Proof Pearl: Proving a Simple Von Neumann Machine Turing Complete

J Strother Moore

Dept. of Computer Science, University of Texas, Austin, TX, USA
moore@cs.utexas.edu
http://www.cs.utexas.edu

Abstract. In this paper we sketch an ACL2-checked proof that a simple but unbounded Von Neumann machine model is Turing Complete, i.e., can do anything a Turing machine can do. The project formally revisits the roots of computer science. It requires re-familiarizing oneself with the definitive model of computation from the 1930s, dealing with a simple "modern" machine model, thinking carefully about the formal statement of an important theorem and the specification of both total and partial programs, writing a verifying compiler, including implementing an X86-like call/return protocol and implementing computed jumps, codifying a code proof strategy, and a little "creative" reasoning about the non-termination of two machines.

Keywords: ACL2, Turing machine, Java Virtual Machine (JVM), verifying compiler

1 Prelude

I have often taught an undergraduate course at the University of Texas at Austin entitled *A Formal Model of the Java Virtual Machine*. In the course, students are taught how to model sophisticated computing engines and, to a lesser extent, how to prove theorems about such engines and their programs with the ACL2 theorem prover [5]. The course starts with a pedagogical ("toy") JVM-like model which the students elaborate over the semester towards a more realistic model, which is then compared to an accurate JVM model[9]. The pedagogical model is called `M1`: a stack based machine providing a fixed number of registers (JVM's "locals"), an unbounded operand stack, and an execute-only program providing the following bytecode instructions `ILOAD`, `ISTORE`, `ICONST`, `IADD`, `ISUB`, `IMUL`, `IFEQ`, `GOTO`, and `HALT`, with unbounded arithmetic.

This set of opcodes was chosen to allow students to easily implement and verify some simple `M1` programs. On the last class day before Spring Break, 2012, the students complained that it was very hard to program `M1`; that in fact, it was "probably impossible" to do "real computations" with it because it lacks a "less than" comparator and procedures![1]

[1] Such judgements are obviously naive and ill-informed; any machine with a branch-if-0, a little arithmetic, and some accessible infinite resource is Turing Complete.

My response was "Well, `M1` can do anything a Turing machine do." But on my way home that evening, I felt guilty:

> *M1 is a pedagogical device, designed to introduce formal modeling to the students and inculcate the idea that expectations on hardware and software can often be formalized and proved. I shouldn't just say it's Turing Complete. I should show them how we can prove it with the tools they're using.*

Fortunately, I had Spring Break ahead of me and thus was born this project.

2 Source Files

The complete set of scripts for this project are part of ACL2's Community Books. See the Community Books link on the ACL2 home page [6]. After downloading and installing the books visit `models/jvm/m1/`. See the `README` file there. References to `*.lisp` files below are for that directory. If you have a running ACL2 session you could `(include-book "models/jvm/m1/find-k!" :dir :system)` and `(in-package "M1")` to inspect everything with ACL2 history commands. This paper is a guide.

3 Related Work

Turing Completeness proofs for various computational models have been a staple of computer science since the time of Turing and Church. Mechanically checked proofs of other important theorems in meta-mathematics (the Church-Rosser theorem, the Cook-Levin theorem, and Gödel's First Incompleteness Theorem) are less common but have been done with several provers. Here I focus on *mechanically checked* formal proofs of the computational completeness of a programming language.

As far as I am aware, the first and only such proof was done in 1984 [2], when Boyer and I proved that Pure Lisp was Turing Complete, using the prover that would become Nqthm. We were asked to prove completeness by a reviewer of [3], in which we proved that the halting problem for Pure Lisp was undecidable; the reviewer objected that we had not proved Pure Lisp Turing Complete.

An important distinction between [2] and the present work is that the "suspect" computational model in the former is the lambda calculus with general recursion (Pure Lisp), whereas here it is a very simple Von Neumann machine (or imperative programming language) similar to the contemporary JVM and its bytecode language [8].

While I'm unaware of other mechanically checked proofs that a given programming language is Turing Complete, this work also involves proofs of properties of low-level assembly code and a verifying compiler. This tradition goes back at least as far as the mechanically checked proof of a compiler by Milner and Weyhrauch in 1972 [10]. Highlights of subsequent systems verification work

involving such mechanically checked reasoning include the "CLI verified (hardware/software) stack" of [1], and of course the even more realistic results of the seL4 microkernel [7] and VCC projects [4]. But even with a verified program one must prove that the specification is Turing Complete.

