Turing Machines and the Decision Problem for First-order Logic

Lecture notes for ECM3404 *Logic and Computation*

1 What is an effective procedure?

We have used the phrase "effective procedure", and this turns up quite a lot in this area. What does it mean? We shall answer this question by defining three other terms, namely "procedure", "algorithm" and "heuristic".

There's a useful rule of thumb for working out roughly how much your trolley-load of supermarket purchases is going to come to. Simply round off all the prices to the nearest pound and add them up. Thus if you have buy ten items, with prices

$$1.24, 3.56, 1.10, 0.67, 0.45, 1.99, 0.36, 2.37, 4.58, 1.85$$

you add up instead $1, 4, 1, 1, 0, 2, 0, 2, 5, 2,$ giving 18, which is close to the true value of 18.17 (and I swear I didn't fiddle the list—I just wrote down prices at random). The point is that in a random list of numbers, roughly half will round up and half will round down and the rounding errors can be expected to cancel out pretty nearly most of the time.

On the other hand it's easy to concoct lists of prices for which this method behaves much less well. If we do it with

$$1.44, 3.96, 1.40, 0.97, 0.45, 1.99, 0.46, 2.47, 4.98, 1.95$$

the rounding method still gives 18, but the true figure is now 20.07.

Whether a method is reliable or not depends on what criteria you use to assess it. A method is a method for performing a task, and it's a good method if it performs the task well. Suppose the task is:

> Given n numbers $a_1, a_2, \ldots, a_n$, find the nearest integer to $a_1 + a_2 + \cdots + a_n$

and the method suggested is

> For $i = 1, \ldots, n$ let b_i be the nearest integer to a_i, and compute $b_1 + b_2 + \cdots + b_n$

then we find, as above, that sometimes the method works and sometimes it doesn't; but when it doesn't the result will typically be not too far off the correct answer.

The point is that the method suggested is a *procedure* but not an *algorithm* for the task in hand. A procedure is a well-defined method which can always be followed, but it only counts as an algorithm if it is guaranteed in every case to terminate and deliver the correct result. The term "effective procedure" is generally used to mean the same as "algorithm"—but usage here is notoriously slippery, and some authors use these words in different ways.

A lot of bread-and-butter computing is about finding effective procedures, although effectiveness isn't the only thing we're looking for (we want *efficiency* and *ease of use* as well—not to mention *cost-effectiveness*). There are effective procedures for doing arithmetic, for manipulating strings and lists, for transforming images in various ways and so on. These are the things generally dignified with the title of algorithm. They are often contrasted with procedures which, though not effective, are in some sense *nearly* effective—e.g., they give the right answer most of the time, or they give answers which are nearly right. These are called *heuristics*.

Note that the terms "algorithm" and "heuristic" are *task-relative*: a procedure does not in itself constitute an algorithm or heuristic, but only in relation to some task. A heuristic for one task can generally be viewed as an algorithm for a different but related task.

An important point about algorithms is that although in general an algorithm can apply to infinitely many different cases (inputs), the procedure that is followed has a finite description that can be given in advance, and when it is applied to any particular case it delivers its output after finitely many steps. To quote Stephen Kleene (Herken p.19):

> [A]n algorithm is a *finitely* described procedure, sufficient to guide us to the answer to any one of *infinitely* many questions, by *finitely* many steps in the case of each question.

We use slightly different terminology for computational tasks depending on whether they apply to functions or predicates. If we have an algorithm which determines the value of some n-place function f for arbitrary input tuples $(x_1, x_2, \ldots, x_n)$, then the function f is said to be *computable*; if on the other hand we have an algorithm which determines whether some n-place predicate P holds for arbitrary input tuples $(x_1, x_2, \ldots, x_n)$, then the predicate P is said to be *decidable*.

The *Church-Turing Thesis* says that, roughly, there is an effective procedure for carrying out a computational task if and only if there is a Turing Machine which can do it. Turing Machines are the gold standard of effectiveness. How widely you can apply the Church-Turing thesis depends on what you understand by the term "computational". By interpreting it very broadly, one might be led to make rash statements concerning the intellectual capabilities of human beings—statements with which, perhaps, Turing, though not Church, might have had some sympathy.

Consider "translation from English to French". Some humans, so the argument might run, can do this effectively. By the Church-Turing thesis, therefore, some Turing machine can do it as well. The trouble with this argument is that the task is not well defined; for that matter neither English nor French is well defined. It is therefore impossible to specify what counts as performing the task effectively. Roughly, we might say that someone has succeeded in translating a certain English text into French if their translation is agreed by the consensus of those qualified to judge that it is a good translation. This isn't the sort of language we use for determining whether the output of a Turing machine is correct!

