Work thus far

We aim to verify the proof of Magic the Gathering's (henceforth MTG) Turing completeness using ACL2. As such, we've been working on implementing robust models of Rogozhin's UTM(2,18) and the Magic the Gathering Turing machine suitable for proof work in ACL2.

The UTM

Our implementation of UTM(2,18) is similar to that of J Strother Moore's 4-tuple Rogers Turing machine, used in his proof that the M1 language is Turing complete, with some key differences. Notably, instead of 4-tuple instructions, we are forced to use 5-tuple instructions: UTM(2,18) is able to both write a symbol and move the head in a single instruction, rather than just one of these operations at a time. We therefore need to perform both of these operations when we produce a new tape, which is also modeled based on the tape Moore used.

Our UTM interpreter, `utmi`, performs `n` operations on a tape with the Turing machine specified by the rules in `*utm-2-18*`. It returns `nil` if the tape has run for `n` steps without terminating, or if the tape has encountered a `HALT` symbol. On a Rogozhin UTM, this is equivalent to the head moving past the leftmost symbol written on the tape, `'c<1`. However, for now, we choose to model this with the `HALT` symbol.

MTG

In compliment to our work on UTM(2,18), we've been developing a model of the Turing machine described by *Magic: The Gathering is Turing Complete*[1] in ACL2. While implementing this Turing machine inside all of the rules of MTG requires several game turns of setup and card-modification (Via card-modifying cards such as Glamerdye), we chose to omit the details of this setup, focusing instead on the resulting structure.

As MTG does not have an ordered list structure anywhere, representing order is tricky. Per *Magic: The Gathering is Turing Complete*[2], we've implemented the *Creature* card type for this purpose. The *name* of a creature represents one of the 18 symbols in the UTM alphabet- creatures have been chosen such that their names start with different letters, for legibility. The *power and toughness* of a creature determine how far offset it is from the head of the Turing machine, with 2 being the position of the head. The *color* of a creature determines whether the symbol is offset to the left or the right of the head- the creatures representing the tape are either *green* or *white*, representing symbols to the left and right of the head respectively.

Now that we have symbols, we need to model the production functions of UTM(2,18). We model the cards Rotlung Reanimator and Xathrid Necromancer, which each create a 2/2 black Zombie token, either tapped or untapped, when a certain creature dies. After using some card-modifying game abilities to tweak both the trigger creature and spawned creature, Rotlung and Xathrid become suitable representations of UTM *production functions*. These cards now have the property that they produce new creatures (tape symbols) whenever a certain creature dies (has its *toughness* reduced to 0). While a Rotlung represents a production function where the state remains the same, a Xathrid represents a production function where the state changes. Additionally, four *spell* cards are used, which we model as functions operating on the machine. Infest allows for the removal of the head of the tape, as it reduces all creatures' power and toughness by 2 until the end of the in game turn, which will reduce the head's toughness to 0, killing it (removing the current symbol there) and triggering a Rotlung Reanimator or Xathrid Necromancer to produce a new creature (the new head). Cleansing Beam combined with Vigor gives all creatures of a certain color 2 additional power and toughness, while Soul Snuffers removes 1 power and toughness from all creatures. This allows the tape to move left/right, as we **increase** the offset from the head in one direction, and **decrease** the offset in the

other. If, after Cleansing Beam, the tape does not change, the machine has halted. Otherwise, the combination of Cleansing Beam and Soul Snuffers establishes a new head, as some other creature which had 3 power and toughness in either direction will now have 2 power and toughness, which we code as the head. All of these cards are modeled through functions which operate on the "tape", or list of creature tokens which represent symbols on a Turing machine, so that we can produce a new tape and have a function `mtgi` which updates our playing board accordingly.