4 Turing Machines

The present work uses the same Turing machine model as [2] (ported from Nqthm to ACL2) which was accepted by the reviewers of that paper. The model is based on Rogers' classic [12] formalization. A Turing machine description, *tm*, (sometimes called an "action table") is a finite list of 4-tuples or *cells*, $\langle st_{in}, sym, op, st_{out} \rangle$. Rogers represents a tape as a pair of half tapes, each being a (finite but extensible) list of 0s and 1s. The concatenation of these two half tapes corresponds to the intuitive notion of a tape (extensible in both directions) with a read/write head "in the middle." Rogers shows one may start with an extensible finite tape. The read/write head is thought of as positioned on the first symbol on the right half. The interpretation of each cell in description *tm* is "if, while in state st_{in}, *sym* is read from the tape, perform operation *op* on the tape and enter state st_{out}." Here, st_{in} and st_{out} are symbolic state names, *sym* is 0 or 1, and *op* is one of four values meaning write a 0, write a 1, shift left, or shift right. The machine halts when the current state and symbol read from the tape do not match any st_{in} and *sym* in *tm*.

We define `tmi` ("Turing machine interpreter") to take a Turing machine state name, tape, and a Turing machine description and a number of steps, n. `Tmi` returns either `nil` ("false") meaning the machine did not reach a halted state in n steps, or the final tape produced after n steps. By our choice of tape representation, a tape can never be `nil` and so the function `tmi` indicates whether the computation halted in n steps and the final tape if it did halt. See the definition `tmi` in `tmi-reductions.lisp`. I will colloquially refer to `tmi` as our "official" model of Turing machines.

In our official model, Turing machine descriptions and cells are lists constructed with `cons`, state names are Lisp symbols (e.g., `Q1`, `LP`, `TEST`), "symbols" on the tape are integers `0` or `1`, and operations are Lisp objects `0`, `1`, `L`, or `R`. See the definition of `*rogers-program*` in `tmi-reductions.lisp` for an example.

5 M1

`M1` is defined in a similar style but takes an `M1` state and a number of steps. An `M1` state contains a program counter ("pc"), a list of integers denoting register values, a stack of integers, and a program; all components of an `M1` state are represented with lists, symbols and numbers in the obvious way. The integers are unbounded, the stack may grow without bound. An arbitrary number of registers may be provided but the number of allocated registers never grows larger than the largest register index used in the program. Programs are finite and fixed ("execute only").

Programs are lists of the *instructions* as described below. The notation "$reg[i]$" denotes the contents of register (JVM local variable) i. "$reg[i] \leftarrow v$" denotes assignment to a register; "$pc \leftarrow v$" denotes assignment to the program counter. The notation "$\ldots, x, y, a \Rightarrow \ldots, v$" describes the manipulation of the stack as per [8] and means that three objects, x, y, and a, are popped from the stack (with a being the topmost) and v is pushed in their place. That portion of the stack ("$\ldots$") deeper than x is unaffected. The first six instructions below always increment the pc by 1, i.e., $pc \leftarrow pc + 1$ is implicit.

instruction	*stack*	*description*
`(ILOAD` n`)` :	$\ldots \Rightarrow \ldots, reg[n]$	
`(ISTORE` n`)` :	$\ldots, v \Rightarrow \ldots$	$reg[n] \leftarrow v$
`(ICONST` k`)` :	$\ldots \Rightarrow \ldots, k$	
`(IADD)` :	$\ldots, x, y \Rightarrow \ldots, x + y$	
`(ISUB)` :	$\ldots, x, y \Rightarrow \ldots, x - y$	
`(IMUL)` :	$\ldots, x, y \Rightarrow \ldots, x \times y$	
`(GOTO` d`)` :	$\ldots \Rightarrow \ldots$	$pc \leftarrow pc + d$
`(IFEQ` d`)` :	$\ldots, v \Rightarrow \ldots$	$pc \leftarrow pc + (\textbf{if}\ v = 0\ \textbf{then}\ d\ \textbf{else}\ 1)$
`(HALT)` :	$\ldots \Rightarrow \ldots$	no change to state

Note that by not changing the state, the `HALT` instruction causes the machine to stop. We consider an `M1` state *halted* if the pc points to a `HALT` instruction.

To *step* an `M1` state the instruction at pc in the program is fetched and executed as described above. We define `(M1` s n`)` to step state s n times and return the final state. See `m1.lisp` for complete details of the `M1` model.

