2.8 Universal Turing Machines and the Halting Problem

Through the hierarchy of Slide 74 we indirectly get a sense that Turing Machines are at least as computationally powerful as any other known model of computation. Thus, the limit of what can be computed can effectively be taken as the limits of what can be computed by a Turing Machine. One way how to delineate this limit is to show when this limit is exceeded, by showing that there are certain problems that a Turing Machine cannot solve.

Thm. 50 has already delineated one of the limits of computability. In an indirect way, i.e., by contradiction, we have shown how there must exist problems that are not computable by a Turing Machine. More precisely, there a membership predicates for certain languages for which we cannot construct an algorithmic solution. Such limit concerns Turing Machine acceptability.

In what follows, we shall be concerned more with the limits of Turing Machine decidability, arguably, a more desirable property. Let us first take a step back to take stock of what we know so far. From Thm. 50 we know that there problems which are not Turing acceptable and, by virtue of Thm. 23, we therefore also know that there problems that are not Turing Decidable (cf. Slide 74).

But the language hierarchy of Slide 74 is still incomplete as it leaves out an important question regarding decidability, namely the question of whether there are languages that are recognisable by a Turing Machine but are not decidable (the class of Turing Acceptable/Recognisable languages is sometime also referred to as semi-decidable in the literature, which is more suggestive in this case). Showing that there are no such cases of semi-decidable languages that are not decidable would effectively prove that $L_{RE} \subseteq L_{Rec}$, which, coupled with $L_{Rec} \subseteq L_{RE}$ of Thm. 23 would equate the two language classes and collapse them into one class, i.e., $L_{Rec} = \overline{L_{RE}} (= L_{PSC})$. On the other hand, showing that there exist semi-decidable languages that are not decidable would fill in the missing relationship on Slide 74 with a strict inclusion $L_{Rec} \subset L_{RE}$. It turns out that the latter case holds true, and the question leading to such a fact is often referred to as the Halting Problem.

Answering the Halting Problem is important irrespective of the theoretical implications of Slide 74. For instance, it has implications on the termination properties of the software we write and use. Computer Scientists (and programmers in the large) use software that they expect to behave in a decidable manner. Consider for example, the compiler for a particular language; we would expect that, irrespective of the source code it is asked to compile, the execution of such software to always terminates, either successfully (with the compiled code) or else unsuccessfully, with an indication of where the error in the source code lies.

For this course, the appeal of this particular problem is that it can be entirely formulated in terms of Turing Machines. For this formulation we will however first need to introduce the notion of a Universal Turing Machines i.e., a class of Turing Machines that are programmable. Intuitively a Universal Turing Machine can be programmed by providing it with a textual description of another Turing Machine i.e., a program. The Universal Turing Machine then executes this program by simulating the behaviour of the Turing Machine that the textual representation denotes. The Halting problem we shall consider can therefore be cast as on Slide 75.

---

**The Halting Problem (Informally)**

Given the textual representation of a Turing Machine, i.e., a program, is there a (universal) Turing Machine that can determine whether the program Turing Machine will halt on all inputs?
2.8.1 Universal Turing Machines

So far we have developed a different Turing Machine for each algorithm. This gives the impression that Turing Machines are more like algorithms than computers (which are able to execute different algorithms on the same setup.) More concretely, the main distinction between an algorithm and a computer is that the latter is programmable; it can take a description of an algorithm and an input and can run the algorithm on the input. As we shall see, Turing Machines can also be assigned this functionality; we can build a Turing Machine whose input consists of two components:

1. the textual description of another Turing Machine. We will refer to this as the program.
2. the input to the program, also expressed in textual format.

The Turing Machine would then run the algorithm on the input much like a computer would. These Turing Machines are called Universal Turing Machines. Note that, by using Turing Machines themselves as means to encode algorithms, we circumvent the need to create a separate programming language with which to express our programs. In addition, equipped with the theorem of Thm. 50, we are safe in the knowledge that Turing Machines are as expressive as any programming language one can construct.\(^7\) Thus, from our setup, Universal Turing Machines can be seen as nothing more than Turing Machine simulators.

Our Turing Machine textual description for \(M\), i.e., our program, must express:

1. the set of states, \(Q\)
2. the Turing Machine alphabet, \(\Sigma\)
3. the transition function, \(\delta\)

We have argued already in Sec. 2.7.3 that, since all of the above sets are finite, this textual encoding of Turing Machine descriptions can always be done. For our technical development we shall assume two translation functions, without being specific as to how this is done for now (we will however consider an example encoding later on):

- a Turing Machine encoding function \(\text{de}_M\), which takes Turing Machines as inputs and returns their respective encoding in some predefined alphabet of the Universal Turing Machine.
- a tape string encoding \(\text{de}_T\), which takes tape strings expressed in the alphabet of the program Turing Machine and returns their encoding in the predefined alphabet of the Universal Turing Machine.