The safest bet is to confine one's discussion to *Formal Languages*, i.e., well defined sets of strings of symbols. This actually covers everything that happens inside computers as we now know them, although the operation of some types of peripheral devices goes beyond this. To illustrate, suppose you do a watercolour painting and digitise it using a scanner; with the digital image you then apply various transformations, e.g., inverting the colours, distorting it by means of a shear transformation. The original watercolour is not a string of symbols, it is not even "equivalent" to one. The digitised image essentially *is* a string of symbols taken from some predetermined finite set. All the transformations you do on it are instances of effective procedures, and come within the scope of the Church-Turing thesis. They could certainly be accomplished by an appropriately configured Turing machine. It's abstract and mathematical through and through. The original digitisation, however, is not something that could be done by a Turing machine: the Turing machine only operates on input that is already digital.

So from now on, this is what we mean by a computational task: the mapping of strings from some well-defined formal input language into strings from some well-defined formal output language. More exactly, a *specification* for a computational task consists of

1. A formal language I, called the *input language*
2. A formal language O, called the *output language*
3. A function $f : I \to O$, called the *transformation*

Digression. One might object that the above characterisation only really applies to so-called *transformational computation*, whereas a lot of the computation we do has a more complicated structure; it is

reactive computation. Here there is an ongoing interaction between the computer and the outside world; an example is provided by an operating system, where we don't ask "what does an operating system compute?" but "how does it interact with its users?". The specification for a reactive system is more like

1. A formal language I, called the *input language*
2. A formal language O, called the *output language*
3. A formal language S, called the *state-space*
4. A function $f : I \times S \to O \times S$, called the *transformation*

A typical transaction of such a system goes like this: if the system is in state s and receives input i then it moves into state s' and delivers output o, where $f(i, s) = (o, s')$.

This apparently more complicated scenario can be assimilated into the one given earlier by the observation that both $I \times S$ and $O \times S$ are also formal langauges.
End of digression.

2 Turing Machines

A **Turing Machine** (TM) consists of a *black box* equipped with an *input/output device* (or *head*) and an infinite supply of *tape* divided into a linear sequence of discrete *cells*. On each tape cell may be written one out of a finite alphabet of *tape symbols* (including the *blank symbol* Δ, standing for an empty cell); at any stage the machine is in one of a finite number of possible *internal states* and the head is positioned over one of the tape cells, which it is said to be *scanning*.

A *computational step* consists of the following sequence of operations:

1. The head *reads* the symbol written on the cell it is scanning and passes this information into the black box;
2. The black box instructs the head to *erase* the symbol on the tape, to *replace* it with a new symbol, or to *leave* it as it is;
3. The machine then either changes to a new state or remains in its present state;
4. The head either *moves* one cell to the right or left along the tape, or remains scanning the same cell.

Exactly which action takes place at steps 2, 3, and 4 depends on the state the machine is in, what input it receives from the tape at step 1, and *nothing else*. So the behaviour of the machine is completely determined by specifying, for some set of state/symbol pairs (q, a), a new state/symbol pair (q', a') (where $a' = a$ if the symbol is to be left as it is, $a' = \Delta$ if it is to be erased), together with a displacement $d \in \{\text{left,right,no move}\}$. The collection of *quintuples* (q, a, q', a', d) is called the *program* of the machine.

One of the states is designated as the *start state*, and one state is designated as the *halting state*.

A computation is started with the TM in its start state, scanning a particular cell on the tape called the *start cell*. The tape can have only finitely many non-blank cells initially. The TM stops as soon as either it enters the halting state or there is no quintuple in the program beginning with the current state/symbol pair (q, a).

What happens depends on the machine's program together with the sequence of symbols initially written on the tape. There are three possibilities:

(a) The TM halts in the halting state (*halt and succeed*);

(b) The TM halts in a non-halting state (*halt and fail*);

(c) The computation goes on for ever.

In case (a) we say that the TM *accepts* as input the string of symbols initially on the tape, from the initial cell up to the first blank to the right, and delivers as output the string of symbols on the tape when it halts, from the final scanned cell up to the first blank to the right.

Formally, a Turing Machine is a sextuple $(Q, \Sigma, \Gamma, q_0, h, \delta)$ where

– Q is a finite set of *states*;

– Σ is a finite set of *input symbols*

– Γ is a finite set of *tape symbols*, such that $\Sigma \subset \Gamma$ and $\Delta \in \Gamma \setminus \Sigma$;

– $q_0 \in Q$ is the start state;

– $h \in Q$ is the halting state;

– $\delta : (Q \setminus \{h\}) \times \Gamma \to Q \times \Gamma \times \{-, 0, +\}$ is the program, a partial function.

