Proof Pearl: Proving a Simple Von Neumann Machine Turing Complete

J Strother Moore

Department of Computer Science
University of Texas at Austin

presented by

Matt Kaufmann

at
ITP 2014, Vienna
July, 2014

Introduction

M1 is a simple (“toy”) model of the JVM, developed by Moore to teach formal modeling and mechanized code proof.

Details are in the paper and in ACL2 input scripts distributed with the ACL2 Community Books (as per the paper).

Feel free to email questions to moore@cs.utexas.edu.

Typical M1 Programming Challenge

Write a program that takes two natural numbers, i and j, in $reg[0]$ and $reg[1]$ and halts with 1 on the stack if $i < j$ and 0 on the stack otherwise.

Typical M1 Programming Challenge

Write a program that takes two natural numbers, i and j, in $reg[0]$ and $reg[1]$ and halts with 1 on the stack if $i < j$ and 0 on the stack otherwise.

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

Typical M1 Programming Challenge

Write a program that takes two natural numbers, i and j, in $reg[0]$ and $reg[1]$ and halts with 1 on the stack if $i < j$ 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 reg[1]=0, jump to 13;
ILOAD_0    //  2
IFEQ 12    //  3      if reg[0]=0, jump to 15;
ILOAD_0    //  4
ICONST_1   //  5
ISUB       //  6
ISTORE_0   //  7      reg[0] := reg[0] - 1;
ILOAD_1    //  8
ICONST 1   //  9
ISUB       // 10
ISTORE_1   // 11      reg[1] := reg[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;
```

Output by `javac` compiler, except pcs shown as instruction counts.

An M1 Programming Solution

