Part I

Turing Machines

Chapter 1

Turing Machine Computations

1.1 Introduction

tur:mac:int:sec

What does it mean for a function, say, from $\mathbb{N}$ to $\mathbb{N}$ to be *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 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.

explanation

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 *tape* and a *read-write head*. In our version of Turing machines, the tape is infinite in one direction (to the right), and it is divided into *squares*, each of which may contain a symbol from a finite *alphabet*. Such alphabets can contain any number of different symbols, 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 *input*. At any time, the mechanism is in one of a finite number of *states*. At the outset, the head scans the leftmost square and in a specified *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 σ and outputs a triple $\langle q', \sigma', D\rangle$. Whenever the mechanism is in

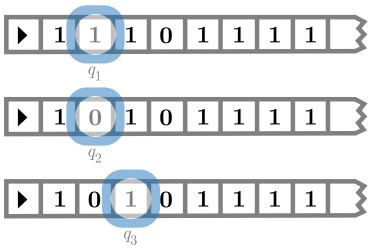


Figure 1.1: A Turing machine executing its program.

tur:mac:int:
fig:tm

state q and reads symbol σ, it replaces the symbol on the current square with σ', 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 $\langle q_1, 1\rangle$, the triple $\langle q_2, 0, N\rangle$, 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 $\langle q_3, 0, R\rangle$ for input $\langle q_2, 0\rangle$, 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, σ such that there is no instruction for $\langle q_n, \sigma\rangle$, i.e., the transition function for input $\langle q_n, \sigma\rangle$ 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): digression

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."

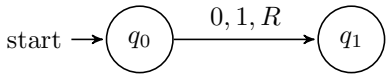
Historical Remarks Alan Turing invented Turing machines in 1936. While his interest at the time was the decidability of first-order logic, the paper has been described as a definitive paper on the foundations of computer design. In the paper, Turing focuses on computable real numbers, i.e., real numbers 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

explanation

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:

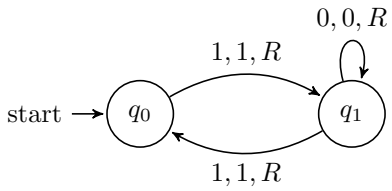
tur:mac:rep:sec



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:

$$\delta(q_0, 1) = \langle q_1, 1, R\rangle,$$
$$\delta(q_1, 1) = \langle q_0, 1, R\rangle,$$
$$\delta(q_1, 0) = \langle q_1, 0, R\rangle$$

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. explanation

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:

$$\triangleright 1_0 1110\ldots$$

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.

$$\triangleright 11_1 110\ldots$$

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

$$\triangleright 111_0 10\ldots$$

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

$$\triangleright 1111_1 0\ldots$$

$$\triangleright 11110_0 \ldots$$

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:

$$\triangleright 1_0 110 \ldots$$

$$\triangleright 11_1 10 \ldots$$

$$\triangleright 111_0 0 \ldots$$

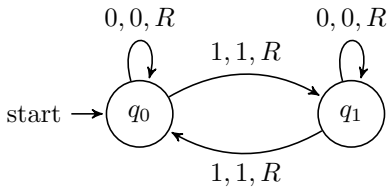
$$\triangleright 1110_1 \ldots$$

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 $\delta(q_1, 0) = \langle q_1, 0, R \rangle$. 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.

explanation

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.



explanation

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.

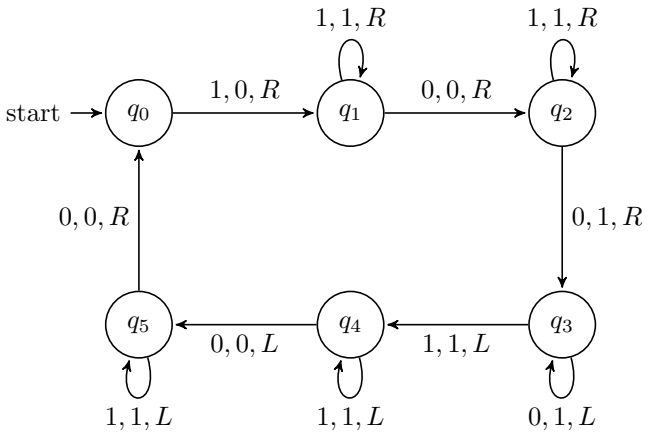


Figure 1.2: A doubler machine

tur:mac:rep: fig:doubler

Example 1.3. The machine table for the even machine is:

	0	1	▷
q_0		$1, q_1, R$	
q_1	$0, q_1, R$	$1, q_0, R$	

As we can see, the machine halts when scanning a 0 in state q_0.

explanation

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.

tur:mac:rep: ex:doubler

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. Design a Turing-machine with alphabet $\{\triangleright, 0, A, B\}$ 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 $AABB$, and $AAABBB$, but reject both AAB and $AABBAABB$.)

Problem 1.3. Design a Turing-machine with alphabet $\{\triangleright, 0, A, B\}$ that takes as input any string α 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 1.4. *Alphabetical?:* Design a Turing-machine with alphabet $\{\triangleright, 0, A, B\}$ 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 1.5. *Alphabetizer:* Design a Turing-machine with alphabet $\{\triangleright, 0, A, B\}$ 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

explanation

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.

tur:mac:tur:sec

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* Σ which includes $\triangleright$ and 0,
3. an *initial state* $q_0 \in Q$,
4. a finite *instruction set* $\delta\colon Q \times \Sigma \nrightarrow Q \times \Sigma \times \{L, R, N\}$.

The partial function δ is also called the *transition function* of M.

explanation

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 textbooks do this, we do not. You can simply be careful when constructing your Turing machine that it never 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

$$\begin{aligned} Q &= \{q_0, q_1\} \\ \Sigma &= \{\triangleright, 0, 1\}, \\ \delta(q_0, 1) &= \langle q_1, 1, R\rangle, \\ \delta(q_1, 1) &= \langle q_0, 1, R\rangle, \\ \delta(q_1, 0) &= \langle q_1, 0, R\rangle. \end{aligned}$$

1.4 Configurations and Computations

cmp:tur:con:sec

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. explanation

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 Σ,
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 explanation

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 $I \in \Sigma^*$ is

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

explanation The $\frown$ symbol is for *concatenation*—the input string begins immediately to the left end marker.

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 σ,
2. the instruction set of M specifies $\delta(q, \sigma) = \langle q', \sigma', D \rangle$,
3. the m-th symbol of C' is σ', 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. cmp:tur:con: defn:run-output

We say that M *halts on input I after k steps* if $C_k = \langle C, m, q \rangle$, the mth symbol of C is σ, 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 ending in 0 such that $C = \triangleright \frown O \frown 0^j$ for some $j \in \mathbb{N}$. (0^j is a sequence of j 0's.)

explanation According to this definition, the output O of M always 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

explanation 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 *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. tur:mac:una: sec

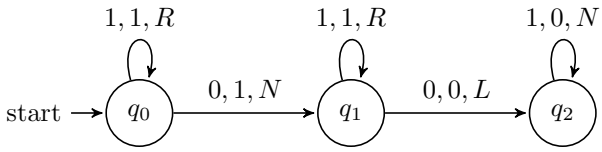


Figure 1.3: A machine computing $f(x, y) = x + y$

tur:mac:una: fig:adder

Definition 1.11 (Computation). A Turing machine M *computes* the function $f\colon \mathbb{N}^k \to \mathbb{N}$ iff M halts on input

$$1^{n_1}01^{n_2}0\ldots01^{n_k}$$

with output $1^{f(n_1,\ldots,n_k)}$.

Problem 1.6. Give a definition for when a Turing machine M computes the function $f\colon \mathbb{N}^k \to \mathbb{N}^m$.

tur:mac:una: ex:adder

Example 1.12. *Addition:* Let's build a machine that computes the function $f(n, m) = n + m$. This requires a machine that starts with two blocks of 1's of length n and m on the tape, and halts with one block consisting of $n + m$ 1's. The two input blocks of 1's are separated by a 0, so one method would be to write a stroke on the square containing the 0, and erase the last 1.

Problem 1.7. Trace through the configurations of the machine from Example 1.12 for input $\langle 3, 2\rangle$. What happens if the machine computes $0 + 0$?

explanation

In Example 1.4, we gave an example of a Turing machine that takes as input a sequence of 1's and halts with a sequence of twice as many 1's on the tape—the doubler machine. However, because the output contains 0's to the left of the doubled block of 1's, it does not actually compute the function $f(x) = 2x$, as you might have assumed. We'll describe two ways of fixing that.

