An ACL2 Proof of the Turing Equivalence of M1

J Strother Moore

Department of Computer Science
University of Texas at Austin

March, 2012

What is Turing Equivalence?

To say that a computational device is Turing equivalent means that it can do anything a Turing machine can.

You typically prove Turing equivalence by showing how to implement a Turing machine interpreter on the device in question. Then you can use the device to simulate any Turing machine.

Why Prove M1 Turing Equivalent

Because in class a few weeks ago I said "M1 can probably do anything a Turing machine can."

But M1 was designed to demonstrate that we can prove things about computational models.

I should have said "I've proved that M1 can do anyting a Turing machine can."

Outline

- M1
- Turing Machines
- formalized Turing equivalence
- implementation issues
- proof issues

An M1 Program to Compute "$v[0] < v[1]$"

```
'((ILOAD 1)   ;  0  loop:
  (IFEQ 12)   ;  1     if v[1]=0, goto false
  (ILOAD 0)   ;  2
  (IFEQ 12)   ;  3     if v[0]=0, goto true
  (ILOAD 0)   ;  4
  (ICONST 1)  ;  5
  (ISUB)      ;  6
  (ISTORE 0)  ;  7     v[0] := v[0] - 1;
  (ILOAD 1)   ;  8
  (ICONST 1)  ;  9
  (ISUB)      ; 10
  (ISTORE 1)  ; 11     v[1] := v[1] - 1;
  (GOTO -12)  ; 12     goto loop
  (ICONST 0)  ; 13  false:
  (HALT)      ; 14     halt with 0 on stack
  (ICONST 1)  ; 15  true:
  (HALT)      ; 16     halt with 1 on stack
```

Schedules versus Clocks

For purposes of this talk,

```
(defun M1 (s m)
  (run (repeat 'TICK m) s))
```

i.e.,

```
(defun M1 (s m)
  (if (zp m)
      s
      (M1 (step s) (- m 1))))
```

This talk conflates “schedule functions” (lists) with “clock functions” (numbers), because the proof was actually done with schedules but the top-level results are “advertised” with clocks to keep the presentation simpler. Here a “clock” is just the length of the corresponding schedule.

Instead of referring to M1 “local variables” I call them “registers,” but they’re just what the JVM calls locals variables.

Typical M1 Theorem

$\forall$ i,j : $\exists$ k :

```
  (natp i) ∧ (natp j)
```

$\rightarrow$

```
  (M1 (make-state 0
                  (list i j)
                  nil
                  '((ILOAD 1) ... (HALT)))
```

k)

=

```
  (make-state (if (< i j) 16 14)
              (list (- i (min i j)) (- j (min i j)))
              (push (if (< i j) 1 0) nil)
              '((ILOAD 1) ... (HALT)))
```

Typical M1 Theorem

```
∀ i,j :
  (natp i) ∧ (natp j)
  →
  (M1 (make-state 0
                  (list i j)
                  nil
                  '((ILOAD 1) ... (HALT)))
      (lessp-clock i j))
  =
  (make-state (if (< i j) 16 14)
              (list (- i (min i j)) (- j (min i j)))
              (push (if (< i j) 1 0) nil)
              '((ILOAD 1) ... (HALT)))
```

Turing Machines

```
Turing Machine*     st  tape
((Q0 1 0 Q1)       Q0: ([ 1 ] 1 1 1 1)
 (Q1 0 R Q2)
 (Q2 1 0 Q3)
 (Q3 0 R Q4)
 (Q4 1 R Q4)
 (Q4 0 R Q5)
 (Q5 1 R Q5)
 (Q5 0 1 Q6)
 (Q6 1 R Q6)
 (Q6 0 1 Q7)
 (Q7 1 L Q7)
 (Q7 0 L Q8)
 (Q8 1 L Q1)
 (Q1 1 L Q1))
```

*from *A Theory of recursive functions and effective computability*, Hartley Rogers, McGraw-Hill, 1967

Turing Machines

```
Turing Machine*     st  tape
((Q0 1 0 Q1)       Q0: ([ 1 ] 1 1 1 1)
 (Q1 0 R Q2)       Q1: ([ 0 ] 1 1 1 1)
 (Q2 1 0 Q3)
 (Q3 0 R Q4)
 (Q4 1 R Q4)
 (Q4 0 R Q5)
 (Q5 1 R Q5)
 (Q5 0 1 Q6)
 (Q6 1 R Q6)
 (Q6 0 1 Q7)
 (Q7 1 L Q7)
 (Q7 0 L Q8)
 (Q8 1 L Q1)
 (Q1 1 L Q1))
```