The machine starts in a configuration

$$\cdots \Delta\Delta\Delta \quad \underset{q_0}{\boxed{t_1}} \quad t_2 \cdots t_n \Delta\Delta\Delta \cdots$$

where $t_i \in \Sigma$, and the machine is scanning symbol t_1 in state q_0.

During a computation, a typical configuration is

$$\cdots \Delta\Delta\Delta t_1 t_2 \cdots t_{h-1} \quad \underset{q}{\boxed{t_h}} \quad t_{h+1} \cdots t_n \Delta\Delta\Delta \cdots$$

The configuration assumed next is determined by the value of $\delta(q, t_h)$:

1. If $\delta(q, t_h) = (q', t'_h, -)$ then the new configuration is

$$\cdots \Delta\Delta\Delta t_1 t_2 \cdots \quad \underset{q'}{\boxed{t_{h-1}}} \quad t'_h t_{h+1} \cdots t_n \Delta\Delta\Delta \cdots$$

2. If $\delta(q, t_h) = (q', t'_h, 0)$ then the new configuration is

$$\cdots \Delta\Delta\Delta t_1 t_2 \cdots t_{h-1} \quad \underset{q'}{\boxed{t'_h}} \quad t_{h+1} \cdots t_n \Delta\Delta\Delta \cdots$$

3. If $\delta(q, t_h) = (q', t'_h, +)$ then the new configuration is

$$\cdots \Delta\Delta\Delta t_1 t_2 \cdots t_{h-1} t'_h \quad \underset{q'}{\boxed{t_{h+1}}} \quad \cdots t_n \Delta\Delta\Delta \cdots$$

Like a finite-state automaton (FA), a TM may be represented be means of a transition network. In the network, there is a node for each state, and whenever $\delta(q, a) = (q', a', d)$ there is an arrow in the network from q to q' and labelled (a, a', d).

Turing Machine Example 1.
$M_1 = (Q, \Sigma, \Gamma, q_0, h, \delta)$, where $\Sigma = \{0, 1\}$, $\Gamma = \Sigma \cup \{\Delta\}$, $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$, $q_0 = 1$, $h = 0$. The program δ is given by the transition diagram shown in Figure 1.

What language does this TM accept?

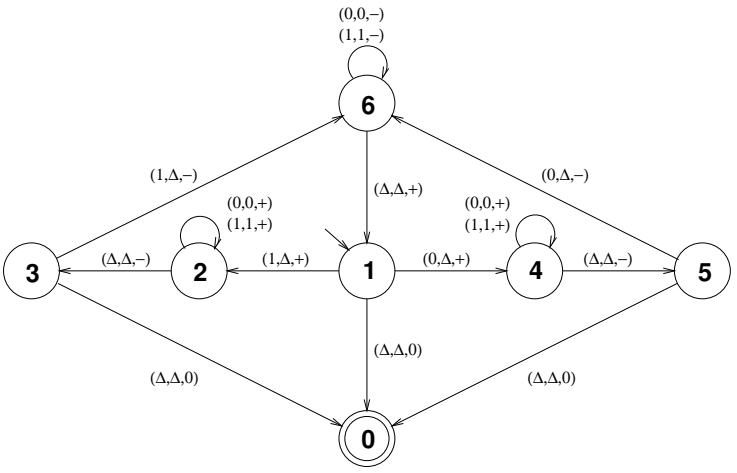

Figure 1: *A Turing Machine which accepts palindromes over* $\{0, 1\}$.

1. If ever it reads a symbol other than 0, 1, or Δ, it halts and fails. So $L(M_1) \subseteq \{0, 1\}^*$.

2. To succeed in traversing the loop 1-2-3-6-1, the configuration at state 1 must be $\cdots \Delta 1 X 1 \Delta \cdots$, where $X \in \{0, 1\}^*$; and on completing the loop the configuration is $\cdots \Delta X \Delta \cdots$. Thus each traversal of the loop strips off a 1 from each end. Similarly, traversal of the loop 1-4-5-6-1 strips off a 0 from each end of the string whose first symbol is scanned in state 1.

3. Suppose that after some number of traversals of the two loops, the machine reads a blank in state 1. Then it will immediately proceed to the halt state. It will have reached this state by successively stripping off pairs of 0s or 1s from the two ends of the string. This can only happen if the string is initially an *even-length palindrome*, e.g. 1101001011.

