# ACL2 Code Proofs

J Strother Moore
Fall, 2013

*Lecture 2*

# Today

I will continue to focus on total correctness, but shift from "constructor-style" to "updater-style" models and theorems.

We'll gradually descend to tools that can help in code proofs.

Next week I'll conclude with partial correctness and inductive invariants.

But the techniques being discussed are useful in both total and partial correctness proofs.

# An Unremarked Feature of M1

M1 has unbounded arithmetic – `IADD` is ACL2's "+", etc.

With an arithmetic library included, this makes simple M1 programs simple to verify.

*We need a book to support reasoning about modular arithmetic with the same ease.*

My code proofs involving bounded arithmetic invariably expand the modular operators to expose `floor`, `mod`, `18446744073709551616`, and many case splits.

I wish I could reason more algebraically, canonicalize arithmetic expressions and still get the benefits of linear arithmetic.

# Moving Parts of M1 Proof Engine

- clock functions:

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

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

- symbolic evaluation:

(m1 $s'$ 11) $=$ $s''$ semi-explicit states

- sequential composition:

$$\texttt{(m1 s (clk+ i j))} = \texttt{(m1 (m1 s i) j)}$$

- how they mesh:
  - induction drives the machinery
  - clocks expand
  - sequential composition demarks paths
  - symbolic execution runs paths
  - induction hypothesis applies

- behavior v intention
  - behavior (what the code *does*):

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

  - intention (what the author *wants*):

```
(* x y)
```

- dealing with loops:
  specify their effects *completely* and prove them first

- separation of concerns:
  don't mix behavior proofs with spec proofs

# Our Example Program

```
(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
```

# Constructor-Style States

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

# Constructor-Style States

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

But today we are interested in models in which the state is a single-threaded object modified via "updates."

# Demo

# Constructor-Style Loop Behavior

```
((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
```

# Updater-Style Loop Behavior

```
 ((good-statep s) ∧ (pc s) = 2)
→
 (m1 s (loop-clk (loi 0 s)))
 =
 (!pc 14
  (!loi 0 0
   (!loi 2 (g (loi 0 s) (loi 1 s) (loi 2 s))
    (!stack (push (g (loi 0 s) (loi 1 s) (loi 2 s))
                  (stack s))
            s))))
```

# Good-Statep

```
(defun good-statep (s)
       (declare (xargs :stobjs (s)))
       (and (sp s)
            (natp (rd :pc s))
            (natp-listp (rd :locals s))
            (<= 3 (len (rd :locals s)))
            (natp-listp (rd :stack s))
            (equal (rd :program s) *pi*)))
```

# Updater-Style Loop Behavior

```
 ((good-statep s) ∧ (pc s) = 2)
→
 (m1 s (loop-clk (loi 0 s)))
 =
 (!pc 14
  (!loi 0 0
   (!loi 2 (g (loi 0 s) (loi 1 s) (loi 2 s))
    (!stack (push (g (loi 0 s) (loi 1 s) (loi 2 s))
                  (stack s))
            s))))
```

How do we prove this? How did we do it for the constructor style?

# Constructor-Style Loop Induction

```
((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*)
```

# Constructor-Style Loop Induction

```
((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*)
```

`(g x y a)` suggests induction on `x` with:

$\{$`x` $\leftarrow$ `(- x 1)`, `y` $\leftarrow$ `y`, `a` $\leftarrow$ `(+ a y)`$\}$

# Constructor-Style Loop Induction Hyp

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

(g x y a) suggests induction on x with:

$\{x \leftarrow (- x 1), y \leftarrow y, a \leftarrow (+ a y)\}$

# Constructor-Style Loop Induction Hyp

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

This is the perfect call of M1: The completion of the computation after going around the loop once.