Unless required, our discussion hereafter will often do not distinguish between the two encoding functions and write \([M]\) and \([x]\) for \(\text{de}_M(M)\) and \(\text{de}_T(x)\) resp.

Universal Turing Machines are formally defined in Def. 51 on Slide 76. Note that the definition does not preclude a Universal Turing Machine from simulating itself by executing an encoding of itself as input; indeed we shall be using this fact as a key step in our proof for the undecidability of the Halting problem. Note also that Def. 51 does not impose an condition upon those inputs to the Universal Turing Machine which do not constitute a valid encoding of a (program) Turing Machine.

We now demonstrate that Universal Turing Machines can indeed be constructed by considering one possible way how to go about it. The example approach we shall consider encodes descriptions of Turing Machines as binary encodings of the states and the alphabets, and then, encodes the transition function in terms of these encodings. The approach also provides an encoding for the non-blank portion of the tape.

The encoding is based on the (finite) alphabet \(\{0, 1, :, >, \ast\}\).

Turing Machine states and alphabet symbols are encoded in the same manner. We first order the respective sets and then give a straightforward binary encoding over binary strings of length \(k = \log_2(n)\) (rounded up to the nearest integer). The details are outlined on Slide 77 and Slide 78.

We encode a Turing Machine by simply encoding its transition function, using the state and alphabet encodings just discussed; this is outlined on Slide 79. Recall that a transition function consists of a map

\(^7\)Strictly speaking, we have not shown this so far, but for the time being we will assume that we can infer this.
Universal Turing Machines

**Definition 51.** A Turing Machine $M_U = (Q_U, \Sigma_U, \delta_U)$ is a Universal Turing Machine just when, for any Turing Machine $M = (Q, \Sigma, \delta)$, there exists

- a Turing Machine encoding $[-]_M : \text{TMACh} \to \Sigma_U$;
- an input string encoding $[-]_\tau : \Sigma \to \Sigma_U$

such that

$$
\langle q_0, \#x\# \rangle \xrightarrow{\tau}_M \langle q_{\#}, y\#z \rangle \\
\iff \langle q^{\#}_0, \# [M]_M [x] \# \rangle \xrightarrow{\tau}_{M_U} \langle q^{\#}_{\#}, \# [y\#z] \# \rangle
$$

---

**Slide 76**

Encoding Turing Machine States

- Order $Q \cup \{h\}$ as $q_0, q_1, \ldots, q_{n-1}, h$.
- Take base $k = \log_2(n)$ rounded up to the nearest integer.
- Give binary encoding (over strings of length $k$) of states $[q]$ according to order, whereby
  - $[q_0] = 0^k$
  - $[h] = 1^k$

---

**Slide 77**

Encoding Turing Machine Alphabet

- Order $\Sigma \cup \{L, R\}$ as $L, a_1, a_2, \ldots, a_{n-2}, R$.
- Take base $k = \log_2(n)$ rounded up to the nearest integer.
- Give binary encoding (over strings of length $k$) of alphabet $[a]$ according to order, whereby
  - $[L] = 0^k$
  - $[R] = 1^k$

---

**Slide 78**
Encoding Transition function, the Turing Machine and the tape

Transitions: \([(q_1, a) \rightarrow (q_2, \mu)] = [q_1 : [a] > [q_2] : [\mu]]

Transition Set: Order \(\delta = t_1, t_2, \ldots, t_n\)

Machine: \([M] = *[t_1]*[t_2]*\ldots*[t_n]*

Tape: \([a_1a_2 \ldots a_n] = *[a_1]*[a_2]*\ldots*[a_n]*

between tuples of the form \((q_1, a)\) and tuples of the form \((q_2, \mu)\) where \(q_1 \in Q, q_2 \in Q \cup \{h\}, a \in \Sigma\) and \(\mu \in \Sigma \cup \{L, R\}\). For each association \((q_1, a) \rightarrow (q_2, \mu)\) in the mapping, we can provide an encoding using the symbol \(:\) as a separator for the elements of the tuple and the symbol \(>\) as a separator between the tuples of the association. Moreover, since this mapping can be finitely described as a series of distinct associations, we can easily order these associations and encode each one of them, using the symbol \(*\) as a separator between such encodings.