4. At state 3, the machine has removed a 1 from the beginning of the string and is reading the symbol at the end; if this is also a 1 it proceeds round the loop. If it is a zero, then the original string cannot have been a palindrome, in which case the machine cannot proceed and remains stuck in state 3. The same thing happens if it reads a 1 in state 5.

5. If in state 3, the machine, having removed a 1, is now scanning a blank cell, then it has deleted the whole initial string, and the 1 it has just removed must have been the middle symbol in an *odd-length palindrome* such as 00110101100. It proceeds to the halt state. The same thing happens if it reads a blank in state 5.

In sum, the machine will reach the halt state if and only if the input string is a palindrome over $\{0,1\}$. It will get stuck in state 3 or 5 if it is a non-palindrome over $\{0,1\}$. If the input string is not in the language $\{0, 1\}^*$, the machine will get stuck as soon as it reads the first symbol not in $\{0,1\}$. Thus the language accepted by M_1 is the set of palindromes over $\{0,1\}$. This is not a regular language.

The TM in Example 1 is a language acceptor. We can also use TMs as computers, delivering output in response to input. The **input to a TM** is the string of symbols on the tape at the start, reading from the initially scanned cell right as far as the first blank, but not including that blank (if the initially scanned cell is blank, then the input is Λ). If the TM is started with input x and run, then it delivers output iff it accepts x, in which case the **output of the TM** is again the string of symbols on the tape starting from the cell scanned at termination and reading to the right as far as, but not including, the first blank. The **function computed by the TM** is the function $f : \Sigma^* \to (\Gamma \setminus \{\Delta\})^*$ such that for input $x \in \Sigma^*$ the value of $f(x)$ is defined as the output given by the TM for input x, if it exists, otherwise it is undefined.

Turing Machine Example 2. The transition network shown in Figure 2 shows a TM which takes as input an expression of the form $M + N$, where M and N are natural numbers in binary notation, and delivers as output the binary numeral S representing the sum of M and N. For example, the input string '10110+101' yields the output string '11001' (the computation corresponding to 22+5=27 in decimal notation).

The machine works by fetching bits from the second argument one by one and adding them to the corresponding bits in the first argument, using i and o to represent 1 and 0 respectively. It does this in either a 'no carry' state (states 2–7) or a 'carry' state (states 9–11), according to the following addition table:

	0, no carry	0, carry 1, no carry	1, carry
0	0, no carry	1, no carry	0, carry
1	1, no carry	0, carry	1, carry

When there are no more bits in the second argument (states 12, 13), any remaining carries are handled, o and i are changed back to 0 and 1, and the machine halts.