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
 (φ x y a
    (m1 (make-state ... x y a ...)
        (loop-clk x)))
```

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
(φ x y a
   (m1 (make-state ... x y a ...)
       (loop-clk x)))
```

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
(φ x y a
   (m1 (make-state ... x y a ...)
       (clk+ 11 (loop-clk (- x 1)))))
```

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
(φ x y a
   (m1 (make-state ... x y a ...)
       (clk+ 11 (loop-clk (- x 1)))))
```

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
(φ x y a
   (m1 (m1 (make-state ... x y a ...) 11)
       (loop-clk (- x 1))))
```

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
(φ x y a
   (m1 (m1 (make-state ... x y a ...) 11)
       (loop-clk (- x 1))))
```

# Induction Step

```
(¬(zp x)
 ∧
 (φ (- x 1) y (+ a y)
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1)))))
→
 (φ x y a
    (m1 (make-state ... (- x 1) y (+ a y) ...)
        (loop-clk (- x 1))))
```

# Updater-Style Loop Behavior

```
 ((good-statep s) ∧ (pc s) = 2)
→
 (m1 s (loop-clk (loi 0 s)))
 =
 (!pc 14
  (!loi 0 0
   (!loi 2 (g (loi 0 s) (loi 1 s) (loi 2 s))
    (!stack (push (g (loi 0 s) (loi 1 s) (loi 2 s))
                  (stack s))
            s))))
```

Induct with s ← (m1 s 11)

# Updater-Style Loop Behavior

```
 ((good-statep s) ∧ (pc s) = 2)
→
 (m1 s (loop-clk (loi 0 s)))
 =
 (!pc 14
  (!loi 0 0
   (!loi 2 (g (loi 0 s) (loi 1 s) (loi 2 s))
    (!stack (push (g (loi 0 s) (loi 1 s) (loi 2 s))
                  (stack s))
            s))))
```

Induct with `s` ← `(m1 s 11)` and make sure you can prove `(good-statep s)` implies `(good-statep (m1 s 11))`.

# Demo

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2))


   (equal
    (m1 s
        (loop-clk (loi 0 s)))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

## Target in Next Proof:

```
(M1 (WR :PC 2
        (WR :LOCALS (UPDATE-NTH 2 0 (RD :LOCALS S))
            S))
    (LOOP-CLK (NTH 0 (RD :LOCALS S))))
```

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2))

   (equal
    (m1 s
        (loop-clk (loi 0 s)))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

**Target in Next Proof**:

```
(M1 (WR :PC 2
        (WR :LOCALS (UPDATE-NTH 2 0 (RD :LOCALS S))
            S))
    (LOOP-CLK (NTH 0 (RD :LOCALS S)))))
```

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2))


   (equal
    (m1 s
        (loop-clk (nth 0 (rd :locals s))))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

**Target in Next Proof**:

```
(M1 (WR :PC 2
        (WR :LOCALS (UPDATE-NTH 2 0 (RD :LOCALS S))
            S))
    (LOOP-CLK (NTH 0 (RD :LOCALS S))))
```

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2))

   (equal
    (m1 s
        (loop-clk (nth 0 (rd :locals s))))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

**Target in Next Proof**:

```
(M1 (WR :PC 2
        (WR :LOCALS (UPDATE-NTH 2 0 (RD :LOCALS S))
            S))
    (LOOP-CLK (NTH 0 (RD :LOCALS S))))
```

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2))


   (equal
    (m1 s
        (loop-clk (loi 0 s)))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

## Target in Next Proof:

```
(M1 (WR :PC 2
        (WR :LOCALS (UPDATE-NTH 2 0 (RD :LOCALS S))
            S))
    (LOOP-CLK (NTH 0 (RD :LOCALS S)))))
```

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2)
                (equal x (loi 0 s)))
   (equal
    (m1 s
        (loop-clk x))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

## Target in Next Proof:

```
(M1 (WR :PC 2
        (WR :LOCALS (UPDATE-NTH 2 0 (RD :LOCALS S))
            S))
    (LOOP-CLK (NTH 0 (RD :LOCALS S)))))
```

# Why Loop Lemma is Unused

```
(defthm loop-correct
  (implies (and (good-statep s)
                (equal (rd :pc s) 2)
                (equal x (loi 0 s)))
   (equal
    (m1 s
        (loop-clk x))
    (!pc 14 (!loi 0 0 ...)))) ...)
```

Unfortunately, this lemma can't be proved by induction on `s` alone: you must induct on `s` and `x`.

Or you can prove the original version and then store this version as its `:rewrite :corollary`.

# Demo

# Inefficient Induction

```
(defun hint (s)
  (if (and (good-statep s)
           (equal (pc s) 2))
      (if (zp (loi 0 s))
          s
          (let ((s (m1 s 11)))
            (hint s)))
      s))
```

Is an "inefficient" induction hint if `s` occurs many times in $\phi$.

