4. Universal Turing machines

We nowadays accept that a single computer can solve a vast range of problems, ranging from astronomical calculations to graphics and process control. But before computers were invented there were many kinds of problem-solving machines, with quite different 'hardware'. Turing himself helped to design code-breaking equipment with dedicated hardware during the second world war. These machines could do nothing but break codes. Turing machines themselves come in different kinds, with different alphabets, state sets, and even hardware (many tapes, etc).

It was Turing's insight that this proliferation is unnecessary. In his 1936 paper he described a single general-purpose Turing machine, that can solve all problems that any Turing machine could solve.[8]

This machine is called a *universal Turing machine.* We call it U. U is not magic — it is an ordinary Turing machine, with a state set, alphabet, etc, as usual. If we want U to calculate $f_M(w)$ for some arbitrary Turing machine M and input w to M, we give U the input w plus a description of M. We can do this because $M = (Q,\Sigma,I,q_0,\delta,F)$ can be described by a finite amount of information. U then evaluates $f_M(w)$ by calculating what M would do, given input w — rather in the way that the 1-tape Turing machine simulated a 2-tape Turing machine in §3.3.5.

So really, U is programmable: it is an interpreter for arbitrary Turing machines. In this section we will show how to build U.

4.1. Standard Turing machines

In fact we have been lying! U will not be able to handle arbitrary Turing machines. For example, if M has a bigger input alphabet than U does, then some legitimate input words for M cannot be given to U at all. There's a similar problem with the output. So when we build U, we will only deal with the restricted case of *standard Turing machines.* This just means that their alphabet is fixed. Though the 'computer alphabet' {0,1} is often used for this purpose, we will use the following, more convenient *standard character set.* In §4.4 we will indicate why using a fixed alphabet is not really a restriction at all.

8 The idea led to his later work on building real computers.

4.1.1. Definition

1. We let C be the alphabet {a,b,c,…,A,B,…,0,1,2,…,!,@,£,…} of characters that you would find on any typewriter (about 88 in all; note that $\wedge$ is not included in C).

2. A Turing machine S is said to be standard if:
(a) it conforms exactly to the definition in §2.1, and
(b) its input alphabet is C and its full alphabet is $C \cup \{\wedge\}$.

4.1.1.1. Warning

By (a) of this definition, we know that:

- S has a single one-way infinite tape (a multi-tape TM is a *variant* of the TM defined in §2.1).
- the tape of S has only one track
- any marking of square 0 is done explicitly.

Extra tracks and implicit marking of square 0 are implemented by adding symbols to the alphabet (see §2.4). There is no point in fixing our alphabet as C, and then changing it by adding these extra symbols.

4.2. Codes for standard Turing machines

We need a way of describing a standard Turing machine to U. So we introduce a key notion, that of *coding* a Turing machine, so we can represent it as data. We will code each standard Turing machine S by a word *code*(S) of C, in such a way that the operations of S can be reconstructed from *code*(S) by an algorithm. So:

- S is a standard Turing machine;
- *code*(S) will be a word of C, representing S.

Then we will design U so that $f_U(code(S)*w) = f_S(w)$ for all standard Turing machines S and all words $w \in C^*$.[9]

4.2.1. Details of the coding

Let $S = (Q,C\cup\{\wedge\},C,q_0,\delta,F)$ be any standard Turing machine. Let us suppose that $Q = \{0,1, \dots, n\}$, $q_0 = 0$ and $F = \{f,f+1, \dots, n\}$ for some $n\geq 0$ and some $f\leq n$. (There is no loss of generality in making this assumption: see §4.2.3.1 below.)

Much as in §1.3.3.1, we think of the instruction table δ as a list of 5-tuples, of the form

$$(q,s,q',s',d)$$

9 We will input the pair (*code*(S),w) to U by giving it the string *code*(S) concatenated with the string w, with a delimiting character, say *, in between. We could just give U an extra tape and put *code*(S) on tape 1 and w on tape 2.

where $\delta(q,s) = (q',s',d)$. For each 5-tuple in the list we have:

$$0 \le q < f \qquad 0 \le q' \le n \qquad s, s' \in C \cup \{\wedge\} \qquad d \in \{-1, 0, 1\}.$$

S is then specified completely by this list, together with the numbers n and f.

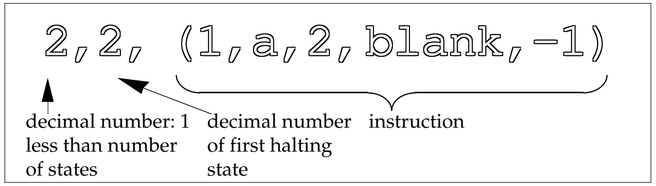
There are many ways of coding this information. We will use a simple one. Consider the word

$$n,f,t_1,t_2, \ \ldots, \ ,t_N$$

where the list of 5-tuples is $t_1,t_2, \ldots, t_N$ in some arbitrary order[10], and all numbers (n, f and the numbers q, q' and d in the 5-tuples) are written in decimal (say). This is a word of our coding alphabet $C \cup \{\wedge\}$. We let *code(S)* be the word of C obtained from this by replacing every '$\wedge$' by the five-letter word 'blank'. (As we will be giving *code*(S) as input to Turing machines, we don't want $\wedge$ to appear in it.)

4.2.2. Checking whether a word codes a TM

If we have a word w of C, we can check by an algorithm if w is the code for a Turing machine. Eg. ';;()101,y%-)' is no good, whilst



4.1 *code*(a very simple TM)

is OK (3 states, 0, 1, 2; state 2 is halting; if in state 1 and read 'a', go to state 2, write '$\wedge$' and move left). In general, w must have the form

$$n,f,(x,y,x,y,d),(x,y,x,y,d), \ldots ,(x,y,x,y,d)$$

where n, f, x and d are decimal numbers with $0 \le f \le n$, $0 \le x \le n$ and $-1 \le d \le 1$, and y is some single character of C, or 'blank'.

4.2.2.1. Exercise

There are several other checks (to do with final states, functionality of δ, and more) to be made before we are sure w is a code for a genuine TM. Explain these.

In a 5-tuple (x,y,x,y,d), each y could be any of:

( ) , blank

Is the code of S really unambiguous?

[10] Unlike in a conventional computer, the order of the instructions t_i is not part of S and clearly does not affect the way S works.

4.2.3. Remarks on the coding

4.2.3.1. Generality of coding

We assumed that Q = {0, 1, …, n} and F = {f, f+1, …, n} for some f≤n. This is not really a restriction, because given any old standard Turing machine S = $(Q, C\cup\{\wedge\}, C, q_0, \delta, F)$, we can always rename its states without changing its behaviour, so long as we then adjust q_0, δ and F accordingly.

We can therefore rename the states in Q to 0, 1, …, n (where S has n+1 states), so that

- the initial state is 0
- the final states come at the end of the listing — i.e., they are f, f+1, …, n for some f≤n.

Note: if q_0 were also a final state of S, we could not assume that F consists of the states at the end of the list. Eg. we might have $F = \{q_0, q_{273}\}$. So our coding would not work. But such a TM would halt immediately, so we can take its code to be '0,0,' or the code of any other Turing machine that halts immediately, as all such machines have the same input-output function (namely the identity). Similarly, if F=Ø then the Turing machine never halts and succeeds, so it never outputs anything, and we can take its code to be '1,1', or the code of any other Turing machine that never halts and succeeds.

4.2.3.2. (Non-)uniqueness of code of S

There are many ways of renaming the states Q of S to 0, 1, …, n. And for a given S we can list the instruction 5-tuples $t_1,\ldots,t_N$ in many different orders. We get a different word *code*(S) representing the same S, for each possible renaming and ordering.

So *code*(S) is really a *relation*, because it is not uniquely defined. We don't mind this. Below, where *code*(S) comes up, we don't care which actual code is used; any order of instructions or states will do. There is still an algorithm to tell whether any given w is the code of some Turing machine S (with the instructions listed in <u>some</u> order). If we really wanted a *function* from standard Turing machines to words, we could let *code*(S) be the <u>first</u> word of C (in alphabetical order) that codes S.

Exercise: could we design a TM to decide whether words w_1, w_2 of C code the <u>same</u> TM?

4.2.3.3. Standard is needed

We could not code a non-standard TM without more definitions. First, an arbitrary M may have alphabet $\Sigma \ne C$ — or it may have alphabet C but uses many tracks, which comes to the same thing. As *code*(M) must be a word of C, not of Σ, we'd need to represent each symbol in Σ by a symbol or word of C. If M had alphabet C but used more than one tape, we'd have to change the instruction format: eg. the instruction table of a 3-tape machine is a list of 11-tuples!