Example 1.13. The machine in Figure 1.4 computes the function $f(x) = 2x$. Instead of erasing the input and writing two 1's at the far right for every 1 in the input as the machine from Example 1.4 does, this machine adds a single 1 to the right for every 1 in the input. It has to keep track of where the input ends, so it leaves a 0 between the input and the added strokes, which it fills with a 1 at the very end. And we have to "remember" where we are in the input, so we temporarily replace a 1 in the input block by a 0.

tur:mac:una: ex:mover

Example 1.14. A second possibility for computing $f(x) = 2x$ is to keep the original doubler machine, but add states and instructions at the end which move the doubled block of strokes to the far left of the tape. The machine in Figure 1.5 does just this last part: started on a tape consisting of a block of 0's

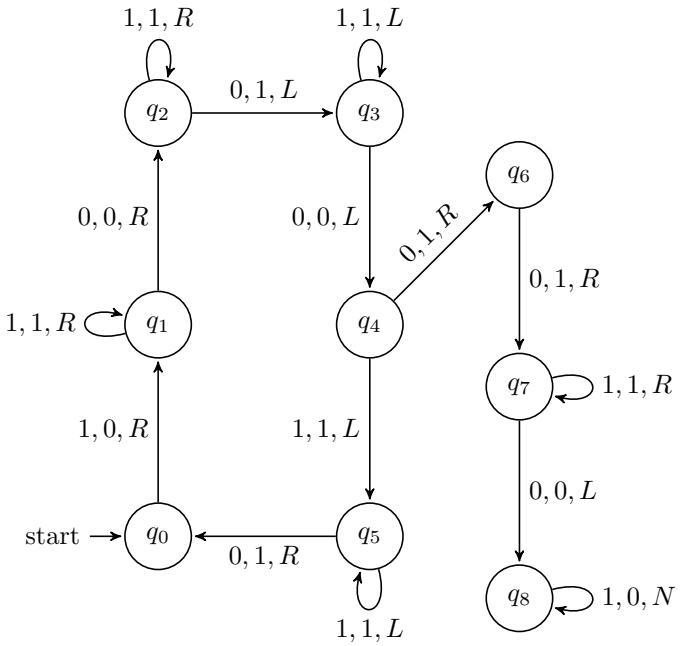


Figure 1.4: A machine computing $f(x) = 2x$

tur:mac:una:
fig:doubler-disc

followed by a block of 1's (and the head positioned anywhere in the block of 0's), it erases the 1's one at a time and writes them at the beginning of the tape. In order to be able to tell when it is done, it first marks the end of the block of 1's with a ▷ symbol, which gets deleted at the end. We've started numbering the states at q_6, so they can be added to the doubler machine. All you'll need is an additional instruction $\delta(q_0, 0) = \langle q_6, 0, N\rangle$, i.e., an arrow from q_0 to q_6 labelled $0, 0, N$. (There is one subtle problem: the resulting machine does not work for input $x = 0$. We'll leave this as an exercise.)

Problem 1.8. In Example 1.14 we described a machine consisting of a combination of the doubler machine from Figure 1.4 and the mover machine from Figure 1.5. What happens if you start this combined machine on input $x = 0$, i.e., on an empty tape? How would you fix the machine so that in this case the machine halts with output $2x = 0$? (You should be able to do this by adding one state and one transition.)

Problem 1.9. *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 $f(n, m) = n - m$.

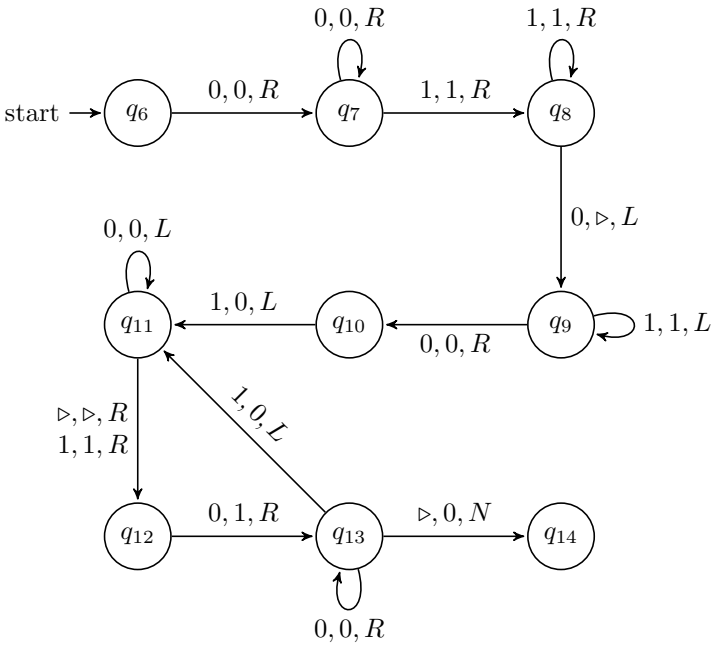


Figure 1.5: Moving a block of 1's to the left

tur:mac:una:fig:mover

Problem 1.10. *Equality:* Design a Turing machine to compute the following function:

$$\text{equality}(n, m) = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{if } n \neq m \end{cases}$$

where n and $m \in \mathbb{Z}^+$.

Problem 1.11. 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.

Definition 1.15. A Turing machine M computes the partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ iff,

1. M halts on input $1^{n_1} \frown 0 \frown \ldots \frown 0 \frown 1^{n_k}$ with output 1^m if $f(n_1, \ldots, n_k) = m$.
2. M does not halt at all, or with an output that is not a single block of 1's if $f(n_1, \ldots, n_k)$ is undefined.

1.6 Halting States

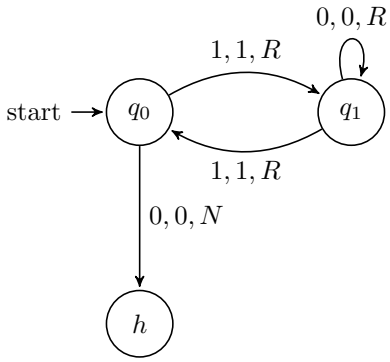
explanation

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 $h \in Q$.

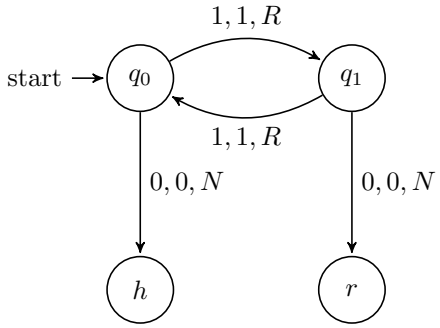
tur:mac:hal:sec

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 1.16. *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 halting 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 r.



explanation

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 Disciplined Machines

tur:mac:dis:sec

In section section 1.6, we considered Turing machines that have a single, designated halting state h—such machines are guaranteed to halt, if they halt at all, in state h. In this way, machines with a single halting state are more "disciplined" than we allow Turing machines in general to be. There are other restrictions we might impose on the behavior of Turing machines. For instance, we also have not prohibited Turing machines from ever erasing the tape-end marker on square 0, or to attempt to move left from square 0. (Our definition states that the head simply stays on square 0 in this case; other definitions have the machine halt.) It is likewise sometimes desirable to be able to assume that a Turing machine, if it halts at all, halts on square 1. explanation

tur:mac:dis:defn:disciplined

Definition 1.17. A Turing machine M is *disciplined* iff

1. it has a designated single halting state h,
2. it halts, if it halts at all, while scanning square 1,
3. it never erases the ▷ symbol on square 0, and
4. it never attempts to move left from square 0.

We have already discussed that any Turing machine can be changed into one with the same behavior but with a designated halting state. This is done simply by adding a new state h, and adding an instruction $\delta(q, \sigma) = \langle h, \sigma, N\rangle$ for any pair $\langle q, \sigma\rangle$ where the original δ is undefined. It is true, although tedious to prove, that any Turing machine M can be turned into a disciplined Turing machine M' which halts on the same inputs and produces the same output. For instance, if the Turing machine halts and is not on square 1, we can add some instructions to make the head move left until it finds the tape-end marker, then move one square to the right, then halt. We'll leave you to think about how the other conditions can be dealt with. explanation

Example 1.18. In Figure 1.6, we turn the addition machine from Example 1.12 into a disciplined machine.

tur:mac:dis:prop:disciplined

Proposition 1.19. *For every Turing machine M, there is a disciplined Turing machine M' which halts with output O if M halts with output O, and does not halt if M does not halt. In particular, any function $f\colon \mathbb{N}^n \to \mathbb{N}$ computable by a Turing machine is also computable by a disciplined Turing machine.*

Problem 1.12. Give a disciplined machine that computes $f(x) = x + 1$.

