3. Variants of Turing machines

In this section we examine some variants of the TM we considered before. The main examples we have in mind are machines with a two-way infinite tape, or more than one tape. We will see that in computational power they are all the same as the ordinary model. This is in line with Church's thesis, and provides some evidence for the 'truth' of the thesis.

Nonetheless, variants of the basic Turing machine are still useful. Just as in real life, the more complex (expensive) versions can be easier to program (user-friendly), whilst the simpler, cheaper models can be easier to understand and so prove things about. For example, suppose we wanted to prove that 'register machines' are equivalent in power to Turing machines. This amounts to showing that Turing machines are no better and no worse than register machines (with respect to computational power). We could show this directly. But clearly it might be easier to prove that a cheap Turing machine is no better than a register machine, which in turn is no better than an expensive Turing machine. As in fact both kinds of Turing machine have equal computational power, this is good enough.

First we need to make precise what we mean by *equal computational power.*

3.1. COMPUTATIONAL POWER

Comparing two different kinds of machine can be like comparing a car and a cooker. How can we begin? But a function $f : I^* \to \Sigma^*$ is a more abstract notion than a machine. We will compare different kinds of computing machine by comparing their input-output functions, as with the formal version of Church's thesis (§2.2.5). To show that cheap and expensive Turing machines are equivalent, we will show that any function computable by one kind is computable by the other.

Formally:

3.1.1. Definition

Let M_1, M_2 be Turing machines, possibly of different kinds, with the same input alphabet. We say that $M_1$and M_2 are *equivalent* if $f_{M_1} = f_{M_2}$. That is, M_1 and M_2 compute the same function. They have the same input-output function.

So to show that two kinds of Turing machine have equal computational power, we will show that for any machine of one kind there is an equivalent one of the other kind.

3.1.2. Proving different machines have equal computational power

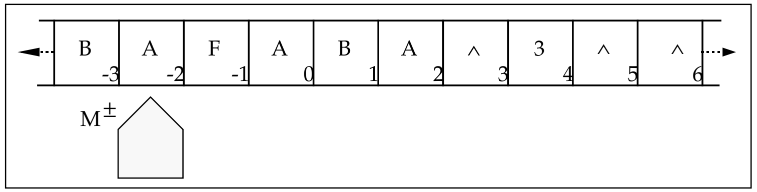
As one might expect, it is usually easy to show that an expensive machine M^+ can compute any function that a cheap machine M can. We must work harder to prove that any algorithm performable by an expensive M^+ can done by a cheaper machine M. We will see several examples below.

The details of the proofs are not so important. The important point is that in each case, although M^+ is (presumably) solving some problem, we do not try to make the cheap machine M solve the problem directly. Instead we cheat and make M mimic or *simulate* M^+, parrot fashion. The same happens when a Macintosh emulates a Sun workstation. We need no deep understanding of what kind of algorithms M^+ can perform, but only of the nuts-and-bolts design of M^+ itself. This is a very profound idea, and we will see it again later (UTMs in §4, and *reduction*, in §5 and Part III of the course).

Aside: it helps if M^+ is not too much more complex than M. So getting a 1-tape Turing machine M to imitate a Cray YMP would be best done by going through several increasingly complex machine designs $M_1, \ldots, M_n$. We would show that an ordinary Turing machine M is equivalent toM_1 (each can simulate the other), M_1 is equivalent to $M_2, \ldots, M_{n-1}$ is equivalent to M_n, and that M_n is equivalent to the Cray. This will show that a Turing machine is equivalent to a Cray.

3.2. TWO-WAY TAPE TURING MACHINES

We could easily (with a little more cash) allow the tape of a Turing machine to be infinite in both directions. In fact this is a common definition of 'Turing machine', and is used in Rayward-Smith's book (our definition, using a one-way infinite tape, is used in Hopcroft & Ullman's). Here's a picture of a 2-way infinite tape Turing machine:



3.1 a two-way infinite tape TM, $M^{\pm}$

3.2.1. Definition

A *two-way infinite tape Turing machine* has the form $M^{\pm} = (Q,\Sigma,I,q_0,\delta,F)$ exactly as before. The tape now goes right and left forever, so the head of $M^{\pm}$ can move left from square 0 to squares -1, -2, etc., without halting and failing. (You can't tell from the 6-tuple definition what kind of tape the machine has; this information must be added as a rider. Of course, by default the tape is 1-way infinite, as that's our *definition* of a Turing machine.)