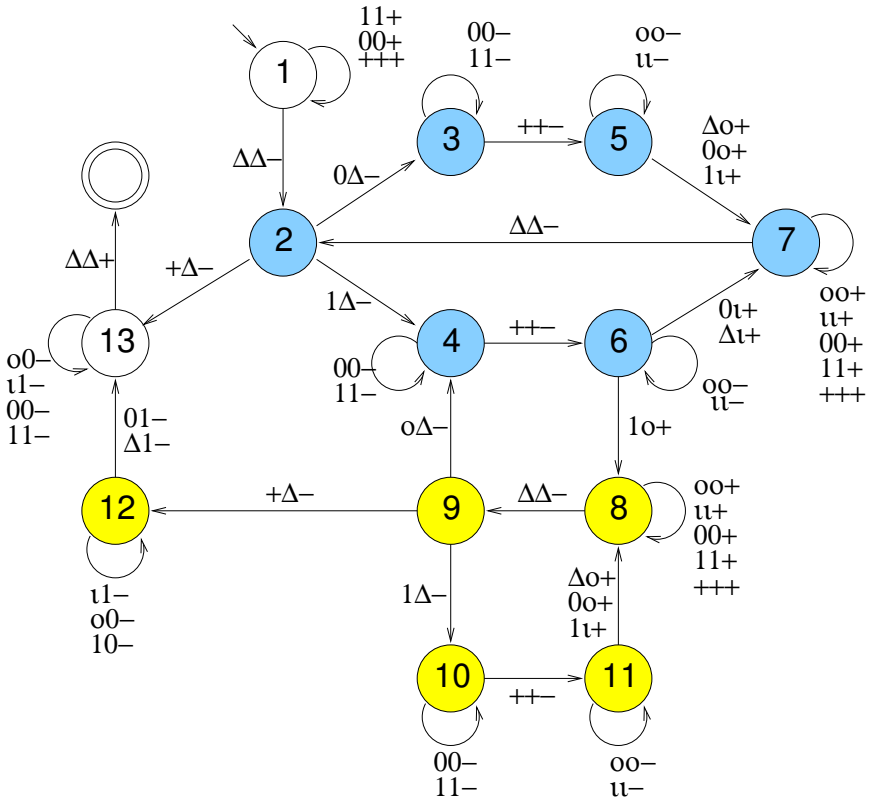

Figure 2: *A Turing Machine for Base-2 Addition.*

3 The Halting Problem

3.1 The Halting Problem for Turing Machines

We assume that we have an agreed encoding of Turing machines as strings over some fixed finite alphabet T. We shall assume a canonical representation of Turing machines in which the tape alphabet is also T (in fact we can work with $T = \{0, 1\}$, where 0 is used for the blank symbol). Though rather fiddly to work out the details, this is an essentially trivial matter. If $x \in T^*$ is an encoding of a Turing Machine, we shall write M_x to refer to the Turing Machine it encodes.

The **Halting function** h takes as input two strings $x, y \in T^*$ and returns as output 0 or 1 as follows:

- $h(x, y) = 1$ if the Turing machine M_x halts when presented with input y;
- $h(x, y) = 0$ otherwise.

We shall show that *the Halting function is uncomputable*. (This is often expressed as *the Halting problem is insoluble*.)

Notation: We write $M(x) \downarrow y$ to mean that when TM M is started with input x it halts with output y; we write $M(x) \downarrow$ to mean that it halts with unspecified output; and $M(x) \uparrow$ to mean that it does not halt.

Theorem: the uncomputability of the Halting function. We must show that no TM can compute h. (Note: this means that no TM will deliver output $h(x, y)$ for *all* possible inputs $x, y \in T^*$.) A machine H which computes h must satisfy

$$\begin{aligned} H(x, y) \downarrow 0 \quad &\text{if} \quad M_x(y) \uparrow \\ H(x, y) \downarrow 1 \quad &\text{if} \quad M_x(y) \downarrow \end{aligned}$$

If H exists then it can easily be modified to a machine H^* such that

$$\begin{aligned} H^*(x) \downarrow \quad &\text{if} \quad H(x, x) \downarrow 0 \\ H^*(x) \uparrow \quad &\text{if} \quad H(x, x) \downarrow 1 \end{aligned}$$

All H^* need do is to copy its argument and then act like H except that when H would deliver output 1, H^* goes into a loop (this is easy to arrange).

Now since H^* is a Turing machine, $H^* = M_w$ for some $w \in T^*$. Giving H^* its own representation, w, as input, we get:

$$\begin{aligned} H^*(w) \downarrow \text{ if } H(w, w,) \downarrow 0 \text{ if } M_w(w) \uparrow \text{ if } H^*(w) \uparrow \\ H^*(w) \uparrow \text{ if } H(w, w,) \downarrow 1 \text{ if } M_w(w) \downarrow \text{ if } H^*(w) \downarrow \end{aligned}$$

so $H^*(w)$ halts if and only if it doesn't halt. From this contradiction we deduce that H doesn't exist, i.e., the Halting function is uncomputable.

3.2 The Halting Problem for Computer Programs

You may find it more convincing if we recast it in terms closer to the programming languages you are familiar with. I shall assume we are working in a programming language $Proglang$ whose syntax includes the following constructions:

input x	Input a new value for the variable x
output x	Output the current value of the variable x
$x \leftarrow V$	Assign the value V to the variable x
$x = y$	The values of variables x and y are equal
while C **do** A	Repeat action A for as long as condition C holds

The problem is to find a $Proglang$ program which, when given as input the code of an arbitrary $Proglang$ program P and input string i, will determine whether or not P would halt when run with input i.

Assume we have such a program, which we shall call H. This can be written in the form

input P;
input i;
A;
output h

where A is itself a well-formed piece of $Proglang$ code, and h is "yes" or "no" according as P does or does not halt when run with input i.

We now modify H to the following program I:

input P;
$i \leftarrow P$;
A;
while $h =$"yes" **do** $h \leftarrow$"yes";
output h

Assuming H does what is claimed for it, then I behaves as follows:

> For any program P, the program I halts when run with input P if and only if P fails to halt when run with input P.

But this behaviour is impossible! For there is nothing to stop us from entering I as input to itself; in which case we would have

> I halts when run with input I if and only if I fails to halt when run with input I

which is a blatant contradiction. We have proved that (1) if H exists, then I exists, and (2) I does not exist, from which it follows that (3) H does not exist.