Problem 1.13. Find a disciplined machine which, when started on input 1^n produces output $1^n \frown 0 \frown 1^n$.

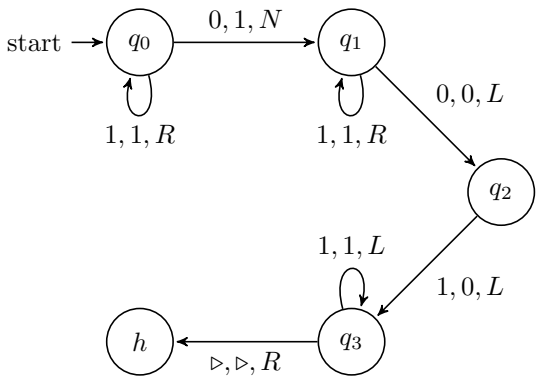


Figure 1.6: A disciplined addition machine

tur:mac:dis: fig:adder-disc

1.8 Combining Turing Machines

explanation

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.

tur:mac:cmb: sec

How do we combine Turing machines $M = \langle Q, \Sigma, q_0, \delta \rangle$ and $M' = \langle Q', \Sigma', q_0', \delta' \rangle$? We now use the configuration of the tape after M has halted as the input configuration of a run of machine M'. To get a single Turing machine $M \frown M'$ that does this, do the following:

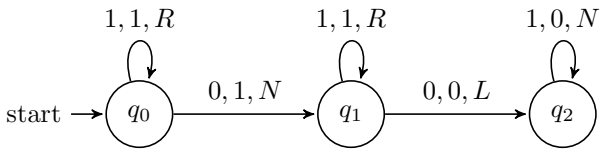
1. Renumber (or relabel) all the states Q' of M' so that M and M' have no states in common ($Q \cap Q' = \emptyset$).
2. The states of $M \frown M'$ are $Q \cup Q'$.
3. The tape alphabet is $\Sigma \cup \Sigma'$.
4. The start state is q_0.
5. The transition function is the function δ'' given by:

$$\delta''(q, \sigma) = \begin{cases} \delta(q, \sigma) & \text{if } q \in Q \\ \delta'(q, \sigma) & \text{if } q \in Q' \\ \langle q_0', \sigma, N \rangle & \text{if } q \in Q \text{ and } \delta(q, \sigma) \text{ is undefined} \end{cases}$$

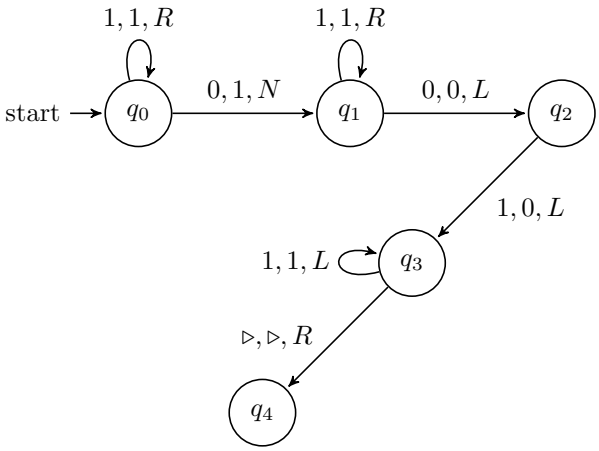
The resulting machine uses the instructions of M when it is in a state $q \in Q$, the instructions of M' when it is in a state $q \in Q'$. When it is in a state $q \in Q$ and is scanning a symbol σ for which M has no transition (i.e., M would have halted), it enters the start state of M' (and leaves the tape contents and head position as it is).

Note that unless the machine M is disciplined, we don't know where the tape head is when M halts, so the halting configuration of M need not have the head scanning square 1. When combining machines, it's important to keep this in mind.

Example 1.20. *Combining Machines:* We'll design a machine which, when started on input consisting of two blocks of 1's of length n and m, halts with a single block of $2(m+n)$ 1's on the tape. 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 in 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.



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 1.7.

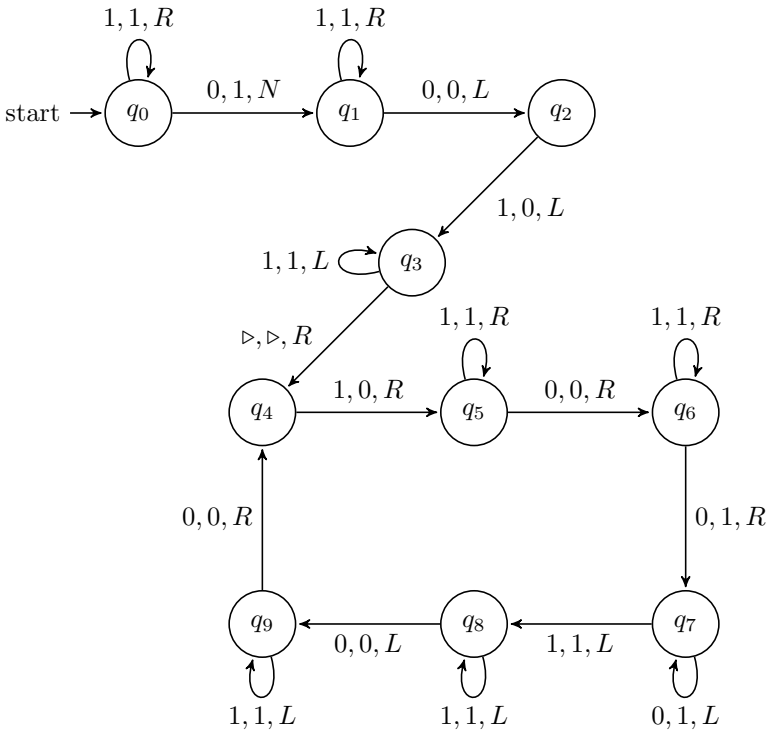


Figure 1.7: Combining adder and doubler machines

tur:mac:cmb:fig:combined

Proposition 1.21. *If M and M' are disciplined and compute the functions $f\colon \mathbb{N}^k \to \mathbb{N}$ and $f'\colon \mathbb{N} \to \mathbb{N}$, respectively, then $M \frown M'$ is disciplined and computes $f' \circ f$.*

Proof. Since M is disciplined, when it halts with output $f(n_1, \ldots, n_k) = m$, the head is scanning square 1. If we now enter the start state of M', the machine will halt with output $f'(m)$, again scanning square 1. The other conditions of Definition 1.17 are also satisfied. □

Problem 1.14. Give a disciplined Turing machine computing $f(x) = x + 2$ by taking the machine M from Problem 1.12 and construct $M \frown M$.

1.9 Variants of Turing Machines

tur:mac:var:sec

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 σ, $\delta(q, \sigma)$ 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. We have explained in section 1.6 why requiring a designated halting state is not a restriction which impacts what Turing machines can compute. 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 of "falling off", but we could have defined it so that it halts when that happens. This definition is also equivalent: we could simulate the behavior of a Turing machine that halts when it attempts to move left from square 0 by deleting every transition $\delta(q, \triangleright) = \langle q', \sigma, L \rangle$—then instead of attempting to move left on $\triangleright$ the machine halts.[1]

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 $\triangleright$.

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.*

We won't go into the details of verifying this. Here's just one example: we gain no additional computing power by allowing a tape that is infinite in both directions, or multiple tapes. The reason is, roughly, that a Turing machine

[1] This doesn't *quite* work, since nothing prevents us from writing and reading $\triangleright$ on squares other than square 0 (see Example 1.14). We can get around that by adding a second $\triangleright'$ symbol to use instead for such a purpose.

with a single one-way infinite tape can simulate multiple or two-way infinite tapes. E.g., using additional states and instructions, we can "translate" a program for a machine with multiple tapes or two-way infinite tape into one with a single one-way infinite tape. The translated machine can use the even squares for the squares of tape 1 (or the "positive" squares of a two-way infinite tape) and the odd squares for the squares of tape 2 (or the "negative" squares).