The input to $M^{\pm}$ is written initially in squares 0,1,…,n. All squares >n and <0 are blank. If $M^{\pm}$ terminates, the output is taken to be whatever is in squares 0,1,…,m-1, where the first blank square ≥0 is square m. So we can define the input-output function $f_{M^{\pm}}$ for a two-way infinite tape machine as before.

3.2.1.1. Exercise (this is too easy!)

Since we can't tell from the 6-tuple definition what kind of tape the machine has, we can alter an ordinary TM by giving it a two-way infinite tape to run on: the result is a working machine. Find a 1-way infinite tape Turing machine that has a different input-output function if we give it a two-way infinite tape in this way.

Now 2-way infinite tape Turing machines still seem algorithmic in nature, so if Church's thesis is true, they should be able to compute exactly the same functions as ordinary Turing machines. Indeed they can: for every two-way infinite Turing machine there is an equivalent ordinary Turing machine, and vice versa. But we can't just quote Church's thesis for this, as we are still gathering evidence for the thesis! We must prove it. If we can do this, it will provide some type (b) evidence (see §1.5.3) for the correctness of Church's thesis as a definition of *algorithm.*

Two-way machines seem intuitively more powerful than ordinary ones. So it should be easy to prove:

3.2.2. Theorem

If $M = (Q,\Sigma,I,q_0,\delta,F)$ is an ordinary Turing machine then there is a two-way infinite Turing machine $M^{\pm}$ equivalent to M.

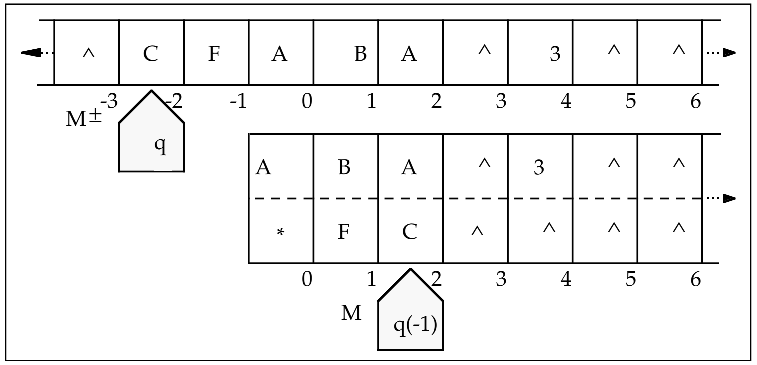
And it is. We take $M^{\pm} = (Q,\Sigma\cup\{\text{fail}\},I,q_0,\delta,F)$, where 'fail' is a new symbol not in Σ. $M^{\pm}$ begins by moving left to square -1, writing 'fail' there, and moving right to square 0 again. Then it behaves exactly as M, except that if it ever reads 'fail' it halts and fails. Clearly $f_M = f_{M^{\pm}}$. QED.

As we might expect, the converse is a little harder.

3.2.3. Theorem

Let $M^{\pm} = (Q,\Sigma,I,q_0,\delta,F)$ be a two-way infinite Turing machine. Then there is an ordinary Turing machine M equivalent to $M^{\pm}$.

Proof The idea is to curl the two-way tape round in a U-shape, making it 1-way infinite but with two tracks. The top track will have the same contents as squares 0,1,2,… of the two-way infinite tape of $M^{\pm}$. The bottom track will have a special symbol '*' in square 0, to mark the end of the tape, and squares 1,2,… will contain the contents of squares -1, -2, … of $M^{\pm}$'s tape. In a figure:



3.2 tape contents of $M^{\pm}$ and its shadow

The 1-way tape of M holds the same information as $M^{\pm}$'s 2-way tape. M will use it to follow $M^{\pm}$, move for move. It keeps track of whether $M^{\pm}$ is currently left or right of square 0, by remembering this (a finite amount of information!) in its state, as in §2.4.1.

