DYNAMICAL SYSTEMS AND COMPUTABILITY THEORY

Forest Kobayashi and Matthew LeMay
Department of Mathematics, Harvey Mudd College, Claremont, CA, 91711

Abstract

Abstract goes here Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

1. Introduction

We'll begin by introducing some of the basic vocabulary used in formal language theory. We then discuss the layers of the *Chomsky Hierarchy*, and how the associated models of computation for each layer can be viewed as dynamical systems.

1.1. Strings and Alphabets

First, we give the definitions for *alphabets* and *strings*. These form the backdrop for all of our discussions of computability theory.

Definition 1.1 (Alphabet)**.** An *alphabet* is a finite set, often denoted by Σ. The elements of Σ are called *characters*. ◊

We make no assumptions about the characters in Σ; they will *exclusively* serve as formal symbols for use in strings.

Definition 1.2 (Strings)**.** Let Σ be an alphabet. We define an operation $\cdot$ on Σ as follows: Let $\sigma_1, \sigma_2 \in \Sigma$ be arbitrary (not necessarily distinct). Then define a *new* symbol $\sigma_1\sigma_2$, and let

$$\sigma_1 \cdot \sigma_2 = \sigma_1\sigma_2.$$

Also define a special symbol $\epsilon \notin \Sigma$ with the property that for all $\sigma \in \Sigma$,

$$\epsilon\sigma = \sigma = \sigma\epsilon.$$

Define $\Sigma^\star$ to be the collection of all formal symbols generated by $\Sigma \cup \{\epsilon\}$ under the operation $\cdot$ described above. Explicitly, if $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, then we have

$$\begin{aligned}\Sigma^\star &= \{\epsilon,\ [\sigma_1],\ \ldots,\ [\sigma_n],\ [\sigma_1\sigma_1],\ \ldots,\ [\sigma_i\sigma_j],\ \ldots\} \\ &= \bigcup_{k=0}^{\infty} \left\{ [\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}] \;\middle|\; \sigma_{i_j} \in \Sigma \right\}\end{aligned}$$

Note: we are using the tortoise shell brackets 〔〕 just to help visually separate the elements. Also, when $k = 0$, we take the convention that $[] = \epsilon$. In any case, we call

1) $\Sigma^\star$ the *Kleene star* of Σ,
2) The $\cdot$ operation the *concatenation operation*, and
3) The elements of $(\Sigma^\star, \cdot)$ *strings*. ◊

Remark. This is the same as defining $(\Sigma^\star, \cdot)$ to be the *free monoid* over Σ. This isn't a terribly enriching perspective, we just think the word "monoid" sounds funny so we wanted to say it.

Definition 1.3 (Length of a string)**.** Let Σ be an alphabet, and let $\sigma \in \Sigma^\star$. Suppose σ is of the form

$$\sigma = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}$$

where $\sigma_{i_j} \in \Sigma$ for each $j = 1, \ldots, k$. Then we define the *length* of σ (denoted $|\sigma|$) to be

$$|\sigma| = k. \qquad ◊$$

Remark. Note, since $\epsilon \notin \Sigma$, our assumption that each of the $\sigma_{i_j} \in \Sigma$ makes the length function well-defined.

Definition 1.4 (Language)**.** Let Σ be an alphabet, and let $L \subseteq \Sigma^\star$. Then we call L a *language* over Σ. ◊

On their own, alphabets and strings are not very interesting. Hence most of our questions center on *languages*. One might note that in our definition for a language, we made no requirements on L other than $L \subseteq \Sigma^\star$. Hence, L could be something extremely simple, such as

$$L_{\text{easy}} = \left\{ \sigma \in \Sigma \;\middle|\; |\sigma| \equiv_2 0 \right\},$$

or something fiendishly complex, like[1]

$$L_{\text{hard}} = \left\{ \sigma \;\middle|\; \sigma \text{ is a valid proof of Hartman–Grobman.} \right\}$$

We think of L_{easy} as having a much simpler "structure" when compared to L_{hard}. This is a byproduct of the differences in the predicates we've use to define our sets. In the first, the rule is very simple to state, and very simple to verify. We could imagine a program that determines membership in L_{easy} simply by scanning left-to-right and flipping a parity bit until reaching the end

[1] Assume that we have some agreed-upon encoding scheme by which we can interpret strings in $\Sigma^\star$ as proofs.

of the string. Then, it would simply checks the final value of the bit to determine whether the length of the string was even or odd. See Fig. 1 for an illustration.

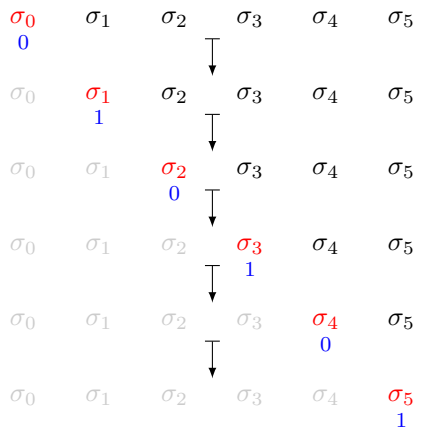

Figure 1: An illustration of the process. In each step, gray represents characters that've already been read, red symbols represent the current character, and blue symbol indicate the value of the parity bit. The final parity is odd, so we reject.