An example of an `M1` program to compute the factorial of register 0 and leave the result on top of the stack is:

program	*pc*	*pseudo-code*
`'((ICONST 1)`	`; 0`	
`(ISTORE 1)`	`; 1`	$reg[1] \leftarrow 1$
`(ILOAD 0)`	`; 2`	
`(IFEQ 10)`	`; 3`	**if** $reg[0] = 0$, **then jump to** 13
`(ILOAD 1)`	`; 4`	
`(ILOAD 0)`	`; 5`	
`(IMUL)`	`; 6`	
`(ISTORE 1)`	`; 7`	$reg[1] \leftarrow reg[1] \times reg[0]$
`(ILOAD 0)`	`; 8`	
`(ICONST 1)`	`; 9`	
`(ISUB)`	`; 10`	
`(ISTORE 0)`	`; 11`	$reg[0] \leftarrow reg[0] - 1$
`(GOTO -10)`	`; 12`	**jump to** 2
`(ILOAD 1)`	`; 13`	
`(HALT))`	`; 14`	**halt** *with* $reg[1]$ *on top of stack*

This program runs forever (never reaches the `HALT`) if $reg[0]$ is negative.

If we require as a precondition that $reg[0]$ is a natural number, a statement of total correctness can be paraphrased as: If s is an `M1` state with pc 0, the natural number n in $reg[0]$ and the list above as the program, then there exists

a natural number i such that `(M1 s i)` is a halted state with $n!$ on top of the stack.

To state and prove such a theorem it is convenient to define a witness for the existentially quantified i. This witness is delivered by a user-defined *clock function* that takes n as an argument and returns a natural number.

The ACL2 Community Books directory `models/jvm/m1/` contains many example `M1` programs along with machine checked proofs of their correctness via such clock functions and other methods[2].

6 The Correspondence Conventions

To state Turing equivalence I followed the approach of [2]. Paraphrasing it into the `M1` setting, I set up a correspondence between official Turing machine representations of certain objects (e.g., machine descriptions and state names) and their `M1` representations. The former are composed of lists, symbols, and integers; the latter are strictly numeric.

Consider an arbitrary cell, $\langle st_{in}, sym, op, st_{out}\rangle$, in a Turing machine description tm. Given that tm contains only a finite number of state name symbols, we can allocate a unique natural to each and represent these naturals in binary in a field of width w (which depends on the number of state names in tm). We could represent each of the four possible op as naturals in 2 bits but we allocate 3 bits. Let the numeric encodings of the four elements of *cell* be $st'_{in}, sym', op', st'_{out}$, respectively. Then the encoded *cell* is $cell' = st'_{in}+2^w sym'+2^{w+1}op'+2^{w+4}st'_{out}$.

Using this convention we can represent a list of cells, tm, as follows. The empty list is represented as an encoded "cell" of 0s with $op' = 4$ (using the otherwise unnecessary 3rd bit of op'). We call this value $nnil$. A non-empty list whose first cell is represented by $cell'$ and whose remaining elements are recursively represented by $tail$ is represented by $cell' + 2^{4+2w}tail$.

The tape (which, recall, also encodes the read/write head "in the middle") is represented on M1 as two natural numbers, one specifying (via its binary expansion) the contents of the tape and the other specifying the head position (via the number of bits in the left-half tape). Henceforth I use these conventions:

level	*variable*	*value*
Official	tm	: a Turing machine description
	st	: a Turing machine state name
	tape	: a Turing machine tape (with encoded head)
M1	w	: width of a state symbol encoding req'd by st and tm
	$nnil$	: the marked encoded "cell" ($op' = 4$) (wrt w)
	tm'	: the `M1` (numeric) representation of tm (wrt w and $nnil$)
	st'	: the `M1` (numeric) representation of st
	tape$'$	: the `M1` (numeric) representation of *tape* contents
	pos'	: the `M1` (numeric) representation of the head position
	s_0	: the initial `M1` state described below

[2] See [11].

The initial M1 state s_0 is an M1 state with program counter 0, thirteen registers set to 0, the stack in which st', $tape'$, pos', tm', w, and $nnil$ have been pushed, and finally, as the program, a certain, fixed list of M1 instructions. That list of M1 instructions, called Ψ and described below, is (allegedly) a Turing machine interpreter in the programming language of M1. Note that s_0 does not specify how long the Turing machine is to run.

