Part I

Turing Machines
The material in this part is a basic and informal introduction to Turing machines. It needs more examples and exercises, and perhaps information on available Turing machine simulators. The proof of the unsolvability of the decision problem uses a successor function, hence all models are infinite. One could strengthen the result by using a successor relation instead. There probably are subtle oversights; use these as checks on students’ attention (but also file issues!).
Chapter 1

Turing Machine Computations

1.1 Introduction

What does it mean for a function, say, from \( \mathbb{N} \) to \( \mathbb{N} \) to be \textit{computable}? Among the first answers, and the most well known one, is that a function is computable if it can be computed by a Turing machine. This notion was set out by Alan Turing in 1936. Turing machines are an example of a \textit{model of computation}—they are a mathematically precise way of defining the idea of a “computational procedure.” What exactly that means is debated, but it is widely agreed that Turing machines are one way of specifying computational procedures. Even though the term “Turing machine” evokes the image of a physical machine with moving parts, strictly speaking a Turing machine is a purely mathematical construct, and as such it idealizes the idea of a computational procedure. For instance, we place no restriction on either the time or memory requirements of a Turing machine: Turing machines can compute something even if the computation would require more storage space or more steps than there are atoms in the universe.

It is perhaps best to think of a Turing machine as a program for a special kind of imaginary mechanism. This mechanism consists of a \textit{tape} and a \textit{read-write head}. In our version of Turing machines, the tape is infinite in one direction (to the right), and it is divided into \textit{squares}, each of which may contain a symbol from a finite \textit{alphabet}. Such alphabets can contain any number of different symbols, say, but we will mainly make do with three: \( \triangleright \), 0, and 1. When the mechanism is started, the tape is empty (i.e., each square contains the symbol 0) except for the leftmost square, which contains \( \triangleright \), and a finite number of squares which contain the \textit{input}. At any time, the mechanism is in one of a finite number of \textit{states}. At the outset, the head scans the leftmost square and in a specified \textit{initial state}. At each step of the mechanism’s run, the content of the square currently scanned together with the state the mechanism is in and the Turing machine program determine what happens next. The Turing machine program is given by a partial function which takes as input a state \( q \) and a symbol \( \sigma \) and outputs a triple \( (q', \sigma', D) \). Whenever the mechanism is in
state $q$ and reads symbol $\sigma$, it replaces the symbol on the current square with $\sigma'$, the head moves left, right, or stays put according to whether $D$ is $L$, $R$, or $N$, and the mechanism goes into state $q'$.

For instance, consider the situation in Figure 1.1. The visible part of the tape of the Turing machine contains the end-of-tape symbol ⊢ on the leftmost square, followed by three 1's, a 0, and four more 1's. The head is reading the third square from the left, which contains a 1, and is in state $q_1$—we say “the machine is reading a 1 in state $q_1$.” If the program of the Turing machine returns, for input $⟨q_1, 1⟩$, the triple $⟨q_2, 0, N⟩$, the machine would now replace the 1 on the third square with a 0, leave the read/write head where it is, and switch to state $q_2$. If then the program returns $⟨q_3, 0, R⟩$ for input $⟨q_2, 0⟩$, the machine would now overwrite the 0 with another 0 (effectively, leaving the content of the tape under the read/write head unchanged), move one square to the right, and enter state $q_3$. And so on.

We say that the machine halts when it encounters some state, $q_n$, and symbol, $\sigma$ such that there is no instruction for $⟨q_n, \sigma⟩$, i.e., the transition function for input $⟨q_n, \sigma⟩$ is undefined. In other words, the machine has no instruction to carry out, and at that point, it ceases operation. Halting is sometimes represented by a specific halt state $h$. This will be demonstrated in more detail later on.

The beauty of Turing’s paper, “On computable numbers,” is that he presents not only a formal definition, but also an argument that the definition captures the intuitive notion of computability. From the definition, it should be clear that any function computable by a Turing machine is computable in the intuitive sense. Turing offers three types of argument that the converse is true, i.e., that any function that we would naturally regard as computable is computable by such a machine. They are (in Turing’s words):

1. A direct appeal to intuition.
2. A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).
3. Giving examples of large classes of numbers which are computable.
Our goal is to try to define the notion of computability “in principle,” i.e., without taking into account practical limitations of time and space. Of course, with the broadest definition of computability in place, one can then go on to consider computation with bounded resources; this forms the heart of the subject known as “computational complexity.”

**Historical Remarks**  Alan Turing invented Turing machines in 1936. While his interest at the time was the decidability of first-order logic, the paper has been described as a definitive paper on the foundations of computer design. In the paper, Turing focuses on computable real numbers, i.e., real numbers whose decimal expansions are computable; but he notes that it is not hard to adapt his notions to computable functions on the natural numbers, and so on. Notice that this was a full five years before the first working general purpose computer was built in 1941 (by the German Konrad Zuse in his parent’s living room), seven years before Turing and his colleagues at Bletchley Park built the code-breaking Colossus (1943), nine years before the American ENIAC (1945), twelve years before the first British general purpose computer—the Manchester Small-Scale Experimental Machine—was built in Manchester (1948), and thirteen years before the Americans first tested the BINAC (1949). The Manchester SSEM has the distinction of being the first stored-program computer—previous machines had to be rewired by hand for each new task.

### 1.2 Representing Turing Machines

Turing machines can be represented visually by *state diagrams*. The diagrams are composed of state cells connected by arrows. Unsurprisingly, each state cell represents a state of the machine. Each arrow represents an instruction that can be carried out from that state, with the specifics of the instruction written above or below the appropriate arrow. Consider the following machine, which has only two internal states, \( q_0 \) and \( q_1 \), and one instruction:

\[
\begin{array}{c}
\text{start} \\
\rightarrow
\end{array} \quad \begin{array}{c}
 q_0 \\
\rightarrow 0, 1, R \\
 q_1
\end{array}
\]

Recall that the Turing machine has a read/write head and a tape with the input written on it. The instruction can be read as *if reading a 0 in state \( q_0 \), write a 1, move right, and move to state \( q_1 \)*. This is equivalent to the transition function mapping \( \langle q_0, 0 \rangle \) to \( \langle q_1, 1, R \rangle \).

**Example 1.1.** *Even Machine:* The following Turing machine halts if, and only if, there are an even number of 1’s on the tape (under the assumption
that all 1’s come before the first 0 on the tape).

The state diagram corresponds to the following transition function:

\[
\begin{align*}
\delta(q_0, 1) &= \langle q_1, 1, R \rangle, \\
\delta(q_1, 1) &= \langle q_0, 1, R \rangle, \\
\delta(q_1, 0) &= \langle q_1, 0, R \rangle
\end{align*}
\]

The above machine halts only when the input is an even number of strokes. Otherwise, the machine (theoretically) continues to operate indefinitely. For any machine and input, it is possible to trace through the configurations of the machine in order to determine the output. We will give a formal definition of configurations later. For now, we can intuitively think of configurations as a series of diagrams showing the state of the machine at any point in time during operation. Configurations show the content of the tape, the state of the machine and the location of the read/write head.

Let us trace through the configurations of the even machine if it is started with an input of four 1’s. In this case, we expect that the machine will halt. We will then run the machine on an input of three 1’s, where the machine will run forever.

The machine starts in state \(q_0\), scanning the leftmost 1. We can represent the initial state of the machine as follows:

\[\triangleright 1_01110\ldots\]

The above configuration is straightforward. As can be seen, the machine starts in state one, scanning the leftmost 1. This is represented by a subscript of the state name on the first 1. The applicable instruction at this point is \(\delta(q_0, 1) = \langle q_1, 1, R \rangle\), and so the machine moves right on the tape and changes to state \(q_1\).

\[\triangleright 1_1110\ldots\]

Since the machine is now in state \(q_1\) scanning a 1, we have to “follow” the instruction \(\delta(q_1, 1) = \langle q_0, 1, R \rangle\). This results in the configuration

\[\triangleright 11_110\ldots\]

As the machine continues, the rules are applied again in the same order, resulting in the following two configurations:

\[\triangleright 1111_10\ldots\]
The machine is now in state $q_0$ scanning a $0$. Based on the transition diagram, we can easily see that there is no instruction to be carried out, and thus the machine has halted. This means that the input has been accepted.