We note that a computer built to execute this algorithm could be very simple: it would only need enough memory to store the program and keep track of the parity bit, and this would be sufficient to determine membership in L_{easy} for arbitrary input.

By contrast, L_{hard} seems a lot harder. Since the input can be made arbitrarily long[2] while still remaining valid,[3] we need to be able to keep track of claims that are made arbitrarily far apart in the input string. E.g., suppose we're handed something like the following:

- Axioms: A_0, A_1, ..., A_n.
- Desired result: Z.
- Proof:
  - $1 + 1 = 2$
  - $1 + 2 = 3$
  - (A bunch of [other statements that are true but irrelevant] interspersed randomly with helpful ones, until finally)
  - $\ldots \implies Z$.

A priori, a verifier program has no knowledge of which claims will be relevant later to the proof. Hence, it needs to store all of the ones it encounters — even silly things like $1 + 1 = 2$. This makes it impossible to create an algorithm that looks exclusively at local data to verify whether a given string $\sigma \in L_{\text{hard}}$. Thus, we would need a more sophisticated computer to execute a verifier for L_{hard} than we did for L_{easy}.

This difference is made formal by the *Chomsky Hierarchy*, which gives an ordering to languages based on the complexity of the "computers" needed to identify their strings.[4] The situation is summarized by the following graphic:

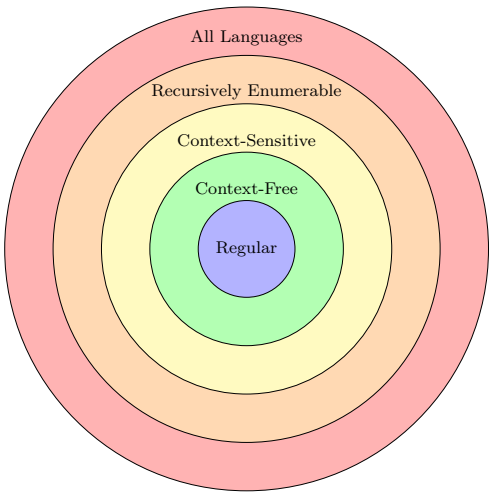

Figure 2: The Chomsky Hierarchy

L_{easy} is a *regular language*, which corresponds to one of the simplest possible models of computation (a *Deterministic Finite Automaton*). By contrast, L_{hard} requires a *Turing Machine*, one of the most powerful models of computation. All of these can be unified into a single conceptual framework by viewing them as *dynamical systems*. The reader might consider how as we begin with our formal definitions. But first, a quote from Prof. Ran:

> "There's this beautiful onion... and if you cut it open, it will make you cry."
>
> *-Prof. Ran, on the Chomsky Hierarchy*

We will strive to ensure that the reader experiences only tears of joy.

1.2. Deterministic Finite Automata

Deterministic finite automata correspond to the innermost ring of Fig. 2, the *regular languages*. These can be described formally in a number of ways; we'll choose the following:

Definition 1.5 (Regular Language)**.** Let Σ be an alphabet. We'll define the set of all *regular languages over* Σ (denoted $\mathcal{L}_{\text{reg}}(\Sigma)$) by an iterative process:

[2] See: this document
[3] See: maybe a different document
[4] Technically "computer" should be replaced with "computational model," but the distinction isn't important for us today

1) First, define $\varnothing$ and $\{\epsilon\}$ to be elements of $\mathcal{L}_{\text{reg}}(\Sigma)$.
2) Next: For all $\sigma \in \Sigma$, define $\{\sigma\}$ to be an element of $\mathcal{L}_{\text{reg}}(\Sigma)$.
3) Now, for all L_0, $L_1 \in \mathcal{L}_{\text{reg}}(\Sigma)$, define
   - $L_0 \cup L_1$,
   - $L_0 \cdot L_1 = \{\ell_0\ell_1 \mid \ell_0 \in L_0, \ell_1 \in L_1\}$, and
   - $L_0^\star$, $L_1^\star$

   to be elements of $\mathcal{L}_{\text{reg}}(\Sigma)$.
4) Finally, if L is yielded by a finite sequence of the rules above, then $L \in \mathcal{L}_{\text{reg}}(\Sigma)$. ◊

The idea with regular languages is that they are only slightly more complicated than Σ itself. Indeed, one might note the similarity between the axioms above and those for Definition 1.2 (strings). We'll give a few examples of regular languages before we introduce the associated model of computation. To that end, we'll first introduce a shorthand that will reappear many times throughout this paper.

Definition 1.6 (Exponent notation)**.** Let Σ be an alphabet. Then for all $\sigma \in \Sigma$, we interpret the notation σ^n by

$$\sigma^n = \underbrace{\sigma \cdot \sigma \cdots \sigma}_{n \text{ times}}$$ ◊

In light of the remark about m o n o i ds made earlier, this notation is actually quite reasonable.