Unimplemented MTG

We made a conscious decision **not** to implement certain other parts of MTG (turn-taking, phase/tap state, etc) which the proof uses into ACL2 at the current time. Our reasoning for this is twofold: first, this allows us to build up our model, rather than starting with a more abstract model of MTG and needing to reduce it down to a UTM. This means we can be sure a given reduction works, before implementing more MTG features into our code. Second, we need to be sure that we are able to deliver a product. Implementing the full system for MTG might be outside the scope, so we want to ensure we can deliver a final product by targeting this reduced version of MTG ***first***, and building up from there if we can successfully prove that.

Current concerns, TODO

Our next steps are as follows: - Reduce the MTG machine to a simpler machine, which eliminates ordering by toughness/color and orders based on that - Prove theorems A and B for the two machines - If time permits, implement tapped creatures and phasing into the MTG model - reduce this model to our initial MTG model

These first two steps are pivotal to completing our project, and we have begun work on the first one, but don't have any code which is truly deliverable at this time, as we're still working out the logistics of this proof. This step only requires mapping the unordered list of creatures to an ordered list of creatures, with all information about their position removed. We first write a function which processes a list and orderes the creatures accordingly. We then develop a proof that the output of a machine which processes these creatures is equivalent to mapping the first pass of `mtgi` to output on the same tape/program. The second proof we believe is easier than the existing Moore proof, as that proof needed to find an existing natural that caused both programs to terminate with the same tapes, as the M1 machine ran much slower than the Rogers TM, while the UTM/MTG machine are 1-to-1 in terms of steps: the UTM is directly embedded in MTG. Therefore, we do not need to develop the clock lemmas, or find-j/find-k functions which Moore does. We only need to prove theorems A and B, which require some intermediate lemmas, in particular that the MTG machine stays halted when further instructions are run.

The next two steps we began work on, but decided to set aside to focus on getting results from the proof first. These steps would require modelling the tapped state for creatures, as well as *phasing* for the production functions, which is how MTG handles state changes (only one set of production functions is active at one time). This would require adding a boolean to the representation of creatures and production functions, and slowing down the speed at which the MTG TM operates to accurately reflect turns in MTG. This would require developing the clocks similar to what Moore did, although this too would be simpler--we know that any instruction with a state change takes 3 turns, while any instruction without takes 4 turns. We will attempt to implement this following the success of the initial proof. If we aren't able to implement this step, we still have a working proof of a reduced representation for Magic: The Gathering.

Overall status

We are in a good position to finish this theorem and present it. We've worked on the data definitions, and a

lot of time has gone into understanding and synthesizing the results from 3 papers: *Magic: The Gathering is Turing Complete*,Rogozhin's 1996 proof that the UTM(2, 18) is Turing complete, and Moore's proofof the Turing equivalence of the M1 system. As such, we feel prepared to write this theorem, and constructed our data with this end goal in mind.

Code

The code for this project is also available at https://github.com/soulwa/mtg-tm.