4.2.3.4. *Other codings*

Our coding has some redundancy. Eg. we don't really need the brackets '(' and ')', or the number n at the front (why not?). There are also other, rather different codings available. For example, Rayward-Smith's book gives one using prime factorisation, in which the code of S is always a number, usually called the *Gödel number* of S. Ex: how could we turn our *code*(S) into a number?

We stress that these are only details. U only needs to be able to recover the workings of S from *code*(S). We can use any coding that allows this.

4.2.4. Summary

The point behind these details is that each standard Turing machine S can be represented by a finite piece of information, and hence can be coded by a word *code*(S) of C, in such a way that we can reconstruct S from code(S). The word *code*(S) $\in$ C* carries all the information about S. It is really a *name* or *plan* of S.

4.3. THE UNIVERSAL TURING MACHINE

Now we can build the universal machine U. It has the following specification:

> If the input to U is *code*(S)*w,[11] where S is a standard Turing machine and w $\in$ C*, then U will output $f_S(w)$.
> If the input is not of this form, we don't care what U does.
> That is,
> • $f_U(code(S)*w) = f_S(w)$
> for all standard Turing machines S and all w $\in$ C*.

The input alphabet of U will be C, but U will not be standard, as it will have 3 tapes with square 0 (implicitly) marked.[12]

How does U work? Suppose that

$$S = (\{0,1,\ldots,n\}, C \cup \{\wedge\}, C, 0, \delta, \{f, f+1, \ldots, n\})$$

for some f≤n. Assume the input to U is *code*(S)*w. U will simulate the run of S on input w. We will ensure that at each stage during the simulation:

- tape 1 keeps its original contents *code*(S), for reference;
- tape 2 is always the same as the current tape of S;
- head 2 of U is always in the same position as the head of S;
- tape 3 holds the current state of S, in the same decimal format as in the instructions on tape 1. So eg. if S is in state 56, the contents of tape 3 of U are '5' in square 0 and '6' in square 1, the rest being blank.

Step 1: Initialisation: U begins by writing 0 in square 0 of tape 3. The rest of tape 3 is already blank, so it now represents the initial state 0 of S. U then copies w from tape 1 to tape 2. (The word w must be whatever is after the pair of characters ')*' or a string of the form 'n,f,*' on tape 1, so U can find it.) It then returns all three of its heads to square 0. The three tapes (and head 2) are now set up as above.

Step 2: Simulation: For each execution step of S, U does several things.

• Maybe the current state q of S is a halting state. So first, U compares the number q on tape 3 with the number f in *code*(S). U can find what f is by looking just after the first ',' on tape 1. They are in the same decimal format, so U can use a simple string comparison to check whether q<f or q≥f.

• If q≥f, this means that S is now in a halting state. Because tape 2 of U is always the same as the tape of S, the output of S is now on tape 2 of U. U now copies tape 2 to tape 1, terminated by a blank, and halts & succeeds.

• If q<f then q is not a halting state, and S is about to execute its next instruction. So head 1 of U scans through the list of instructions (the rest of *code*(S), still on tape 1) until it finds a 5-tuple of the form (q,s,q',s',d) where:

- q (as above) is S's current state as held on tape 3. Head 3 repeatedly moves along in parallel with head 1, to check this.
- s is the symbol that head 2 is now scanning — i.e., S's current symbol. A direct comparison of the symbols read by heads 1 and 2 will check this. (If s is 'blank', U tests whether head 2 is reading $\wedge$.)

• If no such tuple is found on tape 1, this means that S has no applicable instruction, and will halt and fail. Hence U halts and fails too (eg. by moving heads left forever).

• So assume that U has found on tape 1 the part '(q,s' of the instruction (q,s,q',s',d) that S is about to execute. S will write s', move its head by d, and change state to q'. To match this, U needs to know what s', q' and d are. It finds out by looking further along the instruction 5-tuple it just found on tape 1, using the delimiter ',' to keep track of where it is in the 5-tuple.[13] Head 2 of U can now write s' at its current location (by just copying it from tape 1), and then move by d (d is also got from tape 1). Finally, U copies the

---

[11] Recall that *code*(S) is not unique. But any code for S carries all the information about S. In fact, U will output $f_S(w)$ given input s*w, where s is any code for S.

[12] If we wish we can use Theorem 3.3.5 to find a one-tape equivalent of U, and then, as the output of U will be a word of C (why?), apply §4.4.2.1 below to find a standard TM equivalent to U.

[13] The awful possibility that s and/or s' is the delimiter ',' can be got round by careful counting.

decimal number q' from tape 1 to tape 3, replacing tape 3's current contents. After returning head 1 to square 0, U is ready for the next step of the run of S. It now repeats Step 2 again.

Thus every move of S is simulated by U. Clearly the output of U is just $f_S(w)$, and U halts and succeeds if and only if S does. Hence $f_U(code(S)*w) = f_S(w)$, and U is the universal machine we wanted.

Questions

1. What does U do if S tries at some step to move its head left from square 0 of its tape?

2. (Important) Why do we not hold the state of S in the state of U (cf. storing a finite amount of information in the states, as in §2.4.1)? After all, the state set of S is finite!

3. By using §3.3.5 and §4.4.2.1 below we can replace U with an equivalent standard TM. So we can assume that U is standard, so that *code*(U) exists and is a word of C.

Let $w \in C^*$. What is $f_U(code(U)*code(S)*w)$?

Using an interpreter was a key step in our original paradox, and so we are now well on the way to rediscovering it in the TM setting. In fact we will not use U to do this, but will give a direct argument. Nonetheless, U is an ingenious and fascinating construction — and historically it led to the modern programmable computer.

4.4. CODING

A Turing machine can have any finite alphabet, but the machine U built above can only 'interpret' standard Turing machines, with alphabet C. This is not a serious restriction. Computers use only 0 and 1 internally, yet they can work with English text, Chinese, graphics, sound, etc. They do this by *coding.* Coding is not the secret art of spies — that is *cryptography. Coding* means turning information into a different (eg. condensed) format, but in such a way that nothing is lost, so that we can *decode* it to recover the original form. (Cryptography seeks codings having decodings that are hard to find without knowing them.) Examples of codings are ASCII, Braille, hashing and condensing techniques, Morse code, etc. A computer stores graphics in coded (eg. bit-mapped) form. Here we will indicate briefly how to use coding to get round the restriction that U can only 'do' standard machines.

4.4.1. Using the alphabet C for coding

Just as ASCII codes English text into words of {0,1}, so the characters and words of any finite alphabet Σ can be coded as words of C.

1. C has about 88 characters, so we choose a whole number k such that:
   number of words of C of length k $= 88^k \geq$ size of Σ.
2. We can then assign to each $a \in \Sigma$ a unique word of C of length k. We write this word as *code*(a). There are enough words of C of length k to ensure that no two different symbols of Σ get the same code. (Formally we choose a 1-1 function $code : \Sigma \to C^k$; exactly what function we choose is not important).
3. We can now code any word $w = a_1a_2\ldots a_n$ of Σ, by concatenating the codes of the letters of w:
   $$code(\varepsilon) = \varepsilon$$
   $$code(a_1a_2\ldots a_n) = code(a_1).code(a_2)\ldots code(a_n) \in C^*$$
   This is just as in ASCII, Morse, etc. $Code(a_1a_2\ldots a_n)$ is a word of C of length kn.
4. We also need to *decode* the codes. There is a unique partial function $decode : C^* \to \Sigma^*$ given by:
   - $decode(code(w)) = w$ for all $w \in \Sigma^*$
   - *decode*(v) is undefined if v is a word of C* that is not of the form *code*(w).

For any finite Σ, we can choose k and a function *code* as above, and define *decode* accordingly.

As an example of the same idea, we can code words of C itself as words of {0,1}, using ASCII. We have eg. *code*(<space>) = 01000000, *code*(AB) = 0100000101000010, and *decode*(01000100) = D.

4.4.2. Scratch characters

Coding helps us in two ways. First, a Turing machine will often need to use *scratch characters:* characters that are used only in the machine's calculations, and do not appear in its input or output. Examples are characters like $(a_1,\ldots,a_n)$ used in multi-track work (§2.4.2); we also use scratch characters for marking square 0 (§2.4.3). We now show that in fact, scratch characters are never strictly needed. We do this only for standard Turing machines, but the idea in general is just the same.

4.4.2.1. Theorem (elimination of scratch characters)

Let $M = (Q,\Sigma,C,q_0,\delta,F)$ be a Turing machine with input alphabet C and any full alphabet Σ. Suppose that $f_M : C^* \to C^*$. I.e., the output of M is always a word of C. Then there is a standard Turing machine S that is equivalent to M.

