Final Project: Proving Turing Completeness of Magic: The Gathering in ACL2

Authors: Leonid Belyaev and Samuel Lyon

Introduction

The aim of our project is to demonstrate that the game Magic: The Gathering (henceforth MTG) is capable of embedding a Universal Turing machine, and is thus both Turing complete in its ruleset and undecidable. Our project specifically proves that operations on a sequence of "creature tokens", cards in the game which encode symbols on a Turing machine, leave the sequence of cards **well-formed**: each time the game advances analogously to the UTM(2, 18), the cards describe a properly constructed input tape for a Turing machine.

Our project was completed in ACL2, and requires a great deal of lemmata to get working. We develop an interpreter for a UTM(2, 18), or a Universal Turing machine with 2 states and 18 symbols, as described in Yurii Rogozhin's *Small universal Turing machines.*[1] We use existing interpreters, such as the one found here and on Replit to ensure the soundness of our interpreter, since it serves as the source of truth for our project.

We also write various lemmata to build up to our final proof, and ensure the soundness of our additional functions. These will be discussed in detail below.

We use the `lists-light` library, which is part of the ACL2 community books. Specifically, we use the `perm` and `memberp` functions, along with their associated lemmata and equivalence/congruence relations to prove our own lemmata and main theorem.

Finally, our work is inspired by the paper *Magic: The Gathering is Turing Complete* by Churchill et al.[2] as we model the system laid out in this paper which embeds a UTM(2, 18) in the MTG card game. J Strother Moore's *Proof Pearl: Proving a Simple Von Neumann Machine Turing Complete*[3] and its accompanying code in the ACL2 community book `models/jvm/m1` served as the starting point and inspiration for the structure of our project, though we have since pivoted in both direction and scope.

Background

There is no existing implementation of the proof that MTG is Turing complete, in any automated theorem prover, as far as we know. As such, this project

[1] (J Strother Moore, 2014)
[2] (Churchill et al., 2019)
[3] (Yurii Rogozhin, 1996)

serves as a first attempt to implement a working proof (or part of one) of Turing equivalence in a system where it was not intended. The UTM(2, 18) is a popular machine used to show that systems such as video games are Turing equivalent. This project also stands as the first attempt of modelling one of these systems alongside a UTM(2, 18) in an automated theorem prover.

Our data types begin with a working implementation of a UTM(2, 18). We model the *production functions* for the universal Turing machine, often called *instructions* in other machines, as 5-tuples, with a state *q1* and a symbol *s1* expressing that this function should be read when the machine is in state *q1*, and the interpreter reads *s1* from the input tape. The remaining values are another symbol to write to the tape, a direction for the head position to move in, and a new state to transition the interpreter into. We model the tape with a cons list of symbols, corresponding to those used by Rogozhin in his paper.

Our other data types are analogous to those presented in MTG, with some simplifications made. We model the production functions, or Rotlung Reanimators/Xathrid Necromancers similarly to those in the UTM. These are 4-tuples, consisting of a state and creature name expressing that this Rotlung/Xathrid card has its ability triggered, a single *creature* which it produces, and a new state to transition into. *Creatures* represent a symbol on the tape much like the UTM tape, but they also contain positional information regarding where they are on the tape- for MTG's "battlefield" is not ordered. The creature with 2 power and 2 toughness is always at the head, and other creatures all have power and toughness greater than this, representing how far offset from the head they are. All creatures are either *green*, indicating that they are to the left of the head, *white*, indicating that they are to the right, or *blue*, which is only the case when the creature token which halts the game, the assassin, is written to the tape. A creature's color additionally indicates what direction the tape should move in once the creature is written to the head of the tape.

To move the tape, the players cast spells which deal damage to or empower creatures, raising their *power* or *toughness*, which will shift them along the tape. These are represented by functions which operate on every creature on the tape (`snuffers`) or creatures of a specified color (`vigor-beam`) so that at each movement step, one set of creatures is shifted left and another is shifted right.

