state 14 (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)  ✓
```

Entry Correctness – :REWRITE Rule

```
((natp x) ∧ (natp y))
→
 (m1 (make-state 0 (list x y) nil *pi*)
     (clk x))
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *pi*)
```

Note: Disable `clk`!

Demo

Moving Parts of M1 Proof Engine

- clock functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- how they mesh $\surd$
- behavior v intention $\surd$
- dealing with loops $\surd$
- separation of concerns

Separation of Concerns

Once we have dealt with behavior (code implements algorithm), we turn to intention (algorithm implements spec).

During the behavior proofs, you want ACL2's manipulation of the algorithm to mimic its manipulation of code.

But by proving lemmas to relate the algorithm to the spec, you may start changing how ACL2 manipulates the algorithm.

Changing ACL2's View of the Algorithm

Lemma: `(g x y a)`=`(+ a (* x y))`

Thm: `(g x y 0)`=`(* x y)`

Main Goal: Code Satisfies the Spec

(s_i = `(make-state 0`
`(list x y)`
`nil`
`*pi*)`

$\wedge$ `(natp x)` $\wedge$ `(natp y)`

$\wedge$ s_f = `(m1` s_i `(clk x)))`

$\rightarrow$

(`(haltedp` s_f`)`

$\wedge$

`(top (stack` s_f`))` = `(* x y)).`

Main Goal: Code Satisfies the Spec

```
(   si = (make-state 0
                     (list x y)
                     nil
                     *pi*)
  ∧ (natp x) ∧ (natp y)
  ∧ sf = (m1 si (clk x)))
→
( (haltedp sf)
  ∧
   (top (stack sf)) = (* x y)).
```

Main Goal: Code Satisfies the Spec

```
((natp x) ∧ (natp y)
 ∧ s_f = (m1 (make-state 0
                         (list x y)
                         nil
                         *pi*)
             (clk x)))
→
( (haltedp s_f)
 ∧
  (top (stack s_f)) = (* x y)).
```

Main Goal: Code Satisfies the Spec

```
 ((natp x) ∧ (natp y)
  ∧ $s_f$ = (m1 (make-state 0
                            (list x y)
                            nil
                            *pi*)
               (clk x))
→
 ( (haltedp $s_f$)
  ∧
   (top (stack $s_f$)) = (* x y)).
```

Main Goal: Code Satisfies the Spec

```
((natp x) ∧ (natp y)
 ∧ s_f = (m1 (make-state 0
                         (list x y)
                         nil
                         *pi*)
              (clk x)) {Entry Correctness}
→
( (haltedp s_f)
 ∧
  (top (stack s_f)) = (* x y)).
```

Main Goal: Code Satisfies the Spec

((natp x) $\wedge$ (natp y)

 $\wedge$ s_f $=$ (make-state

               14

               (list 0 y (g x y 0))

               (push (g x y 0) nil)

               pi)

$\rightarrow$

((haltedp s_f)

 $\wedge$

  (top (stack s_f)) $=$ (* x y)).

Main Goal: Code Satisfies the Spec

$$
\begin{array}{l}
\texttt{((natp x)} \wedge \texttt{(natp y)} \\
\quad \wedge\ s_f = \texttt{(make-state} \\
\qquad\qquad\qquad \texttt{14} \\
\qquad\qquad\qquad \texttt{(list 0 y (g x y 0))} \\
\qquad\qquad\qquad \texttt{(push (g x y 0) nil)} \\
\qquad\qquad\qquad \texttt{*pi*)} \\
\rightarrow \\
\texttt{((haltedp } s_f\texttt{)} \\
\quad \wedge \\
\quad \texttt{(top (stack } s_f\texttt{))} = \texttt{(* x y)).}
\end{array}
$$

Main Goal: Code Satisfies the Spec

```
((natp x) ∧ (natp y)
 ∧ s_f = (make-state
              14
              (list 0 y (* x y))
              (push (* x y) nil)
              *pi*)
→
( (haltedp s_f)
 ∧
  (top (stack s_f)) = (* x y)).
```

Main Goal: Code Satisfies the Spec

((natp x) $\wedge$ (natp y)

$\wedge$ s_f $=$ (make-state
14
(list 0 y (* x y))
(push (* x y) nil)
pi)

$\rightarrow$

((haltedp s_f)

$\wedge$

(top (stack s_f)) $=$ (* x y)).

Main Goal: Code Satisfies the Spec

((natp x) $\wedge$ (natp y)
 $\wedge$ s_f = (make-state
                14
                (list 0 y (* x y))
                (push (* x y) nil)
                pi)

$\rightarrow$

((haltedp s_f)
 $\wedge$
  (top (stack s_f)) = (* x y)).

Q.E.D.

Demo