# Updater-Style Loop Behavior

```
 ((good-statep s) ∧ (pc s) = 2)
→
 (m1 s (loop-clk (loi 0 s)))
 =
 (!pc 14
  (!loi 0 0
   (!loi 2 (g (loi 0 s) (loi 1 s) (loi 2 s))
    (!stack (push (g (loi 0 s)
                     (loi 1 s)
                     (loi 2 s))
                  (stack s))
            s))))
```

Induct with s ← (m1 s 11)

# Updater-Style Loop Behavior

```
 ((good-statep s') ∧ (pc s') = 2)
→
 (m1 s' (loop-clk (loi 0 s')))
 =
 (!pc 14
  (!loi 0 0
   (!loi 2 (g (loi 0 s') (loi 1 s') (loi 2 s'))
    (!stack (push (g (loi 0 s')
                     (loi 1 s')
                     (loi 2 s'))
                  (stack s'))
            s'))))
```

Induct with s $\leftarrow s'$ where $s' =$ simp[(m1 s 11)]

# Demo

# New M1 Induction Principle

```
(defun iclk (s)
  (if (pre s)
      (if (test s)
          (κ s)
          (clk+ (η s) (iclk (m1 s (η s)))))
      0))

(defthm irule
   (implies (pre s)
            (post (m1 s (iclk s)))))
```

# Constraints

```
(natp (κ s))

(natp (η s))

(implies (pre s)
         (o-p (m s)))

(implies (and (pre s)
              (not (test s)))
         (o< (m (m1 s (η s)))
             (m s)))
```

```
(implies (and (pre s)
              (not (test s)))
         (pre (m1 s (η s))))

(implies (and (pre s)
              (test s))
         (post (m1 s (κ s))))
```

# Demo

# Further Improvements

For best results with special-purpose induction principles:

- define the notion of a syntactically well-formed program that will recognize many of the programs you wish to verify

- pre-prove that well-formed programs do not modify themselves, e.g., `(program (m1 s n))` is `(program s)` when that program is "well-formed"

- pre-prove that "`good-statep`" is invariant under the execution of "well-formed" programs, e.g., so `(good-statep (m1 s 11))` doesn't have to be proved each time

- that "`(m1 s 11)`" is equal to its semi-explicit value so it doesn't have to be computed each time

# General Advice on Inductive Code Proofs

I believe that developing special-purpose induction principles – and convenient macros for invoking them – for oft-used code idioms is worthwhile

# General Advice on Inductive Code Proofs

I believe that developing special-purpose induction principles – and convenient macros for invoking them – for oft-used code idioms is worthwhile – after you've gained enough experience with your machine to know what idioms to support and which principles work best!

# Aside

I will now briefly demonstrate some tools that I have developed.

Not all are ready for prime time:

- some are fragile
- to make them behave as demonstrated you must prove the 'right' lemmas
- they are not documented

They will not be released as part of this talk.

# Snorkeling

a simple trick to avoid stack overflows in long symbolic evaluation runs

$$\texttt{(m1 s 880)} = \underbrace{\texttt{(step (step ... (step s)...))}}_{880}$$

```
(m1 s 880)
=
(m1 s (clk+ 400 400 80)) =
=
(m1 (m1 (m1 s 400) 400) 80)
```

# Demo

# Lift-Subterm-from-Clause

an ACL2 book that allows you to completely simplify a subterm of a goal before working on the rest of the goal, e.g., as we might want to to with `(m1 s 11)` in the induction hypothesis

# Demo

# Terminatricks

an ACL2 book that guesses measures to justify recursive definitions

# Demo

# Simplify-under-hyps

an ACL2 book that simplifies a term under a hypothesis and returns an equal term, e.g., to simplify `(m1 s 11)` under `(good-statep s)` and `(equal (pc s) 2)`.

# Demo

# Codewalker

an ACL2 book that derives clock and semantic functions from code, given only the operational semantics of the machine

# Demo

# Projector

an ACL2 book (part of Codewalker) that 'projects' out the effects of some code on a given machine resource, often helpful in figuring out what a piece of code does

# Demo

# Next Week

Inductive Invariants and Partial Correctness