Proving a Simple von Neumann Machine Turing Equivalent

J Strother Moore

Department of Computer Science
University of Texas at Austin

presented at
University of Edinburgh
August, 2012

Prologue

I teach a course on how to build formal models of computing machines.

We construct a formal model of the Java Virtual Machine (JVM), starting from a very simple stack machine, M1, *designed to show that properties of machines and their programs can be proved.*

If the purpose of M1 is to show the students we can prove properties of machines and programs, I shouldn't just say M1 is Turing equivalent. I should prove it.

Typical M1 Programming Challenge

Write a program that takes two natural numbers in local variables $v[0]$ and $v[1]$ and halts with 1 on the stack if $v[0] < v[1]$ and 0 on the stack otherwise.

Difficulty: The only test in the M1 language is “jump if top-of-stack equals 0”!

Solution: Count both variables down by 1 and stop when one or the other is 0.

Java Bytecode Solution

```
ILOAD_1    //  0
IFEQ 12    //  1     if v[1]=0, jump to 13
ILOAD_0    //  2
IFEQ 12    //  3     if v[0]=0, jump to 15
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     jump to 0
ICONST_0   // 13
IRETURN    // 14     halt with 0 on stack
ICONST_1   // 15
IRETURN    // 16     halt with 1 on stack
```

JVM pcs are byte addresses but instruction counts are shown here

An M1 Programming Solution

```
'((ILOAD 1)   ;  0
  (IFEQ 12)   ;  1     if v[1]=0, jump to 13
  (ILOAD 0)   ;  2
  (IFEQ 12)   ;  3     if v[0]=0, jump to 15
  (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     jump to 0
  (ICONST 0)  ; 13
  (HALT)      ; 14     halt with 0 on stack
  (ICONST 1)  ; 15
  (HALT)      ; 16     halt with 1 on stack
```

An M1 Programming Solution

```
'((ILOAD 1)    ;  0
  (IFEQ 12)    ;  1     if v[1]=0, jump to 13
  (ILOAD 0)    ;  2
  (IFEQ 12)    ;  3     if v[0]=0, jump to 15
  (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     jump to 0
  (ICONST 0)   ; 13
  (HALT)       ; 14     halt with 0 on stack
  (ICONST 1)   ; 15
  (HALT)       ; 16     halt with 1 on stack
```

An M1 Programming Solution

```
'((ILOAD 1)     ;  0
  (IFEQ 12)     ;  1      if v[1]=0, goto false
  (ILOAD 0)     ;  2
  (IFEQ 12)     ;  3      if v[0]=0, jump to 15
  (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      jump to 0
  (ICONST 0)    ; 13
  (HALT)        ; 14      halt with 0 on stack
  (ICONST 1)    ; 15
  (HALT)        ; 16      halt with 1 on stack
```

An M1 Programming Solution

```
'((ILOAD 1)    ;  0
  (IFEQ 12)    ;  1      if v[1]=0, jump to 13
  (ILOAD 0)    ;  2
  (IFEQ 12)    ;  3      if v[0]=0, jump to 15
  (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      jump to 0
  (ICONST 0)   ; 13
  (HALT)       ; 14      halt with 0 on stack
  (ICONST 1)   ; 15
  (HALT)       ; 16      halt with 1 on stack
```

An M1 Programming Solution

```
'((ILOAD 1)    ;  0
  (IFEQ 12)    ;  1      if v[1]=0, jump to 13
  (ILOAD 0)    ;  2
  (IFEQ 12)    ;  3      if v[0]=0, jump to 15
  (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      jump to 0
  (ICONST 0)   ; 13
  (HALT)       ; 14      halt with 0 on stack
  (ICONST 1)   ; 15
  (HALT)       ; 16      halt with 1 on stack
```

An M1 Programming Solution

```
'((ILOAD 1)    ;  0
  (IFEQ 12)    ;  1     if v[1]=0, jump to 13
  (ILOAD 0)    ;  2
  (IFEQ 12)    ;  3     if v[0]=0, jump to 15
  (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     jump to 0
  (ICONST 0)   ; 13
  (HALT)       ; 14     halt with 0 on stack
  (ICONST 1)   ; 15
  (HALT)       ; 16     halt with 1 on stack
```