The macro `with-conventions` in `theorems-a-and-b.lisp` formally defines these conventions. The macro binds the ACL2 variable `s_0` (aka s_0) to the value above, in terms of the variables `tm`, `st`, and `tape`. Technically, it binds `w`, `nnil`, `tm'`, `st'`, `tape'`, and `pos'` as specified in terms of `tm`, `st`, and `tape`, and binds `s_0` in terms of those auxiliary variables.

7 Theorems Proved

The discussion in [2] requires us to prove:

Theorem A. If `tmi` runs forever on st, $tape$, and tm then M1 runs forever on s_0. More precisely, we phrase this in the contrapositive and say that if M1 halts on s_0 in i steps then there exists a j such that `tmi` halts in j steps.

Theorem B. If `tmi` halts on st, $tape$, and tm in n steps, there exists a k such that M1 halts on s_0 in k steps and computes the corresponding tape.

```
(defthm 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))))))
  :hints ...)

(defthm 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)))))))
```

Note that when the `tmi` expressions are used as literals (e.g., in the conclusion of `theorem-A` and the hypothesis of `theorem-B`) it is equivalent to asserting termination (non-`nil` returned value) of the `tmi` run. When `tmi` is used in the equality, we know the value is a tape and the equality checks the correspondence with what M1 computes.

In formalizing these statements there is an opportunity to subvert our goal by defining a devious sense of correspondence! The correspondence has access to the full power of the logic and *could*, for example, compute the right answer from `tm`, `st`, and `tape` and encode it into `s_0`. The correspondence above is not "devious."

It remains to explain the fixed `M1` program, Ψ, and the witness functions `find-j` and `find-k` which constructively establish the existence of the step counts mentioned in the informal statements of the theorems. But first, it is convenient to refine `tmi` into a function that operates on the kind of data `M1` has: numbers.

8 Refinement

We refine the official definition of `tmi` into a function named `tmi3` and verify that it corresponds to `tmi` modulo the representational issues. The proof is done in several steps which successively implement the change of representations of *tm* and *tape*.

- `tmi1` is like `tmi` but for a renamed *tm* with numeric state names
- `tmi2` is like `tmi1` but for tm', w and $nnil$
- `tmi3` is like `tmi2` but for $tape'$ and pos'

This concludes with the theorem `tmi3-is-tmi` in `tmi-reductions.lisp`. It is `tmi3` we will implement on `M1`.

9 The `M1` Program Ψ

Key to our proof is the definition of an `M1` program Ψ for interpreting arbitrary Turing machine descriptions on a given starting state and tape. Ψ either runs forever or `HALT`s; and when it halts, the representation of the official final tape can be recovered from the `M1` state.

Given the limited instruction set of `M1`, it is necessary to implement some simple arithmetic utilities as `M1` programs. Ψ is then the concatenation of all these programs together with "glue code" permitting procedure call and return.

name	*stack*			*description*
LESSP :	$\ldots, x, y$	$\Rightarrow$	$\ldots, v$	$v = (\mathbf{if}\ x < y\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$
MOD :	$\ldots, x, y$	$\Rightarrow$	$\ldots, (x\ mod\ y)$	
FLOOR :	$\ldots, x, y, a$	$\Rightarrow$	$\ldots, (a + \lfloor x/y \rfloor)$	
LOG2 :	$\ldots, x, a$	$\Rightarrow$	$\ldots, (a + log_2(x))$	
EXPT :	$\ldots, x, y, a$	$\Rightarrow$	$\ldots, (a + x^y)$	

We underline program names to help the reader; MOD names an `M1` program, `mod` names an ACL2 function[3]. For brevity, the descriptions above do not include the effects of these programs on the pc or registers. In addition, certain obvious

[3] ACL2 is case insensitive; formally MOD is `'MOD`

preconditions obtain (e.g., for FLOOR, all operands are natural numbers and y is non-0).

With these primitives and subroutine call/return it is not difficult to define slightly higher level M1 programs for accessing encoded Turing machine descriptions, states, and tapes. The names below are all prefixed with 'n' because these functions are the numeric correspondents of functions in the official model of tmi. In the following, $cell'$ is the numeric encoding of some cell $\langle st_{in}, sym, op, st_{out}\rangle$, $st'_{in}, sym', op', st'_{out}$ are the corresponding numeric encodings, tm is assumed non-empty (and so its car is a cell with encoding car' and its cdr is a list of cells with encoding cdr', and tm' is not $nnil$), w is the width of the state symbol encoding, and $nnil$ is the marked cell.

name	:	*stack*		*description*
NST-IN	: $\ldots, cell', w$	$\Rightarrow$	$\ldots, st'_{in}$	
NSYM	: $\ldots, cell', w$	$\Rightarrow$	$\ldots, sym'$	
NOP	: $\ldots, cell', w$	$\Rightarrow$	$\ldots, op'$	
NST-OUT	: $\ldots, cell', w$	$\Rightarrow$	$\ldots, st'_{out}$	
NCAR	: $\ldots, tm', w$	$\Rightarrow$	$\ldots, car'$	
NCDR	: $\ldots, tm', w$	$\Rightarrow$	$\ldots, cdr'$	

With these programs we can implement M1 programs for implementing the numeric version of tmi.