We also encode Turing Machine input and output strings i.e., a finite concatenation of tape symbols in a similar manner, separating the encoding of each symbol in the string again by the symbol \(*\). The details are outlined on Slide 79.

*Example 52 (Encoding \(M_{\text{erase}}\)). As an example, the description on Slide 80 outlines the encoding of the Turing Machine \(M_{\text{erase}}\), formalised earlier on Slide 11. More precisely

\([M_{\text{erase}}] = *00:10>01:01*00:01>11:01*01:10>01:10*01:01>00:11*\)

Note how, in the case of the state encoding, the binary string 10 is not used. The encoding of the tape \#a\# would result in

\([\#a\#] = *01*10*01*\).

Universal Turing Machine Encoding of \(M_{\text{erase}}\)

States: \([q_0] = 00, [q_1] = 01\) and \([h] = 11\)

Alphabet: \([L] = 00, [\#] = 01, [a] = 10\) and \([R] = 11\)

Transitions: \([(q_0, a) \rightarrow (q_1, \#)] = 00:10>01:01\)

\([(q_0, \#) \rightarrow (h, \#)] = 00:01>11:01\)

\([(q_1, a) \rightarrow (q_1, \#)] = 01:10>01:10\)

\([(q_1, \#) \rightarrow (q_0, R)] = 01:00>00:11\)

Having given particular encodings for both Turing Machines and their inputs, we shall now proceed to construct a Universal Turing Machine that satisfies the conditions of Def. 51 on Slide 76; as stated earlier,\(^8\) Strictly speaking this separator is not necessary since we have fixed length encoding, but is uniform with the Turing Machine encoding.
this definition outlines the simulation required by a Universal Turing Machine. Instead of describing this
universal machine directly, we shall describe a 3-tape Turing Machine that does the job. This choice allows
us to structure our description in a more straightforward manner - recall that by Lemma 34, for every 3-tape
Turing Machine there exists a 1-tape Turing Machine that does the same job.

Universal Turing Machine with 3 tapes

<table>
<thead>
<tr>
<th>Input/output tape:</th>
<th>Keeps the encoding of the program tape. Initially also contains the program Turing Machine encoding.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program tape:</td>
<td>Keeps a copy of the program being simulated.</td>
</tr>
<tr>
<td>State Tape:</td>
<td>Keeps an encoding of the current state of the program.</td>
</tr>
</tbody>
</table>

The 3 tapes on the Universal Turing Machine are organised as follows:

**Input/output tape:** On the first tape we keep the encoding of the tape of the program Turing Machine.
To start with, however, this tape will also contain the encoding of the program Turing Machine; an
initialisation procedure will transfer this chunk of the tape to the second tape. When the head of this
tape is on an * symbol, we call the string to the right up to the next * “the encoding of the current
program tape symbol.”

**Program tape:** This tape will contain the encoding of the program Turing Machine, i.e., an encoding of
the transition function. Once the program is written on this tape, no further writes will occur on it.

**State tape:** This tape will contain (the encoding of) the current state of the program Turing Machine being
simulated. Upon every simulated transition, (the encoding of) the resulting next state of the program
is written to this tape.

The operations of the Universal Turing Machine on the input #[M][x]# then proceed as follows:

**Initialisation Procedure:** In this procedure, the Universal Turing Machine copies the program portion
[M] of the first tape to the program tape and shunts the remaining input portion, [x], to the left such
that one blank square is left between it and the leftmost location on the tape. The state tape is also
initialised to contain the encoding of q0, [q0]. The procedure then finishes by placing the heads of the
program and state tapes on the leftmost positions and the head of the input/output tape on the first
* to the left of the first encoded symbol.

**Execute Procedure:** Assume that the current program state is q i.e., [q] is currently on the state tape,
and (the encoding of) the current program symbol is a, i.e., the head of the input tape is on a * symbol
followed by the string [a]*.

1. If q = h, i.e., its encoding consists entirely of 1 symbols, then the halting procedure is initialised.
Otherwise, an attempt is made to locate the left pair of a transition \((q, a) \rightarrow (q', \mu)\) on the program
tape by carrying out the following steps (note that since the Turing Machine being encoded is
deterministic, at most one such transition should exist):

   a) An attempt is made to first match [q] from the state tape with some corresponding *[q]:
      segment on the program tape. If no match if found, the Universal Turing Machine hangs.
3-tape Universal Turing Machine Operations

Upon input \#[M][x]\# on the Input/output tape:

**Initialisation:** Transfer \[M\] to the second tape, write \[q_0\] the the third tape, shift \([x]\) on the first tape accordingly.