Outline

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

M1

M1 provides

- a program counter
- local variables whose values are unbounded rationals
- an unbounded push down stack
- a program which is a finite list of instructions

State Transitions

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

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

(defun do-inst (inst s)
 (case (op-code inst)
  (ILOAD  (execute-ILOAD  inst s)) ; (ILOAD i):  push v[i]
  (ICONST (execute-ICONST  inst s)); (ICONST i): push i
  (IADD   (execute-IADD   inst s)) ; (IADD): pop twice, add, push
  (ISUB   (execute-ISUB   inst s)) ; (ISUB): pop twice, sub, push
  (IMUL   (execute-IMUL   inst s)) ; (IMUL): pop twice, mul, push
  (ISTORE (execute-ISTORE  inst s)); (ISTORE i): pop into v[i]
  (GOTO   (execute-GOTO   inst s)) ; (GOTO δ): pc := pc+δ
  (IFEQ   (execute-IFEQ   inst s)) ; (IFEQ δ): pop, if =0, pc := pc+δ
  (otherwise s)))                  ; halt
```

The Semantics of `IADD`

```
(defun execute-IADD (inst s)  ; inst = (IADD)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (+ (top (pop (stack s)))
                       (top (stack s)))
                    (pop (pop (stack s))))
              (program s)))
```

`IADD`, `ISUB` and `IMUL` operate on unbounded rationals.

The Semantics of `ISTORE`

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

Running M1

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

(defun haltedp (s)
  (equal (next-instr s) '(HALT)))
```

[*alternatively:*

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

```
s0 = (make-state 0             ; pc
                 '(5 7)        ; locals, v[0] and v[1]
                 nil           ; stack
                 '((ILOAD 1)   ; program[ 0]
                   ...         ; ...
                   (HALT)))    ; program[16]

s70 = (M1 s0 70)
s70 = (make-state 16           ; pc
                  '(0 2)       ; locals, v[0] and v[1]
                  '(1)         ; stack
                  '((ILOAD 1)  ; program[ 0]
                    ...        ; ...
                    (HALT)))   ; program[16]
```

Formalizing Total Correctness

“state α runs to halt at state β, provided γ”

$\Longrightarrow$

“$\exists n$:

α runs in n steps to halt at β, provided γ”

$\Longrightarrow$

“α runs in κ steps to halt at β, provided γ.”

$\Longrightarrow$

```
(implies γ
         (and (equal (M1 α κ) β)
              (haltedp β)))
```

“α runs in κ steps to β, provided γ” (for “less than” code)

α :

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

“α runs in κ steps to β, provided γ” (for “less than” code)

κ :

```
(lessp-clock i j)
```

```
(defun lessp-clock (i j)
  (if (zp i)
      3
      (if (zp j)
          5
          (clk+ 13
                (lessp-clock (- i 1) (- j 1))))))
```

“α runs in κ steps to β, provided γ” (for “less than” code)

β :

```
(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)))
```

“α runs in κ steps to β, provided γ” (for “less than” code)