NCURRENT-SYM : $\ldots, tape', pos' \Rightarrow \ldots, sym'$

Description: sym' is the symbol at position pos' of $tape'$

NINSTR1 : $\ldots, a, b, tm', w, nnil \Rightarrow \ldots, cell'$

Description: $cell'$ is the first encoded cell in tm with $st'_{in} = a$ and $sym' = b$, if any, or -1 if no such cell exists

NEW-TAPE2 : $\ldots, op', tape', pos' \Rightarrow \ldots, tape'_{nx}, pos'_{nx}$

Description: op' is the encoding of a tape operation; $tape'_{nx}$ and pos'_{nx} are produced by performing that operation on $tape'$ and pos'

TMI3 : $\ldots, st', tape', pos', tm', w, nnil \Rightarrow \ldots, tape'_{nx}, pos'_{nx}$

Description: This is the M1 program that interprets the Turing machine tm with initial state st and input tape $tape'$ and pos'. The program returns the $tape'_{nx}$ and pos'_{nx} representing the final tape if the machine halts, or runs forever otherwise.

Note that "tmi3" is both the name of an M1 program and of a function defined in ACL2 as part of our refinement of tmi to the M1 representations. However, the program TMI3 takes the six arguments listed above, while the function tmi3 takes an additional argument: the number of steps to take, n[4]. The program TMI3 may run forever. The function tmi3 is total.

MAIN : $\ldots, st', tape', pos', tm', w, nnil \Rightarrow \ldots, tape'_{nx}, pos'_{nx}$

[4] Actually, the function tmi3 takes six, not seven, arguments because it does not need $nnil$: it is determined from w.

Description: By convention, our compiler starts execution with the MAIN program and our MAIN just calls TMI3 above.

10 Verifying Compiler

Writing the sixteen programs above is tedious if done directly. Perhaps the main problem is that M1 does not support subroutine call and return: M1 operates on one "flat" program space! Furthermore, the machine does not provide "computed jumps" like the JVM's JSR (which pops the stack into the pc). There is a strict separation of data from pcs. Every GOTO and IFEQ is pc relative, but the distance skipped is always some constant specified in the instruction. Of course, writing the programs is only part of the battle: they must also be verified to implement the ACL2 function tmi3.

I thus decided to write a compiler from a simple "Toy Lisp" subset to M1 code. The compiler takes as input a system description, containing source code and specifications for every subroutine.

The verifying compiler is called defsys (see defsys.lisp). Ψ is generated by the defsys expression in implementation.lisp. Every subroutine to be compiled is given a name, a list of :formals, an :input precondition, an :output specification describing the top of the stack, and the Toy Lisp source :code. It was sufficient and convenient to support only tail-recursive source code functions. As illustrated by main below, a subroutine may return multiple values and provision is made via so-called "ghosts" to model partial programs with total functions. Finally, optional arguments :ld-flg and :edit-commands allow the user to debug and modify the generated events. Inspection of implementation.lisp will reveal that three edit commands were used to augment the automatically generated commands. These generally inserted additional lemmas to prove before certain automatically generated theorems.

```
(defsys :ld-flg nil        ; debugging aid
  :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))
...
(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-added modifications
```

Toy Lisp is just the subset of ACL2 composed of variable symbols, quoted numeric constants, the function symbols `+`, `-`, `*` (primitively supported by `M1`), the form `(MV` a_1 ... a_n`)` for returning multiple values, the form `(IFEQ` a b c`)` (which is just ACL2's `(if (equal` a `0)` b c`)`), and calls of primitive and defined Toy Lisp functions.

In addition to producing the `M1` object code, the compiler (a) provides a call/return protocol, (b) links symbolic names to actual pcs (and generates appropriate relative jumps), and (c) produces the ACL2 commands (definitions and theorems) establishing that the object code is correct with respect to the Toy Lisp and that the Toy Lisp implements the `:output` specification.