Suppose next we start the machine with an input of three $1$’s. The first few configurations are similar, as the same instructions are carried out, with only a small difference of the tape input:

$\triangleright 111010 \ldots$

$\triangleright 111010 \ldots$

$\triangleright 11100 \ldots$

$\triangleright 11101 \ldots$

The machine has now traversed past all the $1$’s, and is reading a $0$ in state $q_1$. As shown in the diagram, there is an instruction of the form $\delta(q_1, 0) = (q_1, 0, R)$. Since the tape is filled with $0$ indefinitely to the right, the machine will continue to execute this instruction forever, staying in state $q_1$ and moving ever further to the right. The machine will never halt, and does not accept the input.

It is important to note that not all machines will halt. If halting means that the machine runs out of instructions to execute, then we can create a machine that never halts simply by ensuring that there is an outgoing arrow for each symbol at each state. The even machine can be modified to run indefinitely by adding an instruction for scanning a $0$ at $q_0$.

Example 1.2.

Machine tables are another way of representing Turing machines. Machine tables have the tape alphabet displayed on the $x$-axis, and the set of machine states across the $y$-axis. Inside the table, at the intersection of each state and symbol, is written the rest of the instruction—the new state, new symbol, and direction of movement. Machine tables make it easy to determine in what state, and for what symbol, the machine halts. Whenever there is a gap in the table is a possible point for the machine to halt. Unlike state diagrams and instruction sets, where the points at which the machine halts are not always immediately obvious, any halting points are quickly identified by finding the gaps in the machine table.
Example 1.3. The machine table for the even machine is:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0, q₁, R</td>
<td>1, q₁, R</td>
</tr>
<tr>
<td>q₁</td>
<td>0, q₀, R</td>
<td>1, q₀, R</td>
</tr>
</tbody>
</table>

As we can see, the machine halts when scanning a blank in state q₀.

So far we have only considered machines that read and accept input. However, Turing machines have the capacity to both read and write. An example of such a machine (although there are many, many examples) is a doubler. A doubler, when started with a block of $n$ 1’s on the tape, outputs a block of $2n$ 1’s.

Example 1.4. Before building a doubler machine, it is important to come up with a strategy for solving the problem. Since the machine (as we have formulated it) cannot remember how many 1’s it has read, we need to come up with a way to keep track of all the 1’s on the tape. One such way is to separate the output from the input with a 0. The machine can then erase the first 1 from the input, traverse over the rest of the input, leave a 0, and write two new 1’s. The machine will then go back and find the second 1 in the input, and double that one as well. For each one 1 of input, it will write two 1’s of output. By erasing the input as the machine goes, we can guarantee that no 1 is missed or doubled twice. When the entire input is erased, there will be $2n$ 1’s left on the tape. The state diagram of the resulting Turing machine is depicted in Figure 1.2.

Problem 1.1. Choose an arbitrary input and trace through the configurations of the doubler machine in Example 1.4.
Problem 1.2. The double machine in Example 1.4 writes its output to the right of the input. Come up with a new method for solving the doubler problem which generates its output immediately to the right of the end-of-tape marker. Build a machine that executes your method. Check that your machine works by tracing through the configurations.

Problem 1.3. Design a Turing-machine with alphabet \{\_\_\_\_\_\_\_\_0, A, B\} that accepts, i.e., halts on, any string of A’s and B’s where the number of A’s is the same as the number of B’s and all the A’s precede all the B’s, and rejects, i.e., does not halt on, any string where the number of A’s is not equal to the number of B’s or the A’s do not precede all the B’s. (E.g., the machine should accept ABBB, and AAABBB, but reject both AAB and AABBAABB.)

Problem 1.4. Design a Turing-machine with alphabet \{\_\_\_\_\_\_\_\_0, A, B\} that takes as input any string \(\alpha\) of A’s and B’s and duplicates them to produce an output of the form \(\alpha\alpha\). (E.g. input ABBA should result in output ABBAABBA).

Problem 1.5. Alphabetical?: Design a Turing-machine with alphabet \{\_\_\_\_\_\_\_\_0, A, B\} that when given as input a finite sequence of A’s and B’s checks to see if all the A’s appear to the left of all the B’s or not. The machine should leave the input string on the tape, and either halt if the string is “alphabetical”, or loop forever if the string is not.

Problem 1.6. Alphabetizer: Design a Turing-machine with alphabet \{\_\_\_\_\_\_\_\_0, A, B\} that takes as input a finite sequence of A’s and B’s rearranges them so that all the A’s are to the left of all the B’s. (e.g., the sequence BABAA should become the sequence AAABB, and the sequence ABBABB should become the sequence AABBBB).

1.3 Turing Machines

The formal definition of what constitutes a Turing machine looks abstract, but is actually simple: it merely packs into one mathematical structure all the information needed to specify the workings of a Turing machine. This includes (1) which states the machine can be in, (2) which symbols are allowed to be on the tape, (3) which state the machine should start in, and (4) what the instruction set of the machine is.

Definition 1.5 (Turing machine). A Turing machine \(M\) is a tuple \(\langle Q, \Sigma, q_0, \delta \rangle\) consisting of

1. a finite set of states \(Q\),
2. a finite alphabet \(\Sigma\) which includes \(\_\_\_\_\_\_\_\_0\) and 0,
3. an initial state \(q_0 \in Q\),
4. a finite instruction set \(\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, N\}\).
The partial function $\delta$ is also called the *transition function* of $M$.

We assume that the tape is infinite in one direction only. For this reason it is useful to designate a special symbol $\triangleright$ as a marker for the left end of the tape. This makes it easier for Turing machine programs to tell when they’re “in danger” of running off the tape. We could assume that this symbol is never overwritten, i.e., that $\delta(q,\triangleright) = \langle q',\triangleright,x \rangle$ if $\delta(q,\triangleright)$ is defined. Some textbook do this, we do not. You can simply be careful when constructing your Turing machine that it nevery overwrites $\triangleright$. Moreover, there are cases where allowing such overwriting provides some convenient flexibility.

**Example 1.6. Even Machine:** The even machine is formally the quadruple $\langle Q, \Sigma, q_0, \delta \rangle$ where

\[
\begin{align*}
Q &= \{ q_0, q_1 \} \\
\Sigma &= \{ \triangleright, 0, 1 \}, \\
\delta(q_0, 1) &= \langle q_1, 1, R \rangle, \\
\delta(q_1, 1) &= \langle q_0, 1, R \rangle, \\
\delta(q_1, 0) &= \langle q_1, 0, R \rangle.
\end{align*}
\]

### 1.4 Configurations and Computations

Recall tracing through the configurations of the even machine earlier. The imaginary mechanism consisting of tape, read/write head, and Turing machine program is really just an intuitive way of visualizing what a Turing machine computation is. Formally, we can define the computation of a Turing machine on a given input as a sequence of *configurations*—and a configuration in turn is a sequence of symbols (corresponding to the contents of the tape at a given point in the computation), a number indicating the position of the read/write head, and a state. Using these, we can define what the Turing machine $M$ computes on a given input.

**Definition 1.7 (Configuration).** A *configuration* of Turing machine $M = \langle Q, \Sigma, q_0, \delta \rangle$ is a triple $\langle C, m, q \rangle$ where

1. $C \in \Sigma^*$ is a finite sequence of symbols from $\Sigma$, 
2. $m \in \mathbb{N}$ is a number $< \text{len}(C)$, and 
3. $q \in Q$

Intuitively, the sequence $C$ is the content of the tape (symbols of all squares from the leftmost square to the last non-blank or previously visited square), $m$ is the number of the square the read/write head is scanning (beginning with 0 being the number of the leftmost square), and $q$ is the current state of the machine.
The potential input for a Turing machine is a sequence of symbols, usually a sequence that encodes a number in some form. The initial configuration of the Turing machine is that configuration in which we start the Turing machine to work on that input: the tape contains the tape end marker immediately followed by the input written on the squares to the right, the read/write head is scanning the leftmost square of the input (i.e., the square to the right of the left end marker), and the mechanism is in the designated start state $q_0$.