Example 1.1. Let $\Sigma = \{0, 1\}$. Then L defined by

$$L = \{0^n 1^m \mid n, m \in \mathbb{N}\}$$

is a regular language. This follows from the fact that we can write L as $L = \{0\}^\star \cdot \{1\}^\star$.

Example 1.2. Let Σ be an arbitrary alphabet. Then all finite sets $L \subseteq \Sigma$ are regular. This follows from the fact that for any $\sigma \in L$, if σ decomposes as

$$\sigma = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k},$$

Then we can state this equivalently as

$$\{\sigma\} = \{\sigma_{i_1}\} \cdot \{\sigma_{i_2}\} \cdots \{\sigma_{i_k}\},$$

from which it follows that $\{\sigma\}$ is regular. Finally, this gives us that

$$L = \bigcup_{\sigma \in L} \{\sigma\}$$

is a regular language.

Example 1.3. Let $\Sigma = \{0, 1\}$. Then L given by $L = \{(01)^n \mid n \in \mathbb{N}\}$ is regular.

As stated earlier, *regular languages* are tied to *Deterministic Finite Automata*. These will lead to a more dynamical systems-esque view of regular languages.

Definition 1.7 (Deterministic Finite Automata)**.** A *deterministic finite automata* is a 5-tuple

$$M = (Q, F, \Sigma, \delta, q_0)$$

such that the following hold:

1) Q, Σ, F are finite sets. We choose the following naming conventions:
   i) Q is called the *set of states* for M,
   ii) F is called the set of *accepting* states for M, and
   iii) Σ is called the *input alphabet* for M;
2) δ is a function $\delta : Q \times \Sigma \to Q$ (we call δ the *transition function*), and
3) $q_0 \in Q$ is a distinguished element that we call the *start state*. ◊

We imagine M performing a computation as follows: Suppose we're given an input string $\sigma = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_k}$. M initializes in the *start state* q_0, and reads the first character. M then changes state to $q_{j_1} = \delta(q_0, \sigma_{i_1})$, and reads the next character σ_{i_2}. M then transitions to state $q_{j_2} = \delta(q_{j_1}, \sigma_{i_2})$, and so on. Once M has exhausted the input, it halts computation. If it halts in an *accepting state* (i.e. $q_{\text{final}} \in F$), then we say M accepts the input; else, we say M rejects it.

It's common to visualize DFAs through diagrams like the following:

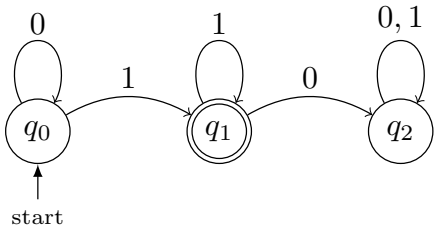

Figure 3: A DFA for Example 1.1.

The convention is that circles represent states of Q; the arrows represent the transitions given by $\delta(q, \sigma)$ (where σ is the label above the arrow), and double-circled states represent elements of F.

One can show that for every regular language L, there exists a DFA M_L such that M_L accepts a string σ iff $\sigma \in L$. We will not give a proof today since our interests are mainly in describing automata by dynamical systems. We now proceed in building this up formally.

1.3. Subshifts of finite type

We want to convert our DFAs into dynamical systems. To that end, we first convert our *alphabets* into structures we can interpret in terms of space. This will be helpful in making our theories "meet in the middle" in some sense: We want to be able to think about our models of computation as dynamical systems on some

phase space; later, we'll want to go the other way. In any case, the correspondence is useful.

To motivate our definitions, let's first think about what we expect things to look like from the dynamical viewpoint. Time is easy enough; we can just start at $t = 0$ and call each step of the computation $+1$ unit of time.

The phase space is a bit sneakier, but we'll manage. Recall that we want this to represent all possible *states* of our system in question. In the 5-tuple formalism (Definition 1.7), we defined *state* by elements of a finite set Q; however, this is insufficient information to fully quantify the behavior of our DFA at any given point of the computation — recall that δ depends not only on the "state" q_i, but also on the current character being read from the input. Hence we need our dynamical version of "state" to include both of these pieces of information. Hence, we might propose something like $Q \times \Sigma$ to be our state space. But there's another problem here.

Say we're given two strings s_0, s_1:

$$s_0 = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}$$
$$s_1 = \sigma_{j_1}\sigma_{j_2}\cdots\sigma_{j_\ell}.$$

Suppose now that $\sigma_{i_1} = \sigma_{j_1}$. We'll call this shared symbol σ_1. Observe that feeding either s_0 or s_1 into some DFA M yields identical behavior in the first step:

$$\delta(q_0, \underbrace{\sigma_{i_1}}) = \delta(q_0, \sigma_1) = \delta(q_0, \underbrace{\sigma_{j_1}})$$
$$\text{all equal}$$

Hence in both cases, the machine enters some new state q'. But now, suppose $\sigma_{i_2} \neq \sigma_{j_2}$, and further that $\delta(q', \sigma_{i_2}) \neq \delta(q', \sigma_{j_2})$. Then M will actually behave differently on the two strings. Hence we see that taking our states to be elements of $Q \times \Sigma$ also fails to give us a full characterization of subsequent behavior.

