Turing Machines and Computability Theory

Luke Mathieson

October 1, 2020

1 Capturing Computation – The Turing Machine

We have now seen two levels of what is know as the Chomsky hierarchy, Type 3 Grammars (regular languages) and Type 2 Grammars (context free languages). We now move directly to the top of the hierarchy: Type 0 Grammars[1].

This level of grammar, the unrestricted grammar, coincides with another, more famous, model of computation: the Turing Machine. Turing Machines (though not called so by Turing of course), were invented by Alan M. Turing as part of a solution to the Entscheidungsproblem. The Entscheidungsproblem was posed by David Hilbert as part of his program to address the "foundational crisis of mathematics", brought to a head by Russell's Paradox and the like. The Entscheidungsproblem can be phrased as follows (in a relatively modern vocabulary):

> Is there an algorithm that takes as input a statement in first-order logic and a finite number of axioms and answer yes or no as to whether the statement is derivable from the axioms?

To answer this question, a definition of algorithm has to be rigorously stated. While the informal idea of an algorithm had been well understood for some time[2], a formal definition remained wanting. Two definitions were given in rapid succession; Alonzo Church developed the λ-calculus in 1936 [**?**] representing the idea of an algorithm through *effective calculability*, and Alan Turing through the *Turing Machine* [**?**, **?**].

Both independently gave negative answers to Hilbert's question; Church showed that there is no computable function that decides whether two λ-calculus expressions are the same, Turing showed that there is no Turing Machine that can decide whether another Turing Machine will halt or not (and also showed that this is equivalent to the general statement of the Entscheidungsproblem above).

Of course this depends on Church and Turing's definitions of an algorithm being sufficient, something we don't strictly know to be true. This assumption is phrased as the Church-Turing Thesis:

> Every algorithm is computable by a Turing Machine[3].

The philosophical implications of this thesis are interesting, as is the debate over whether it is correct. For our purposes though, we shall use it in a much more practical manner, and take Turing Machines to be the definition of algorithms, and hence computation.

1.1 The Formalities

A Turing Machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$ where

- Q is the finite set of states,
- Σ is the finite input alphabet and does not include the *blank symbol* $\sqcup$,
- Γ is the finite tape alphabet with $\Sigma \subset \Gamma$ and $\sqcup \in \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
- $q_{start} \in Q$ is the start state,
- $q_{accept} \in Q$ is the accept state, and
- $q_{reject} \in Q$ is the reject state with $q_{accept} \neq q_{reject}$.

[1] Chomsky, though not a computer scientist, had the decency to index from 0.

[2] It may be reasonably summarised as "what a human can do with some paper, a pencil and a set of rules to follow".

[3] This is far from the original wording, and indeed the evolution of the thesis over time is an interesting tangent in its own right.

The Turing Machine is equipped with a tape, which is infinitely long to the right[4], and a read/write head. The head begins at the first cell of the tape, on which the input has been written. The machine then applies the transition function by reading a symbol, writing a symbol, changing state, and moving the head left or right. The head cannot move off the left hand end of the tape.

1.1.1 Exercises

Construct a Turing Machine for the following languages:

1. $\{0^n 1^n\}$.
2. $\{a^n b^n c^n\}$.

2 Decidability, Recognizability and Undecidability

Up to now, we have been dealing with machines that always process the entire input, and only halt at that point. It is clear that Turing Machines do not have this constraint. For example a Turing Machine that accepts all strings (i.e the language Σ^*) is simply a machine that enters q_{accept} regardless of its input. Similarly a Turing Machine that accepts no strings may simply halt and enter q_{reject} immediately. It is clear that we need slightly more nuance in our language at this point.

Definition 2.1 (Recognizable Language)**.** *A formal language L is* recognizable *if and only if there is a Turing Machine M, that when presented with any $\omega \in L$ as input, halts and accepts, and when presented with $\omega' \notin L$ as input, does not accept.*

Note the asymmetry in this definition; when faced with a string not in the language, M is not even required to halt. It may simply loop infinitely. The set of recognizable languages has many names, thanks to the plethora of competing models of computation: recursively enumerable, semi-decidable, partially decidable, Turing-acceptable and Turing-recognizable. It is also the first level of the Chomsky hierarchy mentioned before, corresponding to unrestricted grammars.