In pseudo-code, it is quite easy to specify M. The variable track will hold 1 if $M^{\pm}$ is now reading a positive square or 0, and -1 if $M^{\pm}$ is reading a negative square. For M, +1 means 'top track' and -1 means 'bottom track'. The variable $M^{\pm}$-state holds the state of $M^{\pm}$. Note that these variables can only take finitely many values, so they can be implemented as a parameter in the states of M, as in §2.4.1.

```
track := 1; $M^{\pm}$-state:= $q_0$          /* initially $M^{\pm}$ reads square 0 in state $q_0$
if reading a then write (a,*)         /* initialise square 0
repeat forever
    if the current square contains a, and a is not a pair of symbols, then
        if track=1 then write (a,$\wedge$) else write ($\wedge$,a) end if  /* dynamic track set-up
    end if
    if track = 1  then
```

```
        case [we are reading (a,b) and δ  (M±-state, a) = (q,a',d)]: /* δ  from M±
            write (a',b)                      /* write to top track
            M±-state:= q
            if b = * and d = -1 then          /* M± in square 0 and moving left…
                move right; track := -1
            else
                move in direction d    /* track = 1 ∴  M moves the same way as M±
            end if
        end case
    else if track = -1 then
        case [we are reading (a,b) and δ  (M±-state, b) = (q,b',d)]:
    write (a,b')                      /* write to bottom track
            M±-state:= q
            move in direction -d              /* track = -1, so M moves 'wrong' way
            if now reading * in track 2 then track := 1  /* M± now in square 0
        end case
    end if
    if M±-state is a halting state of M± then    /* So M± has halted and succeeded
        move left until read * in track 2     /* return to square 0
        repeat while not reading ∧ in track 1
            if reading (a,b) then write a     /* replace two tracks with one
            move right
        end repeat
        write ∧; halt & succeed     /* blank to mark end of output
    end if
end repeat
```

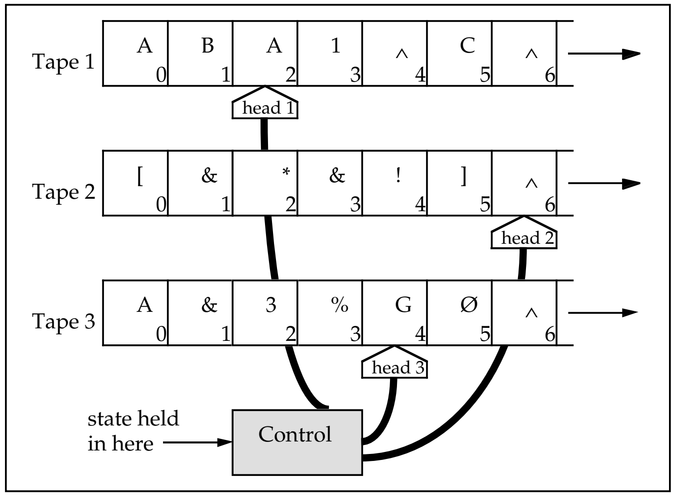
So M mimics M±, move for move. Note that the *case* statements involve a fixed finite number of options, one case for each triple (q,a,b) where $q \in Q$ and $a, b \in \Sigma$. So we can implement them by 'hard-wiring', using finitely many states of M.

When M± halts and succeeds (if ever!), M removes the bottom track, the old top track up to its first ∧ (which is M±'s output) becoming the whole width of the tape. Thus the output of M is the same as M± in all cases, and so M is equivalent to M±.

Try this on a simple example, following M's attempts to keep up with M±. What happens if M± halts and fails?

3.3. MULTI-TAPE MACHINES

With a lot more cash we could allow our machines to have more than one tape. Here is a picture of a 3-tape Turing machine.



3.3 a 3-tape TM caught in action

Notice how it differs from a 1-tape machine with 3 tracks. Here, there are three heads which can move independently on their own tapes. At each step:

- All 3 heads read their squares; the 3 symbols found are passed to *control.*
- Depending on these 3 symbols and on its current state, *control* then:
  - tells each head what to write;
  - tells each head which way to move;
  - moves into a new state.
- The process repeats.