γ :

```
(and (natp i) (natp j))
```

“α runs in κ steps to β, provided γ” (for “less than” code)

```
(implies
 (and (natp i) (natp j))
 (equal (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)      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.

Like Rogers, 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.

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.

Given a Turing machine program `tm`, initial state `st`, and tape `tape`,

“the Turing machine program *halts*”

means

$\exists\ n\ :\ (\texttt{tmi st tape tm}\ n) \neq \texttt{nil}$

“the Turing machine program *runs forever*”

means

$\forall\ n\ :\ (\texttt{tmi st tape tm}\ n) = \texttt{nil}$

The Questions

Can M1 compute anything a Turing machine can compute?

Can we implement a Turing machine interpreter on M1?

Answers: Given unbounded integers, variables, simple arithmetic, jump-if-zero, and iteration? Sure!

But how many times have you seen it proved mechanically?

In fact, how many times have you even seen it *stated* formally?

What do we prove?

Turing Completeness v. Equivalence

In 1984, Boyer and I published a paper on “A Mechanical Proof of the Turing Completeness of Pure Lisp.” It ought to have been titled “A Mechanical Proof of the Turing Equivalence of Pure Lisp.”

A computational paradigm is *Turing equivalent* if it can compute anything a Turing machine can.

A computation is *Turing complete* if it can only be computed by a Turing-equivalent computing system; i.e., if a device can carry out a Turing complete computation, it can compute any other Turing complete computation.

I’m not sure if this terminology was widely accepted in 1984.

“Turing Equivalence”

For any given Turing machine initial state, tape, and machine description, there exists an M1 state such that

(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”

For any given Turing machine initial state, tape, and machine description, there exists an M1 state such that

(a) if M1 halts, then the Turing machine halts,
and

(b) if the Turing machine halts, then M1 halts with the “same” tape.

“Turing Equivalence”

For any given Turing machine initial state, tape, and machine description, there exists an M1 state such that

(a) if M1 halts, then the Turing machine halts,
and

(b) if the Turing machine halts, then M1 halts with the “same” tape.

This must be formalized carefully to make sure M1 does the computation! (It would be cheating to map a given Turing machine configuration to the “final” M1 state with the right answer already in it.)

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.

Basic Idea

Encode initial state, Turing machine description, and tape as numbers,

put those numbers onto the M1 stack, and

invoke a fixed program Ψ to interpret the Turing machine description.

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 – state numbers

```
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 – cells as numbers

```
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))⇒ 0001 010 1 0001
```

Turing Machines – packed cells

```
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))⇒ 0001 010 1 0001

000010000000 000101010001 ... 001001100001 000100010000
```

Turing Machines – packed cells

```
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))⇒ 0001 010 1 0001

000010000000 000101010001 ... 001001100001 000100010000
=
47921276213291088842438974929274042051802613691060496
```

Turing Machines – packed cells

```
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))⇒ 0001 010 1 0001

000010000000 000101010001 ... 001001100001 000100010000
↑
‘‘nnil’’
```

Notational Conventions

- `tm`: a Turing machine description
- `st`: initial state of `tm`
- `tape`: initial tape
- tm': packed cell representation of `tm`
- st': numeric encoding of `st`
- $tape'$: binary encoding of `tape`
- pos': position of read head in `tape`

- w: bit width of state component of cells
- $nnil$: encoding of “NIL” used in packed cells
- Ψ: M1 bytecode program
- s_0:

```
(make-state 0
            '(0 0 0 0 0 0 0 0 0 0 0 0 0)
            (push* nnil w tm' pos' tape' st' nil)
            Ψ)
```

All of the encodings mentioned above are constructive (computable) functions defined in ACL2.

Theorem A

Let i be a natural number. If (M1 s_0 i) is halted, then there exists a j such that (`tmi st tape tm` j) is halted.

Theorem A

Let i be a natural number. If (M1 s_0 i) is halted, then `(tmi st tape tm (find-j st tape tm` i`))` is halted.

Theorem A

```
(with-conventions
   (implies (natp i)
            (let ((s_f (M1 s_0 i)))
              (implies
               (haltedp s_f)
               (tmi st tape tm (find-j st tape tm i)))))
```

Theorem A

```
(let*
 ((map (renaming-map st tm))
  (w (max-width tm map))
  (nnil (nnil w))
  (st′ (ncode-st st map))
  (tm′ (ncode-tm tm map w))
  (tape′ (ncode-tape tape))
  (pos′ (ncode-pos tape))
  (s_0 (make-state 0 ’(0 0 0 0 0 0 0 0 0 0 0 0 0)
                   (push* nnil w tm′
                          pos′ tape′ st′ nil)
                   (psi))))
 (implies (and (symbolp st)
               (tapep tape)
               (turing-machinep tm))
          (implies (natp i)
                   (let ((s_f (M1 s_0 i)))
                        (implies (haltedp s_f)
                                 (tmi st tape tm (find-j st tape tm i))))))))
```

Theorem A

```
(with-conventions
   (implies (natp i)
            (let ((s_f (M1 s_0 i)))
              (implies
               (haltedp s_f)
               (tmi st tape tm (find-j st tape tm i)))))
```

Theorem B

Let n be a natural number. If (`tmi st tape tm` n) is halted with tape τ, then there exists a k such that (`M1` s_0 k) is halted and its final tape and position decode to τ.