3.3 A note on the Church-Turing thesis

What we have proved is that the Halting Problem for Turing Machines cannot be solved *by a Turing Machine*; and that the Halting Problem for Proglang programs cannot be solved *by a Proglang program*. This does not necessarily mean that these problems cannot be solved in some other way. But in fact it is generally believed that they cannot. This is because of the Church-Turing thesis, which asserts that a function is effectively computable if and only if there is a Turing Machine which computes it. This, if true, must apply to the Halting function h, and since we have shown that no Turing Machine can compute this, it means that it is not effectively computable.

Why should we believe the Church-Turing thesis? Turing Machines were developed from a close analysis of the possible kinds of operations that any computing mechanism could use (Turing originally modelled them on *human* computers working things out with pencil and paper). Around the same time as Turing did this, Church developed a theory of recursive functions called the the λ-calculus, and he put forward the claim that a function is effectively computable if and only if it can be defined within the λ-calculus. Turing then proved that the λ-definable functions are precisely the same as the Turing-computable ones, which lent support to the idea that these two independently developed ideas have succeeded in capturing a robust and universally valid notion of effective computability. Subsequently every attempt to lay down a formal definition of effective computability has led to a system that can be proved to be equivalent to Turing Machines.[1]

[1] See also Antony Galton, 'The Church-Turing Thesis: Its Nature and Status', in P. J. R. Millican and A. Clark, *Machines and Thought*, Clarendon Press, Oxford, 1996, pp. 137–164, and Antony Galton, 'The Church-Turing thesis: Still valid after all these years?', *Applied Mathematics and Computation*, 178 (2006), 93–102.

4 The Decision Problem for First-Order Logic

The decision problem for first-order logic (Hilbert's *Entscheidungsproblem*) may be stated as follows:

> Given an arbitrary set Σ of first-order formulae, and a first-order formula P, determine whether or not $\Sigma \models P$.

What is wanted is an algorithm for the decision problem. Thus the theoretical question here is whether or not the relation of validity expressed by "$\Sigma \models P$" is decidable.

We shall show that this problem is *equivalent* to the Halting Problem for Turing machines. For simplicity we shall restrict the discussion to Turing machines having only two symbols in its tape alphabet, which we shall designate "0" (for a blank square) and "1" (for a marked square). We know that any Turing machine can be converted to an essentially equivalent machine of this kind.

Assume that the states of the Turing machine are labelled $0, 1, 2, 3, \ldots, n$, where 1 is the start state and 0 is the halt state. The Turing machine M is fully defined by three functions T, U, D:

- $T : \mathbb{N} \times \{0,1\} \to \mathbb{N}$ is the function which, given state q and currently scanned tape-symbol a, determines what the next state will be. If no transition is defined for the pair (q, a), then $T(q, a) = q$.
- $U : \mathbb{N} \times \{0,1\} \to \{0,1\}$ is the function which, given q and a, determines what symbol is written onto the tape. If no transition is defined for (q, a), we put $U(q, a) = a$.
- $D : \mathbb{N} \times \{0,1\} \to \{-1, 0, 1\}$ is the function which, given q and a, determines the displacement, i.e., whether the machine's scanning head moves left or right along the tape or stays in the same position. If no transition is defined for (q, a), we put $D(q, a) = 0$.

A *configuration* C for the machine M can be specified by means of four items, namely

- $q_C \in \mathbb{N}$, the current state;
- $a_C \in \{0,1\}$, the currently scanned symbol;
- $b_C = b_1 b_2 \ldots b_n$, the string of symbols running to the left of the currently scanned square as far as the leftmost non-blank symbol (if this string is empty, then $b_c = \Lambda$);
- $c_C = c_1 c_2 \ldots c_m$, the string of symbols running to the right of the currently scanned square as far as the rightmost non-blank symbol (if empty, $c_C = \Lambda$).

These items are illustrated in the diagram below.

$$\cdots 000 b_n \cdots b_3 b_2 b_1 \quad \boxed{a} \quad c_1 c_2 c_3 \cdots c_m 000 \cdots$$
$$q$$

An *initial configuration* is any configuration which has $q_C = 1$. A *halting configuration* is any configuration which has $q_C = 0$. If the machine M is started in an initial configuration $C(0)$, it will run through a sequence of configurations which we shall denote

$$C(0), C(1), C(2), C(3), C(4), C(5), \ldots$$

Here C is a function from $\mathbb{N}$ to $\mathcal{C} = \mathbb{N} \times \{0,1\} \times \{0,1\}^* \times \{0,1\}^*$, the set of all possible configurations for M. If for any i, $C(i)$ is a halting configuration, then $C(i+1), C(i+2), C(i+3), \ldots$ will all equal $C(i)$, since there are no transitions out of the halt state. For a general configuration $C(i)$, if no transitions are defined for $(q_{C(i)}, a_{C(i)})$, then all subsequent configurations will equal $C(i)$.