*from *A Theory of recursive functions and effective computability*, Hartley Rogers, McGraw-Hill, 1967

Turing Machines

```
Turing Machine*          st  tape
((Q0 1 0 Q1)          Q0: ([ 1 ] 1 1 1 1)
 (Q1 0 R Q2)          Q1: ([ 0 ] 1 1 1 1)
 (Q2 1 0 Q3)          Q2: (0 [ 1 ] 1 1 1)
 (Q3 0 R Q4)          Q3: (0 [ 0 ] 1 1 1)
 (Q4 1 R Q4)          Q4: (0 0 [ 1 ] 1 1)
 (Q4 0 R Q5)          Q4: (0 0 1 [ 1 ] 1)
 (Q5 1 R Q5)          Q4: (0 0 1 1 [ 1 ])
 (Q5 0 1 Q6)          Q4: (0 0 1 1 1 [ 0 ])
 (Q6 1 R Q6)          Q5: (0 0 1 1 1 0 [ 0 ])
 (Q6 0 1 Q7)          Q6: (0 0 1 1 1 0 [ 1 ])
 (Q7 1 L Q7)          Q6: (0 0 1 1 1 0 1 [ 0 ])
 (Q7 0 L Q8)          ...
 (Q8 1 L Q1)          Q7: (0 0 0 0 0 0 1 1 [ 1 ] 1 1 1 1 1)
 (Q1 1 L Q1))         Q7: (0 0 0 0 0 0 1 [ 1 ] 1 1 1 1 1 1)
                      Q7: (0 0 0 0 0 0 [ 1 ] 1 1 1 1 1 1 1)
                      Q7: (0 0 0 0 0 [ 0 ] 1 1 1 1 1 1 1 1)
                      Q8: (0 0 0 0 [ 0 ] 0 1 1 1 1 1 1 1 1)
```

*from *A Theory of recursive functions and effective computability*, Hartley Rogers, McGraw-Hill, 1967

Note

Rogers shows that it is sufficient to consider only initial (and final) tapes with a finite number of 1s on them.

So, like him, we represent a tape as two lists of 0s and 1s, representing the left and right halves of the tape, with the "read/write head" on the first symbol of the right half tape.

```
(show-tape (cons '(0 0 1 1)  '(1 1 1 0 0 0)))
=
(1 1 0 0 [ 1 ] 1 1 0 0 0)
```

ACL2 Formalization of Turing Machines

```
(defun tmi (st tape tm n)
  (declare (xargs :measure (nfix n)))
  (cond ((zp n) nil)
        ((instr st (current-sym tape) tm)
         (tmi (nth 3 (instr st (current-sym tape) tm))
              (new-tape (nth 2 (instr st (current-sym tape) tm))
                        tape)
              tm
              (- n 1)))
        (t tape)))

(show-tape *example-tape*)
([ 1 ] 1 1 1 1)

(show-tape (tmi 'Q0 *example-tape* *rogers-tm* 78))
(0 0 0 0 [ 0 ] 0 1 1 1 1 1 1 1 1)
```

ACL2 Formalization of Turing Machines

```
(defun tmi (st tape tm n)
  (declare (xargs :measure (nfix n)))
  (cond ((zp n) nil)
        ((instr st (current-sym tape) tm)
         (tmi (nth 3 (instr st (current-sym tape) tm))
              (new-tape (nth 2 (instr st (current-sym tape) tm))
                        tape)
              tm
              (- n 1)))
        (t tape)))
```

But n is just an artifact of ACL2's requirement that all functions be terminating.

"Turing machine `tm` (starting in `st`) on `tape` *halts*"

means

$\exists$ `n : (tmi st tape tm n)` $\neq$ `nil`

"Turing machine `tm` (starting in `st`) on `tape` *runs forever*"

means

$\forall$ `n : (tmi st tape tm n)` $=$ `nil`

The Questions

Can M1 compute anything a Turing machine can compute?

Can we implement a Turing machine interpreter on M1?