Theorem B

Let n be a natural number. If (tmi st tape tm n) is halted with tape τ, then `(M1` s_0 `(find-k st tape tm` n`))` is halted and its final tape and position decode to τ.

Theorem B

```
(with-conventions
   (implies (and (natp n)
                 (tmi st tape tm n))
            (let ((s_f (M1 s_0 (find-k st tape tm n))))
              (and (haltedp s_f)
                   (equal (decode-tape-and-pos
                           (top (pop (stack s_f)))
                           (top (stack s_f)))
                          (tmi st tape tm n))))))
```

Obvious Question

Question: Why not just compile the `tm` to M1 code?

Answer: Wait until we discuss program proofs.

M1 Programming Issues

We must write M1 programs for the following concepts:

- `lessp` – less than on naturals
- `mod` – mod on naturals
- `floor` – floor on naturals
- `log2` – log base 2
- `expt` – exponentiation
- `nst-in`, `nsym`, `nop`, `nst-out`, `ncar`, `ncdr` – subroutines used in reading the cells of a Turing machine description

- `current-symn` – read the current position on the tape
- `ninstr1` – interpret the current Turing machine state
- `new-tape2` – write to the tape and shift position
- `tmi3` – Turing machine interpreter
- `main` – load data and invoke `tmi3`

A Verifying Compiler

I wrote a verifying compiler that let me code all this in a Lisp-like language and compile it to M1 bytecode, Ψ

The compiler generated *clock functions* characterizing the runtime of each subroutine

The compiler generated and proved a correctness theorem for each subroutine with respect to its clock function and Lisp code (see below)

Implementation Notes

The first pass of the compiler generates an assembly language and symbol table and the second pass expands the assembly language to M1 and fills in jump distances to subroutines and labels

Since M1 does not provide subroutine call, my compiler implements an X86-like discipline to provide `CALL` and `RETURN` via the stack

Since M1 does not support the JVM's `JSR` (which jumps to a pc found on the stack), my compiler keeps track every call location and implements `RETURN` by compiling it to a "big switch" sequence of tests and jumps

Example

```
(defsys :ld-flg nil
  :modules
  ((lessp :formals (x y)
          :input (and (natp x)
                      (natp y))
          :output (if (< x y) 1 0)
          :code (ifeq y
                      0
                      (ifeq x
                            1
                            (lessp (- x 1) (- y 1)))))
   (mod :formals (x y)
        :input (and (natp x)
                    (natp y)
                    (not (equal y 0)))
        :output (mod x y)
        :code (ifeq (lessp x y)
                    (mod (- x y) y)
                    x))
```

```
...
(nst-out :formals (cell w)
         :input (and (natp cell) (natp w))
         :output (nst-out cell w)
         :code (mod (floor cell (expt 2 (+ 4 w) 1) 0)
                    (expt 2 w 1)))
...
```

```
...
(main :formals (st tape pos tm w nnil)
       :input (and (natp st)
                   (natp tape)
                   (natp pos)
                   (natp tm)
                   (natp w)
                   (equal nnil (nnil w))
                   (< st (expt 2 w)))
       :output (tmi3 st tape pos tm w n)
       :output-arity 4
       :code (tmi3 st tape pos tm w nnil) ; Numeric version of tmi
       :ghost-formals (n)
       :ghost-base-value (mv 0 st tape pos)))
```