Proof sketch (cf. Rayward-Smith, Theorem 2.6)

The idea is to get S to mimic M by working with codes throughout. Choose an encoding function $code : \Sigma \to C^*$. The input word w is a word of

C. But as $\Sigma \supseteq C$, w is also a word of Σ, so w itself can be coded! S begins by encoding w itself, to obtain a (longer) word *code*(w) of C. S can then simulate the action of M, working with codes of characters all along. Whatever M does with the symbols of Σ, M* does with their codes. If the simulation halts, S can decode the information on the tape to obtain the required output. The decoding only has to cope with codes of characters in $C \cup \{\wedge\}$, as we are told that the output consists only of characters in C. Because S simulates all operations of M, we have $f_S = f_M$, so S and M are equivalent. At no stage does S need to use any other characters than $\wedge$ or those in C. So S can be taken to be standard. QED.

We can now use U to interpret any Turing machine M with input alphabet C and such that $f_M : C^* \to C^*$. We first apply the theorem to obtain an equivalent standard Turing machine S, and then pass *code*(S) to U.

4.4.3. Replacing a Turing machine by a standard one

But what if the input alphabet of M is bigger than C? Maybe ακφδσφγϕ⊥←⊗∂ is a possible input word for M; we are not allowed to pass this to U, as it's not a word of U's input alphabet. But M is presumably executing some algorithm, so we'd like U to have a crack at simulating M.

Well, coding can help here, too. Just as computers can do English (or Chinese) word processing with their limited 0-1 alphabet, so we can design a new Turing machine M* that parallels the action of M, but working with *codes* of the characters that M actually uses. We'll describe briefly how to do this; it's like eliminating scratch characters.

Assume M has full alphabet Σ. Σ could be very large, but it is finite (because the definition of Turing machine only allows finite alphabets). Choose a coding function $code : \Sigma \to C^*$. Where M is given input $w \in C^*$, we'll give *code*(w) to M*. From then on, M* will work with the codes of characters from Σ just as in §4.4.2.1 above. M will halt and succeed on input w if and only if M* halts and succeeds on input *code*(w). The output of M* in this case will be $code(f_M(w))$, the code of M's output, and this carries the same information as the actual output $f_M(w)$ of M.

4.5. Summary of Section

We have built a universal Turing machine U. U can simulate any standard Turing machine S (i.e., one with input alphabet C and full alphabet $C \cup \{\wedge\}$), yielding the same result as S on the same input. We only have to give U the additional information *code*(S) — i.e., the *program* of S. So U serves as an interpreter for TMs.

To this end we explained how to specify a standard TM as a coded word. Standard TMs use the standard alphabet C. We showed how standard machines are in effect as good as any kind of TM, by coding words of an arbitrary alphabet as words of C and having a standard Turing machine work directly on the codes. We used this idea to eliminate the need for scratch characters.

5. Unsolvable problems

5.1. Introduction

In this section we will show that some problems, although not vague in any way, are inherently unsolvable by a Turing machine. Church's thesis then applies, and we conclude that there is no algorithm to solve the problem.

5.1.1. Why is this of interest?

- Many of these problems are not artificial, 'cooked-up' examples, but fundamental questions such as 'will my program halt?' whose solutions would be of great practical use.
- Increasingly, advanced computer systems employ techniques (such as theorem provers and Prolog) based on *mathematical logic*. As logic is an area rich in unsolvable problems, it is important for workers in these application areas to be aware of them.
- The methods for proving unsolvability can be exploited further, in complexity theory (Part III). This is an active area of current research, also very relevant to advanced systems.
- It shows the fundamental limitations of computer science. If we accept Church's thesis, these problems will never be solved, whatever advances in hardware or software are made.
- Even if hardware advances etc. cause Church's thesis to be updated in the fullness of time, the unsolvable problems are probably not going to go away. Their unsolvability arises not because the algorithms we have are not powerful enough, but because they are too powerful! We saw in Section 1 how *paradoxes* cause unsolvability. Paradoxes usually arise because of *self-reference*, and algorithms are powerful enough to allow self-reference. (We saw in Section 4 that a Turing machine can be coded as data, and so given as

input to another Turing machine such as U. Compilers take programs as input — they can even compile themselves!) As any amendment to Church's thesis would probably mean that algorithms are even more powerful than was previously thought, the unsolvable problems would likely remain in some form, and even proliferate.

5.1.2. Proof methods

Our first (algorithmically) unsolvable problems are problems about Turing machines themselves (and so — by Church's thesis — about algorithms themselves). Their unsolvability is proved by assuming that some Turing machine solves the problem, and then obtaining a *contradiction* (eg. 0=1, black = white, etc). Such an impossibility shows that our assumption was wrong, since all other steps in the argument are (hopefully) OK. So there's no Turing machine that solves the problem, after all.

We can then use the method of *reduction* to show that further problems are also unsolvable. The old, unsolvable problem is *reduced* to the new one, by showing that any (Turing machine) solution to the new problem would yield a Turing machine solution to the old. As the old problem is known to be unsolvable, this is impossible; so the new problem has no Turing machine solution either.

A sophisticated example of reduction is used in the proof of Gödel's first incompleteness theorem (§5.4).

5.2. THE HALTING PROBLEM

This is the most famous example of an unsolvable problem. The halting problem (or 'HP') is the problem of whether a given Turing machine will halt on a given input. For the same reasons as in §4, we will restrict attention to standard Turing machines (we saw in §4.3 that this is not really a restriction!) In this setting, the halting problem asks, given the input

- *code*(S) for a standard Turing machine S;
- a word w of C (see Section 4.1 for the alphabet C),

whether or not S halts and succeeds when given input w.

Question: why can we not just use U of Section 4 to do this, by getting U to simulate S running on w, and seeing whether it halts or not?

5.2.1. The halting problem formally

Formally, let $h : C^* \to C^*$ be the partial function given by h(x) =

- 1 if x= *code*(S)*w for some standard Turing machine S, and S halts and succeeds on input w

- 0 if x = *code*(S)*w for some standard Turing machine S, and S does not halt and succeed on input w
- *arbitrary* (e.g., *undefined*) if x is not of the form *code*(S)*w for any standard Turing machine S and word $w \in C^*$.

Big question: is this function h Turing-computable? Is there a Turing machine H such that $f_H = h$? Such an H would *solve the halting problem.*

WARNING: Our choice of values 1, 0 for h is not important. Any two different words of C would do. What matters is that, on input *code*(S)*w, H always halts & succeeds, and we can tell from its output whether or not S would halt & succeed on input w.

The halting problem is not a toy problem. Such an H would be very useful. As we now demonstrate, regrettably there is no such H. This fact has serious repercussions.

5.2.2. Theorem: The halting problem is unsolvable.

This means that there is no Turing machine H such that $f_H = h$. Informally, it means that there's no Turing machine that will decide, for arbitrary S and w, whether S halts & succeeds on input w or not.

Proof Assume for contradiction that the partial function h (as above) is Turing computable. Clearly, if h is computable it is trivial to compute the partial function $g : C^* \to C^*$ given by:

$$g(w) = \begin{cases} 1 \text{ if } h(w{*}w) = 0 \\ \text{undefined otherwise} \end{cases}$$

(Here, 'w*w' is just w followed by a '*', followed by w.) So let M be a Turing machine with $f_M = g$. By §4.4.2.1 (scratch character elimination) we can assume that M is standard, so it has a code.

There are two cases, according to whether g(*code*(M)) is defined or not.

Case 1: g(*code*(M)) is defined.

Then	g(*code*(M)) = 1	[by def. of g],
so	h(*code*(M)***code*(M)) = 0	[also by def. of g],
so	M does not halt & succeed on *code*(M)	[by def. of h],
so	f_M(*code*(M)) is undefined	[by def. of Turing machines],
so	g(*code*(M)) is undefined	[because $f_M = g$].

This contradicts the case assumption (which was 'g(*code*(M)) is defined'). So we can't be in case 1.

Case 2: g(*code*(M)) is not defined.

Then	f_M(*code*(M)) is undefined	[because $f_M = g$],
so	M does not halt & succeed on *code*(M)	[by def. of TMs],
so	h(*code*(M)***code*(M)) = 0	[by def. of h],

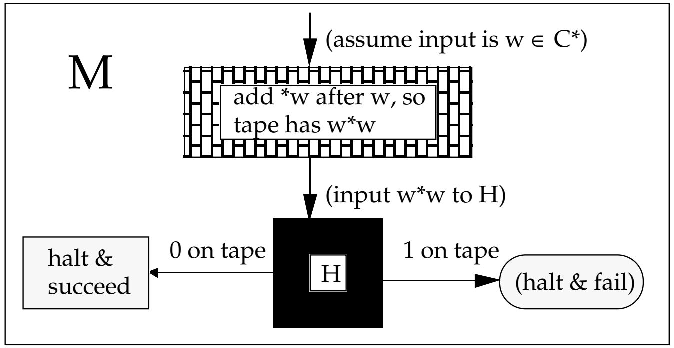
so $g(code(M)) = 1$ [by def. of g],

so $g(code(M))$ is defined! This contradicts the case assumption, too, so we cannot be in case 2 either.