1.10 The Church–Turing Thesis

tur:mac:ctt:sec

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 1.22 (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 for it. 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

tur:und:int:sec

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

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

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 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 φ valid?". There is no Turing machine which, given as input a first-order formula φ, is guaranteed to halt with output 1 or 0 according to whether φ 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

explanation

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

tur:und:enu:sec

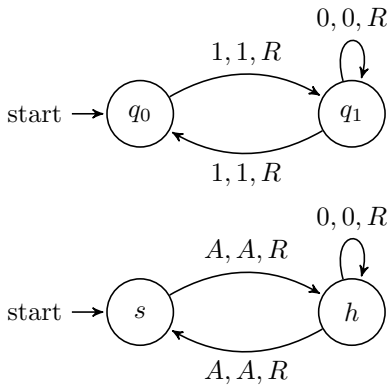


Figure 2.1: Variants of the *Even* machine

tur:und:enu:
fig:variants

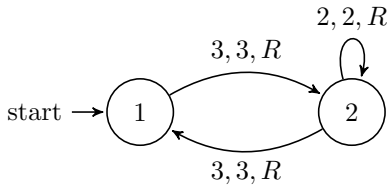


Figure 2.2: A standard *Even* machine

tur:und:enu:
fig:standard-even

specified by listing its values for the finitely many argument pairs for which it is defined.

This is true as far as it goes, but there is a subtle difference. The definition of Turing machines made no restriction on what elements the set of states and tape alphabet can have. So, e.g., for every real number, there technically is a Turing machine that uses that number as a state. However, the *behavior* of the Turing machine is independent of which objects serve as states and vocabulary. Consider the two Turing machines in Figure 2.1. These two diagrams correspond to two machines, M with the tape alphabet $\Sigma = \{\triangleright, 0, 1\}$ and set of states $\{q_0, q_1\}$, and M' with alphabet $\Sigma' = \{\triangleright, 0, A\}$ and states $\{s, h\}$. But their instructions are otherwise the same: M will halt on a sequence of n 1's iff n is even, and M' will halt on a sequence of n A's iff n is even. All we've done is rename 1 to A, q_0 to s, and q_1 to h. This example generalizes: we can think of Turing machines as the same as long as one results from the other by such a renaming of symbols and states. In fact, we can simply think of the symbols and states of a Turing machine as positive integers: instead of σ_0 think 1, instead of σ_1 think 2, etc.; $\triangleright$ is 1, 0 is 2, etc. In this way, the *Even* machine becomes the machine depicted in Figure 2.2. We might call a Turing

machine with states and symbols that are positive integers a *standard* machine, and only consider standard machines from now on.[1]

We wanted to show that the set of Turing machines is enumerable, and with the above considerations in mind, it is enough to show that the set of standard Turing machines is enumerable. Suppose we are given a standard Turing machine $M = \langle Q, \Sigma, q_0, \delta \rangle$. How could we describe it using a finite string of positive integers? We'll first list the number of states, the states themselves, the number of symbols, the symbols themselves, and the starting state. (Remember, all of these are positive integers, since M is a standard machine.) What about δ? The set of possible arguments, i.e., pairs $\langle q, \sigma \rangle$, is finite, since Q and Σ are finite. So the information in δ is simply the finite list of all 5-tuples $\langle q, \sigma, q', \sigma', d \rangle$ where $\delta(q, \sigma) = \langle q', \sigma', D \rangle$, and d is a number that codes the direction D (say, 1 for L, 2 for R, and 3 for N).

In this way, every standard Turing machine can be described by a finite list of positive integers, i.e., as a sequence $s_M \in (\mathbb{Z}^+)^*$. For instance, the standard *Even* machine is coded by the sequence

$$2, \underbrace{1,2}_{Q}, 3, \overbrace{1,2,3}^{\Sigma}, 1, \underbrace{1,3,2,3,2}_{\delta(1,3)=\langle 2,3,R\rangle}, \overbrace{2,2,2,2,2}^{\delta(2,2)=\langle 2,2,R\rangle}, \underbrace{2,3,1,3,2}_{\delta(2,3)=\langle 1,3,R\rangle}.$$

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 sequences of positive integers $(\mathbb{Z}^+)^*$ is enumerable (**??**). This gives us that the set of descriptions of standard Turing machines, as a subset of $(\mathbb{Z}^+)^*$, is itself enumerable. Every Turing computable function $\mathbb{N}$ to $\mathbb{N}$ is computed by some (in fact, many) Turing machines. By renaming its states and symbols to positive integers (in particular, $\triangleright$ as 1, 0 as 2, and 1 as 3) we can see that every Turing computable function is computed by a standard Turing machine. 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 (**??**). If all functions were computable by some Turing machine, we could enumerate the set of all functions by listing all the descriptions of Turing machines that compute them. So there are some functions that are not Turing computable. □

Problem 2.1. Can you think of a way to describe Turing machines that does not require that the states and alphabet symbols are explicitly listed? You may define your own notion of "standard" machine, but say something about why every Turing machine can be computed by a "standard" machine in your new sense.

[1] The terminology "standard machine" is not standard.

2.3 Universal Turing Machines

tur:und:uni:sec

In section 2.2 we discussed how every Turing machine can be described by a finite sequence of integers. This sequence encodes the states, alphabet, start state, and instructions of the Turing machine. We also pointed out that the set of all of these descriptions is enumerable. Since the set of such descriptions is denumerable, this means that there is a surjective function from $\mathbb{N}$ to these descriptions. Such a surjective function can be obtained, for instance, using Cantor's zig-zag method. It gives us a way of enumerating all (descriptions) of Turing machines. If we fix one such enumeration, it now makes sense to talk of the 1st, 2nd, ..., eth Turing machine. These numbers are called *indices*.

Definition 2.2. If M is the eth Turing machine (in our fixed enumeration), we say that e is an *index* of M. We write M_e for the eth Turing machine.

A machine may have more than one index, e.g., two descriptions of M may differ in the order in which we list its instructions, and these different descriptions will have different indices.

Importantly, it is possible to give the enumeration of Turing machine descriptions in such a way that we can effectively compute the description of M from its index, and to effectively compute an index of a machine M from its description. By the Church–Turing thesis, it is then possible to find a Turing machine which recovers the description of the Turing machine with index e and writes the corresponding description on its tape as output. The description would be a sequence of blocks of 1's (representing the positive integers in the sequence describing M_e).

Given this, it now becomes natural to ask: what functions of Turing machine indices are themselves computable by Turing machines? What properties of Turing machine indices can be decided by Turing machines? An example: the function that maps an index e to the number of states the Turing machine with index e has, is computable by a Turing machine. Here's what such a Turing machine would do: started on a tape containing a single block of e 1's, it would first decode e into its description. The description is now represented by a sequence of blocks of 1's on the tape. Since the first element in this sequence is the number of states. So all that has to be done now is to erase everything but the first block of 1's and then halt.

A remarkable result is the following:

tur:und:uni:thm:universal-tm

Theorem 2.3. *There is a* universal Turing machine U *which, when started on input* $\langle e, n\rangle$

1. *halts iff* M_e *halts on input* n*, and*
2. *if* M_e *halts with output* m*, so does* U*.*

U thus computes the function $f : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N}$ given by $f(e, n) = m$ if M_e started on input n halts with output m, and undefined otherwise.

Proof. To actually produce U is basically impossible, since it is an extremely complicated machine. But we can describe in outline how it works, and then invoke the Church–Turing thesis. When it starts, U's tape contains a block of e 1's followed by a block of n 1's. It first "decodes" the index e to the right of the input n. This produces a list of numbers (i.e., blocks of 1's separated by 0's) that describes the instructions of machine M_e. U then writes the number of the start state of M_e and the number 1 on the tape to the right of the description of M_e. (Again, these are represented in unary, as blocks of 1's.) Next, it copies the input (block of n 1's) to the right—but it replaces each 1 by a block of three 1's (remember, the number of the 1 symbol is 3, 1 being the number of ▷ and 2 being the number of 0). At the left end of this sequence of blocks (separated by 0 symbols on the tape of U), it writes a single 1, the code for ▷.

U now has on its tape: the index e, the number n, the code number of the start state (the "current state"), the number of the initial head position 1 (the "current head position"), and the initial contents of the "tape" (a sequence of blocks of 1's representing the code numbers of the symbols of M_e—the "symbols"—separated by 0's).