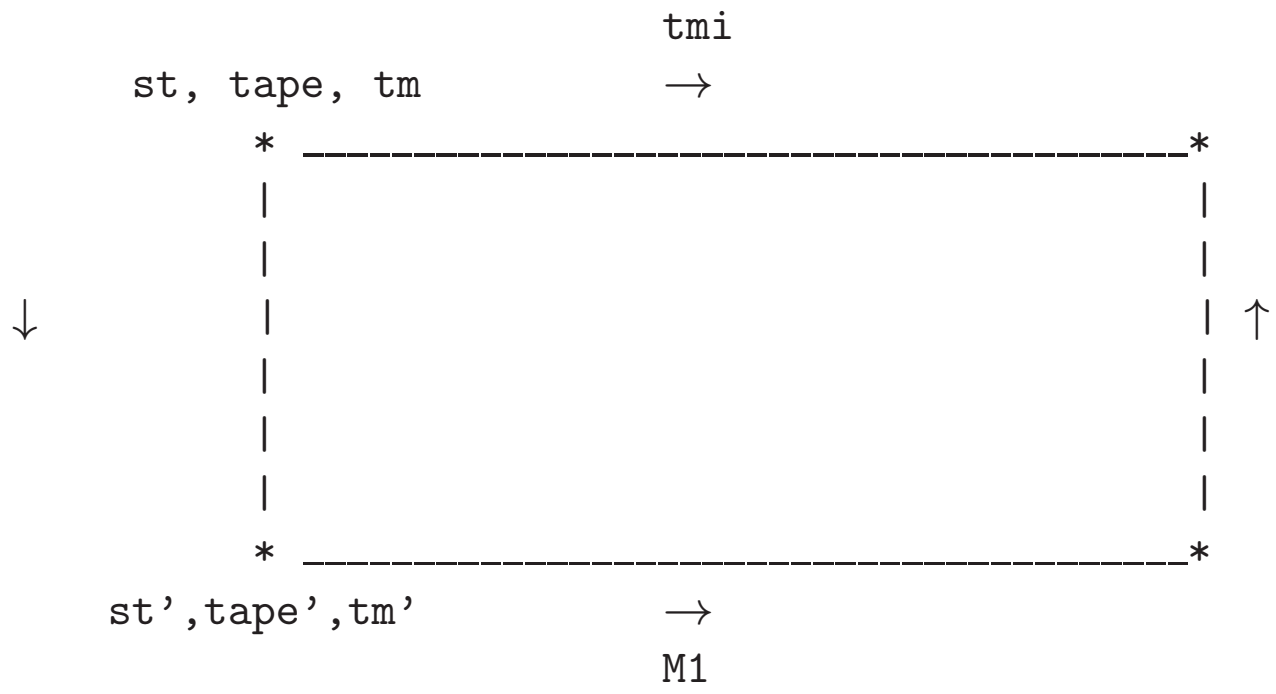
What do we have to prove about it?

Turing Equivalence

We can set up a correspondence between Turing machine resources (descriptions and tapes) and M1 resources (local variables) so that anything `tmi` can do, M1 can do.

(a) if the Turing machine runs forever, then M1 runs forever, and

(b) if the Turing machine halts then M1 halts with the "same" tape.


tmi
st, tape, tm
st',tape',tm'
M1

Turing Equivalence

We can set up a correspondence between Turing machine resources (descriptions and tapes) and M1 resources (local variables) so that anything `tmi` can do, M1 can do.

(a) if the Turing machine runs forever, then M1 runs forever, and

(b) if the Turing machine halts then M1 halts with the "same" tape.

Turing Equivalence

We can set up a correspondence between Turing machine resources (descriptions and tapes) and M1 resources (local variables) so that anything `tmi` can do, M1 can do.

(a) if the Turing machine runs forever, then M1 runs forever

(a') if M1 halts in m steps, then the Turing machine halts in some number J steps

(b) if the Turing machine halts then M1 halts with the "same" tape

Turing Equivalence

We can set up a correspondence between Turing machine resources (descriptions and tapes) and M1 resources (local variables) so that anything `tmi` can do, M1 can do.

(a') if M1 halts in m steps, then the Turing machine halts in some number J steps

(b) if the Turing machine halts then M1 halts with the "same" tape

Turing Equivalence

We can set up a correspondence between Turing machine resources (descriptions and tapes) and M1 resources (local variables) so that anything `tmi` can do, M1 can do.