Our source code is available here.

Walkthrough

We first implement a tag system and UTM(2, 18) in ACL2, to ensure that our domain knowledge of the task ahead was sufficient. Using existing models, we set both of these machines up to terminate on certain example programs. Once these worked correctly, they served as a source of truth for the output of the MTG interpreter.

We now shift files to MTG, where our first steps mirror those performed with UTM. We define an interpreter for this "MTGTM" following the structure described above. The first step in defining the well-formedness of MTGI is the re-ordering of the battlefield- our tape. After defining a comparator on creatures, and testing it for symmetry and transitivity, we define an insertion sort on the battlefield. We now have everything we need to transfer the creatures of the battlefield into the tape representation used in UTM- that is, a list of two half-tapes, where the green creatures make up the left half-tape, and the white creatures the right. Another reduction- that from creature types to simple characters- gets us closer to the UTM.

Now, we delve into lemmas with the objective of separating us from `MTGI`- that is, we wish to prove ultimately that `MTGI` is equivalent to a new machine `mtg-ord`, which operates on the modified tape that we just generated. This `mtg-ord`, in turn, would be connected to `utmi`- and so, by transitivity, `mtgi` would be isomorphic to `utmi`, and `mtgi` would be Turing complete.

With this objective in mind, we wish to assert the well-formedness of our insertion sort. That is, given a battlefield of creatures, the insertion sort operation should always sort that list correctly. This goal is composed of two main sublemmas- the insertion sort produces an ordered list of creatures, and the insertion sort produces a permutation of its input- these goals in turn require a few sublemmas in order to pass.

Given that our insertion sort is a well-formed operation, we move on to reasoning about our new half-tape model. Particularly, we wish to reason about the exclusion of our two half-tapes from one another, their exclusion of the head of the tape, and its termination behavior. This requires us to reason about several of our functions and data representations, and our work culminates in `new-tape-is-well-formed`- `new-tape` being the function that uses the Turing machine interpreter's overall state in order to generate the new tape after one step.

We reason about the left and right halves of the tape, ensuring that they are well-formed- they include only the creatures of the appropriate color, and not the head. Furthermore, they never intersect. We move on to reasoning about halting behavior- halting occurs whenever the assassin creature type is spawned by our production function creatures, a behavior that corresponds to the writing of the HALT symbol to the tape of the UTM. The tape halts only if this is the case, in which our Turing machine runs out of instructions. Furthermore, given a halted tape, stepping that tape by any amount of steps will not modify that tape- it stays halted for good.

Finally, we move into reasoning about `infest` and `move`, the functions that actually drive the computation, and generate the new tape. As `:instances` of previous results, we show that infest does not modify the left or right halves of the tape- only the head. Further, changing the head afterwards does not modify the the and right halves of the tape.

The remaining piece is to reason about the well-formedness of `move`. However, we found this result very challenging, and elected to `skip-proofs` it- to demonstrate it from smaller lemmas would require more invariants to be maintained over the structure of `mtgi`, which we elected to avoid.

Results

We show that a machine running purely based on MTG cards and spells demonstrates some properties of a Turing machine. We show that it is capable of halting, and remains halted after termination, as well as reasoning about the behavior of its tape. We show that, through breaking up each step of the MTG machine into its smallest parts, one can mechanically prove that it behaves like a Turing machine in how it reads, writes, and moves the tape. We show all of these properties are upheld despite the tape having a fundamentally different structure, being ordered by values contained within creatures rather than being ordered by default, as a Turing machine tape ordinarily is. Our results are not what we originally set out to prove, however, as we aimed to prove the Turing equivalence of MTG to the UTM(2, 18). By this metric, we have fallen short, but we were able to explore and demonstrate an essential property of MTG that allows one to properly embed a Turing machine in the game.