Final System

```
(defun Psi ()      ; Ψ  = 896 lines of M1 code
 '((ICONST 2)
   (GOTO 843)
   (HALT)
   (ISTORE 12)
   (ISTORE 7)
   (ISTORE 6)
   (ILOAD 0)
   (ILOAD 1)
   (ILOAD 12)
   (ILOAD 6)
   (ILOAD 7)
   (ISTORE 1)
   (ISTORE 0)
   (ILOAD 1)
   (IFEQ 14)
   (ILOAD 0)
   (IFEQ 10)
(ILOAD 0)
(ICONST 1)
(ISUB)
(ILOAD 1)
(ICONST 1)
...
(ILOAD 6)
(ILOAD 7)
(ILOAD 8)
(ILOAD 9)
(GOTO -891)
(GOTO 0)
(GOTO 0)))
```

Mod's Correctness

The input to the compiler for `MOD` is:

```
(MOD :formals (x y)
     :input (and (natp x)
                 (natp y)
                 (not (equal y 0)))
     :output (mod x y)
     :code (ifeq (lessp x y)
                 (mod (- x y) y)
                 x))
```

After generating M1 code for `MOD`, the compiler defines – in the logic – the “algorithm” `!MOD` for the generated code:

```
(DEFUN !MOD (X Y)
  (IF (AND (NATP X)
           (NATP Y)
           (NOT (EQUAL Y 0)))
      (IF (EQUAL (!LESSP X Y) 0)
          (!MOD (- X Y) Y)
          X)
      NIL))
```

The compiler defines a “clock” function, first for the loop:

```
(DEFUN MOD-LOOP-CLOCK (X Y)
  (IF (AND (NATP X)
           (NATP Y)
           (NOT (EQUAL Y 0)))
      (IF (EQUAL (!LESSP X Y) 0)
          (clk+ 4
                (LESSP-CLOCK '(0 1) X Y)
                8
                (MOD-LOOP-CLOCK (- X Y) Y))
          (clk+ 4
                (LESSP-CLOCK '(0 1) X Y)
                3))
      0))
```

and then for a call of `MOD` from any `ret-pc`:

```
(DEFUN MOD-CLOCK (RET-PC X Y)
  (clk+ 10
        (MOD-LOOP-CLOCK X Y)
        5
        (EXIT-CLOCK 'MOD RET-PC)))
```

The compiler generates a theorem that establishes that running the code for the right number of steps produces a state described in terms of the code's algorithm function. (It does that first for the loop, then for the call; here is the theorem for a call.)

```
(DEFTHM MOD-IS-!MOD
  (IMPLIES
   (AND (READY-AT *MOD* (LOCALS S) 3 S)
        (MEMBER (CDR (ASSOC CALL-ID *ID-TO-LABEL-TABLE*)
                (CDR (ASSOC 'MOD *SWITCH-TABLE*)))
        (EQUAL (TOP (STACK S))
               (FINAL-PC 'MOD CALL-ID))
        ...
```

```
(IMPLIES
 (AND ...
      (EQUAL Y (TOP (POP (STACK S))))
      (EQUAL X (TOP (POP (POP (STACK S)))))
      (AND (NATP X)
           (NATP Y)
           (NOT (EQUAL Y 0))))
 (EQUAL (M1 S (MOD-CLOCK CALL-ID X Y))
        β))
```

β :

```
(MAKE-STATE
 (TOP (STACK S))
 (UPDATE-NTH* 0
              (LIST (NTH 0 (LOCALS S))
                    (NTH 1 (LOCALS S))
                    (NTH 2 (LOCALS S))
                    (NTH 3 (LOCALS S))
                    (NTH 4 (LOCALS S))
                    (NTH 5 (LOCALS S)))
              (MOD-FINAL-LOCALS CALL-ID X Y S))
 (PUSH (!MOD X Y) (POPN 3 (STACK S)))
 (PSI))
```

Then it generates the theorem that the algorithm function is equal to the specification expression provided to the compiler.

```
(DEFTHM !MOD-SPEC
  (IMPLIES (AND (NATP X)
                (NATP Y)
                (NOT (EQUAL Y 0)))
           (EQUAL (!MOD X Y) (MOD X Y))))
```

The compiler generates code for the `main` program, which pops arguments off the M1 stack and calls the (alleged) Turing machine interpreter

The compiler generates clock functions and theorems for every module, including `main`, ultimately producing a clock, `psi-clock`, and a Correctness Theorem for Ψ

Handling Partial Programs

The M1 code for `tmi3` and `main` may not halt.