(a') if M1 halts in m steps, then the Turing machine halts in some number J steps

(b) if the Turing machine halts then M1 halts with the "same" tape

(b') if the Turing machine halts in n steps then M1 halts with the "same" tape in some number K steps

Turing Equivalence

We can set up a correspondence between Turing machine resources (descriptions and tapes) and M1 resources (local variables) so that anything `tmi` can do, M1 can do.

(a') if M1 halts in m steps, then the Turing machine halts in some number J steps

(b') if the Turing machine halts in n steps then M1 halts with the "same" tape in some number K steps

Theorem A

(a') if M1 halts in m steps, then the Turing machine halts in some number J steps

```
(implies
 (and (symbolp st)
      (tapep tape)
      (turing-machinep tm)
      (natp m)
      (m1-haltedp (m1 (down st tape tm) m)))
 (tm-haltedp (tmi st tape tm (J st tape tm m))))
```

Theorem B

(b') if the Turing machine halts in n steps then M1 halts with the "same" tape in some number K steps

```
(implies (and (symbolp st)
              (tapep tape)
              (turing-machinep tm)
              (natp n)
              (tm-haltedp (tmi st tape tm n)))
         (let ((s (m1 (down st tape tm)
                      (K st tape tm n))))
           (and (m1-haltedp s)
                (equal (up s)
                       (tmi st tape tm n)))))
```

Evil Ideas

What do you have to trust to believe that M1 can do anything that a Turing machine can do?

We'd like to believe that the definitions of `down`, `up`, `J` and `K` don't matter as long as Theorems A and B hold.

Is that accurate? No. You could define `down`, mathematically, to be "If `tm` halts on `tape`, compute the right tape using any computational device and store it somewhere in the M1 state and have the program return that tape. Else (if `tm` does not halt on `tape`), generate a program that loops forever."

We have to look at `down`, etc., to make sure M1 does the work.

Outline

- M1
- Turing Machines
- formalized Turing equivalence
- implementation issues
- proof issues

Implementation Issues

`TMI` operates on symbolic data and lists:

- a state name `st` (an ACL2 symbol),
- a `tape` (a cons of two half-tapes: lists of 1s and 0s)
- a Turing machine description (a list of 4-tuples $(q_{in}\ bit\ op\ q_{out})$)

M1 only operates on numbers.

Turing Machines

```
st tape                 tm
Q0 ([ 1 ] 1 1 1 1)      ((Q0 1 0 Q1)
                         (Q1 0 R Q2)
                         (Q2 1 0 Q3)
                         (Q3 0 R Q4)
                         (Q4 1 R Q4)
                         (Q4 0 R Q5)
                         (Q5 1 R Q5)
                         (Q5 0 1 Q6)
                         (Q6 1 R Q6)
                         (Q6 0 1 Q7)
                         (Q7 1 L Q7)
                         (Q7 0 L Q8)
                         (Q8 1 L Q1)
                         (Q1 1 L Q1))
```

Turing Machines

```
st tape                tm
 0 ([ 1 ] 1 1 1 1)     ((0 1 0 1)
                        (1 0 3 2)
                        (2 1 0 3)
                        (3 0 3 4)
                        (4 1 3 4)
                        (4 0 3 5)
                        (5 1 3 5)
                        (5 0 1 6)
                        (6 1 3 6)
                        (6 0 1 7)
                        (7 1 2 7)
                        (7 0 2 8)
                        (8 1 2 1)
                        (1 1 2 1))
```

Turing Machines

```
st tape                tm
 0 ([ 1 ] 1 1 1 1)     ((0 1 0 1) ⇒ 0001 000 1 0000
                        (1 0 3 2) ⇒ 0010 011 0 0001
                        (2 1 0 3) ⇒ 0011 000 1 0010
                        (3 0 3 4) ...
                        (4 1 3 4)
                        (4 0 3 5)
                        (5 1 3 5)
                        (5 0 1 6)
                        (6 1 3 6)
                        (6 0 1 7)
                        (7 1 2 7)
                        (7 0 2 8)
                        (8 1 2 1)
                        (1 1 2 1))
```