**Execute transitions:** Find transition from tape 2 that matches the current symbol encoding on tape 1 and the current state encoding on tape 3 and modify tape 3 and tape 1 as dictated by the transition.

**Halting:** Erase Program and State tapes and move head on Input/output tape to the first \# symbol to the right of the string.

(b) If a match is found, an attempt is made to match the encoding of the current program symbol, \([a]\), to a subsequent string \([a]\#\).

- If no match is found, step (a) is repeated.
- Otherwise i.e., is a match is found, the following steps are taken:
  i. the state \([q']\) following the separator \# on the encoded transition is copied to the state tape, overwriting on the existing state.
  ii. Depending on the encoded \([\mu]\) following the separator : of the second part of the encoded transition, we take the following steps:
    - \(\mu = a'\): The Input/Output tape overwrites the string to its right encoding the previous symbol \(a\) with the encoding string \([a']\).
    - \(\mu = L\): The head on the Input/Output tape is moved left until the next \(*\). Note that if the next \(*\) is never encountered, the Universal Turing Machine will crash, but this is what we want since the program being simulated would do so as well.
    - \(\mu = R\): The head on the Input/Output tape is moved right until the next \(*\).
  iii. The heads of the program and state tapes are returned to the leftmost positions and, with \(q'\) being the new state, we goto step 1.

**Halting Procedure:** Erase the program and state tapes and remove trailing blanks on the input/output tape. Then the input/output tape head is moved to the first (Universal Turing Machine) blank symbol to the right of the (encoded) output string.

---

**Example 53 (Simulating \(M_{\text{erase}}\)).** Let us consider the execution of the program \([M_{\text{erase}}]\) upon input \#a\# using a Universal Turing Machine simulation. The input to the Universal Turing Machine, equivalent to \([M_{\text{erase}}]\[#a#\] is:

\*[00:10>01:01*00:01>11:01*01:10>01:01*01:00:11**01*10*01*]

Note that the occurrence of two adjacent \* symbols act as a separator between the encoding of \(M_{\text{erase}}\) and that of its input \#a#. Upon completion of the initialisation procedure, the contents of the Universal Turing Machine tapes at every simulated transitions will be as follows:

---

\[\text{This is, strictly speaking, not necessary, due to our earlier definition of computability on a } k\text{-tape Turing Machine, Def. 31, which allowed junk to be left on tapes that are not the input/output tapes.}\]
1. Initialise:
   
   **Input/Output:** #*01*10*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*00*#

2. Find transition:
   
   **Input/Output:** #*01*10*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*00*#

3. Perform Transition (Write # i.e., 01):
   
   **Input/Output:** #*01*01*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*01*#

4. Find Transition:
   
   **Input/Output:** #*01*01*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*01*#

5. Perform Transition (move right)
   
   **Input/Output:** #*01*01*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*00*#

6. Find transition
   
   **Input/Output:** #*01*01*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*00*#

7. Perform Transition (rewrite # i.e., 01) and move to halt:
   
   **Input/Output:** #*01*01*01*#
   **Program:** #*00:10>01:01*00:01*11:01*01:10*01:01>00:11*#
   **State:** #*11*#

8. Halt
   
   **Input/Output:** #*01*01*#
   **Program:** ##
   **State:** ##
2.8.2 The Halting Problem

Equipped with our notion of Universal Turing Machine, we now revert back to the problem outlined on Slide 75. We start by describing a problem called Universal Halt on Slide 83 and pose the question of whether Universal Halt is algorithmically solvable.

It seems dauntingly difficult to find a method for determining whether a given Turing Machine $M$ halts on any input. For a start, there are infinitely many possible inputs; trying all the possible inputs in sequence is not an acceptable strategy because, even if it is indeed the case that $M$ halts on all inputs, the strategy will never terminate trying all possible inputs and will never determine whether this fact really holds.

For this reason, we first try to deal with a simpler problem called Halt on Slide 84. Crucially, if we cannot find an algorithm to solve Halt, then it is not possible to find an algorithm to solve Universal Halt. This is so because every instance of $M$ and input $x$ pair that need to be considered for Universal Halt have to be solved as an instance of a Halt problem.