The moves, new symbols and state are determined by the *instruction table,* and depend on the old state and all three old symbols. So what one head writes can depend on what the other heads just read. In many ways, a many-tape Turing machine is analogous to concurrent execution, as in effect we have several communicating Turing machines running together.

WARNING

Do not confuse *multi-tape* TMs with *multi-track* TMs. I know they are similar as English words, but these names are in general use and we are stuck with them. They mean quite different things. Think of tape recorders. Two old mono tape recorders (2 tapes, with 1 track each) are not the same as one stereo tape recorder (1 tape, 2 tracks). They are in theory better, because (a) we could synchronise them to get the capabilities of a single stereo machine, but (b) we can use them in other ways too, eg. for editing. Similar considerations apply to Turing machines.

3.3.1. The multi-tape machine formally

A 3-tape machine can be written formally as $M = (Q,\Sigma,I,q_0,\delta,F)$, where all components except solely for the instruction table δ are as for the usual Turing machine. In the 3-tape machine, δ is a partial function

$\delta : Q\text{x}\Sigma\text{x}\Sigma\text{x}\Sigma \rightarrow Q\text{x}\Sigma\text{x}\Sigma\text{x}\Sigma\text{x}\{-1,0,1\}\text{x}\{-1,0,1\}\text{x}\{-1,0,1\}$.

The definition of the n-tape machine is the same, except that we have:

$\delta : Q \text{ x } \Sigma^n \rightarrow Q \text{ x } \Sigma^n \text{ x } \{-1,0,1\}^n$.

3.3.1.1. Remarks

1. For n=1 this is the Turing machine defined in §2.1.

2. How can you tell how many tapes the Turing machine $(Q,\Sigma,I,q_0,\delta,F)$ has? Only by looking at δ. If δ takes 1 state argument and n symbol arguments, there are n tapes.

3. Note that it is NOT correct to write eg. $(Q^3,\Sigma^3,I^3,q_0,\delta,F^3)$ for a 3-tape machine. One Turing machine, one state set, one alphabet. There is a single state set in a 3-tape Turing machine, and so we write it as Q. If each of the three heads were in its own state from Q, then the state of the whole machine would indeed be a triple in Q^3. But remember, everything is linked up, and one head's actions depend on what the other heads read. With so much instantaneous communication between heads, it is meaningless to say that they have individual states. The machine as a whole is in some state — some q in Q.

And there is one alphabet: the set of characters that can occur in the squares on the tapes. We used Σ^3 when there were 3 tracks on a single tape, because this involved changing the symbols we were allowed to write in a single square of a tape. So if you write Σ^3, you should be thinking of a 3-track machine. In fact, after using 3-tape machines for a while you won't want to bother with tracks any more, except to mark square 0.

3.3.1.2. Computations

Consider a 3-tape machine $M = (Q,\Sigma,I,q_0,\delta,F)$. At the beginning each head is over square 0 of its tape. Assume that at some stage, M is in state q and reads the symbol a_i from head number i (for each i = 1,2,3). Suppose that

$$\delta(q, a_1,a_2,a_3) = (q', b_1,b_2,b_3, d_1,d_2,d_3).$$

Then for each i = 1,2,3, head i will write the symbol b_i in its current square and then move in direction d_i (0 or ±1 as usual), and M will go into state q'. M halts and succeeds if q' is a halting state. It halts and fails if there is no applicable instruction, or if any of the three heads tries to move left from square 0. The definition of an n-tape machine is similar.

3.3.1.3. Input/output. The function computed by M

The input, a word w of I, is placed left-justified on tape 1, with only $\wedge$'s afterwards. All other tapes are blank. If M halts and succeeds, the output $f_M(w)$ is taken to be whatever is on tape 1 from square 0 up to the character before the first blank. If M doesn't halt & succeed, its output on w is undefined. So as before, f_M is a partial function from I* into Σ^*: *the function computed by M*.

Notice that the input-output conventions are as for an ordinary Turing machine. So an n-tape machine is just an ordinary Turing machine if n=1.