Turing Machines

```
st tape                tm
 0 ([ 1 ] 1 1 1 1)     ((0 1 0 1) ⇒ 0001 000 1 0000
                        (1 0 3 2) ⇒ 0010 011 0 0001
                        (2 1 0 3) ⇒ 0011 000 1 0010
                        (3 0 3 4) ...
                        (4 1 3 4)
                        (4 0 3 5)
                        (5 1 3 5)
                        (5 0 1 6)
                        (6 1 3 6)
                        (6 0 1 7)
                        (7 1 2 7)
                        (7 0 2 8)
                        (8 1 2 1)
                        (1 1 2 1))

100000000001... 010001100011 001100010010 001001100001 000100010000
```

Turing Machines

```
st tape                tm
 0 ([ 1 ] 1 1 1 1)     ((0 1 0 1) ⇒ 0001 000 1 0000
                        (1 0 3 2) ⇒ 0010 011 0 0001
                        (2 1 0 3) ⇒ 0011 000 1 0010
                        (3 0 3 4) ...
                        (4 1 3 4)
                        (4 0 3 5)
                        (5 1 3 5)
                        (5 0 1 6)
                        (6 1 3 6)
                        (6 0 1 7)
                        (7 1 2 7)
                        (7 0 2 8)
                        (8 1 2 1)
                        (1 1 2 1))
```

```
100000000001... 010001100011 001100010010 001001100001 000100010000
=
47921276213291088842438974929274042051802613691060496
```

Turing Machine Descriptions

Step 1. Convert the symbols (state names and operations) in Turing machine descriptions into numbers

Step 2. Convert each 4-tuple of numbers into a "cell"

Step 3. Pack the cells into a big number

```
(defun make-cell (tuple w)
  (let ((st-in  (nth 0 tuple))
        (sym    (nth 1 tuple))
        (op     (nth 2 tuple))
        (st-out (nth 3 tuple)))
    (+ (* (expt 2 (+ 3 1 w)) st-out)
       (* (expt 2 (+ 1 w)) op)
       (* (expt 2 w) sym)
       st-in)))

(defun ncons (cell tail w)
  (+ cell
     (* (expt 2 (+ 4 (* 2 w)))
        tail)))
```

```
(defun ncode (tm w)
  (cond
   ((endp tm) (nnil w))
   (t (ncons (make-cell (car tm) w)
             (ncode (cdr tm) w)
             w))))

(defun down-tm (st tm)
  (let*
   ((map (renaming-map st tm))
    (renamed-tm (tm-to-tm1 tm map))
    (w (max-state-width renamed-tm)))
   (mv (cdr (assoc st map))
       (ncode renamed-tm w)
       w)))
```

```
*rogers-tm*
 = ((Q0 1 0 Q1) (Q1 0 R Q2) ... (Q8 1 L Q1) (Q1 1 L Q1))

(down-tm 'Q0 *rogers-tm*)
 = (mv 0 47921276213291088842438974929274042051802613691060496 4)
```

Why Numbers instead of Code?

Why not code the Turing machine description as part of the M1 program instead of part of the data?

Answer: Wait until we discuss program proofs.

Down and Up

```
(defun down (st tape tm)
  (mv-let (st! tm! w!)
          (down-tm st tm)
          (mv-let (tape! pos!)
                  (down-tape tape)
                  (make-state 0
                              (list st! tape! pos! tm! w! (nnil w!)
                                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
                              nil
                              *pi*))))

(defun up (s)
  (up-tape (nth 1 (locals s))
           (nth 2 (locals s))))
```

M1 Programming Issues

We must write M1 programs to compute certain arithmetic operations to manipulate Turing machine descriptions and tapes

`FLOOR2`	`(floor x 2)` – count subtractions of 2 until `x`=0 or `x`=1
`MOD2`	`(mod x 2)` – subtract 2 until 0 or 1
`EXPT2`	`(expt 2 x)` – multiply by 2 `x` times
`LOG2`	`(log2 x)` – count `floor2`
`LESSP`	`(< x y)` – subtract 1 from each until `x`=0 or `y`=0
`MOD`	`(mod x y)` – subtract `y` from `x` until `x`$<$ `y`
`RSH`	`(ash x (- y))` – `floor2` `x`, `y` times