utm.lisp

```
;; an implementation of a tag system
;; some possible tags
(defconst *alphabet*
  '(a b c d e f g h i j k l m n o p q r))
;; an example of a tag system program
(defconst *even-huh-tag-sys*
  '(2 ((a ()) (b (b)) (c (b c))) (c c a a a a)))
;; determines if valid tag
;; if an invalid tag has been produced, it's likely
;; HALT
(defun tag-symp (x)
  (member x *alphabet*))
;; listof symbol
;; used for production functions
(defun tag-losp (x)
  (cond ((endp x) T)
        ((consp x) (and (tag-symp (car x))
                        (tag-losp (cdr x))))))
;; remove n elements of a list from the list
(defun remove-n (n x)
  (if (zp n)
      x
      (remove-n (- n 1) (cdr x))))
;; because tag systems "eat" tapes left to right
;; no need for a half-tape
;; a list of symbols
(defun tag-tapep (x)
  (cond ((endp x) T)
        ((consp x) (and (tag-symp (car x))
                        (tag-tapep (cdr x))))))
;; production function
;; a cons of a symp to a los-symp
(defun tag-funcp (x)
  (and (consp x)
       (= (length x) 2)
       (tag-symp (car x))
       (tag-losp (car (cdr x)))))
;; a list or "set" of prod-funcs
(defun tag-funcsp (x)
  (cond ((endp x) T)
        ((consp x) (and (tag-funcp (car x))
                        (tag-funcsp (cdr x))))))
;; does this tag-func x apply to sym y
(defun tag-func-appliesp (x y)
  (and (tag-funcp x)
       (tag-symp y)
       (equal (car x) y)))
;; apply func
(defun tag-func-apply (x)
```

```
  (car (cdr x)))
;; return the applicable func from a set
;; assumes such a func exists
(defun tag-funcs-applicable (x y)
  (cond ((endp x) nil)
        ((consp x) (if (tag-func-appliesp (car x) y)
                       (car x)
                       (tag-funcs-applicable (cdr x) y)))))
;; a tag system has a tape,
;; deletion number, and prod. function
;; Alphabet is included in the defintion of tape
;; '(,deletion-number ,prod-funcs ,tape)
(defun tag-sysp (x)
  (and (consp x)
       (equal (length x) 3)
       (natp (first x))
       (tag-funcsp (second x))
       (tag-tapep (third x))))
;; step the tape
;; find the applicable func in the tag system
;; using the first element of the tape and the set of production functions
;; append the application of that function to the end
;; delete the deletion number from the head
(defun tag-step (m funcs tape)
  (remove-n m (append tape
                      (tag-func-apply
                      (tag-funcs-applicable funcs (car tape))))))
(defun tag-exec (m funcs tape n)
  (declare (xargs :measure (nfix n)))
  (cond
   ;; both of these are termination conditions
   ((< (length tape) m) tape)
   ((zp n) `(nil ,tape))
   (t (utm-exec m funcs (tag-step m funcs tape) (- n 1)))))
;; wrapper for utm-exec, which works on a tag system construct
(defun tag-sys-exec (tag-sys n)
  (tag-exec (first tag-sys)
            (second tag-sys)
            (third tag-sys)
            n))
;; implementation of Rogozhin's UTM(2, 18)
(defconst *rogozhin-alphabet*
  '(1 1> 1< 1>1 1<1 b b> b< b>1 b<1 b2 b3 c c> c< c>1 c<1 c2))
(defconst *rogozhin-states*
  '(q1 q2))
(defconst *utm-2-18-prog*
  '((q1 1 c2 L q1)
    (q1 1> 1<1 R q1)
    (q1 1< c2 L q1)
    (q1 1>1 1 R q1)
    (q1 1<1 1>1 L q1)
    (q1 b b< R q1)
    (q1 b> b<1 R q1)
    (q1 b< b L q1)
    (q1 b>1 b R q1)
    (q1 b<1 b>1 L q1)
    (q1 b2 b3 L q2)
    (q1 b3 b>1 L q2)
    (q1 c 1> L q2)
    (q1 c> c< R q1)
    (q1 c< c>1 L q1)
    (q1 c>1 c<1 L q1)
    (q1 c<1 HALT nil nil)
```

```
    (q1 c2 1< R q1)
    (q2 1 1< R q2)
    (q2 1> 1< R q2)
    (q2 1< 1> L q2)
    (q2 1>1 1<1 R q2)
    (q2 1<1 1 L q2)
    (q2 b b2 R q1)
    (q2 b> b< R q2)
    (q2 b< b> L q2)
    (q2 b>1 b<1 R q2)
    (q2 b<1 b> L q2)