But clearly either $g(code(M))$ is defined, or it isn't. So we must be in one of the two cases. This is a contradiction. So h is not Turing computable. QED.

Another way of seeing the proof:

Suppose for contradiction that H is a Turing machine that solves HP. We don't know how H operates. We only know that $f_H = h$. Consider the following simple modification M of H:



5.1 An impossible TM

If the input to M is w, then M adds a * after w, then adds a copy of w after it, leaving 'w*w' on the tape. It then returns to square 0, calls H as a subroutine, and halts & succeeds/fails according to the output of H, as in the figure. Note that these extra operations (copying etc.) are easy to do with a TM. So if H exists, so does M.

Clearly M outputs only 0, if anything. So $f_M : C^* \to C^*$, and by §4.4.2.1 (elimination of scratch characters) we can assume that M is standard. So M has a code, viz. *code*(M).

Consider the run of M when its input is *code*(M). M will send input '*code*(M)**code*(M)' to H. Now as we assumed H solves HP, the output of H on input *code*(M)**code*(M) says whether M halts and succeeds when given input *code*(M).

But we are now in the middle of this very run — of M on input *code*(M)! H is saying whether the current run will halt & succeed or not! The run hasn't finished yet, but H is supposed to predict how it will end — in success or failure. This is clearly a difficult task for H! In fact, M is designed to find out what H predicts, and then do the exact opposite! Let us continue and see what happens.

The input to H was *code*(M)**code*(M), and *code*(M) is the code for a standard Turing machine (M itself). So H will definitely halt and succeed. Suppose H outputs 1 (saying that M halts and succeeds on input *code*(M)). M now moves to a state with no applicable instruction (look at Figure 5.1). M has now halted and failed on input *code*(M), so H was wrong.

Alternatively, H decides that M halts and fails on input *code*(M). So H outputs 0. In this case, M gleefully halts and succeeds: again, H was wrong.

But H was assumed to be correct for all inputs. This is a contradiction. So H does not exist. QED.

5.2.3. The halting problem is hard

Warning: do not think that HP is an easy problem. It is not. I've heard the following argument:

1. We proved that there's no Turing machine that solves the halting problem.
2. So by Church's thesis, the halting problem is unsolvable by an algorithm.
3. But our brains are algorithmic — just complicated computers running a complex algorithm.
4. We can solve the halting problem, as we can tell whether a program will halt or not. So there is an algorithm to solve the halting problem — us!

2 and 4 are in conflict. So what's going on?

Firstly, many people would not agree with (3). See Roger Penrose's book, listed on page 2. But in any case, I don't believe (4). Consider the following pseudo-code program:

```
n,p: integer.   stp: Boolean          /* 'n is the sum of two primes'
begin
    put 4 into n; put true into stp
    repeat while stp
        put false into stp
        repeat with p = 2 to n-2
                if prime(p) and prime(n-p) then put true into stp
        end repeat
        add 2 to n
    end repeat
end
function prime(p)   /* assume p≥2
    i,p: integer
    repeat with i = 2 to p-1
        if i divides p without remainder then return false
```

```
        end repeat
        return true
    end prime
```

The function *prime* returns true if the argument is a prime number, and false otherwise. The main program halts if some even number > 2 is not the sum of two primes. Otherwise it runs forever. As far as I know, no-one knows whether it halts or not. See Goldbach's conjecture (§5.4). (And of course we could design a Turing machine doing the same job, and no-one would know whether it halts or not.) Exercise: write a program that halts iff Fermat's last theorem is false. (This theorem has only just been proved, after 300 years of effort. So telling if your program halts can be quite hard!)

5.2.4. Exercises

1. What happens if we rewire the Turing machine M of Fig. 5.1, swapping the 0 and 1, so that M halts and succeeds if H outputs 1, and halts and fails if H's output is 0? What if we omit the duplicator that adds '*w' after w? [Try the resulting machines on some sample inputs.]

2. Show that there is no Turing machine X such that for all standard Turing machines S and words w of C, $f_X(\textit{code}(S)*w) = 1$ if S halts (<u>successfully or not</u>) on input w, and 0 otherwise.

3. Let the function $f : C^* \to C^*$ be given by: $f(w) = a.f_M(w)$ if $w = \textit{code}(M)$ for some standard Turing machine M and $f_M(w)$ is defined, and a otherwise (here, $a \in C$ is just the letter a!). Prove that f is not Turing computable.

4. (similar to part of exam question, 1991) Let X be a Turing machine such that $f_X(w) = w*w$ for all $w \in C^*$. Let Y be a *hypothetical* Turing machine such that for every standard Turing machine S and word w of C,

$$f_Y(\text{code}(S)*w) = \begin{cases} 1 \text{ if } f_S(w) = 0 \\ 0 \text{ otherwise} \end{cases}$$

(i) How might we build a standard Turing machine M such that for all $w \in C^*$,

$$f_M(w) = f_Y(f_X(w))?$$

(ii) By evaluating $f_M(\textit{code}(M))$, or otherwise, deduce that Y does not exist.

5. Prove that HP is unsolvable by using the 'Modula_2-style' diagonal paradox of §1. [Use the universal machine of §4.]

6. A *super Turing machine* is an ordinary TM except that I and Σ are allowed to be infinite. Find a super TM that solves HP for ordinary TMs. [Hint: take the alphabet to be C^*.] Deduce that super TMs can 'compute' non-algorithmic functions. What if instead we let Q be infinite?

5.3. <u>Reduction</u>

So the halting problem, HP, is not solvable by a Turing machine. There is no machine H as above. (By Church's thesis, HP has no algorithmic solution.) We can use this fact to show that a range of other problems have no solution by Turing machines.

The method is to *reduce* HP to a special case of the new problem. The idea is very simple. We just show that <u>in order to solve HP (by a Turing machine), it is enough to solve the new problem</u>. We could say that the task of solving HP *reduces* to the task of solving the new problem, or that HP 'is' a special case of the new problem. So if the new problem had a TM solution, so would HP, contradicting §5.2.2. This means that it doesn't have a TM solution.

5.3.1. Reduction and unsolvability

In general, we say that a problem A *reduces* to another problem, B, if we can convert any Turing machine solution to B into a Turing machine solution to A.[14]

Thus, if we knew somehow that A had no Turing machine solution (as we do for A = HP), we could deduce that B had no Turing machine solution either.

5.3.1.1 Example

For example, *multiplication* reduces to *addition*, because we could easily modify an addition algorithm to do multiplication.[15] So if we knew that multiplication could not be done by an algorithm, we couldn't hope to find an algorithm that does addition. Another example: we reduced Goldbach's conjecture to HP in §5.2.3 above.

5.3.1.2. Warning

A reduces to B = you can use B to solve A.
Please get it the right way round!

[14] This is a bit vague, but it will do for now. (The main problem is what 'convert' means.) We will treat reduction more formally in §11, but I can tell you in advance that we will be saying something like this: A reduces to B if there exists a Turing machine M that converts inputs to A into inputs to B in such a way that if M_B is a Turing machine solving B, then the Turing machine given by 'first run M, then run M_B' solves A. This is what happens in most of the examples below.

[15] In this sense, addition is <u>harder</u> than multiplication, because if you can add you can easily multiply, but not vice versa.

5.3.1.3 Warning

It is vital to realise that we can reduce A to B whether or not A and B are solvable. The point is that if we were given a solution to B, we could use it to solve A, so that A is 'no harder' than B. (These free, magic solutions to problems like B are called *oracles*.) Not all unsolvable problems reduce to each other.[16] Some unsolvable problems are more unsolvable than others! We'll see more of this idea in part III.

5.3.2. The Turing machine M[w]

Reduction is useful in showing that problems are unsolvable. We will see some examples in a moment. The following Turing machine will be useful in doing them.

Suppose M is any Turing machine, and w a word of its input alphabet, I. We write M[w] for the new Turing machine[17] that does the following:

1. First, it overwrites its input with w;
2. then it returns to square 0;
3. then it runs M.

It's easy to see that there always is a TM doing this, whatever M and w are. Here's an example.

5.3.2.1. Example of M[w]

Suppose TAIL is a Turing machine such that $f_{TAIL}(w) = tail(w)$ (for all $w \in C^*$). So TAIL deletes the first character of w, and shifts the rest one square to the left — as in §2.

The machine TAIL[hello_world] first writes 'hello_world' on the tape, overwriting whatever was there already. Then it returns to square 0 and calls TAIL as a subroutine.