Each step in this sequence of $C(i)$ is determined from the previous one by means of the functions T, U, D defining the operation of the Turing machine. Let $C(i) = (q, a, b, c)$, where $b = b_1 b_2 \ldots b_n$ and $c = c_1 c_2 \ldots c_m$; then $C(i+1) = (q', a', b', c')$, where

$$
\begin{aligned}
q' &= T(q,a) \\
a' &= \begin{cases} b_1 & (\text{if } D(q,a) = -1) \\ U(q,a) & (\text{if } D(q,a) = 0) \\ c_1 & (\text{if } D(q,a) = 1) \end{cases} \\
b' &= \begin{cases} b_2 \ldots b_n & (\text{if } D(q,a) = -1) \\ b & (\text{if } D(q,a) = 0) \\ U(q,a) b_1 \ldots b_n & (\text{if } D(q,a) = 1) \end{cases} \\
c' &= \begin{cases} U(q,a) c_1 \ldots c_m & (\text{if } D(q,a) = -1) \\ c & (\text{if } D(q,a) = 0) \\ c_2 \ldots c_m & (\text{if } D(q,a) = 1) \end{cases}
\end{aligned}
$$

Clearly C is a computable function: everything we need to know to compute its value for any particular inputs is given by the formulae above, once the value of $C(0)$ is given.

Given a configuration C it is a simple matter to determine whether or not it is a halting configuration: simply determine whether $q_C = 0$. Thus the formula $H(n)$, which says that when machine M is started in configuration $C(0)$ it will be in a halting configuration after n steps, is decidable. In other words, *we can reliably determine, for any Turing machine and any initial tape, whether or not it has reached the halt state after any given number of steps*. This is not the same as solving the Halting Problem, though! To solve the Halting Problem, we need to be able to determine, for any Turing machine and any initial tape, whether or not it *ever* reaches the halt state. This is the problem that Turing showed could not, in general, be solved.

Turing[2] handled the Entscheidungsproblem by showing that a certain class of inferences in first-order logic was equivalent to the halting problem[3] for Turing machines. For each inference in the class, validating it is equivalent to determining whether or not a particular Turing machine halts when given a particular input tape. This means that there can be no decision procedure for that class of inferences, and hence for first-order logic in general, since any such decision procedure would enable us to solve the Halting problem.

Here's how it's done. We shall work with the following non-logical vocabulary:

- Constants 0, 1, -1, and nil.
- Function symbols $suc, head, tail, cons, config, T, U, D, C$.
- Predicate H.

We use suc with 0 to generate an infinite string of terms corresponding to the natural numbers (for denoting machine states). We use nil, $head$, $tail$ and $cons$ for constructing strings to represent the contents of the tape (b and c in the above formalism). We use $config$ to construct a configuration $config(q, a, b, c)$ out of its components. T, U, D, C, H are as above. We use single lower-case letters for variables (e.g., a, b, c, q).

For a machine with n non-halting states we set up our premisses as follows:

[2]Turing's paper is entitled *On Computable Numbers, with an Application to the Entscheidungsproblem* (1936). It is reprinted in M. Davis, *The Undecidable*, Raven Press, New York, 1965 and in B. J. Copeland, *The Essential Turing*, Clarendon Press, Oxford, 2004.

[3]As a matter of fact, Turing's paper used the *Printing Problem*: If machine T is run with tape t, will it ever print a specified tape symbol on the tape? This can be shown to be equivalent to the Halting Problem, and later treatments almost invariably substitute the latter for the former.

1. First, a set of formulae defining the Turing machine we are representing. With q ranging from $suc(0)$ to $suc^n(0)$ and a taking each of the values 0 and 1 in turn, we include the $6n$ formulae of the forms

   (a) $T(q, a) = q'$ (with q' one of $0, suc(0), \ldots, suc^n(0)$ in each case)

   (b) $U(q, a) = a'$ (with a' one of $0, 1$ in each case)

   (c) $D(q, a) = d$ (with d one of $-1, 0, 1$ in each case)

2. Next, a formula giving the initial configuration, of the form

$$C(0) = config(1, a, b, c)$$

   where a is 0 or 1, and both b and c are strings constructed from $cons$, 0, and 1 (e.g., $cons(1, cons(0, cons(0, 1)))$, representing the string 1001).