It now simulates what M_e would do if started on input n, by doing the following:

1. Find the number k of the "current head position" (at the beginning, that's 1),
2. Move to the kth block in the "tape" to see what the "symbol" there is,
3. Find the instruction matching the current "state" and "symbol," tur:und:uni:find-inst
4. Move back to the kth block on the "tape" and replace the "symbol" there with the code number of the symbol M_e would write,
5. Move the head to where it records the current "state" and replace the number there with the number of the new state,
6. Move to the place where it records the "tape position" and erase a 1 or add a 1 (if the instruction says to move left or right, respectively).
7. Repeat.[2]

If M_e started on input n never halts, then U also never halts, so its output is undefined.

If in step (3) it turns out that the description of M_e contains no instruction for the current "state"/"symbol" pair, then M_e would halt. If this happens, U erases the part of its tape to the left of the "tape." For each block of three 1's (representing a 1 on M_e's tape), it writes a 1 on the left end of its own tape,

[2] We're glossing over some subtle difficulties here. E.g., U may need some extra space when it increases the counter where it keeps track of the "current head position"—in that case it will have to move the entire "tape" to the right.

and successively erases the "tape." When this is done, U's tape contains a single block of 1's of length m.

If U encounters something other than a block of three 1's on the "tape," it immediately halts. Since U's tape in this case does not contain a single block of 1's, its output is not a natural number, i.e., $f(e, n)$ is undefined in this case.□

2.4 The Halting Problem

tur:und:halsec

Assume we have fixed some enumeration of Turing machine descriptions. Each Turing machine thus receives an *index*: its place in the enumeration M_1, M_2, M_3, ... of Turing machine descriptions. explanation

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 functions as well. One such function is the halting function.

Definition 2.4 (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 2.5 (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 by a disciplined Turing machine (section 1.7), and the fact that two Turing machines can be hooked together to create a single machine (section 1.8). explanation

Definition 2.6. 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 2.7. *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 (i.e., that it is disciplined). This machine can be "hooked up" to another machine J, which halts if it is started on input 0 (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. $S \frown J$, 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 $S \frown J$, so it should do exactly what S followed by J would do (i.e., in this case, wander off to the right and never halt). So M_e cannot halt for an input of e 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.

In each case we arrive at a contradiction with our assumption. This shows there cannot be a Turing machine S: s is not Turing computable. □

Theorem 2.8 (Unsolvability of the Halting Problem). *The halting problem is unsolvable, i.e., the function h is not Turing computable.* *tur:und:hal: thm:halting-problem*

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 0 symbol). Then move back to the beginning, and run H. We can clearly make a machine that does the former (see Problem 1.13), and if H existed, we would be able to "hook it up" to such a copier 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 2.2. 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 1's on an otherwise blank tape. Prove that the 3-Halt problem is unsolvable.

Problem 2.3. 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 2.4. 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? 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.

Problem 2.5. Show that the *partial* function s' is defined as

$$s'(e) = \begin{cases} 1 & \text{if machine } M_e \text{ halts for input } e \\ \text{undefined} & \text{if machine } M_e \text{ does not halt for input } e \end{cases}$$

is Turing computable.

2.5 The Decision Problem

tur:und:dec:sec

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 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:

$$\vDash \tau(M, w) \to \alpha(M, w) \text{ iff } M \text{ halts for input } w.$$

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

2.6 Representing Turing Machines

tur:und:rep:sec

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.

explanation

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 σ, a predicate symbol S_σ. 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 o, and a 1-place function $'$, the successor function. By convention it is written *after* its argument (and we leave out the parentheses).

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

$$\overline{0} = \mathsf{o}$$
$$\overline{n+1} = \overline{n}'$$

The term $\overline{0}$, i.e., o names the leftmost position on the tape as well as the time before the first execution step (the initial configuration). The term $\overline{1}$, i.e., o' 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 o, $'$, and $<$, sentences that describes the input configuration, and sentences that describe what the configuration of M is after it executes a particular instruction.

Definition 2.9. Given a Turing machine $M = \langle Q, \Sigma, q_0, \delta \rangle$, the language $\mathcal{L}_M$ consists of: tur:und:rep: defn:tm-descr

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

2. A two-place predicate symbol $S_\sigma(x, y)$ for every symbol $\sigma \in \Sigma$. Intuitively, $S_\sigma(\overline{m}, \overline{n})$ expresses "after n steps, the mth square contains symbol σ."

3. A constant symbol o

4. A one-place function symbol $'$

5. A two-place predicate symbol $<$

The sentences describing the operation of the Turing machine M on input $w = \sigma_{i_1} \dots \sigma_{i_k}$ are the following:

1. Axioms describing numbers and $<$:

a) A sentence that says that every number is less than its successor:

$$\forall x\, x < x'$$

b) A sentence that ensures that $<$ is transitive:

$$\forall x \forall y \forall z\, ((x < y \wedge y < z) \to x < z)$$

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 $\triangleright$, σ_{i_1}, …, σ_{i_k}:

$$S_{\triangleright}(\overline{0}, \overline{0}) \wedge S_{\sigma_{i_1}}(\overline{1}, \overline{0}) \wedge \cdots \wedge S_{\sigma_{i_k}}(\overline{k}, \overline{0})$$

c) Otherwise, the tape is empty:

$$\forall x\, (\overline{k} < x \to S_0(x, \overline{0}))$$

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

$$\forall z\, (((z < x \vee x < z) \wedge S_\sigma(z, y)) \to S_\sigma(z, y'))$$

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

tur:und:rep:rep-right

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

$$\begin{aligned}\forall x \forall y\, ((&Q_{q_i}(x, y) \wedge S_\sigma(x, y)) \to \\ &(Q_{q_j}(x', y') \wedge S_{\sigma'}(x, y') \wedge \varphi(x, y)))\end{aligned}$$

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

tur:und:rep:rep-left

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

$$\begin{aligned}\forall x \forall y\, ((&Q_{q_i}(x', y) \wedge S_\sigma(x', y)) \to \\ &(Q_{q_j}(x, y') \wedge S_{\sigma'}(x', y') \wedge \varphi(x, y))) \wedge \\ \forall y\, ((&Q_{q_i}(\overline{0}, y) \wedge S_\sigma(\overline{0}, y)) \to \\ &(Q_{q_j}(\overline{0}, y') \wedge S_{\sigma'}(\overline{0}, y') \wedge \varphi(\overline{0}, y)))\end{aligned}$$

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, ..., 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 σ, then after $y+1$ steps it's still scanning square 0, is now in state q_j, the symbol on square 0 is σ', and the squares other than square 0 contain the same symbols they contained ofter y steps.

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

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

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

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 $\langle q, \sigma\rangle$ such that $\delta(q,\sigma)$ is undefined. Let $\alpha(M,w)$ then be the sentence

$$\exists x\, \exists y\, (\bigvee_{\langle q,\sigma\rangle \in X} (Q_q(x,y) \wedge S_\sigma(x,y)))$$

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

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

expresses that the machine eventually halts.

Proposition 2.10. *If $m < k$, then $\tau(M,w) \vDash \overline{m} < \overline{k}$* tur:und:rep: prop:mlessk

Proof. Exercise. □

Problem 2.6. Prove Proposition 2.10. (Hint: use induction on $k - m$).

2.7 Verifying the Representation

explanation 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 $\tau(M,w) \rightarrow \alpha(M,w)$ is valid. Then, we have to show the converse, i.e., that if $\tau(M,w) \rightarrow \alpha(M,w)$ is valid, then M does in fact eventually halt when run on input w. tur:und:ver: sec

The strategy for proving these is very different. For the first result, we have to show that a sentence of first-order logic (namely, $\tau(M,w) \to \alpha(M,w)$) 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 $\tau(M,w) \to \alpha(M,w)$.

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, $\tau(M,w) \vDash \chi(M,w,n)$, where $\chi(M,w,n)$ correctly describes the configuration of M run on w after n steps. Now if M halts on input w after, say, n steps, $\chi(M,w,n)$ will describe a halting configuration. We'll also show that $\chi(M,w,n) \vDash \alpha(M,w)$, whenever $\chi(M,w,n)$ 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, $\tau(M,w) \vDash \chi(M,w,n)$ where $\chi(M,w,n)$ describes a halting configuration, and since in that case $\chi(M,w,n) \vDash \alpha(M,w)$, we get that $T(M,w) \vDash \alpha(M,w)$, i.e., that $\vDash \tau(M,w) \to \alpha(M,w)$.