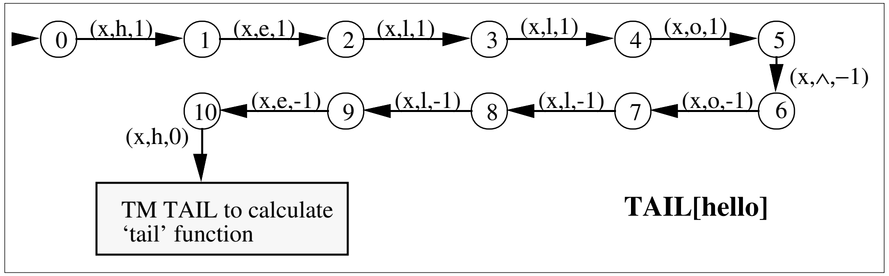
This means that its output will be 'ello_world' on any input.

The input is immediately overwritten, so it doesn't matter what it is.

Fig. 5.2 shows the Turing machine TAIL[hello] as a flowchart.

[16] Eg: the problem 'does the standard TM S return infinitely often to its initial state when run on w?' is unsolvable but does not reduce to HP.

[17] A less succinct notation for M[w] would be 'w, then M'.



5.2 The TM TAIL[hello]

5.3.2.2 Important properties of M[w]

Because the input to M[w] doesn't matter, it is clear that for any Turing machine M and any word w of I, we have:

1. $f_{M[w]}(v) = f_M(w)$ for any word v of I.
2. M halts and succeeds on input w, iff M[w] halts and succeeds on input v for any or all words v of I.

5.3.2.3. Making M[w] from M and w

Given M and w, it's very easy to make M[w]. The part that writes w is always like the 'hello_world' machine — it has w hard-wired in, with a state for each character of w. Note that it adds a blank after w, to kill long inputs (see states 5–6 in Fig. 5.2). Returning to square 0 is easy; and then M is called as a subroutine. In Fig. 5.2, M = TAIL, w=hello.

5.3.2.4. If M is standard, then M[w] can be made standard, too

Moreover, if M is standard, then M[w] can be made[18] standard, too. (See §4.1.)

* The part of M[w] that writes w is certainly standard.
* If we do the return to square 0 by implicitly marking square 0, we will NOT get a standard Turing machine So we don't. Instead, we do a hard-wired return to square 0. Notice how TAIL[hello] in fig. 5.2 returns to square 0: by writing 'olleh' backwards! In general, M[w] writes w forwards, and then returns to square 0 by writing w again, backwards, but with the head moving left.
* As M is known to be standard, there's no problem with that part of M[w].

[18] To show this, we could just use scratch character elimination. But it would make the machine EDIT (below) more complicated. So we try to make M[w] standard in the first place.

5.3.2.5. *The Turing machine EDIT*

It's not only easy for us to get M[w] from M and w; we can even design a Turing machine to do it! There is a Turing machine that takes as input *code*(S)*w, for any standard Turing machine S and word w of C, and outputs *code*(S[w]). We call this machine EDIT — it edits codes of TMs.

How does EDIT work? It adds instructions (i.e., 5-tuples) on to the end of *code*(S). The new instructions are for writing w and returning to square 0. They will involve some number N of new states: a new inital state, and a state for writing each character of w, both ways. So N = 1+2.length(w). (For example, in Fig. 5.2 we added new states 0, 1, …, 10, so N = 11, i.e., one more than twice the number of characters in 'hello'.) The states of the M-part of M[w] were numbered 0,1,2,… in code(S). Now they will be numbered N, N+1, N+2,… . So EDIT must also increase all state numbers in the old *code*(S) by N.

This sounds complicated, but it is really very simple. Rough pseudo-code for EDIT is as follows. Remember, EDIT's input is *code*(S)*w, and its output should be *code*(S[w]).

```
/*    The states of S[w] are those of S plus 1+2.length(w) new ones, numbered
/*    0,1,…, 2.length(w). So first, renumber the states mentioned in code(S)
/*    (currently 0,1,…) as 1+2.length(w), 2+2.length(w), ….
Add 1+2.length(w) to all state numbers in code(S), including n and f at the front

/*  S[w] overwrites its input with 'w∧'. So add instructions to do this to the end of the
/*  code. In fig 5.2 we'd get (0,a,1,h,1),(1,a,2,e,1),(2,a,3,l,1), …, for all a in C∪        {blank}.
s := 0                          /* s will be current state number (s = 0, 1, …, 2.length(w)).
repeat with q = 1 to length(w)
    for each a ∈ C∪{blank},  add an instruction 5-tuple '(s,a,s+1,<q'th char of w>,1)'
    s:= s+1
end repeat
for each a ∈ C∪{blank},  add an instruction 5-tuple '(s,a,s+1,blank,-1)'
s:= s+1
/* S[w] returns to square 0 and hands over to S. Add instructions for this, in the way we said.
repeat with q = length(w) down to 2
    for each a ∈ C∪{blank},  add an instruction 5-tuple '(s,a,s+1,<q'th char of w>,-1)'
    s:= s+1
end repeat
for each a ∈ C∪{blank},  add an instruction 5-tuple '(s,a,s+1,<1st char of w>,0)'
halt & succeed
```

Of course, this does not show all the details of how to transform the input word *code*(S)*w into the output word *code*(S[w]).

5.3.2.6. *Exercises*

1. Write out the code of the *head*-calculating machine M shown in Fig 2.3 (§2.3.3), assuming to keep it short that the alphabet is only {a,b,∧}. Then write out the code of M[ab]. Do a flowchart for it. Does it have the number of states that I claimed above? What is its output on inputs (i) bab, (ii) ∧? (Don't just guess; run your machine and see what it outputs!)
2. Would you implement the variable q in the pseudocode for EDIT using a parameter in one of EDIT's states, or by an extra tape? Why?
3. Write proper pseudocode (or even a flowchart!) for EDIT.

5.3.3. The empty-input halting problem, EIHP

This is the problem of whether or not a Turing machine halts and succeeds on the empty input, ε. As before, we only consider standard Turing machines.

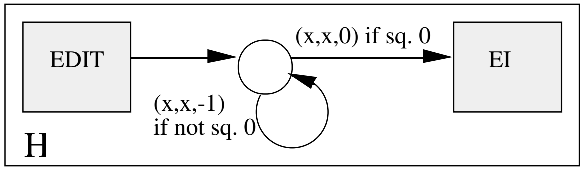
You might think EIHP looks easier than HP, as it only asks about the halting of S on a single fixed input. Well done — you noticed that EIHP reduces to HP!

To say that EIHP is *solvable* is to say that there is a Turing machine EI such that for any standard Turing machine S,

$$f_{EI}(code(S)) = \begin{cases} 1 & \text{if S halts and succeeds on input } \varepsilon \\ 0 & \text{otherwise} \end{cases}$$

5.3.3.1. *EIHP is unsolvable — there is no such Turing machine EI*

We will prove this by showing that HP reduces to EIHP. The proof that HP reduces to EIHP goes like this. Assume we're given a Turing machine EI that solves EIHP. We convert it into a Turing machine H, as follows:



5.3 EIHP solution gives HP solution

H first runs EDIT (see §5.3.2.4), then returns to square 0, then calls EI as a subroutine. We showed how to make EDIT, we're given EI, and the rest is easy. So we can really make H as above.

We claim that H solves HP. Let's feed into H an input *code*(S)*w of HP. H runs EDIT , which converts *code*(S)*w into *code*(S[w]). This word *code*(S[w]) is then fed into EI, which (we are told) outputs 1 if S[w] halts and succeeds on input ε, and 0 otherwise.

But (cf. §5.3.2.1, with v=ε) S[w] halts and succeeds on input ε iff S halts and succeeds on w.

So the output of H is 1 if S halts and succeeds on w, and 0 otherwise. Thus, H solves HP, as claimed, and we have reduced HP to EIHP.

So by the argument at the beginning of §5.3, EIHP has no Turing machine solution. EI does not exist. QED.

5.3.4. Exercises (No. 4 is quite challenging)

1. Reduce EIHP to HP (easy).
2. The *uniform halting problem, UHP,* is the problem of whether a (standard) Turing machine halts & succeeds on every possible input. Show by reduction of HP that UHP is unsolvable. [Hint: use the machine of Fig. 5.3.]
3. The *sometimes-halts problem*, SHP, is the problem of whether a (standard) Turing machine halts & succeeds on at least one input. Show by reduction of HP that SHP is unsolvable.
4. Show that the problem of deciding whether or not two arbitrary standard Turing machines S_1, S_2 are equivalent (§3.1.1) is not solvable by a Turing machine.

5.3.5. Real-life example of reduction

As we can simulate Turing machines on ordinary computers (if enough memory is available), and vice versa, it follows that the halting problem for Turing machines reduces to the HP for Modula_2. For if we had a Modula_2 program to tell whether an arbitrary Modula_2 program halts, we can apply it to the TM simulator, so Modula_2 could solve HP for TMs. But (cf. Church's thesis) the Modula_2 halting program could be implemented on a TM, so we'd have a TM that could solve the TM HP, contradicting §5.2.2.