What we really want is a way to encode the *entire* unread portion of the string into a state variable. We do exactly that with *configuartions* (note, this term is nonstandard). We will first define *unrestricted configurations*, which we'll use later in our definition of Turing machines[5] We'll then define *DFA configurations* as a particular restricted subset of these unrestricted configurations.

Definition 1.8 (Unrestricted Configuration)**.** Given a DFA with states Q and input alphabet Σ, define the space of *unrestricted configurations over* Q, Σ to be

$$X = Q \times \left(\prod_{i=1}^{\infty} \Sigma \cup \{\epsilon\} \right),$$

where the $\prod$ is understood as a cartesian product. ◊

Definition 1.9 (DFA Configuration)**.** Let $X_{\text{DFA}} \subseteq X$ be defined by

$$X_{\text{DFA}} = \left\{ (q, \sigma) \in X \;\middle|\; \sigma = \lceil \sigma_1\sigma_2\cdots\sigma_n \rfloor \cdot \epsilon^\infty \right\}.$$

That is, pairs of states q and finite strings σ over Σ. ◊

This "configuration space" turns out to be the correct choice of phase space. We now seek to define a *topology* on X, which will in turn induce a topology on X_{DFA}. This will give us the necessary framework by which to translate back and forth between dynamical systems and computability later on. First, recall the definition of a topology:

To avoid confusion we'll refer to q_i as the *machine state* from now on.

1.4. Turing Machines

A *Turing machine* (often TM for short) is an abstract model of computation that allows us to encode computer programs as collections of sets with well-defined maps. There are many "equivalent" definitions for Turing machines;[6] we'll give the canonical one first and save the more dynamical-systems-flavored version for later.

Definition 1.10 (Turing Machine)**.** A *Turing machine* is a 5-tuple

$$M = (Q, F, \Gamma, \Sigma, \delta)$$

such that the following conditions hold:

1) Q, F, Γ, Σ and F are finite sets. Note, we choose the following naming conventions:
   i) Q is called the set of *states* for M,
   ii) $F = \{\checkmark, \text{✗}\}$ is called the set of *final states* (read "accept" and "reject" respectively),
   iii) Γ is called the *work alphabet*, and
   iv) Σ is called the *input alphabet*;
2) $F \subseteq Q$ and $\Sigma \subseteq \Gamma$;
3) Q contains a distinguished element q_0, called the *start state* or *initial state*;
4) Γ contains a distinguished element ␣ called the *blank character* (we require $\text{␣} \notin \Sigma$);
5) δ (called the *transition function*) has the following properties:
   i) δ is a partial function $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$. Note the inclusion of the word *partial* — in general we do not require δ to be defined on all of $(Q \setminus F) \times \Gamma$.
   ii) L, R are understood as "shift" directions, as explained below. ◊

[5] We'll repeat the definition there when we need it, so don't worry about memorizing this one.

[6] Here, "equivalent" means that the models of computation they define are equally powerful. That is, given two distinct definitions D_0, D_1 for Turing machines, the behavior of any machine defined with D_0 can be simulated using a machine defined by D_1 and vice versa.

We think of Turing machines as operating on an infinite *work tape*, where the cells of the tape each represent a symbol from the alphabet Γ.[7] One can think of this tape as representing the "memory" of the TM.

The tape is initialized with a finite word $\sigma = \sigma_1\sigma_2\cdots\sigma_n$ representing the input to our program, while the rest of the tape is filled with $\sqcup$ characters. The TM initializes in the *start state*, with the *tape head* (drawn with an $\uparrow$ in the below) on σ_1. We think of the tape head as "pointing" to whatever character the TM is reading currently.

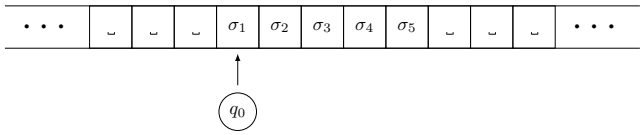

Figure 4: Turing machine initialized with the input string $\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$.

The "program" is then executed by the δ function. For instance, suppose $\delta(q_0, \sigma_1) = (q_i, \gamma_1, R)$. Then we interpret this as the Turing machine writing a γ_1 on the current tape, updating its state to q_1, and moving one space to the right on the tape.