**Definition 1.8 (Initial configuration).** The initial configuration of $M$ for input $I \in \Sigma^\ast$ is

$$\langle \triangleright \sim I, 1, q_0 \rangle.$$

The $\sim$ symbol is for concatenation—we want to ensure that there are no blanks between the left end marker and the beginning of the input.

**Definition 1.9.** We say that a configuration $\langle C, m, q \rangle$ yields the configuration $\langle C', m', q' \rangle$ in one step (according to $M$), iff

1. the $m$-th symbol of $C$ is $\sigma$,
2. the instruction set of $M$ specifies $\delta(q, \sigma) = \langle q', \sigma', D \rangle$,
3. the $m$-th symbol of $C'$ is $\sigma'$, and
4. a) $D = L$ and $m' = m - 1$ if $m > 0$, otherwise $m' = 0$, or
   b) $D = R$ and $m' = m + 1$, or
   c) $D = N$ and $m' = m$,
5. if $m' = \text{len}(C)$, then $\text{len}(C') = \text{len}(C) + 1$ and the $m'$-th symbol of $C'$ is 0. Otherwise $\text{len}(C') = \text{len}(C)$.
6. for all $i$ such that $i < \text{len}(C)$ and $i \neq m$, $C'(i) = C(i)$,

**Definition 1.10.** A run of $M$ on input $I$ is a sequence $C_i$ of configurations of $M$, where $C_0$ is the initial configuration of $M$ for input $I$, and each $C_i$ yields $C_{i+1}$ in one step.

We say that $M$ halts on input $I$ after $k$ steps if $C_k = \langle C, m, q \rangle$, the $m$th symbol of $C$ is $\sigma$, and $\delta(q, \sigma)$ is undefined. In that case, the output of $M$ for input $I$ is $O$, where $O$ is a string of symbols not beginning or ending in 0 such that $C = \triangleright \sim 0^i \sim O \sim 0^j$ for some $i, j \in \mathbb{N}$.

According to this definition, the output $O$ of $M$ always begins and ends in a symbol other than 0, or, if at time $k$ the entire tape is filled with 0 (except for the leftmost $\triangleright$), $O$ is the empty string.
1.5 Unary Representation of Numbers

Turing machines work on sequences of symbols written on their tape. Depending on the alphabet a Turing machine uses, these sequences of symbols can represent various inputs and outputs. Of particular interest, of course, are Turing machines which compute 

\textit{arithmetical} functions, i.e., functions of natural numbers. A simple way to represent positive integers is by coding them as sequences of a single symbol 1. If \( n \in \mathbb{N} \), let \( 1^n \) be the empty sequence if \( n = 0 \), and otherwise the sequence consisting of exactly \( n \) 1’s.

**Definition 1.11 (Computation).** A Turing machine \( M \) computes the function \( f: \mathbb{N}^n \to \mathbb{N} \) if \( M \) halts on input

\[ 1^{k_1}01^{k_2}0\ldots01^{k_n} \]

with output \( 1^{f(k_1,\ldots,k_n)} \).

**Example 1.12. Addition:** Build a machine that, when given an input of two non-empty strings of 1’s of length \( n \) and \( m \), computes the function \( f(n,m) = n + m \).

We want to come up with a machine that starts with two blocks of strokes on the tape and halts with one block of strokes. We first need a method to carry out. The input strokes are separated by a blank, so one method would be to write a stroke on the square containing the blank, and erase the first (or last) stroke. This would result in a block of \( n+m \) 1’s. Alternatively, we could proceed in a similar way to the doubler machine, by erasing a stroke from the first block, and adding one to the second block of strokes until the first block has been removed completely. We will proceed with the former example.

![Turing Machine Diagram](diagram.png)

**Problem 1.7.** Trace through the configurations of the machine for input \( \langle 3,5 \rangle \).

**Problem 1.8. Subtraction:** Design a Turing machine that when given an input of two non-empty strings of strokes of length \( n \) and \( m \), where \( n > m \), computes the function \( f(n,m) = n - m \).

**Problem 1.9. Equality:** Design a Turing machine to compute the following function:

\[ \text{equality}(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases} \]

where \( x \) and \( y \) are integers greater than 0.
Problem 1.10. Design a Turing machine to compute the function \( \min(x, y) \) where \( x \) and \( y \) are positive integers represented on the tape by strings of 1’s separated by a 0. You may use additional symbols in the alphabet of the machine.

The function \( \min \) selects the smallest value from its arguments, so \( \min(3, 5) = 3 \), \( \min(20, 16) = 16 \), and \( \min(4, 4) = 4 \), and so on.

1.6 Halting States

Although we have defined our machines to halt only when there is no instruction to carry out, common representations of Turing machines have a dedicated halting state, \( h \), such that \( h \in Q \).

The idea behind a halting state is simple: when the machine has finished operation (it is ready to accept input, or has finished writing the output), it goes into a state \( h \) where it halts. Some machines have two halting states, one that accepts input and one that rejects input.

Example 1.13. Halting States. To elucidate this concept, let us begin with an alteration of the even machine. Instead of having the machine halt in state \( q_0 \) if the input is even, we can add an instruction to send the machine into a halt state.

Let us further expand the example. When the machine determines that the input is odd, it never halts. We can alter the machine to include a reject state by replacing the looping instruction with an instruction to go to a reject state.
Adding a dedicated halting state can be advantageous in cases like this, where it makes explicit when the machine accepts/rejects certain inputs. However, it is important to note that no computing power is gained by adding a dedicated halting state. Similarly, a less formal notion of halting has its own advantages. The definition of halting used so far in this chapter makes the proof of the Halting Problem intuitive and easy to demonstrate. For this reason, we continue with our original definition.

1.7 Combining Turing Machines

The examples of Turing machines we have seen so far have been fairly simple in nature. But in fact, any problem that can be solved with any modern programming language can also be solved with Turing machines. To build more complex Turing machines, it is important to convince ourselves that we can combine them, so we can build machines to solve more complex problems by breaking the procedure into simpler parts. If we can find a natural way to break a complex problem down into constituent parts, we can tackle the problem in several stages, creating several simple Turing machines and combining them into one machine that can solve the problem. This point is especially important when tackling the Halting Problem in the next section.

Example 1.14. Combining Machines: Design a machine that computes the function \( f(m, n) = 2(m + n) \).

In order to build this machine, we can combine two machines we are already familiar with: the addition machine, and the doubler. We begin by drawing a state diagram for the addition machine.

Instead of halting at state \( q_2 \), we want to continue operation in order to double the output. Recall that the doubler machine erases the first stroke in the input.
and writes two strokes in a separate output. Let’s add an instruction to make
sure the tape head is reading the first stroke of the output of the addition
machine.

\begin{center}
\begin{tikzpicture}[->,thick,main node/.style={circle,draw}]
    \node[main node,fill=white] (q0) at (0,0) {$q_0$};
    \node[main node,fill=white] (q1) at (2,1) {$q_1$};
    \node[main node,fill=white] (q2) at (2,2) {$q_2$};
    \node[main node,fill=white] (q3) at (4,1) {$q_3$};
    \node[main node,fill=white] (q4) at (4,0) {$q_4$};

    \path
    (q0) edge [loop above] node {$0,1,R$} (q0)
    (q0) edge node {$1,1,L$} (q3)
    (q0) edge node {$1,1,R$} (q1)
    (q1) edge node {$0,1,R$} (q0)
    (q1) edge node {$0,0,L$} (q2)
    (q2) edge node {$1,0,L$} (q1)
    (q3) edge node {$1,0,L$} (q1)
    (q3) edge [loop below] node {$\triangleright,\triangleright, R$} (q3)
    (q3) edge node {$1,1, L$} (q4)
    (q4) edge node [swap] {$1,1, R$} (q3);
\end{tikzpicture}
\end{center}

It is now easy to double the input—all we have to do is connect the doubler
machine onto state $q_4$. This requires renaming the states of the doubler machine
so that they start at $q_4$ instead of $q_0$—this way we don’t end up with two
starting states. The final diagram should look as in Figure 1.3.