So there is no Modula_2 program that will take as input an arbitrary Modula_2 program P and arbitrary input x, and tell whether P will halt on input x. For a particular P and x you may be able to tell, but there is no general strategy (algorithm) that will work. The paradox of §1 can also be used to show this. Thus it is better to write well-structured programs that can easily be seen to halt as required!

5.3.6. Sample exam questions on HP etc.

1. (a) Explain what the *halting problem* is. What does it mean to say that the halting problem is *unsolvable*?

(b) Explain what the technique of *reduction* is, and how it can be used to show that a problem is unsolvable.

(c) Let C be the standard typewriter alphabet. The symbol $*$ is used as a delimiter. Let the partial function $f : C^* \to C$ be given by

$$f(\text{code}(S)*w) = \begin{cases} 1 \text{ if S halts and succeeds on input w} \\ \quad \text{and its output contains the symbol 0} \\ 0 \text{ otherwise} \end{cases}$$

for any standard Turing machine S and word w of C.

Prove, either directly or by reduction of the halting problem, that there is no Turing machine M such that $f_M = f$.

2. (a) Explain what the *empty-input halting problem* is. What does it mean to say that the empty-input halting problem is *unsolvable*?

(b) The *empty-output problem* asks, given a standard Turing machine M and input word w to M, whether the output $f_M(w)$ of M on w is defined and is empty (ε).

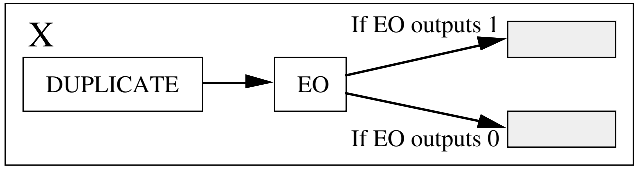
Let EO be a *hypothetical* Turing machine solving the empty-output problem:

$$f_{EO}(\text{code}(M)*w) = \begin{cases} 1 \text{ if } f_M(w) = \varepsilon \\ 0 \text{ otherwise} \end{cases}$$

for any standard Turing machine M and word w of the typewriter alphabet C.

Let DUPLICATE be a Turing machine such that $f_{DUPLICATE}(w) = w*w$ for any word w of C.

By considering the Turing machine X partially described by:



prove that EO cannot exist. (You must decide how to fill in the shaded parts.)

(c) Can there exist a Turing machine EO' such that

$$f_{EO'}(\text{code}(M)*w) = \begin{cases} \varepsilon \text{ if } f_M(w) = \varepsilon \\ 1 \text{ otherwise} \end{cases}$$

for all standard Turing machines M and words w of C? Justify your answer.

The three parts carry, respectively, 30%, 50%, 20% of the marks.

3. In this question, C denotes the standard typewriter alphabet.

a What does it mean to say that a partial function $g : C^* \to C^*$ is (Turing-)computable?

b Let $g : C^* \to C^*$ be a partial function that "tells us whether the output of a standard Turing machine on a given input is 'hello' or not". That is, for any standard Turing machine S and word w of C,

$$g(\text{code}(S)*w) = \begin{cases} y & \text{if } f_S(w) = \text{hello} \\ n & \text{otherwise.} \end{cases}$$

Show that g is not computable.

c Let U be the universal Turing machine. Let $h : C^* \to C^*$ be a partial function such that for any word x of C,

$$h(x) = \begin{cases} y & \text{if } f_U(x) = \text{hello} \\ n & \text{otherwise.} \end{cases}$$

Given that g as in part b is not computable, deduce that h is not computable either.

The three parts carry, respectively, 25%, 50% and 25% of the marks.

4. In this question:

- C denotes the standard typewriter alphabet.
- If S is a standard Turing machine and w a word of C, S[w] is a standard Turing machine that overwrites its input with w and then runs S.
  So $f_{S[w]}(x) = f_S(w)$ for any word x of C.
- EDIT is a standard Turing machine such that for any standard Turing machine S and word w of C, $f_{EDIT}(\text{code}(S)*w) = \text{code}(S[w])$.
- REV is a standard Turing machine that reverses its input (so, for example, $f_{REV}(abc) = cba$).
- U is a (standard) universal Turing machine.

a Define what it means to say that a partial function $g : C^* \to C^*$ is *(Turing-)computable*.

b Let $g : C^* \to C^*$ be a partial function that "tells us whether or not a standard Turing machine halts and succeeds <u>on input w*w</u>". That is, for any standard Turing machine S and word w of C,

$$g(\text{code}(S)*w) = \begin{cases} y & \text{if S halts \& succeeds on input w*w} \\ n & \text{otherwise.} \end{cases}$$

Show that g is not computable.

c Evaluate:

i) $f_U(\text{code}(REV)*\text{deal})$

ii) $f_U(f_{EDIT}(\text{code}(REV)*\text{stock})*\text{share})$

iii) $f_U(f_{EDIT}(\text{code}(U)*\text{code}(REV)*\text{buy})*\text{sell})$

The three parts carry, respectively, 20%, 40% and 40% of the marks. [1994]

5.4. GÖDEL'S INCOMPLETENESS THEOREM

We sketch a final unsolvability result, first proved by the great Austrian logician Kurt Gödel in 1931. One form of the theorem states that <u>there is no Turing machine that prints out all true statements ('sentences') about arithmetic and no false ones.</u>[19] *Arithmetic* is the study of the whole numbers {0,1,2,…} with addition and multiplication. A 'Gödel machine' is a hypothetical Turing machine that prints out all the true sentences (and no false ones) of the *language of arithmetic*. This is a first order language with:

The function symbols '+' and '.' (plus and times), and the successor function S
The relation symbol **<** (and = of course)
The constant symbol 0
The variables x, x', x'', x''', … (infinitely many)
The connectives $\wedge$ (and), $\vee$ (or), $\to$ (implies), $\neg$ (not), $\leftrightarrow$ (iff)
The quantifiers $\forall$ and $\exists$
The brackets (and)

There are 19 symbols here. Think of them as forming an alphabet, Σ. Note that x' is two symbols, x and '. (We'll cheat and use y,z,… as abbreviations for variables x', x'', …) The successor function S represents addition of 1. So SSS0 has value 3.

This is a powerful language. Eg. *x is prime* is expressible by the formula $\pi(x) =_{def.} (x>S0)\wedge(\forall y\forall z(x=y.z \to y=x\vee z=x))$. *Goldbach's conjecture* is expressible by the sentence

$GC =_{def.} \forall x(x>SS0 \wedge \exists y(x = y.SS0) \to \exists y\exists z(\pi(y)\wedge\pi(z)\wedge x=y+z))$. <u>Exercises</u>: what does GC <u>say</u>? Is $\forall y\forall z(SS0.y.y=z.z \to y=0)$ true?

Whether Goldbach's conjecture is true is still unknown, 200 years or so after Goldbach asked it. So if we had a 'Gödel machine', we could wait and see whether GC or ¬GC was in its output. Thus it would solve Goldbach's conjecture (eventually).

Regrettably:

5.4.1. Theorem: There is no Gödel machine[20]

Proof sketch: We reduce HP to the problem a Gödel machine would be solving, if such a machine existed. In fact we show that for any standard Turing machine M and word w of the alphabet C, the statement 'M halts on input w' can be algorithmically written as a sentence X of arithmetic. We could then solve the halting problem by waiting to see which of X or ¬X occurs in the output of a Gödel machine G. But we proved that HP has no Turing machine solution, so this is a contradiction. Hence there is no such G.

[19] Such a machine would never halt. It would print out successive truths, separated on the tape by * say. We just have to keep looking at the output every so often.

[20] Gödel's 1st incompleteness theorem. Equivalently (for mathematicians), there is no recursive axiomatisation of true arithmetic.

We will only be able to <u>sketch</u> the argument for obtaining X from code(S)*w — in full, the proof is quite long. The idea, though, is very simple:

- A *configuration* of M is the combination (current state, tape contents, head position).
- A run of M can be modelled by a sequence of configurations of M. Given any configuration in a run, we can work out the next configuration in a well-defined way using the instruction table δ of M.
- We can *code* any configuration as a number. The relationship between successive configurations in a run of M then becomes an arithmetical relation between their codes. This relation is expressible in the language of arithmetic.
- We can write a formula coding entire sequences of numbers (configurations), of any length, as a single number. Thus the entire run of M can be represented as a single number, c. We can write a formula R(c) expressing that the list of configurations coded by c forms a successful (halting) run of M on input w.
- We then write a sentence $X = \exists x R(x)$ of arithmetic saying that there exists ($\exists$) a run of M on w that ends in a halting state. So X is true if and only if M halts on w, as required.