`NCAR`	"`(car x)`" – `(mod tm (expt 2 (+ 4 (* 2 w))))`
`NCDR`	"`(cdr x)`" – `(rsh tm (+ 4 (* 2 w)))`
`CSYM`	read current symbol on tape
`NINSTR`	search tm for st and sym
`NST-IN`	$(q_{in}\ bit\ op\ q_{out}) \Rightarrow q_{in}$
`NSYM`	$(q_{in}\ bit\ op\ q_{out}) \Rightarrow bit$
`NOP`	$(q_{in}\ bit\ op\ q_{out}) \Rightarrow op$
`NST-OUT`	$(q_{in}\ bit\ op\ q_{out}) \Rightarrow q_{out}$
`NEW-TAPE2`	write/move tape
`TMI`	Turing machine interpreter

Sample Assembly Code – MOD

```
mod  (ILOAD mod.x)
     (ISTORE lessp.x)        ; lessp.x := mod.x (pass args)
     (ILOAD mod.y)
     (ISTORE lessp.y)        ; lessp.y := mod.y
     (goto@ lessp mod-next)  ; (< mod.x mod.y)
mod-next
     (ICONST 0)              ;                  (clear regs used)
     (ISTORE exit-pc)
     (ILOAD lessp.a)
     (ICONST 0)
     (ISTORE lessp.a)
     (IFEQ mod-continue)     ; if lessp.a=0, goto continue
     (ILOAD mod.x)
     (ISTORE mod.a)          ; mod.a := mod.x  (store ans)
     (ICONST 0)              ;                 (clear regs used)
     (ISTORE mod.x)
     (ICONST 0)
     (ISTORE mod.y)
     (exit mod)              ; return ans
```

```
mod-continue
      (ILOAD mod.x)
      (ILOAD mod.y)
      (ISUB)
      (ISTORE mod.x)          ; mod.x := mod.x-mod.y
      (GOTO mod)              ; goto mod
```

Sample Assembly Code – MOD

```
      ...
      (ISTORE lessp.y)
      (goto@ lessp mod-next)  ⇒  (ICONST mod-next)
mod-next                         (GOTO lessp)
      (ICONST 0)
      (ISTORE exit-pc)
      ...l
      (exit mod)                 ⇒  (ISTORE exit-pc)
      ...                           (ILOAD exit-pc)
                                    (ICONST < caller_1 >)
                                    (ISUB)
                                    (IFEQ < caller_1 >)
                                    ...
                                    (ILOAD exit-pc)
                                    (ICONST < caller_2 >)
                                    ...
```

Sample Assembly Code – MOD

```
      ...
      (ISTORE lessp.y)
      (goto@ lessp mod-next)  ⇒  (ICONST 569)
mod-next                         (GOTO -203)
      (ICONST 0)
      (ISTORE exit-pc)
      ...l
      (exit mod)              ⇒  (ISTORE 11)
      ...                        (ILOAD 11)
                                 (ICONST 473)
                                 (ISUB)
                                 (IFEQ -722)
                                 ...
                                 (ILOAD 11)
                                 (ICONST 124)
                                 ...
```

Summary of Code

The M1 Turing machine interpreter uses 41 variables, 17 subroutines, and 804 instructions.

Note: The M1 implementation of `tmi` may loop forever (it does not have the step count `n`); it loops until the Turing machine description tells it to halt.

Why Numbers instead of Code?

Idea: Code `tm` as an M1 program.

Answer: That would make the program a function of `tm`.

All relative jump numbers (generated by the `exits` from kernel code like `expt2` and `mod2`) would change.

The kernel would not be constant.

We would have to verify the "kernel generator" instead of a constant `*pi*`.