```
'((ILOAD 1)    ;  0
  (IFEQ 12)    ;  1      if reg[1]=0, jump to 13;
  (ILOAD 0)    ;  2
  (IFEQ 12)    ;  3      if reg[0]=0, jump to 15;
  (ILOAD 0)    ;  4
  (ICONST 1)   ;  5
  (ISUB)       ;  6
  (ISTORE 0)   ;  7      reg[0] := reg[0] - 1;
  (ILOAD 1)    ;  8
  (ICONST 1)   ;  9
  (ISUB)       ; 10
  (ISTORE 1)   ; 11      reg[1] := reg[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;
```

Call this constant κ.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

M1

The M1 state provides

- a program counter
- a fixed (but arbitrary) number of registers whose values are unbounded integers
- an unbounded push down stack
- a program which is a fixed, finite list of instructions

M1

The M1 instruction set:

- load/store between top-of-stack and registers
- push numeric constants
- add, subtract, and multiply
- jump, conditional jump (if 0), and halt

M1

The M1 instruction set:

- load/store between top-of-stack and registers
- push numeric constants
- add, subtract, and multiply
- jump, conditional jump (if 0), and halt

M1 does not provide subroutine call and return!

Each instruction is formalized with a state transition function.

Given a state s and a natural n, we define $M1(s, n)$ to be the result of stepping n times from s.

Each instruction is formalized with a state transition function.

Given a state s and a natural n, we define $M1(s, n)$ to be the result of stepping n times from s.

It is possible to prove properties of M1 programs, e.g., that if $reg[0]$ and $reg[1]$ contain natural numbers, program κ halts and leaves 1 or 0 on the stack, depending on whether $reg[0] < reg[1]$.

Each instruction is formalized with a state transition function.

Given a state s and a natural n, we define $M1(s, n)$ to be the result of stepping n times from s.

It is possible to prove properties of M1 programs, e.g., that if $reg[0]$ and $reg[1]$ contain natural numbers, program κ halts and leaves 1 or 0 on the stack, depending on whether $reg[0] < reg[1]$.

Partial correctness results can be proved too, e.g., κ does not terminate if $reg[0]$ and $reg[1]$ are negative integers.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Turing Machines

Description* trace of TMI$(st, tape, tm, n)$

tm =*rogers-tm*	n	st	$tape$
((Q0 1 0 Q1)	0	Q0	(1 1 1 1 1)
(Q1 0 R Q2)	1	Q1	(0 1 1 1 1)
(Q2 1 0 Q3)	2	Q2	(0 1 1 1 1)
(Q3 0 R Q4)	3	Q3	(0 0 1 1 1)
(Q4 1 R Q4)	4	Q4	(0 0 1 1 1)
(Q4 0 R Q5)	5	Q4	(0 0 1 1 1)
(Q5 1 R Q5)	6	Q4	(0 0 1 1 1)
(Q5 0 1 Q6)	7	Q4	(0 0 1 1 1 0)
(Q6 1 R Q6)	8	Q5	(0 0 1 1 1 0 0)
(Q6 0 1 Q7)	9	Q6	(0 0 1 1 1 0 1)
(Q7 1 L Q7)	10	Q6	(0 0 1 1 1 0 1 0)
(Q7 0 L Q8)	...	...	...
(Q8 1 L Q1)	75	Q7	(0 0 0 0 0 0 1 1 1 1 1 1 1 1)
(Q1 1 L Q1))	76	Q7	(0 0 0 0 0 0 1 1 1 1 1 1 1 1)
	77	Q7	(0 0 0 0 0 0 1 1 1 1 1 1 1 1)
	78	Q8	(0 0 0 0 0 0 1 1 1 1 1 1 1 1) ⇐ *halted*

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

A Turing Machine Interpreter in ACL2

$$\texttt{tmi}(st, tape, tm, n) = \begin{cases} \text{final tape} & \text{if halts within } n \text{ steps} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

A Turing Machine Interpreter in ACL2

$$\texttt{tmi}(st, tape, tm, n) = \begin{cases} \text{final tape} & \text{if halts within } n \text{ steps} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

A tape is represented as a pair of extensible half-tapes $< Left, Right >$, where the read/write head is at the start of $Right$.

A Turing Machine Interpreter in ACL2

$$\texttt{tmi}(st, tape, tm, n) = \begin{cases} \text{final tape} & \text{if halts within } n \text{ steps} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

A tape is represented as a pair of extensible half-tapes $< Left, Right >$, where the read/write head is at the start of $Right$.

A tape is never `nil`.

A Turing Machine Interpreter in ACL2

$$\texttt{tmi}(st, tape, tm, n) = \begin{cases} \text{final tape} & \text{if halts within } n \text{ steps} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

A tape is represented as a pair of extensible half-tapes $< Left, Right >$, where the read/write head is at the start of $Right$.

A tape is never `nil`.

The definition of `tmi` is the ACL2 translation of the definition of NQTHM's `tmi` used in [Boyer-Moore 1984].

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Turing Completeness

“M1 can emulate TMI”

Approach: Implement TMI as an M1 program and prove it correct.

Turing Completeness