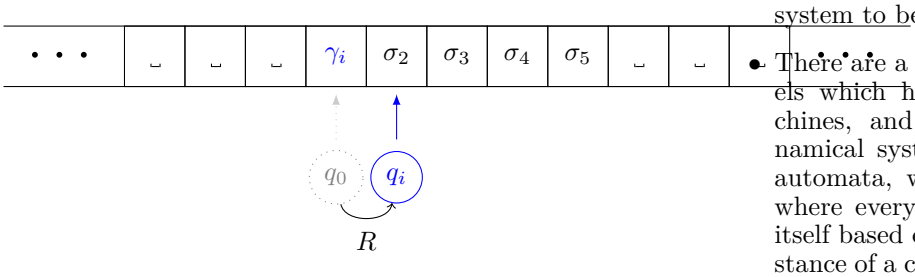

Figure 5: Turing machine after performing a single step. Differences shown in blue.

At each step of the computation, the Turing machine performs a similar process: Read the tape, rewrite and move as dictated by δ, and then repeat. If the Turing machine reaches the state ✓, then it halts and the input is said to be "accepted." If it reaches ✗, the state is "rejected." In the case that the machine never halts, we treat this as a form of rejection.

2. Computational Universality

A Universal Turing Machine is a Turing machine which can simulate any arbitrary Turing Machine. It takes as its Turing Machine code, and an input word, and then simulates the Turing Machine on that input. Intuitively, any modern computer is a (finite memory) Universal Turing Machine: it can store arbitrary computer programs as data, and then run them on other data it stores or with user input.

In a similar vein, a system is *computationally universal* (or just *universal*) if it has the ability to simulate an arbitrary Turing machine. This is a loose definition because it is not immediately clear what it means for a dynamical system to simulate a Turing machine. There are many possible ways to create a more precise definition; we will follow the method of Delvenne et al. [CITATION]. Before we do, though, we note a few reasonable expectations about what simulating a Turing machine should mean:

- If a dynamical system can simulate an arbitrary Turing machine, we should be able to take any combination of Turing machine and input, and convert it into some question about the system. The question should be "equivalent" to the question of whether the Turing machine would accept its input, in the sense that it should have a "yes" answer if the Turing machine accepts, and a "no" answer if the Turing machine rejects or never halts.
- The Universal Turing Machine is a specific Turing Machine that exists, and therefore it is a dynamical system on the space of its possible configurations. We would definitely expect this dynamical system to be considered universal.
- There are a number of other computational models which have ability to simulate Turing Machines, and which can also be considered dynamical systems. One good example is cellular automata, which are like Turing machine tapes where every cell has its own state and updates itself based on its neighbors' states. If a given instance of a computation model is capable of simulating arbitrary Turing machines, we would expect its associated dynamical system to be universal.

Formally defining universality

We will follow the lead of Delvenne et al. [CITATION] in defining what it means for a dynamical system to be universal. Like them, we will consider only *symbolic* dynamical systems, that is, dynamical systems on spaces consisting of infinite words made out of symbols drawn from a finite alphabet. We will need to do some spadework before we're ready to give the full definition.

Spadework Part I - R.E.-completeness

Our ultimate goal is, given the code for a Turing machine M, and the input w, to ask a question about a particular dynamical system which has a "yes" answer exactly when M accepts w. In the terms used in the field of computability, we want to be able to reduce an arbitrary instance of the question "Will Turing machine

[7] Formally, the addition of the tape requires using an infinite tuple of elements from $\prod_{i=-\infty}^{\infty}\Gamma$, with all but finitely-many entries non-blank.

M accept the input w?" to a question about the dynamical system.

Fortunately, much is already known about the question "Will Turing machine M accept the input w?". In particular, it is equivalent to the question "Will Turing machine M halt on the input w?" in the sense that if we had a way to answer either of the questions, we could easily answer the other as well. This second question is called the Halting Problem, and computer scientists say that it is "complete for the class of recursively enumerable problems."

So, depending on your background, you may be asking two questions: "What is the class of recursively enumberable problems?" and "What does it mean to be complete for it?" A problem is called recursively enumerable if it can be solved by a Turing machine that may not ever halt, that is, if the question has a "yes" answer, the Turing machine will accept in finite time, but if the question has a "no" answer, the machine may run indefinitely. They are called "recursively enumerable" because it would be possible for a Turing machine (a computation model capable of *recursion*) to *enumerate* every input that would have a "yes" answer, in such a way that every such input would eventually be listed in finite time. (The machine could do this by listing all inputs lexicographically, and spending half its time simulating the computation on the first input, a quarter of its time on the second input, and so on, printing out any input when it is accepted.)

A given problem, call it P, is complete for a class of problems, C, if two conditions hold. First, P must be an element of C. Second, all problems in C must be reducible to P, meaning that if $P' \in C$, we can convert any instance of P' to an instance of P that will have the same answer, without doing an unreasonable amount of work in the conversion (for our purposes here, any finite computation time is reasonable). The Halting Problem is known to be complete for the recursively enumerable problems, or r.e.-complete.

The question "Will Turing machine M accept input w?" (call it Q) is also r.e.-complete. Recall that our goal was to find a question QDS about a dynamical system, which we could reduce Q to. One way to express this constraint is that QDS should be r.e.-complete: then both QDS and Q could be reduced to each other, so they would be equivalent questions, which is what we are looking for. In fact, we will see that the r.e.-completeness of QDS is exactly what we will require.