1.8 Variants of Turing Machines

There are in fact many possible ways to define Turing machines, of which ours
is only one. In some ways, our definition is more liberal than others. We allow
arbitrary finite alphabets, a more restricted definition might allow only two
tape symbols, 1 and 0. We allow the machine to write a symbol to the tape
and move at the same time, other definitions allow either writing or moving. We
allow the possibility of writing without moving the tape head, other definitions
leave out the $N$ “instruction.” In other ways, our definition is more restrictive.
We assumed that the tape is infinite in one direction only, other definitions
allow the tape to be infinite both to the left and the right. In fact, one can
even allow any number of separate tapes, or even an infinite grid of squares.
We represent the instruction set of the Turing machine by a transition function;
other definitions use a transition relation where the machine has more than one
possible instruction in any given situation.

This last relaxation of the definition is particularly interesting. In our
definition, when the machine is in state $q$ reading symbol $\sigma$, $\delta(q, \sigma)$ determines
what the new symbol, state, and tape head position is. But if we allow the
instruction set to be a relation between current state-symbol pairs $\langle q, \sigma \rangle$ and
new state-symbol-direction triples $\langle q', \sigma', D \rangle$, the action of the Turing machine
may not be uniquely determined—the instruction relation may contain both
$\langle q, \sigma, q', \sigma', D \rangle$ and $\langle q, \sigma, q'', \sigma'', D' \rangle$. In this case we have a non-deterministic
Turing machine. These play an important role in computational complexity theory.

There are also different conventions for when a Turing machine halts: we say it halts when the transition function is undefined, other definitions require the machine to be in a special designated halting state. Since the tapes of our Turing machines are infinite in one direction only, there are cases where a Turing machine can’t properly carry out an instruction: if it reads the leftmost square and is supposed to move left. According to our definition, it just stays put instead, but we could have defined it so that it halts when that happens.

There are also different ways of representing numbers (and hence the input-output function computed by a Turing machine): we use unary representation, but you can also use binary representation. This requires two symbols in addition to 0 and ⊥.

Now here is an interesting fact: none of these variations matters as to which functions are Turing computable. If a function is Turing computable according to one definition, it is Turing computable according to all of them.
1.9 The Church-Turing Thesis

Turing machines are supposed to be a precise replacement for the concept of an effective procedure. Turing thought that anyone who grasped both the concept of an effective procedure and the concept of a Turing machine would have the intuition that anything that could be done via an effective procedure could be done by Turing machine. This claim is given support by the fact that all the other proposed precise replacements for the concept of an effective procedure turn out to be extensionally equivalent to the concept of a Turing machine—that is, they can compute exactly the same set of functions. This claim is called the Church-Turing thesis.

Definition 1.15 (Church-Turing thesis). The Church-Turing Thesis states that anything computable via an effective procedure is Turing computable.

The Church-Turing thesis is appealed to in two ways. The first kind of use of the Church-Turing thesis is an excuse for laziness. Suppose we have a description of an effective procedure to compute something, say, in “pseudo-code.” Then we can invoke the Church-Turing thesis to justify the claim that the same function is computed by some Turing machine, even if we have not in fact constructed it.

The other use of the Church-Turing thesis is more philosophically interesting. It can be shown that there are functions which cannot be computed by Turing machines. From this, using the Church-Turing thesis, one can conclude that it cannot be effectively computed, using any procedure whatsoever. For if there were such a procedure, by the Church-Turing thesis, it would follow that there would be a Turing machine. So if we can prove that there is no Turing machine that computes it, there also can’t be an effective procedure. In particular, the Church-Turing thesis is invoked to claim that the so-called halting problem not only cannot be solved by Turing machines, it cannot be effectively solved at all.
Chapter 2

Undecidability

2.1 Introduction

It might seem obvious that not every function, even every arithmetical function, can be computable. There are just too many, whose behavior is too complicated. Functions defined from the decay of radioactive particles, for instance, or other chaotic or random behavior. Suppose we start counting 1-second intervals from a given time, and define the function \( f(n) \) as the number of particles in the universe that decay in the \( n \)-th 1-second interval after that initial moment. This seems like a candidate for a function we cannot ever hope to compute.

But it is one thing to not be able to imagine how one would compute such functions, and quite another to actually prove that they are uncomputable. In fact, even functions that seem hopelessly complicated may, in an abstract sense, be computable. For instance, suppose the universe is finite in time—some day, in the very distant future the universe will contract into a single point, as some cosmological theories predict. Then there is only a finite (but incredibly large) number of seconds from that initial moment for which \( f(n) \) is defined. And any function which is defined for only finitely many inputs is computable: we could list the outputs in one big table, or code it in one very big Turing machine state transition diagram.

We are often interested in special cases of functions whose values give the answers to yes/no questions. For instance, the question “is \( n \) a prime number?” is associated with the function

\[
isprime(n) = \begin{cases} 
1 & \text{if } n \text{ is prime} \\
0 & \text{otherwise.}
\end{cases}
\]

We say that a yes/no question can be effectively decided, if the associated 1/0-valued function is effectively computable.

To prove mathematically that there are functions which cannot be effectively computed, or problems that cannot effectively decided, it is essential to fix a specific model of computation, and show about it that there are functions
it cannot compute or problems it cannot decide. We can show, for instance, that not every function can be computed by Turing machines, and not every problem can be decided by Turing machines. We can then appeal to the Church-Turing thesis to conclude that not only are Turing machines not powerful enough to compute every function, but no effective procedure can.

The key to proving such negative results is the fact that we can assign numbers to Turing machines themselves. The easiest way to do this is to enumerate them, perhaps by fixing a specific way to write down Turing machines and their programs, and then listing them in a systematic fashion. Once we see that this can be done, then the existence of Turing-uncomputable functions follows by simple cardinality considerations: the set of functions from \( \mathbb{N} \) to \( \mathbb{N} \) (in fact, even just from \( \mathbb{N} \) to \{0, 1\}) are non-enumerable, but since we can enumerate all the Turing machines, the set of Turing-computable functions is only denumerable.

We can also define specific functions and problems which we can prove to be uncomputable and undecidable, respectively. One such problem is the so-called Halting Problem. Turing machines can be finitely described by listing their instructions. Such a description of a Turing machine, i.e., a Turing machine program, can of course be used as input to another Turing machine. So we can consider Turing machines that decide questions about other Turing machines. One particularly interesting question is this: “Does the given Turing machine eventually halt when started on input \( n \)?” It would be nice if there were a Turing machine that could decide this question: think of it as a quality-control Turing machine which ensures that Turing machines don’t get caught in infinite loops and such. The interesting fact, which Turing proved, is that there cannot be such a Turing machine. There cannot be a single Turing machine which, when started on input consisting of a description of a Turing machine \( M \) and some number \( n \), will always halt with either output 1 or 0 according to whether \( M \) machine would have halted when started on input \( n \) or not.

Once we have examples of specific undecidable problems we can use them to show that other problems are undecidable, too. For instance, one celebrated undecidable problem is the question, “Is the first-order formula \( \varphi \) valid?” There is no Turing machine which, given as input a first-order formula \( \varphi \), is guaranteed to halt with output 1 or 0 according to whether \( \varphi \) is valid or not. Historically, the question of finding a procedure to effectively solve this problem was called simply “the” decision problem; and so we say that the decision problem is unsolvable. Turing and Church proved this result independently at around the same time, so it is also called the Church-Turing Theorem.