"M1 can emulate TMI"

Approach: Implement TMI as an M1 program and prove it correct.

The types of objects in the TMI model of computation are different from the types of objects in the M1 model.

TMI deals with *symbols* (e.g., `Q1`, `L`, `R`, etc) and *conses* (e.g., machine descriptions and tapes) whereas M1 only has *integers*.

Turing Completeness

"M1 can emulate TMI"

Approach: Implement TMI as an M1 program and prove it correct.

The types of objects in the TMI model of computation are different from the types of objects in the M1 model.

TMI deals with *symbols* (e.g., `Q1`, `L`, `R`, etc) and *conses* (e.g., machine descriptions and tapes) whereas M1 only has *integers*.

We must set up a correspondence a la Gödel so that TMI objects can be represented as integers and manipulated by M1 programs.

Turing Completeness

"M1 can emulate TMI (modulo correspondence)"

Approach: Implement TMI as an M1 program and prove it correct.

The types of objects in the TMI model of computation are different from the types of objects in the M1 model.

TMI deals with *symbols* (e.g., `Q1`, `L`, `R`, etc) and *conses* (e.g., machine descriptions and tapes) whereas M1 only has *integers*.

We must set up a correspondence a la Gödel so that TMI objects can be represented as integers and manipulated by M1 programs.

Conventions

Let tm, st, and $tape$ be a Turing machine description, initial state symbol, and initial tape.

Define s_0 be the corresponding M1 state with

Conventions

Let tm, st, and $tape$ be a Turing machine description, initial state symbol, and initial tape.

Define s_0 be the corresponding M1 state with

- pc = 0
- 13 registers, initially containing 0s,
- a stack containing (the numeric correspondents of) tm, st, $tape$ and certain constants used to decode them, and
- the program Ψ discussed below.

Theorems

Theorem A: If TMI runs forever on st, $tape$, and tm, then M1 runs forever on s_0.

Theorem B: If TMI halts on st, $tape$, and tm after n steps, then M1 halts on s_0 after some k steps and returns the same tape (modulo correspondence).

Theorems

Theorem A: If TMI runs forever on st, $tape$, and tm, then M1 runs forever on s_0.

Theorem B: If TMI halts on st, $tape$, and tm after n steps, then M1 halts on s_0 after some k steps and returns the same tape (modulo correspondence).

Theorems

Theorem A: If TMI runs forever on st, $tape$, and tm, then M1 runs forever on s_0.

Theorem B: If TMI halts on st, $tape$, and tm after n steps, then M1 halts on s_0 after find-k$(st, tape, tm, n)$ steps and returns the same tape (modulo correspondence).

Theorems

Theorem A: If TMI runs forever on st, $tape$, and tm, then M1 runs forever on s_0.

Theorem B: If TMI halts on st, $tape$, and tm after n steps, then M1 halts on s_0 after find-k$(st, tape, tm, n)$ steps and returns the same tape (modulo correspondence).

Theorems

Theorem A: If M1 halts on s_0 after i steps,
then TMI halts on st, $tape$, and tm after some j steps.

Theorem B: If TMI halts on st, $tape$, and tm after n steps,
then M1 halts on s_0 after find-k$(st, tape, tm, n)$ steps and
returns the same tape (modulo correspondence).

Theorems

Theorem A: If M1 halts on s_0 after i steps,
then TMI halts on st, $tape$, and tm after some j steps.

Theorem B: If TMI halts on st, $tape$, and tm after n steps,
then M1 halts on s_0 after find-k$(st, tape, tm, n)$ steps and returns the same tape (modulo correspondence).

Theorems

Theorem A: If M1 halts on s_0 after i steps,
then TMI halts on st, $tape$, and tm after find-j$(st, tape, tm, i)$ steps.

Theorem B: If TMI halts on st, $tape$, and tm after n steps, then M1 halts on s_0 after find-k$(st, tape, tm, n)$ steps and returns the same tape (modulo correspondence).

Theorems

Theorem A: If M1 halts on s_0 after i steps,
then TMI halts on st, $tape$, and tm after find-j$(st, tape, tm, i)$ steps.

Theorem B: If TMI halts on st, $tape$, and tm after n steps,
then M1 halts on s_0 after find-k$(st, tape, tm, n)$ steps and returns the same tape (modulo correspondence).

Creative Steps:

- reducing TMI to an equivalent numeric version, TMI3, by successive refinements: TMI $\approx$ TMI1 $\approx$ TMI2 $\approx$ TMI3
- defining Ψ and proving it implements TMI3
- defining find-j (to count TMI steps given M1 steps)

See the paper and scripts.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Implementation $\Psi =$