Spadework Part II - Effective Symbolic Systems

A symbolic set can be thought of as a set of words made from a finite alphabet A. We could express such a set as $(A \cup \{B\})^{\mathbf{n}}$, meaning the set of one-sided infinite words with characters which are either drawn from A or are the "blank" symbol B. Finite words would then end be expressed as infinite words ending with an infinite tail of Bs. But we could encode each element of $A \cup \{B\}$ with a binary sequence (with all encodings the same length), so any symbolic set can be encoded as a subset of the set $\{0,1\}^{\mathbf{n}}$ of one-sided infinite binary words.

We give $\{0,1\}^{\mathbf{n}}$ the following distance metric d: $d(x,y) = 0$ if x and y agree at all indices, and otherwise $d(x,y) = 2^{-n}$ where n is the smallest index at which x and y differ. Under this metric, the set of all sequences beginning with a finite binary word w is a both a closed and open set (a "clopen" set) under this metric. Call sets like this, generated by a common prefix, cylinders. It can be shown that the clopen sets of $\{0,1\}^{\mathbf{n}}$ are precisely the finite unions of cylinders. What is special about this is that we can associate with each clopen subset of $\{0,1\}^{\mathbf{n}}$ a unique finite set of finite words, which are the generating prefixes for the cylinders that make up that set. So we can express every clopen set of $\{0,1\}^{\mathbf{n}}$ in a finite way, which means the we can list off all the clopen sets in some lexicographic order (in other words, the set is countable). We'll be exploiting this fact later.

Now we're ready to define the main dynamical systems we'll be working with. If $X \subseteq \{0,1\}^{\mathbf{n}}$ is a symbolic set and $f : X \to X$ is a continuous map on X, then (X, f) is an *effective symbolic system* if:

(i) X is closed.

(ii) Checking whether some clopen set $Y \subseteq \{0,1\}^{\mathbf{n}}$ has a non-zero intersection with X is decidable by a Turing machine in finite time.

(iii) The inverse map f^{-1} on clopen sets of X can be computed in finite time.

These requirements may seem a bit arbitrary, but they are made for a good reason. We will shortly be performing some operations on the clopen sets of X, and it turns out that, since X is closed, these are exactly the intersections of X with the clopen sets of $\{0,1\}^{\mathbf{n}}$, so we know this is a countable set. Requirement (ii) helps ensure that a Turing machine could list out these clopen sets without ever getting stuck checking if a particular clopen set of $\{0,1\}^{\mathbf{n}}$ has a nonempty intersection with X. And requirement (iii) foreshadows operations we will be doing on clopen sets.

Spadework Part III - Temporal Logic

Recall that we are looking for a way to "ask questions" about dynamical systems, whose answers will tell us something about the behavior of Turing machines. In order to be able to construct such questions, we will use subsets of state space to encode logical statements. Since we want to be able to make statements about the *eventual* behavior of the system, we will use a logical framework called temporal logic. Temporal logic includes all of the standard logical operators:

- $\top$: always true
- $\bot$: always false
- $\vee$: or

- $\neg$: not
- $\wedge$: and (which is actually redundant because it is constructible from $\vee$ and $\neg$)

It adds two additional operators to the list:

- $\circ$: next, a unary operator, with $\circ\phi$ meaning that ϕ will be true after one time step.
- $\mathcal{U}$: until, a binary operator, with $\phi\mathcal{U}\psi$ meaning that ψ will be true within finite time, and for all time between the present and one step before the time when ψ is true (inclusive), ϕ will be true

The developers of temporal logic, Arthur Prior and Hans Kamp, have described a set of rules for incorporating these operators into classical logic in a way that is consistent with our interpretation of them [CITATION]. For our purposes, suffice it to say that it works.

How do we propose to use temporal logic to construct statements about effective symbolic systems? Well, as we have suggested above, we are interested in operating on the clopen sets of symbolic systems, and here is where that will come into play. Let's say we have some effective dynamical system (X, f). We will use the fact that the clopen subsets of X are countable, and let $P_0, P_1, P_2, ...$ be a listing of all the clopen subsets of X (including $\emptyset$ and X in some positions).