The strategy for the converse is very different. Here we assume that $\vDash \tau(M,w) \to \alpha(M,w)$ and have to prove that M halts on input w. From the hypothesis we get that $\tau(M,w) \vDash \alpha(M,w)$, i.e., $\alpha(M,w)$ is true in every structure in which $\tau(M,w)$ is true. So we'll describe a structure $\mathfrak{M}$ in which $\tau(M,w)$ is true: its domain will be $\mathbb{N}$, and the interpretation of all the Q_q and S_σ will be given by the configurations of M during a run on input w. So, e.g., $\mathfrak{M} \vDash Q_q(\overline{m},\overline{n})$ iff T, when run on input w for n steps, is in state q and scanning square m. Now since $\tau(M,w) \vDash \alpha(M,w)$ by hypothesis, and since $\mathfrak{M} \vDash \tau(M,w)$ by construction, $\mathfrak{M} \vDash \alpha(M,w)$. But $\mathfrak{M} \vDash \alpha(M,w)$ iff there is some $n \in |\mathfrak{M}| = \mathbb{N}$ so that M, run on input w, is in a halting configuration after n steps.

Definition 2.11. Let $\chi(M,w,n)$ be the sentence

$$Q_q(\overline{m},\overline{n}) \wedge S_{\sigma_0}(\overline{0},\overline{n}) \wedge \cdots \wedge S_{\sigma_k}(\overline{k},\overline{n}) \wedge \forall x\,(\overline{k} < x \to S_0(x,\overline{n}))$$

where q is the state of M at time n, M is scanning square m at time n, square i contains symbol σ_i at time n for $0 \leq i \leq k$ and k is the right-most non-blank square of the tape at time 0, or the right-most square the tape head has visited after n steps, whichever is greater.

tur:und:ver: lem:halt-config-implies-halt

Lemma 2.12. *If M run on input w is in a halting configuration after n steps, then $\chi(M,w,n) \vDash \alpha(M,w)$.*

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

1. After n steps, M is in state q scanning square m on which σ 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 $Q_q(\overline{m}, \overline{n})$ and $S_\sigma(\overline{m}, \overline{n})$. These clauses together imply $\alpha(M, w)$:

$$\exists x \, \exists y \, (\bigvee_{\langle q,\sigma\rangle \in X} (Q_q(x, y) \wedge S_\sigma(x, y)))$$

since $Q_{q'}(\overline{m}, \overline{n}) \wedge S_{\sigma'}(\overline{m}, \overline{n}) \vDash \bigvee_{\langle q,\sigma\rangle \in X} (Q_q(\overline{m}, \overline{n}) \wedge S_\sigma(\overline{m}, \overline{n}))$, as $\langle q', \sigma'\rangle \in X$.□

explanation

So if M halts for input w, then there is some n such that $\chi(M, w, n) \vDash \alpha(M, w)$. We will now show that for any time n, $\tau(M, w) \vDash \chi(M, w, n)$.

Lemma 2.13. *For each n, if M has not halted after n steps, $\tau(M, w) \vDash \chi(M, w, n)$.* *tur:und:ver: lem:config*

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 nth step, then $\tau(M, w) \vDash \chi(M, w, n)$. We have to show that (unless $\chi(M, w, n)$ describes a halting configuration), $\tau(M, w) \vDash \chi(M, w, n + 1)$.

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:

1. $\delta(q, \sigma) = \langle q', \sigma', R\rangle$ tur:und:ver: right
2. $\delta(q, \sigma) = \langle q', \sigma', L\rangle$ tur:und:ver: left
3. $\delta(q, \sigma) = \langle q', \sigma', N\rangle$ tur:und:ver: stay

We will consider each of these three cases in turn.

1. Suppose there is an instruction of the form (1). By Definition 2.9(3a), this means that

$$\begin{aligned}\forall x \, \forall y \, ((Q_q(x, y) \wedge S_\sigma(x, y)) \rightarrow \\ (Q_{q'}(x', y') \wedge S_{\sigma'}(x, y') \wedge \varphi(x, y)))\end{aligned}$$

is a conjunct of $\tau(M, w)$. This entails the following sentence (universal instantiation, $\overline{m}$ for x and $\overline{n}$ for y):

$$\begin{aligned}(Q_q(\overline{m}, \overline{n}) \wedge S_\sigma(\overline{m}, \overline{n})) \rightarrow \\ (Q_{q'}(\overline{m}', \overline{n}') \wedge S_{\sigma'}(\overline{m}, \overline{n}') \wedge \varphi(\overline{m}, \overline{n})).\end{aligned}$$

By induction hypothesis, $\tau(M, w) \vDash \chi(M, w, n)$, i.e.,

$$\begin{aligned}Q_q(\overline{m}, \overline{n}) \wedge S_{\sigma_0}(\overline{0}, \overline{n}) \wedge \cdots \wedge S_{\sigma_k}(\overline{k}, \overline{n}) \wedge \\ \forall x \, (\overline{k} < x \rightarrow S_0(x, \overline{n}))\end{aligned}$$

Since after n steps, tape square m contains σ, the corresponding conjunct is $S_\sigma(\overline{m}, \overline{n})$, so this entails:

$$Q_q(\overline{m}, \overline{n}) \wedge S_\sigma(\overline{m}, \overline{n})$$

We now get

$$\begin{aligned} &Q_{q'}(\overline{m}', \overline{n}') \wedge S_{\sigma'}(\overline{m}, \overline{n}') \wedge \\ &\quad S_{\sigma_0}(\overline{0}, \overline{n}') \wedge \cdots \wedge S_{\sigma_k}(\overline{k}, \overline{n}') \wedge \\ &\quad \forall x\, (\overline{k} < x \to S_0(x, \overline{n}')) \end{aligned}$$

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 $S_{\sigma_m}(\overline{m}, \overline{n}')$—follows from the corresponding conjunct in $\chi(M, w, n)$ together with $\varphi(\overline{m}, \overline{n})$.

If $m < k$, $\tau(M, w) \vdash \overline{m} < \overline{k}$ (Proposition 2.10) and by transitivity of $<$, we have $\forall x\, (\overline{k} < x \to \overline{m} < x)$. If $m = k$, then $\forall x\, (\overline{k} < x \to \overline{m} < x)$ by logic alone. The last line then follows from the corresponding conjunct in $\chi(M, w, n)$, $\forall x\, (\overline{k} < x \to \overline{m} < x)$, and $\varphi(\overline{m}, \overline{n})$. 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, $S_0(\overline{k}', \overline{n}')$, and the last conjunct is $\forall x\, (\overline{k}' < x \to S_0(x, \overline{n}'))$. We have to verify that these two sentences are also implied.

We already have $\forall x\, (\overline{k} < x \to S_0(x, \overline{n}'))$. In particular, this gives us $\overline{k} < \overline{k}' \to S_0(\overline{k}', \overline{n}')$. From the axiom $\forall x\, x < x'$ we get $\overline{k} < \overline{k}'$. By modus ponens, $S_0(\overline{k}', \overline{n}')$ follows.

Also, since $\tau(M, w) \vdash \overline{k} < \overline{k}'$, the axiom for transitivity of $<$ gives us $\forall x\, (\overline{k}' < x \to S_0(x, \overline{n}'))$. (We leave the verification of this as an exercise.)

2. Suppose there is an instruction of the form (2). Then, by Definition 2.9(3b),

$$\begin{aligned} &\forall x\, \forall y\, ((Q_q(x', y) \wedge S_\sigma(x', y)) \to \\ &\quad (Q_{q'}(x, y') \wedge S_{\sigma'}(x', y') \wedge \varphi(x, y))) \wedge \\ &\forall y\, ((Q_{q_i}(\overline{0}, y) \wedge S_\sigma(\overline{0}, y)) \to \\ &\quad (Q_{q_j}(\overline{0}, y') \wedge S_{\sigma'}(\overline{0}, y') \wedge \varphi(\overline{0}, y))) \end{aligned}$$

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:

$$\begin{aligned} &(Q_q(\overline{l}', \overline{n}) \wedge S_\sigma(\overline{l}', \overline{n})) \to \\ &\quad (Q_{q'}(\overline{l}, \overline{n}') \wedge S_{\sigma'}(\overline{l}', \overline{n}') \wedge \varphi(\overline{l}, \overline{n})) \end{aligned}$$

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

$$((Q_{q_i}(\overline{0}, \overline{n}) \wedge S_\sigma(\overline{0}, \overline{n})) \rightarrow \\ (Q_{q_j}(\overline{0}, \overline{n}') \wedge S_{\sigma'}(\overline{0}, \overline{n}') \wedge \varphi(\overline{0}, \overline{n})))$$

Either sentence implies

$$Q_{q'}(\overline{l}, \overline{n}') \wedge S_{\sigma'}(\overline{m}, \overline{n}') \wedge \\ S_{\sigma_0}(\overline{0}, \overline{n}') \wedge \cdots \wedge S_{\sigma_k}(\overline{k}, \overline{n}') \wedge \\ \forall x\, (\overline{k} < x \rightarrow S_0(x, \overline{n}'))$$

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

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

We have shown that for any n, $\tau(M, w) \vDash \chi(M, w, n)$. □

Problem 2.7. Complete case (3) of the proof of Lemma 2.13.