If the maximum number of registers required by any subroutine's body is max, the call/return protocol requires $2max + 1$ registers. We divide them into max so-called A-registers, $max + 1$ B-registers. The A-registers are for use by the subroutine body and the B-registers are used by the call/return protocol. For simplicity we assume (and check) that the maximum number of registers used by a subroutine body is equal to the number of input parameters of the subroutine.

Note that of the sixteen programs sketched above, `TMI3` has the most parameters: 6. Thus, we need 13 registers.

The basic protocol for calling a subroutine *subr* of n arguments with arguments $a_1, \ldots, a_n$, is as follows: the caller pushes $a_1, \ldots, a_n$, and the *pc* to which *subr* should return. The caller then jumps to the pc of *subr*. At that pc, a prelude for *subr* pops $a_1, \ldots, a_n$, and *pc* into the B-registers. It then protects the caller's environment by pushing the first n A-registers onto the stack, followed by the return *pc* from the B-registers. Finally, it moves the other B-registers (containing $a_1, \ldots, a_n$) to the first n A-registers[5]. A symmetric postlude supports returning

[5] The only way to move a value from one register to another is via the stack; only the topmost item on the stack can be accessed per instruction.

$k \leq n$ values on the stack. At the conclusion of the postlude, the code jumps to the return *pc*.

But how can `M1` jump to a pc found on the stack if the ISA firmly separates "data" from "pcs"? The answer is quite tedious: the compiler keeps track of every call of each subroutine; the postlude for each subroutine concludes with a "big switch" which compares the "return *pc*" (data on the stack) to the known pc of each call and then jumps to the appropriate pc.

The compiler works in several passes. The first pass compiles the object code but includes symbolic labels and pseudo-instructions for `CALL` and `RET`. The second pass expands the `CALL` and `RET` "instructions" into appropriate sequences of `M1` code. The last pass removes and replaces labels by relative jumps to the appropriate pcs. The compiler saves the output of the three passes in the ACL2 constants `*ccode*`, `*acode*`, and `*Psi*` respectively. These may be inspected after the compiler is run.

The key to generating the clock functions is just to count instructions in the prelude, loop, and postlude of each subroutine.

`Defsys` generates certain definitions and theorems for each subroutine, admits the definitions under the logic's definitional principle, and proves the theorems. The important ones are noted below for LESSP[6]. Recall that LESSP takes two arguments, x and y. The `:input` condition of the module is that both x and y are naturals. The `:output` condition is that 1 or 0 is on top of the stack, depending on whether $x < y$. The source `:code` for the module is shown above. When *rpc* is mentioned below it is the return pc from some call of LESSP in Ψ. When s is mentioned it is an `M1` state with program Ψ. Toy Lisp translations to ACL2 have names beginning with "`!`".

Def (`!lessp` x y): the ACL2 function `!lessp` is defined

```
(defun !lessp (x y)
  (if (and (natp x) (natp y))  ; :input condition
      (if (equal y 0)          ; translated Toy Lisp
          0
          (if (equal x 0)
              1 (!lessp (- x 1) (- y 1))))
      nil))
```

Def (`lessp-loop-clock` x y): defined to compute the number of `M1` steps from the loop in LESSP to the postlude

Def (`lessp-clock` *rpc* x y): defined to compute the number of `M1` steps to get from the top of the prelude in LESSP through the return to *rpc*

Thm `lessp-loop-is-!lessp`: if the pc in s is at the top of the loop in LESSP, with x and y (satisfying the stated `:input` conditions) in the first two A-registers, then after (`lessp-loop-clock` x y) steps the pc is at the postlude, all of the A-

[6] It is easiest to inspect the results by loading the project into ACL2 (Section 2) typing (`pe '`*name*), where name is the name of an event mentioned here.

registers except the first two are unchanged, and `(!lessp` x y`)` has been pushed on the stack

Thm `lessp-is-!lessp`: if the pc in s is poised at the pc of LESSP and the stack contains at least three values, x, y, and rpc, where x and y satisfy the `:input` conditions on `lessp` and rpc is a known return pc from LESSP, then after `(lessp-clock` rpc x y`)` steps the pc is rpc, the A-registers are unchanged, and `(!lessp` x y`)` has been pushed onto the stack obtained by popping off x, y, and rpc

Thm `!lessp-spec`: if x and y satisfy the `:input` conditions for `lessp`, then `(!lessp` x y`)` is as specified by the `:output`, i.e., it is 1 or 0 depending on $(x < y)$.