Then let $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, ...$ be a set of propositional symbols, which are the logical "units" that along with the operators above are the building blocks of temporal logic formulas. The listing will contain $\top$ and $\bot$ in some positions. We're going to define an "interpretation" operator $|\cdot|$ that takes logical formulas to subsets of X. The intuition behind it will be that if ϕ is a logical formula that expresses something about the "current configuration of the system," $|\phi|$ is the set of possible system configurations (points of X) for which that statement is true. Formally, the operator will behave like this:

- If ϕ is just the symbol $\mathcal{P}_n$, $|\phi| = P_n$, and we stipulate that the orderings should align such that $|\top| = X$ and $|\bot| = \emptyset$. This means that the symbol $\mathcal{P}_n$ represents the statement "The current system configuration is in the set P_n," which is of course always true if $P_n = X$ and never true if $P_n = \emptyset$.
- $|\phi_1 \vee \phi_2| = |\phi_1| \cup |\phi_2|$, because for either ϕ_1 or ϕ_2 to be true, the system can be in any state in which either is true.
- $|\neg\phi| = X \setminus |\phi|$, because the set of states for which ϕ is not true is the complement of the set of states for which it is.
- $|\circ\phi| = f^{-1}(|\phi|)$, which means that $\circ\phi$ represents the statement, ϕ will be true after one application of f, meaning that each application of f corresponds to one "time step" within the temporal logic system. Note that we know that this is computable, by requirement (iii) of effective symbolic systems.
- $|\phi_1 \mathcal{U} \phi_2| = \bigcup_{n \in \mathbf{n}} A_n$ with $A_0 = |\phi_2|$ and the other sets defined by the recurrence relation $A_{n+1} = f^{-1}(A_n) \cap |\phi_1|$. Here A_n can be interpreted as, the set of system states for which ϕ_2 will be true on the n^{th} time step from the current state, and ϕ_1 will be true on the $0, ...n-1$ time steps. The union of all of these represents all states for which "ϕ_1 is true until ϕ_2 is eventually true."

  So, in short, we have used temporal logic to construct a set of formula that express statements about the state of the system. We say a given formula is satisfiable if its interpretation is nonempty, meaning that there is some nonempty set of X that satisfies it.

UNIVERSALITY

Our spadework concluded, we're ready to say what it means for an effective dynamical system to be universal. We will use Delvenne et. al's precise definition here:

"*An effective dynamical system is* universal *if there is a recursive family of temporal formulae such that knowing whether a given formula of the family is satisfiable is an r.e.-complete problem.*"

A "recursive" family of temporal formulae is a set of temporal formula for which membership in the set can be decided by a turing machine in finite time; this is essentially a regularity condition which prevents the family from being outlandishly defined.

To get a feel for what this definition means, and ensure that it is reasonable, let's start by confirming that the Universal Turing Machine dynamical system is, as we expected from the start, universal.

A configuration of the Universal Turing Machine, as with all Turing machines, is defined by finite control state and its tape contents, both of which can be encoded in a binary strings. So the configuration space of a UTM is a subset X of $\{0,1\}^{\mathbf{n}}$, and the code for the UTM is some specific transition function on this space, f.

3. APPENDIX

Here, we include some of the background definitions the reader might not be acquainted with. First, we define a *topology.*

Definition 3.1 (Topology)**.** Let X be an arbitrary set. Let $\mathcal{T}$ be a collection of $U \subseteq X$ such that

1) (Containment of trivial elements): $\varnothing$, $X \in \mathcal{T}$,
2) (Closure under arbitrary unions): For arbitrary collections $\{U_i\}_{i \in I} \subseteq \mathcal{T}$, we have

$$\left(\bigcup_{i \in I} U_i \right) \in \mathcal{T},$$

and

3) (Closure under finite intersection): For finite collections $\{U_i\}_{i=1}^n$, we have

$$\left(\bigcap_{i=1}^{n} U_i\right) \in \mathcal{T}. \qquad \Diamond$$

Then we call $\mathcal{T}$ a *topology* on X.

The definition of a *topology* is meant to axiomatize the properties of *open sets* that we encounter in analysis. Indeed, one might note that in any metric space X, (1) both $\varnothing$ and X are open sets, (2) open sets are closed under arbitrary union, and (3) open sets are closed under finite intersection.

However, one should note that this is a *proper* generalization, in the sense that there exist topologies which cannot arise from a metric. For instance, consider the following:

Definition 3.2 (Double-headed snake)**.** Let $X = \{0_A, 0_B\} \cup (0, 1]$. Here, the labels A and B are just to distinguish the two "copies" of 0 that we've made. Define a topology $\mathcal{T}$ on X as follows:

1) Let $U \subseteq (0, 1]$. Then if U is open in $[0, 1]$ viewed as a metric space, we have $U \in \mathcal{T}$.
2) Similarly: Given such a U, we have $\{0_A\} \cup U$, $\{0_B\} \cup U \in \mathcal{T}$. $\Diamond$

This effectively gives us a version of $[0, 1]$ where we have two copies of 0. One can verify that the topology axioms are satisfied

So, why isn't this topology metrizable? In a nutshell, the problem is that 0_A, 0_B act as if they are distance 0 apart from each other, which breaks the "$d(x, y) = 0 \iff x = y$" axiom for metric spaces.[8]

This underscores that topology is generally a more abstract way of looking at properties of space. There are a number of situations in which this abstraction is desired. First, it helps us to focus on more "abstract" properties of space that are invariant under stretching and deforming (but not gluing or tearing) — e.g., in a topological context, the two shapes shown in Fig. 6 are the "same."