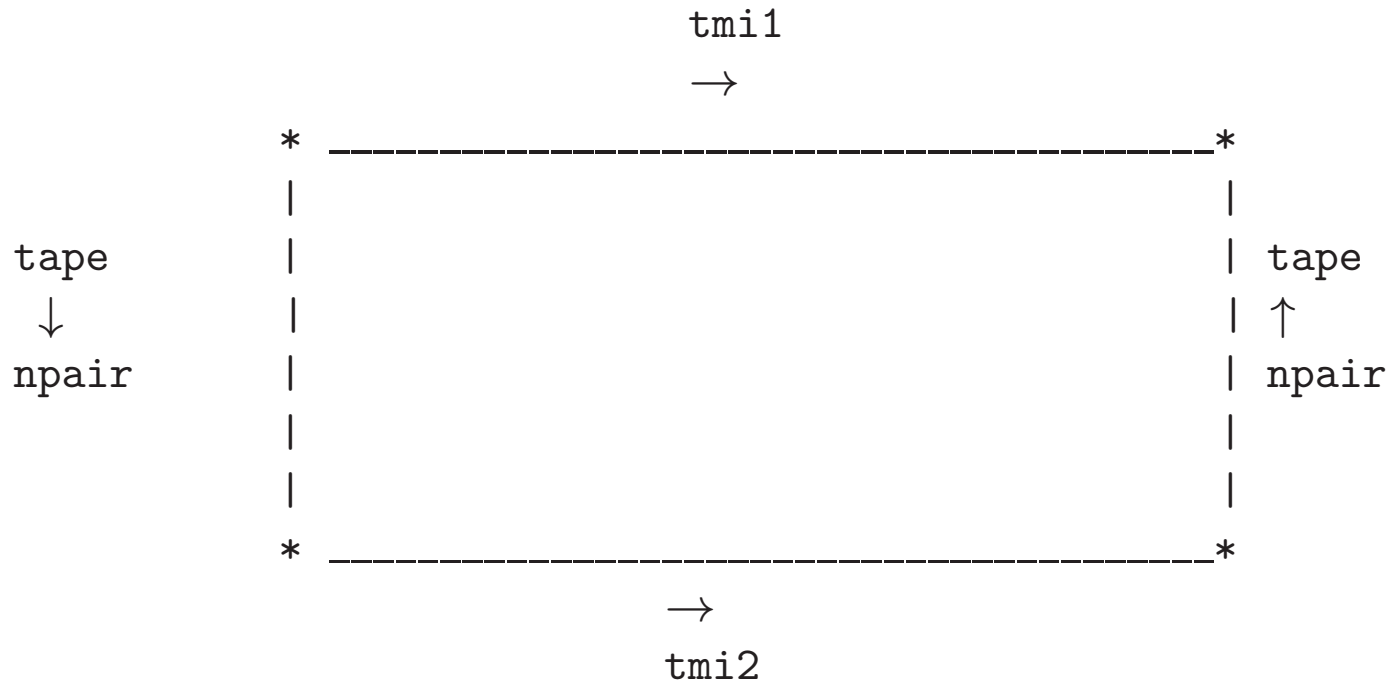
By keeping the program constant we simplified its verification.

Proof Issues

Step 1: Reduce by proof `tmi` to `tmi3`, an ACL2 Turing machine interpreter that operates on numeric representations of tapes and machine descriptions.

- `tmi` official definition of Turing machine
- `tmi1` like `tmi` but state names and `L` and `R` replaced by numbers
- `tmi2` like `tmi1` but tape replaced by an "npair" consisting of a binary number and a position
- `tmi3` like `tmi2` but machine description (list of 4-tuples) replaced by "cells" packed into a binary number

For each adjacent pair we prove a commuting diagram, e.g.,



L1: $(\texttt{tmi2}\ (\downarrow\ \texttt{tape})) = (\downarrow\ (\texttt{tmi1}\ \texttt{tape}))$

L2: $(\uparrow\ (\downarrow\ \texttt{tape})) = \texttt{tape}$

Thm: $(\uparrow\ (\texttt{tmi2}\ (\downarrow\ \texttt{tape}))) = (\texttt{tmi1}\ \texttt{tape})$

```
(defthm tmi3!-is-tmi2-transformed
  (implies (and (natp w)
                (natp st)
                (< st (expt 2 w))
                (tapep tape)
                (natp tm))
           (equal (tmi3! st (convert-tape-to-npair tape)
                  (convert-tape-to-npair (tmi2 st tape t

(defthm tape-conversion-theorem
  (implies (tapep tape)
           (equal (convert-npair-to-tape
                   (convert-tape-to-npair tape))
                  tape)))
```

```
(defthm tmi3!-is-tmi
  (implies
    (and (symbolp st)
         (tapep tape)
         (turing-machinep tm))
    (equal (convert-npair-to-tape
            (tmi3! (cdr (assoc st (renaming-map st tm)))
                   (convert-tape-to-npair tape)
                   (ncode (tm-to-tm1 tm (renaming-map st tm))
                          (max-state-width (tm-to-tm1 tm (renaming-map st tm))))
                   (max-state-width (tm-to-tm1 tm (renaming-map st tm)))
                   n))
           (tmi st tape tm n))))
```