In order to determine whether Halt has an algorithmic solution, the problem must be somehow phrased and encoded in terms of some (formal) language. Moreover, note that the question in Slide 84 is a predicate (returning true or false as an answer) and predicates can be used for defining sets. Thus, using this predicate, we define the language $L_{halt}$ on Slide 85. The observation of greatest interest to us is however that Halt is Turing decidable if and only if $L_{halt}$ is a recursive language. More specifically, if $L_{halt}$ is recursive, then, by Def. 20, we have a general algorithm for determining whether any program $M$ halts on any input $x$, and thus an algorithm which determines whether $M$ halts on $x$ (i.e., we first encode $M$ and $x$ as $dM$ and input this string in the Turing Machine deciding $L_{halt}$).

If, on the other hand, $L_{halt}$ is not recursive it would mean that there is no algorithm for determining, in general, whether a program $M$ halts on input $x$. This, in turn implies that Halt does not have an algorithmic solution. Indeed it turns out to be the case, as stated by Thm. 54 on Slide 86.

(Proof for Thm. 54). Define the language $L_*$ as

$$L_* = \{ [M] \mid M \text{ halts on input } [M] \}$$

Indeed, every set we have described so far, whether Turing recognisable or not, was described using a predicate which acts as a gate keeper, telling us which elements from a certain domain belong to the set and which elements don’t.
The language $L_{\text{halt}}$

$$L_{\text{halt}} \overset{\text{def}}{=} \{ [M][x] \mid M \text{ halts on input } x \}$$

Note that $x \in L_{\text{halt}}$ implies $x \in \{0, 1, :, >, *\}^*$.

HALT is Turing decidable iff $L_{\text{halt}}$ is recursive

Decidability of $L_{\text{halt}}, \text{Halt and Universal Halt}$

**Theorem 54.** $L_{\text{halt}}$ is not recursive

**Corollary 55.** HALT is not decidable

**Corollary 56.** Universal HALT is not decidable
1. If \( L_{\text{halt}} \) is recursive, then so is \( L_\ast \).
   Note that from our definitions of the languages \( L_\ast \) and \( L_{\text{halt}} \), for any \( x \in L_\ast \) iff \( xx \in L_{\text{halt}} \). Now if \( L_{\text{halt}} \) is recursive, there exists a Turing Machine \( M_h \) that can decide the membership of \( L_{\text{halt}} \), which means that the Turing Machine that decided \( L_\ast \) can be constructed from \( M_h \) using the schema \( M_{\text{copy}} M_h \), where \( M_{\text{copy}} \) converts an input \( x \) to the output \( xx \) (cf. Exercise 2.1.1.3 for a similar machine).

2. If \( L_\ast \) is not recursive, then \( L_{\text{halt}} \) is not recursive
   It follows immediately as the contrapositive of 1.

3. If \( L_\ast \) is not recursively enumerable, then \( L_\ast \) is not recursive
   By Corollary 25.

4. For any Turing Machine \( M, [M] \in \overline{L}_\ast \) iff \( M \) does not halt on input \([M]\).
   By definition of \( L_\ast \) and of language complement.

5. \( \overline{L}_\ast \) is not recursively enumerable
   Assume that \( \overline{L}_\ast \) is recursively enumerable. Then there exists some \( M_{\text{halt}} \) computing the membership predicate of \( \overline{L}_\ast \), which means that \( M_{\text{halt}} \) has the following property:
   For any \( x, M_{\text{halt}} \) halts on \( x \) iff \( x \in \overline{L}_\ast \).
   Consider the instance when \( x = [M_{\text{halt}}] \). Then, by the above assumption we have:
   \( M_{\text{halt}} \) halts on \([M_{\text{halt}}]\) iff \([M_{\text{halt}}]\) \( \in \overline{L}_\ast \)
   and by expanding the definition of \( \overline{L}_\ast \) this means
   \( M_{\text{halt}} \) halts on \([M_{\text{halt}}]\) iff \( M_{\text{halt}} \) does not halt on \([M_{\text{halt}}]\)
   which is clearly a contradiction. Thus our assumption must be false and therefore \( \overline{L}_\ast \) is not recursively enumerable.

6. Since \( \overline{L}_\ast \) is not recursively enumerable, by (3), \( L_\ast \) is not recursive and, by (2), \( L_{\text{halt}} \) is not recursive.

What are the repercussions of Thm. 54? For a start, \( \text{Halt} \) cannot be decidable, and as a result the more general \( \text{Universal Halt} \) is not decidable either (see Slide 86). In terms of our language-class hierarchy, Slide 74, we also know that there exists a class of languages that can be semi-decided but are not (fully) decidable.

2.8.3 Exercises

1. Formalise and encode the Turing Machines described on Slide 38 for \( \Sigma = \{\#, a\} \).

2. Show that \( L_{\text{halt}} \) is recursively enumerable.