### 2.2 Enumerating Turing Machines

We can show that the set of all Turing machines is enumerable. This follows from the fact that each Turing machine can be finitely described. The set of states and the tape vocabulary are finite sets. The transition function is a partial function from \( Q \times \Sigma \) to \( Q \times \Sigma \times \{L, R, N\} \), and so likewise can be
specified by listing its values for the finitely many argument pairs for which it is defined. Of course, strictly speaking, the states and vocabulary can be anything; but the behavior of the Turing machine is independent of which objects serve as states and vocabulary. So we may assume, for instance, that the states and vocabulary symbols are natural numbers, or that the states and vocabulary are all strings of letters and digits.

Suppose we fix a denumerable vocabulary for specifying Turing machines: \( \sigma_0 = \triangledown, \sigma_1 = 0, \sigma_2 = 1, \sigma_3, \ldots, R, L, N, q_0, q_1, \ldots \). Then any Turing machine can be specified by some finite string of symbols from this alphabet (though not every finite string of symbols specifies a Turing machine). For instance, suppose we have a Turing machine \( M = \langle Q, \Sigma, q, \delta \rangle \) where

\[
Q = \{ q'_0, \ldots, q'_n \} \subseteq \{ q_0, q_1, \ldots \}
\]

\[
\Sigma = \{ \triangledown, \sigma'_1, \sigma'_2, \ldots, \sigma'_m \} \subseteq \{ \sigma_0, \sigma_1, \ldots \}.
\]

We could specify it by the string

\[
q'_0 q'_1 \ldots q'_n \triangledown \sigma'_1 \ldots \sigma'_m \triangledown q \triangledown S(q'_0, \sigma'_0) \triangledown \ldots \triangledown S(q'_n, \sigma'_m)
\]

where \( S(q'_j, \sigma'_i) \) is the string \( q'_j \sigma'_i \delta(q'_j, \sigma'_i) \) if \( \delta(q'_j, \sigma'_i) \) is defined, and \( q'_j \sigma'_i \) otherwise.

**Theorem 2.1.** There are functions from \( \mathbb{N} \to \mathbb{N} \) which are not Turing computable.

**Proof.** We know that the set of finite strings of symbols from a denumerable alphabet is enumerable. This gives us that the set of descriptions of Turing machines, as a subset of the finite strings from the enumerable vocabulary \( \{ q_0, q_1, \ldots, \triangledown, \sigma_1, \sigma_2, \ldots \} \), is itself enumerable. Since every Turing computable function is computed by some (in fact, many) Turing machines, this means that the set of all Turing computable functions from \( \mathbb{N} \to \mathbb{N} \) is also enumerable.

On the other hand, the set of all functions from \( \mathbb{N} \to \mathbb{N} \) is not enumerable. This follows immediately from the fact that not even the set of all functions of one argument from \( \mathbb{N} \) to the set \( \{ 0, 1 \} \) is enumerable. If all functions were computable by some Turing machine we could enumerate the set of all functions. So there are some functions that are not Turing computable.

**2.3 The Halting Problem**

Assume we have fixed some finite descriptions of Turing machines. Using these, we can enumerate Turing machines via their descriptions, say, ordered by the lexicographic ordering. Each Turing machine thus receives an index: its place in the enumeration \( M_1, M_2, M_3, \ldots \) of Turing machine descriptions.

We know that there must be non-Turing-computable functions: the set of Turing machine descriptions—and hence the set of Turing machines—is enumerable, but the set of all functions from \( \mathbb{N} \to \mathbb{N} \) is not. But we can find specific examples of non-computable function as well. One such function is the halting function.
Definition 2.2 (Halting function). The halting function $h$ is defined as
\[
h(e, n) = \begin{cases} 
0 & \text{if machine } M_e \text{ does not halt for input } n \\
1 & \text{if machine } M_e \text{ halts for input } n 
\end{cases}
\]

Definition 2.3 (Halting problem). The Halting Problem is the problem of determining (for any $e, n$) whether the Turing machine $M_e$ halts for an input of $n$ strokes.

We show that $h$ is not Turing-computable by showing that a related function, $s$, is not Turing-computable. This proof relies on the fact that anything that can be computed by a Turing machine can be computed using just two symbols: 0 and 1, and the fact that two Turing machines can be hooked together to create a single machine.

Definition 2.4. The function $s$ is defined as
\[
s(e) = \begin{cases} 
0 & \text{if machine } M_e \text{ does not halt for input } e \\
1 & \text{if machine } M_e \text{ halts for input } e 
\end{cases}
\]

Lemma 2.5. The function $s$ is not Turing computable.

Proof. We suppose, for contradiction, that the function $s$ is Turing computable. Then there would be a Turing machine $S$ that computes $s$. We may assume, without loss of generality, that when $S$ halts, it does so while scanning the first square. This machine can be “hooked up” to another machine $J$, which halts if it is started on a blank tape (i.e., if it reads 0 in the initial state while scanning the square to the right of the end-of-tape symbol), and otherwise wanders off to the right, never halting. $S \sim J$, the machine created by hooking $S$ to $J$, is a Turing machine, so it is $M_e$ for some $e$ (i.e., it appears somewhere in the enumeration). Start $M_e$ on an input of $e$ 1s. There are two possibilities: either $M_e$ halts or it does not halt.

1. Suppose $M_e$ halts for an input of $e$ 1s. Then $s(e) = 1$. So $S$, when started on $e$, halts with a single 1 as output on the tape. Then $J$ starts with a 1 on the tape. In that case $J$ does not halt. But $M_e$ is the machine $S \sim J$, so it should do exactly what $S$ followed by $J$ would do. So $M_e$ cannot halt for an input of 1’s.

2. Now suppose $M_e$ does not halt for an input of $e$ 1s. Then $s(e) = 0$, and $S$, when started on input $e$, halts with a blank tape. $J$, when started on a blank tape, immediately halts. Again, $M_e$ does what $S$ followed by $J$ would do, so $M_e$ must halt for an input of $e$ 1’s.

This shows there cannot be a Turing machine $S$: $s$ is not Turing computable.

Theorem 2.6 (Unsolvability of the Halting Problem). The halting problem is unsolvable, i.e., the function $h$ is not Turing computable.
Proof. Suppose \( h \) were Turing computable, say, by a Turing machine \( H \). We could use \( H \) to build a Turing machine that computes \( s \): First, make a copy of the input (separated by a blank). Then move back to the beginning, and run \( H \). We can clearly make a machine that does the former, and if \( H \) existed, we would be able to “look it up” to such a modified doubling machine to get a new machine which would determine if \( M_e \) halts on input \( e \), i.e., computes \( s \). But we’ve already shown that no such machine can exist. Hence, \( h \) is also not Turing computable.

**Problem 2.1.** The Three Halting (3-Halt) problem is the problem of giving a decision procedure to determine whether or not an arbitrarily chosen Turing Machine halts for an input of three strokes on an otherwise blank tape. Prove that the 3-Halt problem is unsolvable.

**Problem 2.2.** Show that if the halting problem is solvable for Turing machine and input pairs \( M_e \) and \( n \) where \( e \neq n \), then it is also solvable for the cases where \( e = n \).

**Problem 2.3.** We proved that the halting problem is unsolvable if the input is a number \( e \), which identifies a Turing machine \( M_e \) via an enumeration of all Turing machines. What if we allow the description of Turing machines from section 2.2 directly as input? (This would require a larger alphabet of course.) Can there be a Turing machine which decides the halting problem but takes as input descriptions of Turing machines rather than indices? Explain why or why not.

### 2.4 The Decision Problem

We say that first-order logic is *decidable* iff there is an effective method for determining whether or not a given sentence is valid. As it turns out, there is no such method: the problem of deciding validity of first-order sentences is unsolvable.