Of course we typically don't want to talk about programs that only stop sometimes, thus we need a slightly smaller class of languages.

Definition 2.2 (Decidable Language)**.** *A formal language L is* recognizable *if and only if there is a Turing Machine M, that when presented with any $\omega \in L$ as input, halts and accepts, and when presented with $\omega' \notin L$ as input, halts and rejects.*

Decidable languages (also called recursive languages) are a subset of the recognizable languages: the definition of recognizability doesn't *prevent* the machine from halting when the string is not in the language, it just allows it not to. It should also be clear that regular languages and context free languages are also decidable languages.

This leads to several questions. What languages are decidable? Are there any recognizable languages that are not also decidable?

2.1 Some Decidable Languages

To demonstrate that a language is decidable, it suffices to give a program that decides it. One nice property of Turing Machines is that they are powerful enough that we can be somewhat more relaxed about specifying them. Although formally the machines have states and transitions, it becomes quickly apparent that they are powerful enough to express anything written in any practical programming language, so if we can demonstrate a program in that context, we can be satisfied that a Turing Machine can also be specified.

Using this, we can show the following:

Lemma 2.3. *$EQ_{DFA} = \{\langle D, E\rangle \mid L(D) = L(E)$ and D and E are DFAs$\}$ is decidable.*

Lemma 2.4. *E_{DFA} is decidable.*

Lemma 2.5. *ALL_{DFA} is decidable.*

Lemma 2.6. *A_{DFA} is decidable.*

Lemma 2.7. *A_{CFG} is decidable.*

Lemma 2.8. *E_{CFG} is decidable.*

Lemma 2.9. *A_{PDA} is decidable.*

[4]Being finite to the left is not strictly necessary, but simplifies things to start with.

2.2 An Undecidable Language

Of course it would be surprising if everything were decidable. In fact the existence of languages is given by a relatively straight forward counting argument.

Lemma 2.10. *There are languages which are undecidable.*

Proof. We may assume that our inputs are encoding in binary. Moreover, we can represent any Turing Machine by a binary string (if you are unconvinced, imagine drawing a transition diagram with sufficient detail on a computer, this would be a representation of the Turing Machine, and clearly represented in binary).

A language then is just a subset of the set of all possible binary strings.

Then the set of all Turing Machines has cardinality at most that of the set of all binary strings. The set of all languages however has cardinality of the powerset of the set of all binary strings, which is strictly greater than the number of binary strings. Thus there are some languages which no Turing Machine can decide. □

Corollary 2.11. *There are languages which are unrecognizable.*

Of course this doesn't help us do anything particularly concrete. We would like to know *which* languages are undecidable. To do this will take a bit more effort.

Theorem 2.12. $A_{TM} = \{\langle M, \omega\rangle \mid M \text{ is a Turing Machine that accepts } \omega\}$ *is undecidable.*

Proof. Assume for contradiction that A_{TM} is decidable, then there is a Turing Machine T that decides it:

$$T(\langle M, \omega\rangle) = \begin{cases} accept & \text{if } M \text{ accepts } \omega \\ reject & \text{otherwise} \end{cases}$$

Consider the Turing Machine D that uses T as a subroutine:

$D =$ "On input $\langle R\rangle$ where R is a Turing Machine:

1. Run T on input $\langle R, \langle R\rangle\rangle$.
2. If T accepts, reject.
3. If T rejects, accept.

Consider now what happens if D is given $\langle D\rangle$ as input. It will accept if D rejects $\langle D\rangle$, and reject if D accepts $\langle D\rangle$. This is clearly a contradiction, thus the original assumption is false, and A_{TM} is undecidable. □

2.3 Finding More Undecidable Languages

The proof of undecidability of A_{TM} is amenable to a fairly straightforward diagonalisation argument. It would be surprising if every undecidable language was so easy to deal with. To established the undecidability of other languages, we use the technique of Turing reduction.

A Turing reduction from language A to B (also said "A is (Turing-)reducible to B", written $A \leq_T B$) is a mapping that shows that if B were decidable, A would be too. To put it another way, a reduction is a algorithm that decides A using an algorithm for deciding B as a subroutine. Thus if we have an undecidable problem, and show that it is reducible to another problem (i.e. we could use the second problem to decide the first), the second problem must also be undecidable.