Personal Progress

Our work with UTM and MTG was certainly enlightening. While we set out with the seemingly straightforward objective of proving the Turing equivalence of UTM and MTGI using theorems A and B, per we soon discovered that this objective was a bit more than we could pull off. Once we had internalized all of the relevant research, we set about putting it all together. As a trial run, we worked on implementing a tag system in ACL2- the high level computational model to be embedded into a Turing machine. This exercise did not present significant problems. The next step was to implement the UTM itself- that is, the Turing machine in which this tag system was to be embedded. This too was relatively straightforward. Moore's work[4] served as a useful guide in this exercise, and we adopted several of his design choices that we found useful. We now could begin work on MTG.

In initializing our work, MTG was definitely more difficult than UTM and MTG, as we now had several layers of choices to make- how much of the card game proper should we implement into our Turing machine emulator? It has turn-taking, cloaking states, tapping states, and other details- not to mention that this MTG Turing machine takes a considerable about of in-game setup to pull off, utilizing many modifier cards from across the game's long run[5]. If our models of

[4](Moore, 2014)
[5](Churchill et al., 2019)

MTG are too high-level, we risk failing to achieve much in the way of reasoning about their Turing completeness in relation to UTM. Many, many reductions would have to take place, and the final result would be a computationally heavy proof indeed. On the other hand, we may use a more reduced representation of MTG. At its most basic level, this MTG TM is, after all, a UTM[6]- and we might implicitly reduce nearly all of it down to UTM if we so desired. This approach, however, obviously makes for a less interesting result. We opted for something towards this interesting side of the spectrum. We keep all of the implementation details which made UTM meaningfully different from MTG, but strip out cloaking states, tapping, and turn-taking, which we saw as contributing nothing meaningful to the final product.

One early difficulty we encountered was the speed at which our proofs were submitting. For example, we handwrote `mem`, `permutation`, and `remove` functions, as we were accustomed to doing from class. However, we quickly realized that this was a mistake, as it took upwards of 10 minutes to see whether or not any changes we made to proofs took effect or not. This came to a head when implementing insertion sort, which became incredibly difficult to test as it took extremely long to attempt to submit the function each time. However, our research led us to `lists-light` in the ACL2 community books, and we were able to leverage the libraries to exponentially improve our runtime, thanks to congruences and equivalences we otherwise wouldn't have access to. This allowed us to speed up our proofs across the board, and made developing much easier. Additionally, working with the `perm` library led to another breakthrough, where we learned that removing the type constraint `creaturesp` to prove `isort-is-perm`, then adding it in a later theorem, was able to force ACL2 to prove the theorem using rewrite rules and existing axioms, rather than immediately attempt proof by induction. Our experience setting up insertion sort early in the project allowed us to gain stronger command of ACL2, and learn how to encourage it when it seemed that proofs took far too long.

Throughout the programming process, the design of our data took turns as we looked to find the best way to represent it in the face of our invariants. For example, encoding the tape much like Moore's tape, where the head is at the front of the right half of the tape, seemed like an easy way to go about the data representation for the input, and this is the structure we followed for our UTM. However, much later in MTG, we determined that we needed to represent the right tape as its own entity, in order to uphold the invariants on the left and right halves of the tape being sequential when we modified the head. Another example of this is the `head` function, versus the `find-head` function. Initially, we only had one function which either found the head, or returned the blank symbol. However, for certain parts of our proof, we found it was necessary to also determine whether the tape had a head at any given time, and so we employed this separate function solely for the purpose of upholding the structure of our data.

[6] ibid.

References

Moore, J Strother. 2014. “Proof Pearl: Proving a Simple Von Neumann Machine Turing Complete.” In Interactive Theorem Proving, 406–20. Springer International Publishing. https://doi.org/10.1007/978-3-319-08970-6_26.

Churchill, Alex, Stella Biderman, and Austin Herrick. “Magic: The Gathering Is Turing Complete,” April 23, 2019. https://arxiv.org/abs/1904.09828.

Rogozhin, Y. (1996). Small universal Turing machines. Theoretical Computer Science, 168(2), 215–240. https://doi.org/10.1016/s0304-3975(96)00077-1