In order to establish this important negative result, we prove that the decision problem cannot be solved by a Turing machine. That is, we show that there is no Turing machine which, whenever it is started on a tape that contains a first-order sentence, eventually halts and outputs either 1 or 0 depending on whether the sentence is valid or not. By the Church-Turing thesis, every function which is computable is Turing computable. So if this “validity function” were effectively computable at all, it would be Turing computable. If it isn’t Turing computable, then, it also cannot be effectively computable.

Our strategy for proving that the decision problem is unsolvable is to reduce the halting problem to it. This means the following: We have proved that the function \( h(e, w) \) that halts with output 1 if the Turing machine described by \( e \) halts on input \( w \) and outputs 0 otherwise, is not Turing computable. We will show that if there were a Turing machine that decides validity of first-order sentences, then there is also a Turing machine that computes \( h \). Since \( h \) cannot
be computed by a Turing machine, there cannot be a Turing machine that
decides validity either.

The first step in this strategy is to show that for every input \( w \) and a Turing
machine \( M \), we can effectively describe a sentence \( \tau(M, w) \) representing the
instruction set of \( M \) and the input \( w \) and a sentence \( \alpha(M, w) \) expressing “\( M \)
eventually halts” such that:

\[
\models \tau(M, w) \rightarrow \alpha(M, w) \text{ iff } M \text{ halts for input } w.
\]

The bulk of our proof will consist in describing these sentences \( \tau(M, w) \) and
\( \alpha(M, w) \) and verifying that \( \tau(M, w) \rightarrow \alpha(M, w) \) is valid iff \( M \) halts on input \( w \).

### 2.5 Representing Turing Machines

In order to represent Turing machines and their behavior by a sentence of first-order
logic, we have to define a suitable language. The language consists of
two parts: predicate symbols for describing configurations of the machine, and
expressions for numbering execution steps (“moments”) and positions on the
tape.

We introduce two kinds of predicate symbols, both of them 2-place: For
each state \( q \), a predicate symbol \( Q_q \), and for each tape symbol \( \sigma \), a predicate
symbol \( S_\sigma \). The former allow us to describe the state of \( M \) and the position of
its tape head, the latter allow us to describe the contents of the tape.

In order to express the positions of the tape head and the number of steps
executed, we need a way to express numbers. This is done using a constant
symbol \( 0 \), and a 1-place function \( \prime \), the successor function. By convention it
is written after its argument (and we leave out the parentheses). \( 0 \) names
the leftmost position on the tape as well as the time before the first execution
step (the initial configuration), \( 0' \) names the square to the right of the leftmost
square, and the time after the first execution step, and so on. We also introduce
a predicate symbol \( < \) to express both the ordering of tape positions (when it
means “to the left of”) and execution steps (then it means “before”).

Once we have the language in place, we list the “axioms” of \( \tau(M, w) \), i.e.,
the sentences which, taken together, describe the behavior of \( M \) when run on
input \( w \). There will be sentences which lay down conditions on \( 0, \prime \), and \( < \),
sentences that describes the input configuration, and sentences that describe
what the configuration of \( M \) is after it executes a particular instruction.

**Definition 2.7.** Given a Turing machine \( M = \langle Q, \Sigma, q_0, \delta \rangle \), the language \( L_M \)
consists of:

1. A two-place predicate symbol \( Q_q(x, y) \) for every state \( q \in Q \). Intuitively,
   \( Q_q(m, \bar{m}) \) expresses “after \( n \) steps, \( M \) is in state \( q \) scanning the \( m \)th
   square.”

2. A two-place predicate symbol \( S_\sigma(x, y) \) for every symbol \( \sigma \in \Sigma \). Intuitively,
   \( S_\sigma(m, \bar{m}) \) expresses “after \( n \) steps, the \( m \)th square contains symbol \( \sigma \).”
3. A constant symbol $\circ$

4. A one-place function symbol $'$

5. A two-place predicate symbol $<$

For each number $n$ there is a canonical term $\pi$, the *numeral* for $n$, which represents it in $\mathcal{L}_M$. $\overline{0}$ is $\circ$, $\overline{1}$ is $\circ'$, $\overline{2}$ is $\circ''$, and so on. More formally:

\[
\overline{0} = \circ
\]
\[
\overline{n + 1} = \pi'
\]

The sentences describing the operation of the Turing machine $M$ on input $w = \sigma_1 \ldots \sigma_k$ are the following:

1. Axioms describing numbers:
   a) A sentence that says that the successor function is injective:
      \[
      \forall x \forall y (x' = y' \rightarrow x = y)
      \]
   b) A sentence that says that every number is less than its successor:
      \[
      \forall x x < x'
      \]
   c) A sentence that ensures that $<$ is transitive:
      \[
      \forall x \forall y \forall z ((x < y \land y < z) \rightarrow x < z)
      \]
   d) A sentence that connects $<$ and $=$:
      \[
      \forall x \forall y (x < y \rightarrow x \neq y)
      \]

2. Axioms describing the input configuration:
   a) After 0 steps—before the machine starts—$M$ is in the initial state $q_0$, scanning square 1:
      \[
      Q_{q_0}(\overline{1}, \overline{0})
      \]
   b) The first $k + 1$ squares contain the symbols $\sigma_1, \ldots, \sigma_k$:
      \[
      S_{\circ}(\overline{0}, \overline{0}) \land S_{\sigma_1}(\overline{1}, \overline{0}) \land \cdots \land S_{\sigma_k}(\overline{k}, \overline{0})
      \]
   c) Otherwise, the tape is empty:
      \[
      \forall x (\overline{k} < x \rightarrow S_0(x, \overline{0}))
      \]
3. Axioms describing the transition from one configuration to the next:

For the following, let $\varphi(x, y)$ be the conjunction of all sentences of the form

$$\forall z (((z < x \lor x < z) \land S_\sigma(z, y)) \rightarrow S_\sigma(z, y'))$$

where $\sigma \in \Sigma$. We use $\varphi(m, n)$ to express “other than at square $m$, the tape after $n+1$ steps is the same as after $n$ steps.”

a) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', R \rangle$, the sentence:

$$\forall x \forall y (((Q_{q_i}(x, y) \land S_\sigma(x, y)) \rightarrow \quad (Q_{q_j}(x', y') \land S_{\sigma'}(x', y') \land \varphi(x, y))))$$

This says that if, after $y$ steps, the machine is in state $q_i$ scanning square $x$ which contains symbol $\sigma$, then after $y+1$ steps it is scanning square $x+1$, is in state $q_j$, square $x$ now contains $\sigma'$, and every square other than $x$ contains the same symbol as it did after $y$ steps.

b) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', L \rangle$, the sentence:

$$\forall x \forall y (((Q_{q_i}(x, y) \land S_\sigma(x, y)) \rightarrow \quad (Q_{q_j}(x, y') \land S_{\sigma'}(x, y') \land \varphi(x, y))) \land \forall y \quad (((Q_{q_i}(0, y) \land S_\sigma(0, y)) \rightarrow \quad (Q_{q_j}(0, y') \land S_{\sigma'}(0, y') \land \varphi(0, y))))$$

Take a moment to think about how this works: now we don’t start with “if scanning square $x$ . . .” but: “if scanning square $x+1$ . . .” A move to the left means that in the next step the machine is scanning square $x$. But the square that is written on is $x+1$. We do it this way since we don’t have subtraction or a predecessor function.

Note that numbers of the form $x+1$ are $1, 2, \ldots$, i.e., this doesn’t cover the case where the machine is scanning square $0$ and is supposed to move left (which of course it can’t—it just stays put). That special case is covered by the second conjunction: it says that if, after $y$ steps, the machine is scanning square $0$ in state $q_i$ and square $0$ contains symbol $\sigma$, then after $y+1$ steps it’s still scanning square $0$, is now in state $q_j$, the symbol on square $0$ is $\sigma'$, and the squares other than square $0$ contain the same symbols they contained after $y$ steps.

c) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', N \rangle$, the sentence:

$$\forall x \forall y (((Q_{q_i}(x, y) \land S_\sigma(x, y)) \rightarrow \quad (Q_{q_j}(x, y') \land S_{\sigma'}(x, y') \land \varphi(x, y))))$$

Let $\tau(M, w)$ be the conjunction of all the above sentences for Turing machine $M$ and input $w$. 