Theorem 2.13 (The Halting Problem)**.** $HALT_{TM} = \{\langle M, \omega\rangle \mid$ *Turing Machine* M *halts on input* $\omega\}$ *is undecidable.*

Proof. We will show that $A_{TM} \leq_T HALT_{TM}$, and thus that $HALT_{TM}$ is undecidable (if not, A_{TM} would also be decidable).

Assume that $HALT_{TM}$ is decidable. Then there is a Turing Machine M that decides it. Construct the following Turing Machine R that decides A_{TM}:

$R =$ "On input $\langle M, \omega\rangle$, where M is a Turing Machines:

1. Run M on $\langle M, \omega\rangle$.
2. If M rejects, reject.
3. Run M on input ω.
4. If M accepts, accept.
5. If M rejects, reject.

Thus as this machine decides A_{TM}, which is undecidable by Theorem 2.12, the original assumption gives a contradiction, and thus $HALT_{TM}$ is undecidable. □

2.3.1 Exercises

Determine whether the following languages are decidable or undecidable. Either give a decider, or a reduction proving undecidability.

1. $INF_{DFA} = \{\langle D \rangle \mid D \text{ is a DFA with an infinite language}\}$.
2. $E_{TM} = \{\langle M \rangle \mid M \text{ is a Turing Machine that accepts no strings}\}$.
3. $SUBS_{DFA} = \{\langle D, E \rangle | D \text{ and } E \text{ are DFAs and } LD \subseteq L(E)\}$.
4. $EQ_{TM} = \{\langle M, N \rangle \mid M \text{ and } N \text{ are Turing Machines and } LM = L(N)\}$.
5. $PAL_{DFA} = \{\langle D \rangle \mid D \text{ is a DFA that accepts at least one palindromic string}\}$. You may need the fact that the intersection of a regular language and a context free language is context free.

2.4 Rice's Theorem

For a large class of languages, the undecidability of the language is somewhat divorced from the language and is, to a large extent, a property of the fact that the question we're asking is along the lines of "does this Turing Machine compute such and such a property", and thus it is our inability to analyse Turing Machines that undoes us, rather than the property itself. Formalising this, we obtain a very useful theorem, Rice's Theorem, that allows quick classification of large swathes of similar problems.

First we need some definitions:

Definition 2.14 (Trivial Property)**.** *A [language] property is* trivial *if it is true for [the languages] of all Turing Machines, or false for the languages of all Turing Machines.*

Definition 2.15 (Semantic Property)**.** *A [language] property is* semantic *if, given Turing Machines M and N, where $L(M) = L(N)$, the property holds for $L(M)$ iff it holds for $L(N)$.*

We can now state Rice's Theorem:

Theorem 2.16 (Rice's Theorem [**?**])**.** *Any non-trivial, semantic property of Turing Machines is undecidable.*

Proof. Let P be some property we want to decide, and let M_P be a Turing Machine that satisfies that property, then we can construct the following parameterized machine:

$C_{M_P,M,\omega}$ = "On input x:

1. Run M on ω *until* it accepts. If it rejects, loop forever.
2. If M accepts ω, run M_P on x and accept iff M_P does."

Assume that P is decidable by machine B. The following machine decides A_{TM}, and thus gives a contradiction:

R = "On input $\langle M, \omega \rangle$ where M is a TM and ω is a string:

1. Construct a TM M_P with property P.
2. Use M_P, M and ω to construct $C_{M_P,M,\omega}$.
3. Run B on $\langle C_{M_P,M,\omega} \rangle$ and ACCEPT if it accepts, REJECT if it rejects.

□

With this we reduce the undecidability proofs for many languages to proofs that the properties are non-trivial and semantic:

Lemma 2.17. *E_{TM} is undecidable.*

Proof. First we demonstrate that the property of have an empty language is non-trivial:

1. Let T_{all} be a Turing Machine where the start state is the accept state. This machine accepts on all inputs, and thus its language is Σ^*, and not empty.
2. Let T_{none} be a Turing Machine were the state state is the reject state. This machine rejects on all inputs, and thus its language is $\emptyset$, which is empty.

As $\langle T_{all} \rangle \in E_{TM}$ and $\langle T_{none} \rangle \in E_{TM}$, the property of having an empty language is non-trivial.
It is also clear that the property of having an empty language is vacuously semantic.
Thus, by Theorem 2.16, E_{TM} is undecidable. □