Step 2: For each M1 subroutine in `*pi*`, define a clock function and prove that the code implements the corresponding ACL2 function from the `tmi3` model.

```
(defthm mod2-thm
  (implies
   (and (natp x)
        (member exit (cdr (assoc 'mod2 *switch-table*)))
        (equal exit! (cdr (assoc exit *label-table*))))
   (equal (run (mod2-sched exit x)
               (make-state *mod2*
                           (std-locals :mod2.x x)
                           (push exit! stack)
                           *pi*))
          (make-state exit!
                      (std-locals :exit-pc exit!
                                  :mod2.x 0
                                  :mod2.a (mod x 2))
                      stack *pi*))))
```

Step 3: The last of these code proof theorems is the key one: *TMI Simulation Theorem*. A `tmi3` computation of length `n` produces the "same" answer as an M1 computation on `*pi*` of length `(K st tape tm n)`. By "same" we mean, if either is halted the other is, and if halted, the tapes are the same modulo representation.

Proof Issues, Continued

Step 4: Theorem B is now easy: If `tmi` halts in `n` steps, then M1 halts with same tape in `(K st tape tm n)` steps.

Theorem B

```
(implies (and (symbolp st)
              (tapep tape)
              (turing-machinep tm)
              (tm-haltedp (tmi st tape tm n)))
         (let ((s (m1 (down st tape tm)
                      (K st tape tm n))))
           (and (m1-haltedp s)
                (equal (up s)
                       (tmi st tape tm n)))))
```

Proof Issues, Continued

The TMI Simulation Theorem gives us the equivalence of an `n` step `tmi` computation with a `(K st tape tm n)` step M1 computation.

But Theorem A requires: if M1 halts in `m` steps then `tmi` halts in some number `(J st tape tm m)` steps.

How do we define function `J`?

Step 4: Prove `K` monotonic: if `tmi` hasn't halted after `n` steps, then `(K st tape tm n)` $<$ `(K st tape tm (+ 1 n))`

Step 5: Define `(J st tape tm m` i`)` to *search* upwards for an i such that either `tmi` halts at i or else `m` $<$ `(K st tape tm i)`. This definition is admissible because `K` is monotonic.

Step 6: Observe that `(J st tape tm m 0)` is a j such that either (i) `tmi` halts at j or (ii) `m` $<$ `(K st tape tm` j`)`

Step 7: Theorem A is now easy. Assume M1 is halted after `m` steps. Either (i) or (ii). If (i), we're done: `tmi` halts at j.

Otherwise `tmi` is still running at j.

By the TMI Simulation Theorem, M1 is still running at `(K st tape tm` j`)`.

But `m` $<$ `(K st tape tm` j`)`!

Thus M1 cannot be halted at `m`, because M1 cannot "restart." Contradiction.

Proof Statistics

The proof involves 119 definitions and 160 theorems.

The proof takes about 6 minutes on my laptop.

I spent about 10 days working on it.

Using M1 to Emulate Turing Machines

How practical is it to run the M1 implementation of Turing machines?

Given our constructive clocks, we can determine, for any Turing machine run, how many M1 instructions it takes.

Consider `*rogers-tm*` on the tape `([ 1 ] 1 1 1 1)`, which takes 78 steps to compute the tape

`(0 0 0 0 [ 0 ] 0 1 1 1 1 1 1 1 1)`

It takes M1 `(K 'Q0 *example-tape* *rogers-tm* 78)` steps.

```
(m 'Q0 *example-tape* *rogers-tm* 78)
```

$=$

103,979,643,405,139,456,340,754,264,791,057,682,257,947,240,629,585,359,596

$\approx 10^{56}$ steps!

```
(K 'Q0 *example-tape* *rogers-tm* 78)
```

=

103,979,643,405,139,456,340,754,264,791,057,682,257,947,240,629,585,359,596

$\approx 10^{56}$ steps!

This is because M1 is using repeated subtractions of 1 and 2 to recover bits from 50 digit numbers.

It would be much faster if M1 had built in binary arithmetic (less than, `rsh`, `mod`)

It would be a little faster still if it had `JSR`.

Summary

This is only the second mechanically checked Turing equivalence proof I know.

This is the first one for a von Neuman machine model.

It's a great little project requiring some coding skills and layered abstractions.

The 804 instruction M1 program is the largest M1 program I've ever verified.

This project also clearly demonstrates that we can reason about computations that are impractical to carry out!