*turing-machines rev: 2c33e9e (2021-03-08) by OLP / CC–BY*
In order to express that $M$ eventually halts, we have to find a sentence that says “after some number of steps, the transition function will be undefined.” Let $X$ be the set of all pairs $⟨q, σ⟩$ such that $δ(q, σ)$ is undefined. Let $α(M, w)$ then be the sentence

$$\exists x \exists y ( \bigvee_{⟨q, σ⟩ \in X} (Q_q(x, y) \land S_σ(x, y)))$$

If we use a Turing machine with a designated halting state $h$, it is even easier: then the sentence $α(M, w)$

$$\exists x \exists y Q_h(x, y)$$

expresses that the machine eventually halts.

**Proposition 2.8.** If $m < k$, then $τ(M, w) ⊨ m < k$

**Proof.** Exercise.

**Problem 2.4.** Prove Proposition 2.8. (Hint: use induction on $k − m$).

### 2.6 Verifying the Representation

In order to verify that our representation works, we have to prove two things. First, we have to show that if $M$ halts on input $w$, then $τ(M, w) → α(M, w)$ is valid. Then, we have to show the converse, i.e., that if $τ(M, w) → α(M, w)$ is valid, then $M$ does in fact eventually halt when run on input $w$.

The strategy for proving these is very different. For the first result, we have to show that a sentence of first-order logic (namely, $τ(M, w) → α(M, w)$) is valid. The easiest way to do this is to give a derivation. Our proof is supposed to work for all $M$ and $w$, though, so there isn’t really a single sentence for which we have to give a derivation, but infinitely many. So the best we can do is to prove by induction that, whatever $M$ and $w$ look like, and however many steps it takes $M$ to halt on input $w$, there will be a derivation of $τ(M, w) → α(M, w)$.

Naturally, our induction will proceed on the number of steps $M$ takes before it reaches a halting configuration. In our inductive proof, we’ll establish that for each step $n$ of the run of $M$ on input $w$, $τ(M, w) ⊨ χ(M, w, n)$, where $χ(M, w, n)$ correctly describes the configuration of $M$ run on $w$ after $n$ steps. Now if $M$ halts on input $w$ after, say, $n$ steps, $χ(M, w, n)$ will describe a halting configuration. We’ll also show that $χ(M, w, n) ⊨ α(M, w)$, whenever $χ(M, w, n)$ describes a halting configuration. So, if $M$ halts on input $w$, then for some $n$, $M$ will be in a halting configuration after $n$ steps. Hence, $τ(M, w) ⊨ \chi(M, w, n)$ where $χ(M, w, n)$ describes a halting configuration, and since in that case $χ(M, w, n) ⊨ α(M, w)$, we get that $T(M, w) ⊨ α(M, w)$, i.e., that $τ(M, w) → α(M, w)$.

The strategy for the converse is very different. Here we assume that $τ(M, w) → α(M, w)$ and have to prove that $M$ halts on input $w$. From the hypothesis we get that $τ(M, w) ⊨ α(M, w)$, i.e., $α(M, w)$ is true in every structure...
Lemma 2.10. If \( M \) run on input \( w \) is in a halting configuration after \( n \) steps, then \( \chi(M, w, n) \models \alpha(M, w) \).

Proof. Suppose that \( M \) halts for input \( w \) after \( n \) steps. There is some state \( q \), square \( m \), and symbol \( \sigma \) such that:

1. After \( n \) steps, \( M \) is in state \( q \) scanning square \( m \) on which \( \sigma \) appears.

2. The transition function \( \delta(q, \sigma) \) is undefined.

\( \chi(M, w, n) \) is the description of this configuration and will include the clauses \( Q_q(m, n) \) and \( S_\sigma(m, n) \). These clauses together imply \( \alpha(M, w) \):

\[
\exists x \exists y \left( \bigvee_{(q', \sigma') \in X} (Q_{q'}(m, n) \land S_{\sigma'}(m, n)) \right)
\]

since \( Q_{q'}(m, n) \land S_{\sigma'}(m, n) \models \bigvee_{(q, \sigma) \in X} (Q_q(m, n) \land S_{\sigma}(m, n)), \) as \( (q', \sigma') \in X \). \( \square \)

Lemma 2.11. For each \( n \), if \( M \) has not halted after \( n \) steps, \( \tau(M, w) \models \chi(M, w, n) \).

Proof. Induction basis: If \( n = 0 \), then the conjuncts of \( \chi(M, w, 0) \) are also conjuncts of \( \tau(M, w) \), so entailed by it.

Inductive hypothesis: If \( M \) has not halted before the \( n \)th step, then \( \tau(M, w) \models \chi(M, w, n) \). We have to show that (unless \( \chi(M, w, n) \) describes a halting configuration), \( \tau(M, w) \models \chi(M, w, n + 1) \).

Suppose \( n > 0 \) and after \( n \) steps, \( M \) started on \( w \) is in state \( q \) scanning square \( m \). Since \( M \) does not halt after \( n \) steps, there must be an instruction of one of the following three forms in the program of \( M \):

\[ turing-machines \text{ rev: 2c33e9e (2021-03-08) by OLP / CC–BY} \]
1. \( \delta(q, \sigma) = \langle q', \sigma', R \rangle \)
2. \( \delta(q, \sigma) = \langle q', \sigma', L \rangle \)
3. \( \delta(q, \sigma) = \langle q', \sigma', N \rangle \)

We will consider each of these three cases in turn.

1. Suppose there is an instruction of the form (1). By Definition 2.7(3a), this means that
   \[
   \forall x \forall y ((Q_q(x, y) \land S_\sigma(x, y)) \rightarrow (Q_{q'}(x', y') \land S_{\sigma'}(x', y') \land \varphi(x, y)))
   \]
   is a conjunct of \( \tau(M, w) \). This entails the following sentence (universal instantiation, \( m \) for \( x \) and \( n \) for \( y \)):
   \[
   (Q_q(m, n) \land S_\sigma(m, n)) \rightarrow (Q_{q'}(m', n') \land S_{\sigma'}(m, n) \land \varphi(m, n)).
   \]
   By induction hypothesis, \( \tau(M, w) \models \chi(M, w, n) \), i.e.,
   \[
   Q_q(m, n) \land S_{\sigma_0}(0, n) \land \cdots \land S_{\sigma_k}(k, n) \land \forall x (k < x \rightarrow S_0(x, n))
   \]
   Since after \( n \) steps, tape square \( m \) contains \( \sigma \), the corresponding conjunct is \( S_\sigma(m, n) \), so this entails:
   \[
   Q_q(m, n) \land S_\sigma(m, n))
   \]
   We now get
   \[
   Q_{q'}(m', n') \land S_{\sigma'}(m, n') \land S_{\sigma_0}(0, n') \land \cdots \land S_{\sigma_k}(k, n') \land \forall x (k < x \rightarrow S_0(x, n'))
   \]
   as follows: The first line comes directly from the consequent of the preceding conditional, by modus ponens. Each conjunct in the middle line—which excludes \( S_{\sigma_m}(m, n) \)—follows from the corresponding conjunct in \( \chi(M, w, n) \) together with \( \varphi(m, n) \).

   If \( m < k \), \( \tau(M, w) \vdash m < k \) (Proposition 2.8) and by transitivity of \( < \), we have \( \forall x (k < x \rightarrow \overline{m} < x) \). If \( m = k \), then \( \forall x (k < x \rightarrow \overline{m} < x) \) by logic alone. The last line then follows from the corresponding conjunct in \( \chi(M, w, n) \), \( \forall x (k < x \rightarrow \overline{m} < x) \), and \( \varphi(m, n) \). If \( m < k \), this already is \( \chi(M, w, n + 1) \).