```
((ICONST 2)   ;   0
 (GOTO 843)   ;   1
 (HALT)       ;   2
 (ISTORE 12)  ;   3
 (ISTORE 7)   ;   4
 (ISTORE 6)   ;   5
 (ILOAD 0)    ;   6
 (ILOAD 1)    ;   7
 (ILOAD 12)   ;   8
 (ILOAD 6)    ;   9
 (ILOAD 7)    ;  10
 (ISTORE 1)   ;  11
 (ISTORE 0)   ;  12
 (ILOAD 1)    ;  13
 (IFEQ 14)    ;  14
 (ILOAD 0)    ;  15
 (IFEQ 10)    ;  16
 (ILOAD 0)    ;  17
 (ICONST 1)   ;  18
 (ISUB)       ;  19
 (ILOAD 1)    ;  20
 (ICONST 1)   ;  21
 (ISUB)       ;  22
 (ISTORE 1)   ;  23
 (ISTORE 0)   ;  24
 (GOTO -12)   ;  25
 (ICONST 1)   ;  26
 (GOTO 2)     ;  27
 (ICONST 0)   ;  28
 (ISTORE 6)   ;  29
 (ISTORE 12)  ;  30
 (ISTORE 1)   ;  31
 (ISTORE 0)   ;  32
 (ILOAD 6)    ;  33
 (ILOAD 12)   ;  34
 (ICONST 107);  35
 (ISUB)       ;  36
 (IFEQ 70)    ;  37
 (GOTO 15)    ;  38
 (ISTORE 12)  ;  39
 (ISTORE 7)   ;  40
 (ISTORE 6)   ;  41
 (ILOAD 0)    ;  42
 (ILOAD 1)    ;  43
 (ILOAD 12)   ;  44
 (ILOAD 6)    ;  45
 ...              ...
[824 lines elided]
 ...              ...
 (ISTORE 0)   ;869
 (ILOAD 0)    ;870
 (ILOAD 1)    ;871
 (ILOAD 2)    ;872
 (ILOAD 3)    ;873
 (ILOAD 4)    ;874
 (ILOAD 5)    ;875
 (ICONST 878);876
 (GOTO -132)  ;877
 (ISTORE 9)   ;878
 (ISTORE 8)   ;879
 (ISTORE 7)   ;880
 (ISTORE 6)   ;881
 (ISTORE 12)  ;882
 (ISTORE 5)   ;883
 (ISTORE 4)   ;884
 (ISTORE 3)   ;885
 (ISTORE 2)   ;886
 (ISTORE 1)   ;887
 (ISTORE 0)   ;888
 (ILOAD 6)    ;889
 (ILOAD 7)    ;890
 (ILOAD 8)    ;891
 (ILOAD 9)    ;892
 (GOTO -891)  ;893
 (GOTO 0)     ;894
 (GOTO 0))    ;895
```

If we had some eggs. . .

we could have eggs and ham, . . .

if we had some ham. – Groucho Marx

If we had some eggs. . .

we could have eggs and ham, . . .

if we had some ham. – Groucho Marx

If we had M1 code for less than, mod, floor, $\log_2$, and exponentiation, . . .

we could write M1 code to decode the bit-packed description tm and read/write/shift the $tape$, . . .

if we had subroutine call and return.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Verifying Compiler

To solve these problems, and automate the proofs, we wrote a verifying compiler from “Toy Lisp” to M1.

Verifying Compiler

To solve these problems, and automate the proofs, we wrote a verifying compiler from “Toy Lisp” to M1.

It maps a system of Toy Lisp programs and specifications into M1 code and lemmas to prove that each compiled routine meets its specifications when called properly.

Verifying Compiler

To solve these problems, and automate the proofs, we wrote a verifying compiler from “Toy Lisp” to M1.

It maps a system of Toy Lisp programs and specifications into M1 code and lemmas to prove that each compiled routine meets its specifications when called properly.

It supports symbolic names, formal parameters, multiple return values, and a call/return protocol that protects the caller’s environment.

Verifying Compiler

To solve these problems, and automate the proofs, we wrote a verifying compiler from “Toy Lisp” to M1.

It maps a system of Toy Lisp programs and specifications into M1 code and lemmas to prove that each compiled routine meets its specifications when called properly.

It supports symbolic names, formal parameters, multiple return values, and a call/return protocol that protects the caller’s environment.

It generated and verified Ψ above from input like this:

```
(defsys
: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))))))
```

```
(defsys
: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 ...
   :code (ifeq (lessp x y)(mod (- x y) y) x))
```

```
(defsys
: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 ...
   :code (ifeq (lessp x y)(mod (- x y) y) x))
 ...  ; 12 modules elided, building toward:
  (tmi3 ... :code ...)
```

```
(defsys
: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 ...
   :code (ifeq (lessp x y)(mod (- x y) y) x))
 ...  ; 12 modules elided, building toward:
  (tmi3 ... :code ...)
  (main ...
          :code (tmi3 st tape pos tm w nnil) ...))
:edit-commands ...)   ; user provided hints
```

What the Compiler Generates

```
(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)))))
```

What the Compiler Generates

```
(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)))))
```

- M1 code for :code in Ψ (including call/return support)

What the Compiler Generates

```
(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)))))
```

- M1 code for :code in Ψ (including call/return support)
- Proofs that the generated code satisfies :input/:output specifications

What the Compiler Generates

```
(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)))))
```

- M1 code for :code in Ψ (including call/return support)
- Proofs that the generated code satisfies :input/:output specifications

The compiler succeeds iff all proofs succeed.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Some Statistics

The M1 Turing Machine Interpreter uses 13 registers, 16 subroutines, and 896 M1 instructions.

Some Statistics

The M1 Turing Machine Interpreter uses 13 registers, 16 subroutines, and 896 M1 instructions.

The reduction of TMI to its numeric counterpart, TMI3, requires 92 user-supplied (automatically proved) lemmas.

Some Statistics

The M1 Turing Machine Interpreter uses 13 registers, 16 subroutines, and 896 M1 instructions.

The reduction of TMI to its numeric counterpart, TMI3, requires 92 user-supplied (automatically proved) lemmas.

The correctness of Ψ requires 92 proved lemmas, of which 82 are generated automatically.

Some Statistics

The M1 Turing Machine Interpreter uses 13 registers, 16 subroutines, and 896 M1 instructions.

The reduction of TMI to its numeric counterpart, TMI3, requires 92 user-supplied (automatically proved) lemmas.

The correctness of Ψ requires 92 proved lemmas, of which 82 are generated automatically.

The proofs of Theorems A and B require 37 lemmas.

Some Statistics

The M1 Turing Machine Interpreter uses 13 registers, 16 subroutines, and 896 M1 instructions.

The reduction of TMI to its numeric counterpart, TMI3, requires 92 user-supplied (automatically proved) lemmas.

The correctness of Ψ requires 92 proved lemmas, of which 82 are generated automatically.

The proofs of Theorems A and B require 37 lemmas.

Total proof time is about 3.5 minutes on a Macbook Pro 2.6GHz Intel Core i7 running CCL.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Emulating Turing Machines with M1

Given our constructive clocks, we can determine, for any Turing Machine test run (description tm, initial st, $tape$, and number of steps), how many M1 instructions it will take.

Emulating Turing Machines with M1

Given our constructive clocks, we can determine, for any Turing Machine test run (description tm, initial st, $tape$, and number of steps), how many M1 instructions it will take.

Recall `*rogers-tm*` (slide 12) on the tape `(1 1 1 1 1)` takes 78 steps to compute the tape

`(0 0 0 0 0 0 1 1 1 1 1 1 1 1)`

Emulating Turing Machines with M1

Given our constructive clocks, we can determine, for any Turing Machine test run (description tm, initial st, $tape$, and number of steps), how many M1 instructions it will take.

Recall `*rogers-tm*` (slide 12) on the tape `(1 1 1 1 1)` 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)
```

So how many steps is that?

Emulating Turing Machines with M1

Given our constructive clocks, we can determine, for any Turing Machine test run (description tm, initial st, $tape$, and number of steps), how many M1 instructions it will take.

Recall `*rogers-tm*` (slide 11) on the tape `(1 1 1 1 1)` 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)`

=

29120225358873448421927429750556894535712988861237566388 3

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

$=$

29120225358873448421927429750556894535712988861237566383

$\approx 10^{56}$ steps!

We can compute `find-k` efficiently because of additional theorems reducing each “clock function” to a closed-form algebraic expression.

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

$=$

29120225358873448421927429750556894535712988861237566388 3

$\approx 10^{56}$ steps!

We can compute `find-k` efficiently because of additional theorems reducing each “clock function” to a closed-form algebraic expression.

Good News: ACL2 can execute M1 programs at about 500,000 bytecode instructions/second!

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

$=$

291202253588734484219274297505568945357129888612375663883

$\approx 10^{56}$ steps!

We can compute `find-k` efficiently because of additional theorems reducing each “clock function” to a closed-form algebraic expression.

Good News: ACL2 can execute M1 programs at about 500,000 bytecode instructions/second!

Bad News: It would take about 1.8×10^{43} years to emulate this Turing machine run!

Emulating Turing Machines with M1

Why so long?

Emulating Turing Machines with M1

Why so long?

M1 is using repeated subtractions of 1 and 2 to recover bits from large (e.g., 50 digit) numbers encoding the Turing Machine Description.

Emulating Turing Machines with M1

Why so long?

M1 is using repeated subtractions of 1 and 2 to recover bits from large (e.g., 50 digit) numbers encoding the Turing Machine Description.