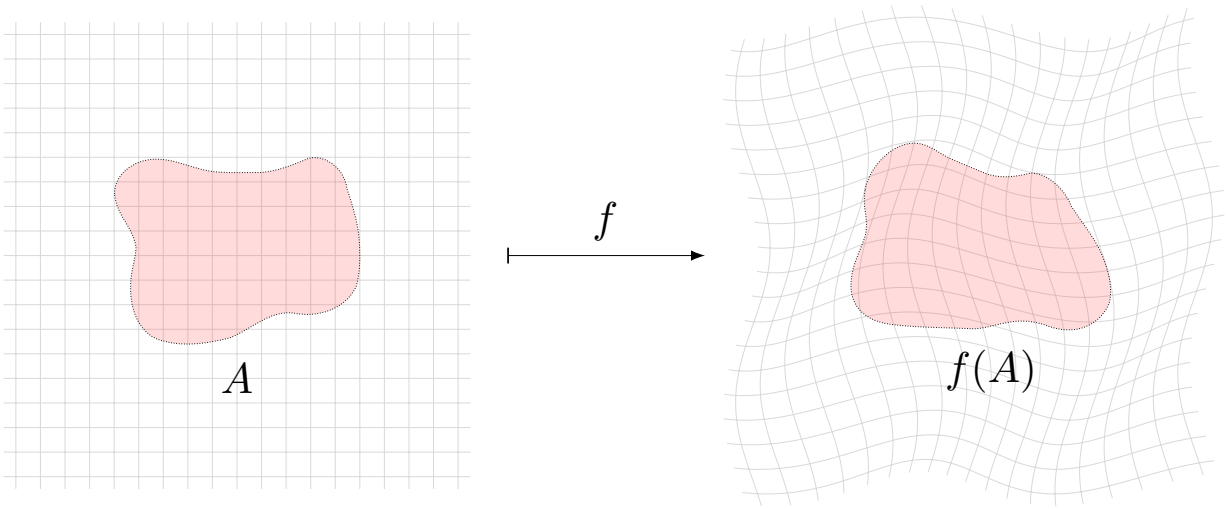

Figure 6: Two shapes that are topologically equivalent

[8] A more rigorous argument can be constructed by noting that $\forall \varepsilon > 0$, $\{0_A\} \cup (0, \varepsilon)$ and $\{0_B\} \cup (0, \varepsilon)$ are open and then doing something like $0_A = \lim_{\varepsilon \to 0} \varepsilon = 0_B$, a contradiction.

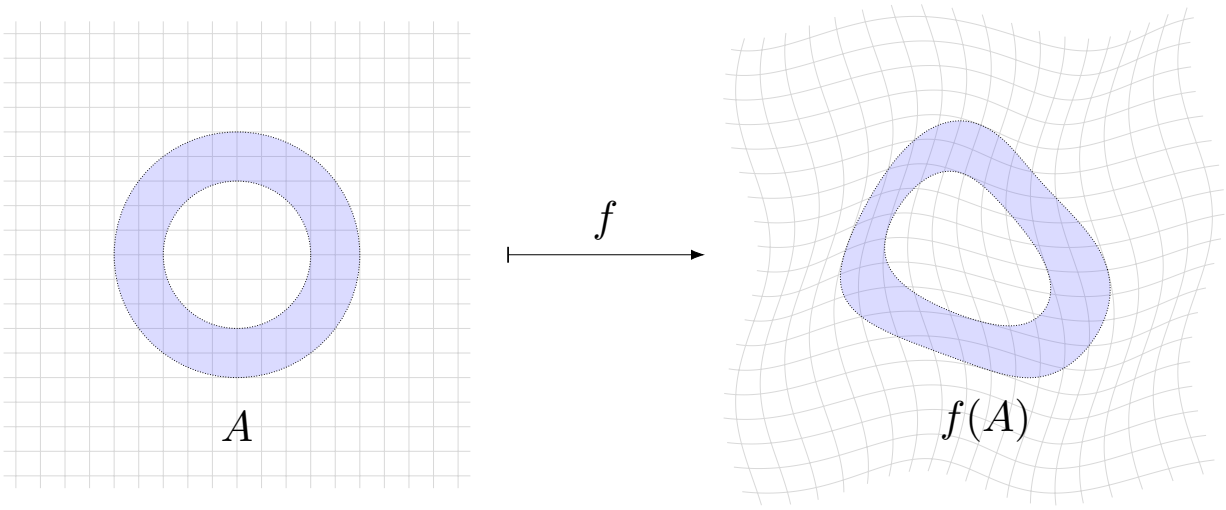

Figure 7: Two shapes that are topologically equivalent to each other, but not to those in Fig. 6

Often, these are the kinds of properties we're interested in when we try to find ways of thinking of random sets X that we've just scooped out of a bush on the side of the street in terms of "spaces."

is there more to say here?

Definition 3.3 (Discrete Topology)**.** Let X be a set. Then we define the *discrete topology* on X to be the powerset:

$$\mathcal{T}_{\text{discrete}} = \{U \mid U \subseteq X\}.$$

◊

In the discrete topology, every set is open.