    (q2 b2 b R q1)
    (q2 b3 b<1 R q2)
    (q2 c c< R q2)
    (q2 c> c< R q2)
    (q2 c< c> L q2)
    (q2 c>1 c2 R q2)
    (q2 c<1 c2 L q1)
    (q2 c2 c L q2)))
;; represents a symbol that's part of a UTM(2, 18)
(defun utm-symbolp (x)
  (member x *rogozhin-alphabet*))
;; represents a direction to move the tape
(defun utm-dirp (x)
  (or (equal x 'L) (equal x 'R) (equal x nil)))
;; represents a state in the UTM(2, 18)
(defun utm-statep (x)
  (or (member x *rogozhin-states*) (equal x nil)))
;; represents an instruction
(defun utm-instrp (x)
  (and
    (true-listp x)
    (= (length x) 4)
    (utm-symbolp (first x))
    (or (utm-symbolp (second x)) (equal x 'HALT))
    (utm-dirp (third x))
    (utm-statep (fourth x))))
;; represents a list of instructions, or: a machine
;; might not be turing complete (no guarantees are made
;; by this function), but a program which runs on a tape
;; we know our *utm-2-18* is complete thanks to Rogozhin's proof
(defun utm-loinstrp (x)
  (cond
    ((endp x) t)
    ((consp x) (and (utm-instrp (first x))
                    (utm-loinstrp (rest x))))))
;; gets the head of the tape, or a new blank symbol
(defun sym (x)
  (if (consp x) (first x) '1<))
;; get the next instruction for a utm, given a symbol, state, and a program
(defun instr (sym st progm)
  (cond
    ((endp progm) nil) ;; error
    ((and
      (equal st (first (first progm)))
      (equal sym (second (first progm))))
     (sym (first progm)))
    (t (instr sym st (rest progm)))))
;; definition of tape based on tmi-reductions.lisp
;; represents half of a turing machine's tape
(defun half-tapep (x)
  (cond
    ((endp x) t)
```

```
    ((consp x) (and (utm-symbolp (first x))
                    (half-tapep (rest x))))))
;; represents a full tape:
;; '(,first-half ,second-half)
;; where first-half is reversed from its current order, and
;; (first second-half) is the head of the tape
(defun tapep (x)
  (and (consp x)
    (half-tapep (first x))
    (half-tapep (rest x))))
;; rev1 based on tmi-reductions.lisp
;; this function adds each element of x to a in reverse order
;; so it returns
;; (append (reverse x) a)
(defun rev1 (x a)
  (cond
    ((endp x) a)
    ((consp x) (rev1 (rest x) (cons (first x) a)))))
;; shows the tape (from tmi-reductions.lisp)
;; for debugging purposes
(defun show-tape (tape)
  (cond ((consp tape)
         (rev1 (car tape)
                  (cons '[ (cons (sym (cdr tape)) (cons '] (cdr (cdr tape)))))
        (t nil)))
;; gets the head of the tape
(defun tape-head (tape)
  (sym (rest tape)))
;; generates a new tape given a new mark to write
;; and a direction to move in
(defun new-tape (mark dir tape)
  ;; shadow tape with the new tape, since we always replace the mark
  (let ((tape (cons (first tape) (cons mark (rest (rest tape))))))
    (case dir
      ;; push head to the beginning of (first tape), which
      ;; represents the order of the symbols backwards
      (L (cons (rest (first tape))
               (cons (sym (first tape))
                     (rest tape))))
      ;; add new symbol to second half of tape
      (R (cons (cons (sym (rest tape))
                     (first tape))
               (rest (rest tape)))))))
;; runs a UTM on a given tape
(defun utmi (st tape tm n)
  (declare (xargs :measure (nfix n)))
  (cond
    ;; we're done taking steps
    ((zp n) nil)
    ;; the symbol appears in the progm
    ((instr (tape-head tape) st tm)
     (let ((inst (instr (tape-head tape) st tm)))
       (utmi (fourth inst)
             (new-tape (second inst) (third inst) tape)
             tm
             (- n 1))))
    ;; the symbol doesn't appear: must be HALT symbol, as it doesn't have look
    (t nil)))
```