The `tmi3` specification, `(!tmi3 ...` n`)`, has a *ghost variable*, n, that the generated code does not.

The `tmi3` theorem proved by the compiler compares the specification to running the M1 code for `(tmi3-clock ...`n`)` steps:

(i) if the specification halts, then M1 halts with same answer

(ii) if the specification does not halt, then M1 is at its loop with same data `!tmi3` has when it times out.

Handling Partial Programs

The M1 code for `tmi3` and `main` may not halt.

The specification, `(!tmi3 ...` n`)`, has a *ghost variable*, n, that the generated code does not.

The `tmi3` theorem proved by the compiler compares the specification to running the M1 code for `(tmi3-clock ...`n`)` steps:

(i) if the specification halts, then M1 halts with same answer

(ii) if the specification does not halt, then M1 is at its loop with same data `!tmi3` has when it times out. This state is obviously not halted!

Summary of Code

The M1 Turing machine interpreter system uses 13 local variables, 16 subroutines, and 896 M1 instructions.

The compiler generates 329 events [9 are provided as “hints” by the user] to prove that its code corresponds to the high level functions compiled.

type	*commands*	*of which [n] are user supplied*
`DEFCONST`	110	
`DEFUN`	94	
`DEFTHM`	88	[7]
`IN-THEORY`	37	[2]

The compiler fails unless all theorems are proved.

Obvious Question

Idea: Compile `tm` as an M1 program instead of interpretting a big numeric constant.

Problem: Instead of verifying a fixed program Ψ I would have to verify the compiler.

(This is possible but would have been a little harder.)

Some Proof Related Issues

A little more work is needed to relate the compiler's theorems to our Theorems A and B:

We compile a "numeric" representation of Turing machines, `tmi3`, but Theorems A and B talk about `tmi`

Our compiler theorem introduces a verified clock function that maps from Turing machine steps to M1 steps but we *also* need one that maps from M1 steps to Turing machine steps

Reductions of TMI

`Tmi` uses lists but M1 deals only with numeric data

We proved a sequence of reductions:

`tmi`

$\downarrow$

`tmi1` – introduce state numbers

$\downarrow$

`tmi2` – introduce packed cell representation of `tm`

$\downarrow$

`tmi3` – binary number and position for `tape`

$=$

the “`tmi`” we compiled

The Simulation Theorem

Assume the conventions on `st`, `tape`, `tm`, `st/`, etc.

Define
(`find-k st tape tm` n)
=(`psi-clock st/ tape/ tm/ ...` n).

Compare the result of running `tmi` n steps to the result of running M1 on s_0 for (`find-k st tape tm n`) steps:

(i) `tmi` is halted ↔ M1 is halted

(ii) `tmi` halted → M1 tape is the “same” as `tmi` tape

Proof: This follows from our `tmi`-to-`tmi3` Reduction Theorem and the compiler's Ψ Correctness Theorem. Q.E.D.

Theorem B

Let n be a natural number. If (`tmi st tape tm` n) is halted with tape τ, then (`M1` s_0 (`find-k st tape tm` n)) is halted and its final tape and position decode to τ.

Proof: See the Simulation Theorem. Q.E.D.

Theorem A – Still Work To Do!

But Theorem A requires: if M1 halts in i steps then `tmi` halts in `(find-j st tape tm` i`)` steps.

How do we define function `find-j`?

Theorem A – Still Work To Do!

But Theorem A requires: if M1 halts in i steps then `tmi` halts in `(find-j st tape tm` i`)` steps.

How do we define function `find-j`?

Aide Memoire:

We have `find-k` and we want `find-j`.