It would be much faster if M1 included more arithmetic primitives (e.g., JVM's `IFLT`, `RSH`, `MOD`).

Emulating Turing Machines with M1

Why so long?

M1 is using repeated subtractions of 1 and 2 to recover bits from large (e.g., 50 digit) numbers encoding the Turing Machine Description.

It would be much faster if M1 included more arithmetic primitives (e.g., JVM's `IFLT`, `RSH`, `MOD`).

It would be a little faster if M1 included the JVM's `JSR` and `RET` or `INVOKESTATIC` and `IRETURN`. Other ACL2 models include these.

Outline

- M1
- Turing Machines
- Turing Completeness
- Implementation
- Verifying Compiler
- Some Statistics
- Emulating Turing Machines with M1
- Conclusion

Conclusion

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

Conclusion

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

This is only the second mechanically checked Turing Complete proof. The other is [Boyer-Moore 1984] which used the same TMI.

Conclusion

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

This is only the second mechanically checked Turing Complete proof. The other is [Boyer-Moore 1984] which used the same TMI.

This is the first one for an imperative machine model.

Conclusion

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

This is only the second mechanically checked Turing Complete proof. The other is [Boyer-Moore 1984] which used the same TMI.

This is the first one for an imperative machine model.

The 896 instruction M1 program is the largest M1 program Moore has verified.

Conclusion

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

This is only the second mechanically checked Turing Complete proof. The other is [Boyer-Moore 1984] which used the same TMI.

This is the first one for an imperative machine model.

The 896 instruction M1 program is the largest M1 program Moore has verified.

The project is a great Formal Methods exercise. Try it with your favorite prover!

Thank You

(Supplemental material follows.)

```
(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))
   ...                                ; 12 modules elided
```

```
(tmi3
 :formals (st tape pos tm w nnil)
 :dcls ((declare (xargs :measure (acl2-count n))))
 :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
 (ifeq
  (- (ninstr1 st (current-symn tape pos) tm w nnil) -1)
  (mv 1 st tape pos)
  (tmi3 (nst-out (ninstr1 st (current-symn tape pos) tm w nnil) w)
        (new-tape2 (nop (ninstr1 st (current-symn tape pos))
                               tm w nnil w)
                        tape pos)
        tm w nnil))
 :ghost-formals (n)
 :ghost-base-test (zp n)
 :ghost-base-value (mv 0 st tape pos)
 :ghost-decr ((- n 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)
       :ghost-formals (n)
       :ghost-base-value (mv 0 st tape pos)))

:edit-commands ...)            ; user provided hints
```

What the Compiler Generates

```
(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)))))
```

- M1 code for :code in Ψ (incl call/return support)
- clock function (number of steps from call through ret)
- algorithm function, `!LESSP` (ACL2 translation of Toy Lisp)
- proof that code implements algorithm:
  "good call leaves `!LESSP`(x, y) on stack"
- proof that algorithm implements :input/:output spec:
  "`!LESSP`(x, y) is `(if (< x y) 1 0)`"

M1 Code for `LESSP` (within Ψ)

```
...                  || (ICONST 1)   ; 21         color coding
(ISTORE 12)  ;  3    || (ISUB)       ; 22  ||  entry prelude
(ISTORE 7)   ;  4    || (ISTORE 1)   ; 23  ||  loop
(ISTORE 6)   ;  5    || (ISTORE 0)   ; 24  ||  exit postlude - regs
(ILOAD 0)    ;  6    || (GOTO -12)   ; 25  ||  exit postlude - returning
(ILOAD 1)    ;  7    || (ICONST 1)   ; 26  ||
(ILOAD 12)   ;  8    || (GOTO 2)     ; 27  ||
(ILOAD 6)    ;  9    || (ICONST 0)   ; 28  ||
(ILOAD 7)    ; 10    || (ISTORE 6)   ; 29  ||
(ISTORE 1)   ; 11    || (ISTORE 12)  ; 30  ||
(ISTORE 0)   ; 12    || (ISTORE 1)   ; 31  ||
(ILOAD 1)    ; 13    || (ISTORE 0)   ; 32  ||
(IFEQ 14)    ; 14    || (ILOAD 6)    ; 33  ||
(ILOAD 0)    ; 15    || (ILOAD 12)   ; 34  ||
(IFEQ 10)    ; 16    || (ICONST 107) ; 35  ||
(ILOAD 0)    ; 17    || (ISUB)       ; 36  ||
(ICONST 1)   ; 18    || (IFEQ 70)    ; 37  ||
(ISUB)       ; 19    || (GOTO 15)    ; 38  ||
(ILOAD 1)    ; 20    || ...
```