mtg.lisp

```
;; An ACL2 implementation of an MTGTM.
;; https://arxiv.org/abs/1904.09828
;; MTGTM creature types.
(defconst *types*
          '(aetherborn basilisk cephalid demon elf faerie giant harpy illusion
                       juggernaut kavu leviathan myr noggle orc pegasus rhino
                       silver))
;; MTGTM tape-creature colors
(defconst *tokencolor*
          '(white green))
;; MTGTM creature type recognizer
(defun ctypep (x)
  (member x *types*))
;; MTGTM creature color recognizer
(defun ccolorp (x)
  (member x *tokencolor*))
;; Example tape-creature
(defconst *aetherborn-ex*
  '(aetherborn green 2 2))
;; example rotlung
(defconst *rotlung-1<q1*
  '(T 'cephalid *aetherborn-ex* T))
;; example xathrid
;; note that (rotlungp *xathrid-b2q1*) will be true, since
;; they both produce a creature, just that xathrid produces a tapped
;; creature (transitioning the state)
(defconst *xathrid-b2q1*
  '(nil kavu (leviathan white 2 2) nil))
;; creature cards (or tokens) are of a type, have a color,
;; and have power and toughness
(defun creaturep (x)
  (and (= (length x) 4)
       (ctypep (first x))
       (ccolorp (second x))
       (natp (third x))
       (natp (fourth x))))
;; a collection of creatures, where order does not matter
(defun creaturesp (x)
  (cond ((endp x) T)
        ((consp x) (and (creaturep (first x))
                        (creaturesp (rest x))))))
;; rotlung renanimator- writes symbols when infest causes
;; the head to die
;; two booleans represent state
(defun rotlungp (x)
  (and (= (length x) 4)
       (booleanp (first x))
       (ctypep (second x))
       (creaturep (third x))
       (booleanp (fourth x))))
;; list of rotlungs
(defun rotlungsp (x)
  (cond ((endp x) T)
        ((consp x) (and (rotlungp (first x))
                        (rotlungsp (rest x))))))
;; find the rotlung appropriate rotlung given state and type
(defun applicable (tm typ st)
  (cond ((endp tm) 'error)
        ((consp tm) (if (and (equal (first (first tm)) st)
```

```
                            (equal (second (first tm)) typ))
                      (first tm)
                      (applicable (rest tm) typ st)))))
;; finds the head creature, of power and toughness 2/2
(defun find-head (tape)
  (cond ((endp tape) 'error)
        ((consp tape) (if (and (= 2 (third (first tape)))
                               (= 2 (fourth (first tape))))
                          (first tape)
                          (find-head (rest tape))))))
;; move the computation
;; return the new tape and a new state
(defun infest (tape st tm)
  (let* ((hd (find-head tape))
         (rlg (applicable tm (first hd) st)))
  (cons (remove hd (append (list rlg) tape)) (fourth rlg))))
;; cast beam on creatures
(defun vigor-beam (tape color)
  (cond ((endp tape) nil)
        ((consp tape) (if (equal color (second (first tape)))
                      (cons (list (first (first tape))
                                  (second (first tape))
                                  (+ 2 (third (first tape)))
                                  (+ 2 (fourth (first tape))))
                            (vigor-beam (rest tape) color))
                      (cons (first tape) (vigor-beam (rest tape) color))))))
;; termination condition
(defun victoryp (tape1 tape2)
  (equal tape1 tape2))
;; cast snuffers, dealing -1 -1 to all
(defun snuffers (tape)
  (cond ((endp tape) nil)
        ((consp tape) (cons (list (first (first tape))
                                  (second (first tape))
                                  (- (third (first tape)) 1)
                                  (- (fourth (first tape)) 1))
                            (snuffers (rest tape))))))
;; interpret this MTGTM for n steps or until termination, whichever comes firs
(defun mtgi (st tape tm n)
  (declare (xargs :measure (nfix n)))
  (let* ((head (find-head tape))
         (new (infest tape st tm))
         (advanced (vigor-beam (first new) (second head))))
    (cond ((zp n) nil)
          ((victoryp tape advanced) nil)
          (T
            (mtgi (second new)
                  advanced
                  tm
                  (- n 1))))))
```

---

1. https://arxiv.org/pdf/1904.09828.pdf ?