3. Next, three formulae defining the relationship between each machine configuration and the one which follows it, one formula for each of the possible displacements:

   (a) $\forall q, a, b, c, d, i(C(i) = config(q, a, b, c) \wedge D(q, a) = -1 \rightarrow$
   $C(suc(i)) = config(T(q, a), head(b), tail(b), cons(U(q, a), c)))$

   (b) $\forall q, a, b, c, d, i(C(i) = config(q, a, b, c) \wedge D(q, a) = 0 \rightarrow$
   $C(suc(i)) = config(T(q, a), U(q, a), b, c))$

   (c) $\forall q, a, b, c, d, i(C(i) = config(q, a, b, c) \wedge D(q, a) = 1 \rightarrow$
   $C(suc(i)) = config(T(q, a), head(c), cons(U(q, a), b), tail(c)))$

4. We also need some general (i.e., not machine-specific) axioms to ensure that suc, $head$, $tail$ and $cons$ behave in the right way:

   (a) $\forall x(\neg suc(x) = 0)$

   (b) $\forall x, y(suc(x) = suc(y) \rightarrow x = y)$

   (c) $\forall x, y(head(cons(x, y)) = x)$

   (d) $\forall x, y(tail(cons(x, y) = y)$

   (e) $head(nil) = 0$

   (f) $tail(nil) = nil$

5. Finally, we define what it is for the computation to halt after a given number of steps:

$$\forall i(H(i) \leftrightarrow \exists a, b, c\, C(i) = config(0, a, b, c)).$$

All the above formulae together define completely the computation performed by a given machine (the one defined by the formulae giving the values of T, U, D) with a given starting configuration (defined by the formula giving the value of $C(0)$). This computation either halts or it does not, that is, exactly one of the formulae $\exists i H(i)$ and $\neg\exists i H(i)$ must be a logical consequence of that set of formulae. Note that a set of formulae like this can be set up for *every* Turing machine and *every* possible initial configuration.

Now suppose there were a decision procedure for first-order logic, that is, an algorithm for determining whether or not an arbitrary conclusion follows from an arbitrary set of premisses. Then this procedure could be applied to the premisses describing any particular Turing machine configuration to determine which of the formulae $\exists i H(i)$ and $\neg\exists i H(i)$ is implied by them. In this way the Halting Problem could be solved for arbitrary Turing machine computations. But we know that the Halting problem can't be solved in general. It follows that there can be no general decision procedure for first-order

logic. That is how Turing solved Hilbert's *Entscheidungsproblem* — or rather, proved that it could not be solved.

Although the decision problem for first-order logic is *in general* insoluble, quite a lot is known about the cases for which it is soluble, the *decidable classes of first-order formulae*. For example, the decision problem is soluble for formulae containing only one-place predicates, formulae whose prenex form contains only existential quantifiers, formulae whose prenex form does not have an existential quantifier in front of a universal quantifier, formulae whose prenex form does not contain more than one existential quantifier, as well as numerous other types[4]. By the **prenex form** of a formula is meant the equivalent formula in which all the quantifiers have been moved to the beginning, and modified as necessary—e.g., the prenex form of $\forall x((\exists y P(y)) \rightarrow \exists z Q(z))$ is $\forall x \forall y \exists z (P(y) \rightarrow Q(z))$. All first-order formulae can be converted into prenex form. Some examples are listed in the table below.

Original formula	Prenex form
$\forall x(P(x) \rightarrow \exists y Q(x,y))$	$\forall x \exists y(P(x) \rightarrow Q(x,y))$
$\forall x P(x) \wedge \forall x Q(x)$	$\forall x(P(x) \wedge Q(x))$
$\exists x P(x) \wedge \exists x Q(x)$	$\exists x \exists y(P(x) \wedge Q(y))$
$\exists x P(x) \vee \exists x Q(x)$	$\exists x(P(x) \vee Q(x))$
$\forall x P(x) \vee \forall x Q(x)$	$\forall x \forall y(P(x) \vee Q(y))$
$\exists x P(x) \rightarrow \forall x Q(x)$	$\forall x \forall y(P(x) \rightarrow Q(y))$
$\forall x P(x) \rightarrow \exists y Q(y)$	$\exists x \exists y(P(x) \rightarrow Q(y))$

[4] See W. Ackermann, *Soluble Cases of the Decision Problem*, 1954.