The same idea comes up again in §12 (Cook's theorem).

Gödel's theorem can also be proved using our old paradox *the least number not definable by a short English sentence:* see Boolos' paper in the reading list. For assume that there was such a Turing machine: G say. Because of G's mechanical Turing-machine nature, it turns out that properties of G are themselves statements about arithmetic. Crudely, the statement 'this statement is not in the output of G' can be written as a sentence S of arithmetic. This leads to a contradiction, since S is in the output of G iff S is false. Of course, Gödel's proof didn't use Turing machines — they hadn't been invented in 1931.

5.4.1.1. Details of the proof

These three pages of details are for interest only; they're not likely to be examined!! There are eleven easy steps.

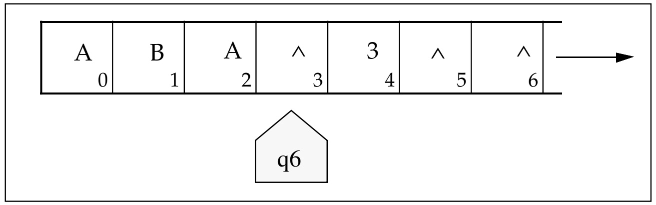
1. As in §4.2, let M have state set Q = {0,1,2,…,q}, where 0 is the initial state and the halting states are f,f+1,…,q.

2. Let's begin by <u>coding a configuration of M as a sequence of numbers</u>. We can code the state by itself, and the head position by the number of the tape square that the head is reading. And as the alphabet of M is C, we can code the symbols on the tape by their ASCII equivalents, using 0 for ∧ (say). (Any ASCII code is a number: e.g., the ASCII code for 'A' is ASCII(A) = 01000001 in binary, i.e., 64+1 = 65. ASCII(B) = 66, etc.)

So the configuration 'in state k, with head over square r, the tape contents up to the last non-∧ being a_0 (square 0), a_1 (square 1), …, a_m (square m)' can be represented by the sequence of numbers:

$$(k,r,\mathrm{ASCII}(a_0),\mathrm{ASCII}(a_1),\ldots,\mathrm{ASCII}(a_m))$$

For example, if squares 5, 6, … are blank, the configuration



5.4 A Turing machine configuration

can be represented by the sequence (6,3,65,66,65,0,51), the ASCII codes for A, B, 3 being 65, 66, 51 respectively.

3. <u>Useful Technical Fact</u>

There is a formula SEQ(x,y,z) of the language of arithmetic with the following useful property. Given any sequence $(a_0,a_1,\ldots,a_n)$ of numbers, there is a number c such that for all numbers z:

- SEQ(c,0,z) is true if and only if $z = a_0$
- SEQ(c,1,z) is true if and only if $z = a_1$

  … … … … … …
- SEQ(c,n,z) is true if and only if $z = a_n$.

We can use such a formula SEQ to *code* the sequence $(a_0,a_1,\ldots,a_n)$ by the single number c. We can recover $(a_0,a_1,\ldots,a_n)$ from c using SEQ.

Finding such a formula SEQ is not easy, but it can be done. For example, we might try to code $(a_0,a_1,\ldots,a_n)$ by the number $c = 2^{a_0+1}.3^{a_1+1}.\ \ldots.p_n{}^{a_n+1}$, where the first n primes are $p_0=2$, $p_1=3$, …, p_n.[21] E.g., the sequence (2,0,3) would be coded by $2^{2+1}.3^{0+1}.5^{3+1}$ = 15,000. Because any whole number factors uniquely into primes, we can recover a_0+1, …,a_n+1, and hence $(a_0,a_1,\ldots,a_n)$ itself, from the number c. So it is enough if we can write a formula SEQ(x,y,z) saying 'the highest power of the y^{th} prime that divides x is z+1'. <u>In fact we can write such a formula</u>, but there are simpler ones available (and usually the simple versions are used in proving that there is a formula SEQ like this!)

4. In fact we want to <u>code a configuration of M as a single number</u>. Using SEQ we can code the configuration $(k,r,\mathrm{ASCII}(a_0), \mathrm{ASCII}(a_1), \ldots, \mathrm{ASCII}(a_m))$ as a single number, c. If we do this, then SEQ(c,0,k), SEQ(c,1,r), SEQ(c,2,ASCII(a_0)), …, SEQ(c,m+2,ASCII(a_m))

---

[21] We use a_1+1 (etc.) because a_1 may be 0; if we just used a_1 then 0 wouldn't show up as a power of 2 dividing c. So we'd have code(2,0,3) = code(2,0,3,0,0,0) — not a good idea.

are true, and in each case the number in the third slot is the only one that makes the formula true.

5. Relationship between successive configurations. Suppose that M is in a configuration coded (as in (4)) by the number c. If M executes an instruction successfully, without halting & failing, it will move to a new configuration, coded by c', say. What is the arithmetical relationship between c and c'?

Let c code (k,r,ASCII(a_0), ASCII(a_1), …, ASCII(a_m)). Assume that r ≤ m.[22] So M is in state k, and its head is reading the symbol a_r. But we know the instruction table δ of M. Assume that δ(k,a_r) is (k',b,d) where k' is the new state, b is the symbol written, and d the move. So if c' codes the next configuration (k',r',ASCII(a'_0), ASCII(a'_1), …, ASCII(a'_m)) of M, we know that r' = r+d, and $a'_i = a_i$ unless i=r, in which case a'_r = b. So in this case, the arithmetical relationship between c and c' is expressible by this formula:[23]

$F(c,c',k,a_r,k',b,d) =_{def.}$

$\forall r\{SEQ(c,0,k)\wedge SEQ(c,1,r)\wedge SEQ(c,r+2,ASCII(a_r))$	[state k, head in sq. r reads a]
$\to [SEQ(c',0,k')\wedge SEQ(c',1,r+d)\wedge SEQ(c',r+2,ASCII(b))$	[new state, head pos & char]
$\wedge\ \forall i(i\geq 2 \wedge i\neq r+3 \to \forall x(SEQ(c,i,x) \leftrightarrow SEQ(c',i,x))]\}$	[rest of tape is unchanged]

Note that we obtain the values k', b and d from δ.

To express — for arbitrary codes c, c' of configurations — that c' codes the next configuration after c, we need one statement F(c,c',k,a,k',b,d) like this for each line (k,a,k',b,d) of δ. Let N(c,c') be the conjunction ('and') of all these F's. N is the formula we want, because N(c,c') is true if and only if, whenever M is in the configuration coded by c then its next configuration (if it has one) will be the one coded by c'.

6. Coding a successful run. A successful (halting) run of M is a certain finite sequence of configurations. We can code each of these configurations as a number c, so obtaining a sequence of codes, $c_0,c_1,\ldots,c_n$. For these to be the codes of a successful (halting) run of M on w, we require:

- c_0 codes the starting configuration of M. So we want SEQ(c_0,0,0) [state is 0 initially] and SEQ(c_0,1,0) [head over square 0 initially] and also SEQ(c_0,2,ASCII(w_0)), SEQ(c_0,3,ASCII(w_1)),…, SEQ(c_0,m+2,ASCII(w_m)), where w is the word $w_0w_1\ldots w_m$. We can write all this as a finite conjunction I(c_0) (I for 'initial').
- c_{i+1} is always the 'next' configuration of M after c_i. We can write this as 'N(c_i,c_{i+1}) holds for each i<n'.

[22] If r>m, M is reading ∧. This is a special case which we omit for simplicity. Think about what we need to do to allow for it.

[23] We've cheated and used 1,2,3 in this formula, rather than S0, SS0 and SSS0. We will continue to cheat like this, to simplify things. Also, the formula only works if d≥ 0, as there's no "–" in the language of arithmetic so if d=-1 we can't write r+d. If d = -1 we replace SEQ(c,2,r) in line 1 by SEQ(c,2,r+1) and SEQ(c',2,r+d) in line 2 by SEQ(c',2,r).

- c_n codes a halting configuration of M. Recalling that f, f+1, …, q are the halting states of M, we can write this as a finite disjunction ('or'),
  $H(c_n) = SEQ(c_n,0,f) \vee SEQ(c_n,0,f+1) \vee \ldots \vee SEQ(c_n,0,q)$,
  saying that c_n is a configuration in which M is in a halting state.

7. Coding a successful run as a single number. If we now use the formula SEQ to code the entire n+2-sequence (n,c_0,…,c_n) as a single number, g, say, we can express the constraints in (6) as properties of g:

- $\forall x(SEQ(g,1,x) \to I(x))$
- $\forall n\forall i\forall x\forall y(SEQ(g,0,n) \wedge 1\leq i<n+1 \wedge SEQ(g,i,x) \wedge SEQ(g,i+1,y) \to N(x,y))$
- $\forall n\forall x(SEQ(g,0,n) \wedge SEQ(g,n+1,x) \to H(x))$

(Note that $c_0,\ldots,c_n$ are entries 1, 2, …, n+1 of g.)