Problem 2.8. Give a derivation of $S_{\sigma_i}(\overline{i}, \overline{n}')$ from $S_{\sigma_i}(\overline{i}, \overline{n})$ and $\varphi(m, n)$ (assuming $i \neq m$, i.e., either $i < m$ or $m < i$).

Problem 2.9. Give a derivation of $\forall x\, (\overline{k}' < x \rightarrow S_0(x, \overline{n}'))$ from $\forall x\, (\overline{k} < x \rightarrow S_0(x, \overline{n}'))$, $\forall x\, x < x'$, and $\forall x \forall y \forall z\, ((x < y \wedge y < z) \rightarrow x < z)$.)

Lemma 2.14. *If M halts on input w, then $\tau(M, w) \rightarrow \alpha(M, w)$ is valid.* *tur:und:ver: lem:valid-if-halt*

Proof. By Lemma 2.13, 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. At that point, it will be scanning square m, for some $m \in \mathbb{N}$. Then $\chi(M, w, k)$ describes a halting configuration of M, i.e., it contains as conjuncts both $Q_q(\overline{m}, \overline{k})$ and $S_\sigma(\overline{m}, \overline{k})$ with $\delta(q, \sigma)$ undefined. Thus, by Lemma 2.12, $\chi(M, w, k) \vDash \alpha(M, w)$. But since $\tau(M, w) \vDash \chi(M, w, k)$, we have $\tau(M, w) \vDash \alpha(M, w)$ and therefore $\tau(M, w) \rightarrow \alpha(M, w)$ is valid. □

explanation

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 w.

Lemma 2.15. *If $\vDash \tau(M, w) \rightarrow \alpha(M, w)$, then M halts on input w.* *tur:und:ver: lem:halt-if-valid*

Proof. Consider the $\mathcal{L}_M$-structure $\mathfrak{M}$ with domain $\mathbb{N}$ which interprets o as 0, $\mathsf{'}$ as the successor function, and $<$ as the less-than relation, and the predicates Q_q and S_σ as follows:

$$Q_q^{\mathfrak{M}} = \{\langle m, n\rangle : \begin{array}{l}\text{started on } w\text{, after } n \text{ steps,}\\ M \text{ is in state } q \text{ scanning square } m\end{array}\}$$

$$S_\sigma^{\mathfrak{M}} = \{\langle m, n\rangle : \begin{array}{l}\text{started on } w\text{, after } n \text{ steps,}\\ \text{square } m \text{ of } M \text{ contains symbol } \sigma\end{array}\}$$

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, $\mathfrak{M} \vDash \tau(M, w)$. If $\vDash \tau(M, w) \to \alpha(M, w)$, then also $\mathfrak{M} \vDash \alpha(M, w)$, i.e.,

$$\mathfrak{M} \vDash \exists x\, \exists y\, (\bigvee_{\langle q,\sigma\rangle \in X} (Q_q(x, y) \wedge S_\sigma(x, y))).$$

As $|\mathfrak{M}| = \mathbb{N}$, there must be m, $n \in \mathbb{N}$ so that $\mathfrak{M} \vDash Q_q(\overline{m}, \overline{n}) \wedge S_\sigma(\overline{m}, \overline{n})$ for some q and σ 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 σ, and the transition function is undefined, i.e., M has halted. □

2.8 The Decision Problem is Unsolvable

tur:und:uns:sec
tur:und:uns:thm:decision-prob

Theorem 2.16. *The decision problem is unsolvable: There is no Turing machine D, which when started on a tape that contains a sentence ψ of first-order logic as input, D eventually halts, and outputs* 1 *iff ψ is valid and* 0 *otherwise.*

Proof. Suppose the decision problem were solvable, i.e., suppose there were a Turing machine D. 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 $E \frown D$ 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 2.15 and Lemma 2.14, $\tau(M_e, w) \to \alpha(M_e, w)$ is valid iff M_e halts on input w. Thus, $E \frown D$, 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, $E \frown D$ would solve the halting problem. But we know, by Theorem 2.8, that no such Turing machine can exist. □

tur:und:uns:cor:undecidable-sat

Corollary 2.17. *It is undecidable if an arbitrary sentence of first-order logic is satisfiable.*

Proof. Suppose satisfiability were decidable by a Turing machine S. Then we could solve the decision problem as follows: Given a sentence B as input, move ψ to the right one square. Return to square 1 and write the symbol $\neg$.

Now run the Turing machine S. It eventually halts with output either 1 (if $\neg\psi$ is satisfiable) or 0 (if $\neg\psi$ is unsatisfiable) on the tape. If there is a 1 on square 1, erase it; if square 1 is empty, write a 1, then halt.

This Turing machine always halts, and its output is 1 iff $\neg\psi$ is unsatisfiable and 0 otherwise. Since ψ is valid iff $\neg\psi$ is unsatisfiable, the machine outputs 1 iff ψ is valid, and 0 otherwise, i.e., it would solve the decision problem. □

explanation

So there is no Turing machine which always gives a correct "yes" or "no" answer to the question "Is ψ a valid sentence of first-order logic?" However, there *is* a Turing machine that always gives a correct "yes" answer—but simply does not halt if the answer is "no." This follows from the soundness and completeness theorem of first-order logic, and the fact that derivations can be effectively enumerated.

Theorem 2.18. *Validity of first-order sentences is semi-decidable: There is a Turing machine E, which when started on a tape that contains a sentence ψ of first-order logic as input, E eventually halts and outputs 1 iff ψ is valid, but does not halt otherwise.*

tur:und:uns: thm:valid-ce

Proof. All possible derivations of first-order logic can be generated, one after another, by an effective algorithm. The machine E does this, and when it finds a derivation that shows that $\vdash \psi$, it halts with output 1. By the soundness theorem, if E halts with output 1, it's because $\vDash \psi$. By the completeness theorem, if $\vDash \psi$ there is a derivation that shows that $\vdash \psi$. Since E systematically generates all possible derivations, it will eventually find one that shows $\vdash \psi$, so will eventually halt with output 1. □

2.9 Trakhtenbrot's Theorem

explanation

In section 2.6 we defined sentences $\tau(M, w)$ and $\alpha(M, w)$ for a Turing machine M and input string w. Then we showed in Lemma 2.14 and Lemma 2.15 that $\tau(M, w) \to \alpha(M, w)$ is valid iff M, started on input w, eventually halts. Since the Halting Problem is undecidable, this implies that validity and satisfiability of sentences of first-order logic is undecidable (Theorem 2.16 and Corollary 2.17).

tur:und:tra: sec

But validity and satisfiability of sentences is defined for arbitrary structures, finite or infinite. You might suspect that it is easier to decide if a sentence is satisfiable in a finite structure (or valid in all finite structures). We can adapt the proof of the unsolvability of the decision problem so that it shows this is not the case.