Putting the last two theorems together allows ACL2 to deduce that every jump to LESSP in Ψ just advances the pc to the return pc, pops the arguments and the return pc off the stack, and pushes 1 or 0 according to the specification, without changing the A-registers.

`Defsys` compiles M1 code and generates and proves analogous definitions and theorems for every module. Thus, it compiles TMI3 and proves that running that code produces the results specified by `tmi3`. The only wrinkle in this story is that `tmi3` takes a step-count argument while the program TMI3 does not. However, provision is made for this via the user-supplied "ghost" parameters of `defsys`. The clock function `tmi3-clock` and the `:code` function `!tmi3` are augmented by an additional formal parameter, the user-supplied `:ghost-formal` n. In recursion (once per iteration), these functions decrement n and halt if $n = 0$. No such parameter exists in the compiled code. But `defsys` proves that the code, when run according to `tmi3-clock`, returns the same result as `tmi3` (both wrt n), or else is left "still running" at the top of its loop.

11 Finishing the Proof

From the theorems in `tmi-reductions.lisp` we get that the official Turing machine interpreter, `tmi`, is equal to `tmi3` modulo the representations, for any Turing step count n.

From `implementation.lisp` we get theorems about MAIN, its ACL2 analogue, `!main` and its `:output` specification function `tmi3`. In particular the theorem `main-is-!main` tells us that if invoked appropriately and run for `main-clock` `M1`-steps (for exactly n iterations), the result is exactly described by its Lisp analogue `!main`: If `!main` reports halting after n iterations, then the final `M1` state has as its pc the return pc of the call of MAIN in Ψ, and the stack contains same tape and position computed by `!main`; and if `!main` reports that it did not halt (in n iterations) the `M1` state is poised at the top of the loop in the TMI3 program.

Meanwhile, `!main-spec` tells us that `!main` computes the same thing as `tmi3`.

Since `main-clock` starts counting from the pc of MAIN and Ψ just pushes the return pc, jumps to MAIN, and `HALT`s, we define `(find-k` st $tape$ tm n`)` to be just 2 more than `main-clock` on the corresponding arguments st', $tape'$, pos', tm', w, $nnil$ and n.

In `theorems-a-and-b.lisp` we combine these results in the `simulation` theorem, which states that an `M1` run starting in initial state s_0 and taking `(find-k` *st tape tm n*`)` steps is halted precisely if `tmi` halts in n steps, and furthermore, that if `tmi` halts in n steps, then the answer in the final `M1` state corresponds to the tape computed by `tmi`.

Now we wish to prove theorem A and B. In fact, `theorem-B` (see page 6) follows easily from the `simulation` theorem.

Theorem A (page 6) requires more work. Recall that it deals with the non-termination of the two machines. Informally, it says that if `tmi` fails to terminate, then so does `M1`. But we phrased it in the contrapositive: if `M1` terminates, then so does `tmi`.

Here we know that `M1` halts on s_0 after i steps and we must define `find-j` to return a number of steps sufficient to insure that `tmi` halts. Notice that the previously defined `find-k` counts `M1`-steps and now we seek to count `tmi` steps.

Two observations are important in defining `find-j`. The first is a theorem called `find-k-monotonic` in `theorems-a-and-b.lisp` which states that if `tmi` has not halted after n steps then `(find-k` *st tape tm n*`)` $<$ `(find-k` *st tape tm* $n+1$`)`. This is actually an interesting non-trivial theorem to prove, whose proof involved the only use of traces in the script.

The second observation is an easy one called `m1-stays-halted`: once `M1` has halted, it stays halted. Thus, if `M1` is halted after i steps it is halted after any greater number of steps.

We can then define `find-j` to find a `j` at which `(tmi` *st tape tm j*`)` is halted given that we know `(M1` s_0 i`)` is halted. The definition searches upwards from $j = 0$: if `tmi` is halted at j, return j; if `(find-k` *st tape tm j*`)` $\geq i$, return j; else search from $j + 1$.

This is a well-defined function: the recursion terminates because the `find-k` expression is growing monotonically and will therefore eventually reach the fixed i, if the earlier exit is not taken first.

It is easy to see that if `M1` is halted at i, then `tmi` is halted at `(find-j` *st tape tm i*`)`: either `find-j` returns a j (in the first exit) known to be sufficient or else it returns a j such that `(find-k` *st tape tm j*`)` $\geq i$. But our second observation above shows that `M1` must thus be halted at `(find-k` *st tape tm j*`)`. And if `M1` is halted there, then `tmi` must be halted at j, by the `simulation` theorem.