8. Let the conjunction (∧) of these three formulas in (7) be R(g). So for any number g, R(g) holds just when g codes a successful run of M on input w. So the statement 'M halts on input w' is equivalent to the truth of ∃xR(x).

9. R can be constructed algorithmically. Notice that what I've just described is an algorithm (implementable by a Turing machine) to construct R(x), given the data: (a) how many states M has, (b) which states are halting states, (c) the instruction table δ of M, and (d) the input word, w. This information is exactly what *code*(M)*w contains!

10. Reducing HP to the Gödel machine. If we had a Gödel machine, G, we could now solve the halting problem by an algorithm as follows.

- Given *code*(M) and w, where M is a standard Turing machine and w a word of C, we construct the sentence ∃xR(x).
- Then we wait and see whether ∃xR(x) or ¬∃xR(x) turns up in the output of G. This tells us which of them is true. (One of them will turn up, because one of them is true, and G prints all and only true statements. So we won't have to wait forever.)
- If ∃xR(x) turns up, then it's true, so by (8) M must halt & succeed on input w. So we print 'halts' in this case — and we'd be right! We print 'doesn't halt' if ¬∃xR(x) turns up; similarly, we'd be right in this case too.

So these algorithmic steps would solve the halting problem by a Turing machine.

11. Conclusion. But we know the halting problem can't be solved by a Turing machine. This is a contradiction. So G does not exist (because this is the only assumption we made). QED.

5.4.2. Other unsolvable problems

- Deciding whether a sentence of first order logic is valid or not. Church showed that any algorithm to do this could be modified to print out (roughly) all true statements of arithmetic and no false ones. We've shown this is impossible.
- Post's correspondence problem. This has the stamp of a real problem about it — it doesn't mention algorithms or Turing machines. Given words

$a_1,\ldots,a_n$, $b_1,\ldots,b_n$ of C, the question is: is there a non-empty sequence i(1), …, i(k) of numbers ≤ n such that the words $a_{i(1)}a_{i(2)}\ldots a_{i(k)}$ and $b_{i(1)}b_{i(2)}\ldots b_{i(k)}$ are the same? There is no algorithm to decide, in the general case, whether there is or not. This can be shown by reducing HP to this problem; see Rayward-Smith for details.

5.4.3. Exercises

1. Show that there is no algorithm to decide whether a sentence A of arithmetic is true or false. [Hint: any Turing machine to do this could be modified to give a Gödel machine.]
2. Show that there is no Turing machine N such that for all sentences A of arithmetic, $f_N(A) = 1$ if A is true, and undefined if A is false.

5.5. SUMMARY OF SECTION

We saw the significance of unsolvable problems for computing. We proved that there is no Turing machine that says whether an arbitrary Turing machine will halt on a given input ('HP is unsolvable'). By showing that a solution to certain other problems would give a solution to HP (the technique of reduction), we concluded that they were also unsolvable. In this way, we proved that EIHP is unsolvable, and that there is no Turing machine that prints out exactly the true sentences of arithmetic. So doing this is another unsolvable problem.

Part I in a nutshell

§1–2: A Turing machine (TM) is a 6-tuple $M = (Q,\Sigma,I,q_0,\delta,F)$ where Q is a finite set (of states), Σ is a finite set of symbols (the full alphabet), $I \neq \varnothing$ is the input alphabet, $\Sigma \supseteq I$, $\wedge \in \Sigma \setminus I$ is the blank symbol, $q_0 \in Q$ is the initial state, $\delta: Q\text{x}\Sigma \to Q\text{x}\Sigma\text{x}\{0,1,-1\}$ is a partial function (the instruction table), and F (a subset of Q) is the set of final states. We view M as having a 1-way infinite tape with squares numbered 0,1,2,… from left to right. In each square is written a symbol from Σ; all but finitely many squares contain $\wedge$. Initially the tape contains a word w of I (a finite sequence of symbols from I), followed by blanks. w is the input to M. M has a read/write head, initially over square 0. At each point, M is in some state in Q, initially q_0. At each stage, if the head is over a square containing $a \in \Sigma$, and M is in state $q \in Q$, then if $q \in F$, M halts and succeeds. Otherwise, let $\delta(q,a) = (q',b,d)$; M writes b to the square, goes to state q', and the head moves left, right, or not at all, according as d = -1, 1, or 0 respectively. If $\delta(q,a)$ is undefined or if the move would take M's head off the tape, M halts and fails. The output of M is the final tape contents, terminated by the first $\wedge$; the output is only defined if M halts & succeeds. We write $f_M : I^* \to \Sigma^*$ for the partial function taking the input word to the output word; here, Σ^* is the set of words of Σ, and similarly for I. A partial function $f : I^* \to \Sigma^*$ is said to be Turing computable if it is of the form f_M for some Turing machine M.

Church's thesis (or better, the Church-Turing thesis) says that all algorithmically computable functions are Turing computable. As *algorithm* is a vague, intuitive concept, this can't be proved. But there is evidence for it, and it is generally accepted. The evidence has 3 forms:

- A wide class of functions are Turing-computable. No known algorithm cannot be implemented by a TM.
- The definition of computability provided by the TM is equivalent to all other definitions so far suggested.
- Intuitively, any algorithm ought to be implementable by a TM.

Various tricks for simplifying TM design have been suggested. They help the user to design TMs, without changing the definition of a TM. We can divide the tape into finitely many tracks without changing the definition of a TM, since for n tracks, if a_i is in track i of a square (for each i≤n) the tuple $(a_1,\ldots,a_n) \in \Sigma^n$ can be viewed as a single symbol. As Σ^n is finite, it can be the alphabet of a legitimate TM. Using many tracks simplifies comparing and marking characters and square 0. Often we mark square 0 implicitly. Track copying operations etc. are easy to do.

Similarly we can structure the states in Q. This amounts to replacing Q by a set of the form QxX for some non-empty X. Typically X will be Σ or Σ^n for some n. This allows M's behaviour to be more easily specified: when in a state $(q,a) \in Q\text{x}\Sigma$, the behaviour depending on q can be specified separately from that depending on a. Since $Q\text{x}\Sigma$ is a finite set, it can be the state set of a legitimate TM.

We often use 'flowcharts' to specify TMs. A pseudo-code representation is also possible but care is needed to ensure that the code represents with precision a real TM.

§3: Variants of TMs have been suggested, making for easier TM design. In particular, the *multi-tape TM* is extremely useful: there are available n tapes, each with its own head, and the instruction table δ now has the form $\delta : Q \text{ x } \Sigma^n \to Q \text{ x } \Sigma^n \text{ x } \{0,\pm1\}^n$, interpreted in the natural way. The input and output are by convention on tape 1. An n-tape machine M_n can be simulated by an ordinary Turing machine M . M divides its single tape into 2n tracks. For each i in the range $1 \le i \le n$, track 2i-1 contains the contents of tape i of M_n, and track 2i contains a marker denoting the position of head i of M_n over its tape. For each step of M_n's run, M updates its tape accordingly, keeping the specified format. As M simulates every step of the computation of M_n, it can duplicate the input-output function of M_n.

In a similar way it an be shown that a 2-way infinite tape machine, a machine with a 2-dimensional tape, etc, can all be simulated by an ordinary TM. Thus they all turn out to be equivalent in computational power to the original TM. This provides evidence for Church's thesis.

§4: A 'universal' Turing machine U can be designed, which can simulate any ordinary Turing machine M. Clearly, for this to be possible we must restrict the alphabets Σ and I of M to a standard form; but essentially no loss of generality results, since we can code the symbols and words of any finite alphabet by words of a fixed standard alphabet. Given a word w and a description of a Turing machine S with standard alphabet, U will output $f_S(w)$. The description of S is also a word 'code(S)' in the standard alphabet. U works by using the description 'code(S)' to simulate the action of S on w.

§5: One advantage of formalising 'computable' is that theorems about 'computable' can be proved. We could not hope to show that some problems (functions) were <u>not</u> algorithmically solvable (computable) without a formal definition. In fact many problems are not solvable by a TM. The halting problem (HP), *will M halt on input w,* is not. This is shown by contradiction: assuming that the Turing machine H solves HP, we construct a Turing machine M that on a certain input (viz. code(M)) halts iff H says M will not halt on this input. This is impossible, so H does not exist.

Once a problem is known to be unsolvable, other problems can also be shown unsolvable, by reducing them to the first one. Or one can proceed from first principles in each case. One such is the famous Gödel theorem, that there is no algorithm to print all true statements of arithmetic. We proved this by reducing HP to the problem.