   Now suppose \( m = k \). In that case, after \( n + 1 \) steps, the tape head has also visited square \( k + 1 \), which now is the right-most square visited. So
\[\chi(M, w, n + 1)\] has a new conjunct, \(S_0(k', \pi')\), and the last conjunct is \(\forall x (k' < x \rightarrow S_0(x, \pi'))\). We have to verify that these two sentences are also implied.

We already have \(\forall x (k < x \rightarrow S_0(x, \pi'))\). In particular, this gives us \(k < k' \rightarrow S_0(k', \pi')\). From the axiom \(\forall x x < x'\) we get \(k < k'\). By modus ponens, \(S_0(k', \pi')\) follows.

Also, since \(\tau(M, w) \vdash k < k'\), the axiom for transitivity of \(<\) gives us \(\forall x (k' < x \rightarrow S_0(x, \pi'))\). (We leave the verification of this as an exercise.)

2. Suppose there is an instruction of the form (2). Then, by Definition 2.7(3b),

\[
\forall x \forall y \left( (Q_q(x', y) \land S_\sigma(x', y)) \rightarrow (Q_q'(x', y') \land S_\sigma'(x', y') \land \varphi(x, y)) \right) \land \\
\forall y \left( (Q_q(o, y) \land S_\sigma(o, y)) \rightarrow (Q_q'(o, y') \land S_\sigma'(o, y') \land \varphi(o, y)) \right)
\]

is a conjunct of \(\tau(M, w)\). If \(m > 0\), then let \(l = m - 1\) (i.e., \(m = l + 1\)). The first conjunct of the above sentence entails the following:

\[
(Q_q(l', \pi') \land S_\sigma(l', \pi')) \rightarrow \\
(Q_q'(l, \pi') \land S_\sigma'(l', \pi') \land \varphi(l, \pi))
\]

Otherwise, let \(l = m = 0\) and consider the following sentence entailed by the second conjunct:

\[
\left( (Q_q(o, \pi) \land S_\sigma(o, \pi)) \rightarrow (Q_q(o, \pi') \land S_\sigma'(o, \pi') \land \varphi(o, \pi)) \right)
\]

Either sentence implies

\[
Q_q'(l, \pi') \land S_\sigma'(\tilde{m}, \pi') \land \\
S_{\sigma_0}(\tilde{m}, \pi') \land \cdots \land S_{\sigma_k}(\overline{k}, \pi') \land \\
\forall x (k < x \rightarrow S_0(x, \pi'))
\]

as before. (Note that in the first case, \(l' \equiv l + 1 \equiv \overline{m}\) and in the second case \(l \equiv 0\).) But this just is \(\chi(M, w, n + 1)\).

3. Case (3) is left as an exercise.

We have shown that for any \(n\), \(\tau(M, w) \vdash \chi(M, w, n)\). \(\Box\)

**Problem 2.5.** Complete case (3) of the proof of Lemma 2.11.
**Problem 2.6.** Give a derivation of $S_{\sigma_{i}(i, n')}$ from $S_{\sigma_{i}(i, n)}$ and $\varphi(m, n)$ (assuming $i \neq m$, i.e., either $i < m$ or $m < i$).

**Problem 2.7.** Give a derivation of $\forall x (K < x \rightarrow S_{0}(x, n'))$ from $\forall x (K < x \rightarrow S_{0}(x, n'))$, $\forall x x < x'$, and $\forall x \forall y \forall z ((x < y \land y < z) \rightarrow x < z)$.

**Lemma 2.12.** If $M$ halts on input $w$, then $\tau(M, w) \rightarrow \alpha(M, w)$ is valid.

*Proof.* By Lemma 2.11, we know that, for any time $n$, the description $\chi(M, w, n)$ of the configuration of $M$ at time $n$ is entailed by $\tau(M, w)$. Suppose $M$ halts after $k$ steps. It will be scanning square $m$, say. Then $\chi(M, w, k)$ describes a halting configuration of $M$, i.e., it contains as conjuncts both $Q_{q}(m, k)$ and $S_{\sigma}(m, k)$ with $\delta(q, \sigma)$ undefined. Thus, by Lemma 2.10, $\chi(M, w, k) \models \alpha(M, w)$. But since $\tau(M, w) \models \chi(M, w, k)$, we have $\tau(M, w) \models \alpha(M, w)$ and therefore $\tau(M, w) \rightarrow \alpha(M, w)$ is valid.

To complete the verification of our claim, we also have to establish the reverse direction: if $\tau(M, w) \rightarrow \alpha(M, w)$ is valid, then $M$ does in fact halt when started on input $m$.

**Lemma 2.13.** If $\models \tau(M, w) \rightarrow \alpha(M, w)$, then $M$ halts on input $w$.

*Proof.* Consider the $L_{M}$-structure $\mathfrak{M}$ with domain $\mathbb{N}$ which interprets $\circ$ as 0, $'$ as the successor function, and $<$ as the less-than relation, and the predicates $Q_{q}$ and $S_{\sigma}$ as follows:

- $Q_{q}^{\mathfrak{M}} = \{ (m, n) : \text{started on $w$, after $n$ steps, $M$ is in state $q$ scanning square $m$} \}$
- $S_{\sigma}^{\mathfrak{M}} = \{ (m, n) : \text{started on $w$, after $n$ steps, square $m$ of $M$ contains symbol $\sigma$} \}$

In other words, we construct the structure $\mathfrak{M}$ so that it describes what $M$ started on input $w$ actually does, step by step. Clearly, $\mathfrak{M} \models \tau(M, w)$. If $\models \tau(M, w) \rightarrow \alpha(M, w)$, then also $\mathfrak{M} \models \alpha(M, w)$, i.e.,

$$\mathfrak{M} \models \exists x \exists y ( \bigvee_{(q, \sigma) \in X} (Q_{q}(x, y) \land S_{\sigma}(x, y))).$$

As $|\mathfrak{M}| = \mathbb{N}$, there must be $m$, $n \in \mathbb{N}$ so that $\mathfrak{M} \models Q_{q}(m, n) \land S_{\sigma}(m, n)$ for some $q$ and $\sigma$ such that $\delta(q, \sigma)$ is undefined. By the definition of $\mathfrak{M}$, this means that $M$ started on input $w$ after $n$ steps is in state $q$ and reading symbol $\sigma$, and the transition function is undefined, i.e., $M$ has halted. \hfill $\square$

### 2.7 The Decision Problem is Unsolvable

**Theorem 2.14.** The decision problem is unsolvable.
Proof. Suppose the decision problem were solvable, i.e., suppose there were a Turing machine $D$ of the following sort. Whenever $D$ is started on a tape that contains a sentence $\psi$ of first-order logic as input, $D$ eventually halts, and outputs 1 iff $\psi$ is valid and 0 otherwise. Then we could solve the halting problem as follows. We construct a Turing machine $E$ that, given as input the number $e$ of Turing machine $M_e$ and input $w$, computes the corresponding sentence $\tau(M_e, w) \to \alpha(M_e, w)$ and halts, scanning the leftmost square on the tape. The machine $E \vdash D$ would then, given input $e$ and $w$, first compute $\tau(M_e, w) \to \alpha(M_e, w)$ and then run the decision problem machine $D$ on that input. $D$ halts with output 1 iff $\tau(M_e, w) \to \alpha(M_e, w)$ is valid and outputs 0 otherwise. By Lemma 2.13 and Lemma 2.12, $\tau(M_e, w) \to \alpha(M_e, w)$ is valid iff $M_e$ halts on input $w$. Thus, $E \vdash D$, given input $e$ and $w$ halts with output 1 iff $M_e$ halts on input $w$ and halts with output 0 otherwise. In other words, $E \vdash D$ would solve the halting problem. But we know, by Theorem 2.6, that no such Turing machine can exist. \qed
Bibliography