That completes our proof sketch of theorem A.

12 Efficiency Considerations

`M1` is an executable operational model which ACL2 can execute at about 500,000 M1 bytecodes/sec. We can therefore run Ψ to simulate Turing machines. The clock function `find-k` tells us exactly how long we must run it to simulate a given `tmi` run of n steps.

Consider Rogers' Turing machine description for doubling the number on the tape [12]. Suppose the tape starts with Rogers' representation for 4 on the tape.

Running `tmi` experimentally reveals it takes 78 Turing steps to reach termination and compute a tape representing 8. We can use `find-k` to determine how long it takes `M1` to simulate this computation. And the answer is:

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

or slightly more than 10^{56} steps!

The primary reason our implementation is so inefficient is that tapes and Turing machine descriptions are represented as large (bit-packed) integers and must be unpacked on `M1` with programs that use `LESSP`. But the only way to answer the question "is $x < y$?" for two naturals x and y on `M1` is to subtract 1 from each until one or the other becomes 0, because the only test `M1` programs can perform is equality against 0. Thus it takes exponential time to unpack[7].

The efficiency of our `M1` Turing machine interpreter would be much improved if `M1` provided the JVM instruction `IFLT` (branch if negative) or `IF_ICMPLT` (branch if $x < y$). Further improvement could be made by having `IDIV` (floor), or bit-packing operations like `ISHR` (shift right), `IAND` (bit-wise and), etc., and perhaps arrays (with `IALOAD` and `IASTORE`), to represent the tape. Minor further improvements could be had by supporting `JSR` or even `INVOKESTATIC` or `INVOKEVIRTUAL` to make call/return simpler. All of these features are supported on our most complete JVM model, M6[9].

Another obvious approach would have been to compile the Turing machine description *tm* into an `M1` program. Had I done so, the proofs of theorems A and B would have required proving that the Turing machine compiler was correct for all possible Turing machine descriptions. By representing Turing machine descriptions as data to be interpreted, I could limit my compiler's task to proving that its output was correct on the 16 Toy Lisp modules discussed. Put succinctly, it is easier to write a verifying compiler than to verify a compiler.

13 Project History

I developed `M1` in 1997 to teach my JVM modeling course, which I subsequently taught about ten times. While the ISA of `M1` changed annually to make homeworks harder or easier, programming `M1` and proving correctness of my programs became almost second nature to me.

The question of `M1`'s computational power arose in class in March, 2012. I completed the first version of this proof March 10–18, 2012 after coding Ψ by hand in 804 `M1` instructions and manually typing the specifications and lemmas. I was helped enormously by the 1984 paper [2] and my experience with `M1`.

After Spring Break, I gave two talks on the proof: one to the Austin ACL2 research group and one to my undergraduate JVM class. Neither talk went smoothly and I learned a lot about the difficulty of presenting the work. A few weeks later, in early April, 2012, I decided to implement the verifying compiler. The present version of the proof was polished by April 14, 2012.

[7] And ACL2 would take exponential time evaluating `find-k` except for the theorems in `find-k!.lisp`

14 Conclusion

Aside from the satisfaction of formally revisiting the roots of computer science, this work allowed me to go back into class after Spring Break and say:

`M1` *can do anything a Turing machine can do. Here's a proof.*

References

1. W. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.
2. R. S. Boyer and J. S. Moore. A mechanical proof of the turing completeness of pure lisp. In W. W. Bledsoe and D. W. Loveland, editors, *Contemporary Mathematics: Automated Theorem Proving: After 25 Years*, volume 29, pages 133–168, Providence, Rhode Island, 1984. American Mathematical Society.
3. R. S. Boyer and J. S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, 31(3):441–458, 1984.
4. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*. Springer, 2009.
5. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
6. M. Kaufmann and J. S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2014.
7. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an os kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, October 2009.
8. T. Lindholdm and F. Yellin. *The Java Virtual Machine Specification, 2nd edition*. Prentice Hall, 1999.
9. H. Liu. *Formal Specification and Verification of a JVM and its Bytecode Verifier*. PhD thesis, University of Texas at Austin, 2006.
10. R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In *Machine Intelligence 7*, pages 51–72. Edinburgh University Press, 1972.
11. S. Ray and J. S. Moore. Proof styles in operational semantics. In A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD-2004)*, volume 3312 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2004.
12. H. Rogers. *A Theory of Recursive Functions and Effective Commputability*. McGraw-Hill, 1967.