First, if you go back to the proof of Lemma 2.15, you'll see that what we did there is produce a model $\mathfrak{M}$ of $\tau(M, w)$ which describes exactly what machine M does when started on input w. The domain of that model was $\mathbb{N}$,

i.e., infinite. But if M actually halts on input w, we can build a finite model $\mathfrak{M}'$ in the same way. Suppose M started on input w halts after k steps. Take as domain $|\mathfrak{M}'|$ the set $\{0, \ldots, n\}$, where n is the larger of k and the length of w, and let

$$\prime^{\mathfrak{M}'}(x) = \begin{cases} x+1 & \text{if } x < n \\ n & \text{otherwise,} \end{cases}$$

and $\langle x, y\rangle \in <^{\mathfrak{M}'}$ iff $x < y$ or $x = y = n$. Otherwise $\mathfrak{M}'$ is defined just like $\mathfrak{M}$. By the definition of $\mathfrak{M}'$, just like in the proof of Lemma 2.15, $\mathfrak{M}' \vDash \tau(M, w)$. And since we assumed that M halts on input w, $\mathfrak{M}' \vDash \alpha(M, w)$. So, $\mathfrak{M}'$ is a finite model of $\tau(M, w) \wedge \alpha(M, w)$ (note that we've replaced $\rightarrow$ with $\wedge$).

We are halfway to a proof: we've shown that if M halts on input w, then $\tau(M, w) \wedge \alpha(M, w)$ has a finite model. Unfortunately, the converse of this does not hold, i.e., there are Turing machines that don't halt on some input w, but $\tau(M, w) \wedge \alpha(M, w)$ still has a finite model. For instance, consider the machine M with the single state q_0 and instruction $\delta(q_0, 0) = \langle q_0, 0, N\rangle$. Started on empty input $w = \Lambda$, this machine never halts: it is in an infinite loop, but does not change the tape or move the head. All configurations are the same (same state, same head position, same tape contents). We can define a finite structure $\mathfrak{M}''$ that satisfies $\tau(M, \Lambda) \wedge \alpha(M, \Lambda)$ (exercise). We can, however, change $\tau(M, w)$ in a suitable way so that such structures are ruled out.

Problem 2.10. Let M be a Turing machine with the single state q_0 and single instruction $\delta(q_0, 0) = \langle q, 0, N\rangle$. Let $|\mathfrak{M}''| = \{0, 1, 2\}$, $\prime^{\mathfrak{M}''}(0) = \prime^{\mathfrak{M}'}(1) = 1$ and $\prime^{\mathfrak{M}''}(2) = 2$, and $<^{\mathfrak{M}''} = \{\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle\}$. Define $Q_{q_0}^{\mathfrak{M}''}$, $S_0^{\mathfrak{M}''}$, and $S_\triangleright^{\mathfrak{M}''}$ so that $\tau(M, \Lambda)$ and $\alpha(M, \Lambda)$ become true and explain why they are. Hint: Observe that $\delta(q_0, \triangleright)$ is undefined. Ensure that

$$\begin{aligned} &Q_{q_0}(\overline{1}, \overline{n}) \wedge S_\triangleright(\overline{0}, \overline{n}) \wedge \forall x\, (\overline{0} < x \rightarrow S_0(x, \overline{n})) \quad \text{for all } n \in \mathbb{N} \\ &\exists y\, (Q_{q_0}(\overline{0}, y) \wedge S_\triangleright(\overline{0}, y)) \end{aligned}$$

are both true in $\mathfrak{M}''$.

Consider the sentences describing the operation of the Turing machine M on input $w = \sigma_{i_1} \ldots \sigma_{i_k}$:

1. Axioms describing numbers and $<$ (just like in the definition of $\tau(M, w)$ in section 2.6).

2. Axioms describing the input configuration: just like in the definition of $\tau(M, w)$.

3. Axioms describing the transition from one configuration to the next:

   For the following, let $\varphi(x, y)$ be as before, and let

   $$\psi(y) \equiv \forall x\, (x < y \rightarrow x \neq y).$$

a) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', R\rangle$, the sentence: tur:und:tra: rep-right

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

b) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', L\rangle$, the sentence tur:und:tra: rep-left

$$\begin{aligned}&\forall x\, \forall y\, ((Q_{q_i}(x', y) \wedge S_\sigma(x', y)) \rightarrow \\ &\qquad (Q_{q_j}(x, y') \wedge S_{\sigma'}(x', y') \wedge \varphi(x, y))) \wedge \\ &\forall y\, ((Q_{q_i}(\overline{0}, y) \wedge S_\sigma(\overline{0}, y)) \rightarrow \\ &\qquad (Q_{q_j}(\overline{0}, y') \wedge S_{\sigma'}(\overline{0}, y') \wedge \varphi(\overline{0}, y) \wedge \psi(y')))\end{aligned}$$

c) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', N\rangle$, the sentence: tur:und:tra: rep-stay

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

As you can see, the sentences describing the transitions of M are the same as the corresponding sentence in $\tau(M, w)$, except we add $\psi(y')$ at the end. $\psi(y')$ ensures that the number y' of the "next" configuration is different from all previous numbers o, o', ...

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

Lemma 2.19. *If M started on input w halts, then $\tau'(M, w) \wedge \alpha(M, w)$ has a finite model.* tur:und:tra: lem:halts-sat

Proof. Let $\mathfrak{M}'$ be as in the proof of Lemma 2.15, except

$$\begin{aligned}|\mathfrak{M}'| &= \{0, \ldots, n\},\\ \prime^{\mathfrak{M}'}(x) &= \begin{cases} x+1 & \text{if } x < n \\ n & \text{otherwise,}\end{cases}\\ \langle x, y\rangle \in <^{\mathfrak{M}'} &\text{ iff } x < y \text{ or } x = y = n,\end{aligned}$$

where $n = \max(k, \text{len}(w))$ and k is the least number such that M started on input w has halted after k steps. We leave the verification that $\mathfrak{M}' \vDash \tau'(M, w) \wedge E(M, w)$ as an exercise. □

Problem 2.11. Complete the proof of Lemma 2.19 by proving that $\mathfrak{M}' \vDash \tau(M, w) \wedge E(M, w)$.

Lemma 2.20. *If $\tau'(M, w) \wedge \alpha(M, w)$ has a finite model, then M started on input w halts.* tur:und:tra: lem:sat-halts

Proof. We show the contrapositive. Suppose that M started on w does not halt. If $\tau'(M,w) \wedge \alpha(M,w)$ has no model at all, we are done. So assume $\mathfrak{M}$ is a model of $\tau(M,w) \wedge \alpha(M,w)$. We have to show that it cannot be finite.

We can prove, just like in Lemma 2.13, that if M, started on input w, has not halted after n steps, then $\tau'(M,w) \vDash \chi(M,w,n) \wedge \psi(\overline{n})$. Since M started on input w does not halt, $\tau'(M,w) \vDash \chi(M,w,n) \wedge \psi(\overline{n})$ for all $n \in \mathbb{N}$. Note that by Proposition 2.10, $\tau'(M,w) \vDash \overline{k} < \overline{n}$ for all $k < n$. Also $\psi(\overline{n}) \vDash \overline{k} < \overline{n} \to \overline{k} \neq \overline{n}$. So, $\mathfrak{M} \vDash \overline{k} \neq \overline{n}$ for all $k < n$, i.e., the infinitely many terms $\overline{k}$ must all have different values in $\mathfrak{M}$. But this requires that $|\mathfrak{M}|$ be infinite, so $\mathfrak{M}$ cannot be a finite model of $\tau'(M,w) \wedge \alpha(M,w)$. □

Problem 2.12. Complete the proof of Lemma 2.20 by proving that if M, started on input w, has not halted after n steps, then $\tau'(M,w) \vDash \psi(\overline{n})$.

tur:und:tra: thm:trakhtenbrodt

Theorem 2.21 (Trakhtenbrot's Theorem). *It is undecidable if an arbitrary sentence of first-order logic has a finite model (i.e., is finitely satisfiable).*

Proof. Suppose there were a Turing machine F that decides the finite satisfiability problem. Then given any Turing machine M and input w, we could compute the sentence $\tau'(M,w) \wedge \alpha(M,w)$, and use F to decide if it has a finite model. By Lemmata 2.19 and 2.20, it does iff M started on input w halts. So we could use F to solve the halting problem, which we know is unsolvable. □

tur:und:tra: cor:fproof-incomp

Corollary 2.22. *There can be no derivation system that is sound and complete for* finite *validity, i.e., a derivation system which has* $\vdash \psi$ *iff* $\mathfrak{M} \vDash \psi$ *for every finite structure* $\mathfrak{M}$.

Proof. Exercise. □

Problem 2.13. Prove Corollary 2.22. Observe that ψ is satisfied in every finite structure iff $\neg\psi$ is not finitely satisfiable. Explain why finite satisfiability is semi-decidable in the sense of Theorem 2.18. Use this to argue that if there were a derivation system for finite validity, then finite satisfiability would be decidable.

Photo Credits

Bibliography