If `tmi` halts in n steps, M1 halts in `(find-k ...` n`)` steps.

If M1 halts in i steps, `tmi` halts in `(find-j ...` i`)` steps.

Observation: Simulation Theorem

Compare the result of running `tmi` n steps to the result of running M1 on s_0 for `(find-k st tape tm n)` steps:

(i) `tmi` is halted $\leftrightarrow$ M1 is halted

(ii) `tmi` halted $\rightarrow$ M1 tape is the “same” as `tmi` tape

Observation: Simulation Theorem

Compare the result of running `tmi` n steps to the result of running M1 on s_0 for `(find-k st tape tm n)` steps:

(i) `tmi` is halted at $n \leftrightarrow$ M1 is halted at `(find-k` $\ldots n$`)`

(ii) `tmi` halted $\rightarrow$ M1 tape is the "same" as `tmi` tape

Observation: Find-k is Monotonic

If `tmi` has not halted after n steps, then

(`find-k st tape tm` n)

$<$

(`find-k st tape tm (+ 1` n`))`

Note: This is actually an interesting-to-prove "self-evident" theorem whose proof involved the introduction of the notion of "trace" into my script.

Observation: Halted Means Halted!

If M1 is halted at i steps and $c \geq i$, then M1 is halted at c steps.

Defining Find-j

In Theorem A, M1 is known to halt after i steps (and thus M1 is halted at any $c \geq i$)

To find a j for which `tmi` halts:

Try successively larger j starting from 0:

- If `tmi` halts at j, return j. (Obviously a good answer.)
- If `(find-k st tape tm` j`)` $\geq i$, return j. (See below.)
- Else, keep searching upwards.

Defining Find-j

In Theorem A, M1 is known to halt after i steps (and thus M1 is halted at any $c \geq i$)

To find a j for which `tmi` halts:

Try successively larger j starting from 0:

- If `tmi` halts at j, return j. (Obviously a good answer.)
- If `(find-k st tape tm` j`)` $\geq i$, return j. (See below.)
- Else, keep searching.

This search terminates because of find-k Monotonicity.

Defining Find-j

In Theorem A, M1 is known to halt after i steps (and thus M1 is halted at any $c \geq i$)

To find a j for which `tmi` halts:

Try successively larger j starting from 0:

If `tmi` halts at j, return j. (Obviously a good answer.)

If `(find-k st tape tm` j`)` $\geq i$, return j. (See below.)

Else, keep searching.

Exiting on the second test contradicts the Simulation Theorem: M1 is halted at `(find-k` $\ldots j$`)` $\geq i$ and thus `tmi` is halted at j.

Proof Discovery

book	`defun`	`defthm`
`defsys-utilities`	4	20
`defsys-v2`	101	22
`tmi-reductions`	58	92
`implementation`	3+`defsys`	11
`theorems-a-and-b`	14	37
	180	182

The proof takes about 6 minutes on my laptop.

I spent about 10 days working on the first version and have cleaned it up twice since then.

Using M1 to Emulate Turing Machines

We can run M1.

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)`

M1 requires

`(find-k 'Q0 *example-tape* *rogers-tm* 78)` steps

So how many steps is that?

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

$=$

291,202,253,588,734,484,219,274,297,505,568,945,357,129,888,612,375,663,883

$\approx 10^{56}$ steps!

```
(find-k ’Q0 *example-tape* *rogers-tm* 78)
```

$=$

291,202,253,588,734,484,219,274,297,505,568,945,357,129,888,612,375,663,883

$\approx 10^{56}$ steps!

This is because M1 is using repeated subtractions of 1 and 2 to recover bits from large (e.g., 50 digit) numbers encoding `tm`.

It would be much faster if M1 had built-in binary arithmetic (`IFLT`, `RSH`, `MOD`)

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

Compiling `tm` to M1 would also produce a faster emulator (but we would still need log-time operations on the tape).

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 requires some coding skills and layered abstractions.

The 896 instruction M1 program is the largest M1 program I've ever verified.

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

This project shows that clock functions facilitate certain kinds of proofs.