Defs of Clock and Algorithm Functions

```
(DEFUN LESSP-CLOCK (RET-PC X Y)
  (CLK+ 10                              ; cost of entry
        (LESSP-LOOP-CLOCK X Y)          ; cost of loop
        4                               ; cost of restoring regs
        1                               ; cost of returning to right pc
        (EXIT-CLOCK 'LESSP RET-PC)))
```

```
(DEFUN !LESSP (X Y)
  (IF (AND (NATP X) (NATP Y))           ; :input pre-condition
      (IF (EQUAL Y 0)                   ; Toy Lisp :code trans'd to ACL2
          0
          (IF (EQUAL X 0)
              1
              (!LESSP (- X 1) (- Y 1))))
      NIL))                             ; Don't-care value
```

Thm: Code Implements Semantics

```
(IMPLIES
 (AND
  (READY-AT *LESSP* (LOCALS S) 3 S)         ; well-formed call stack
  (MEMBER (CDR (ASSOC CALL-ID *ID-TO-LABEL-TABLE*)); this call known
          (CDR (ASSOC 'LESSP *SWITCH-TABLE*)))     ;  to compiler
  (EQUAL (TOP (STACK S))                    ; top of stack is ret pc
         (FINAL-PC 'LESSP CALL-ID))         ;  for this call
  (EQUAL Y (TOP (POP (STACK S))))           ; actuals on rest
  (EQUAL X (TOP (POP (POP (STACK S)))))     ;  of stack
  (AND (NATP X) (NATP Y)))                  ; pre-conditions ok
 (EQUAL (M1 S (LESSP-CLOCK CALL-ID X Y))    ; running M1 for clock steps
        (MAKE-STATE                         ; produces a state with
         (TOP (STACK S))                    ;  pc set to ret pc
         (UPDATE-NTH* 0                     ;  restored locals
          (LIST (NTH 0 (LOCALS S)) ... (NTH 5 (LOCALS S)))
          (LESSP-FINAL-LOCALS CALL-ID X Y S))
         (PUSH (!LESSP X Y)                 ;  alg'm val pushed after
               (POPN 3 (STACK S)))          ;   popping actuals & ret pc
         (PSI))))                           ;  our program Ψ
```

Thm: Semantics Implements Spec

```
(IMPLIES (AND (NATP X) (NATP Y))     ; :input pre-condition implies
         (EQUAL (!LESSP X Y)         ; semantic function equals
                (IF (< X Y) 1 0)))   ; :output spec
```

Ghost Parameters

Two Toy Lisp programs, `TMI3` and `MAIN`, describe algorithms – and generate compiled code – that may not terminate.

Their translations to ACL2 (`!TMI3` and `!MAIN`) must be total.

The ghost parameters insure termination of the ACL2 functions used to express the programs' correctness.

The :input/:output spec for `TMI3` establishes that if `!TMI3` halts, Ψ halts with the same answer, and if `!TMI3` runs out of time, Ψ runs out of time.

See paper or `TMI3-IS-!TMI3` (gen'd by `defsys`).

Some Statistics

The M1 Turing Machine Interpreter uses 13 registers, 16 subroutines, and 896 M1 instructions.

book (i.e., file)	defun	defthm	defconst	in-theory	time
m1	29	10	0	5	1.12
tmi-reductions	56	92	2	6	88.40
defsys-utilities	4	21	0	2	0.42
defsys	54	0	0	0	0.87
implementation	1	10	0	5	2.82
autogenerated	94	81	108	33	68.28
theorems-a-and-b	15	37	0	6	16.25
find-k!	34	67	0	34	29.75
totals	287	318	110	91	207.91

Proof times in seconds on Macbook Pro 2.6GHz Intel Core i7 running CCL. Total proof time is about 3.5 minutes.