3.3.2. Old tricks on the new machine

We can still write many-tape machines as flowcharts or in pseudo-code; pseudo-code is very suitable as it can easily refer to individual tapes (as for tracks). Indeed, all the programming techniques we saw in Section 2 can still be used for multi-tape machines. For example, we can store information in states, and can divide each tape into tracks (each tape can have a different number of tracks). So eg. by adding one extra track to each tape and putting * in square 0 of that track, a many-tape machine can tell at any stage whether one of its heads is in square 0.

However, having many tapes is itself one of the most useful ‘tricks’ of all for programming Turing machines. For this reason we will use many-tape machines very often. An example may illustrate how they can help.

3.3.3. Example: detecting palindromes (a yes/no problem)

Let I be an alphabet with at least 2 symbols. A *palindrome* of I is a word w of I such that w is the *reverse* of itself (if $w = a_1a_2\ldots a_n$ then reverse(w) = $a_na_{n-1}\ldots a_1$). Eg. *abracadabra* is (sadly) not a palindrome; *level* is.

We describe a 2-tape Turing machine $M = (Q,\Sigma,I,q_0,\delta,F)$ that halts and succeeds if w is a palindrome, and halts and fails otherwise. (The actual output $f_M(w)$ of M is unimportant; what matters is whether $f_M(w)$ is defined. Problems like this are called *yes/no problems*, or *decision problems*.)

The idea is very simple. Initially the word w is on tape 1. M first copies w to tape 2. Then it moves head 1 back to the beginning of the original copy of w on tape 1, and head 2 to the end of the new copy on tape 2. It now executes the following:

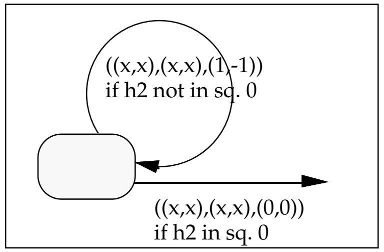
```
if the symbols read by the two heads are different then halt and fail
```

```
repeat  until head 2 reaches square 0
    move head 1 right and move head 2 left
    if the symbols read by the two heads are different then halt and fail
end repeat
```

Note that a 2-track machine could not do this, as it has only one head. Note also that we can't take a shortcut by stopping when the heads meet in the middle: a 2-tape Turing machine doesn't know when its heads cross, unless it has arranged to count moves.

3.3.3.1. Exercise

Draw a flowchart for this machine. The arrow labels will be 6-tuples from $\Sigma^4 x\{0,\pm1\}^2$, usually written ((a,b),(a',b'),(d,d')), which means *take this arrow if head 1 reads a and head 2 b; get head 1 to write a' and head 2 b', and move head 1 in direction d (0, ±1) and head 2 in direction d'* . Here's the main bit:



3.5 main part of palindrome tester

(This halts & fails if the heads read different characters — there's no applicable instruction.) Now try to design a single-tape Turing machine that does the same job. That should convince you that many-tape machines can help programming considerably. (For a solution see Harel's book, p.202.)

3.3.3.2. Exercise

Let I be an alphabet. Design a 2-tape Turing machine M such that $f_M(w)$ = reverse(w) for all w ∈ I*. Can you design an M such that $f_M(w)$ = w*w? (The output is w, followed by a *, followed by a copy of w. Eg: abc*abc)

3.3.4. Many-tape versus 1-tape Turing machines

In the exercise, the 1-tape Turing machine that you came up with probably didn't look much like the original 2-tape machine. If we tried to find 1-tape equivalents for more and more complex Turing machines with more and more tapes, our solutions (if any) would probably look less and less like the original. Can we be sure that any n-tape Turing machine has a 1-tape equivalent — as it should have by Church's thesis?

In the following important theorem we prove that for any n≥1, n-tape Turing machines have exactly the same computational power as 1-tape machines. Thus Church's thesis survives, and in fact the theorem provides further evidence for the thesis.

Just as in the two-way-infinite tape case (§3.2.3), we will design a 1-tape machine that mimics or simulates the n-tape machine itself, rather than trying to solve the same problem directly, perhaps in a very different way. But now, each single step of the n-tape machine will be mimicked by many steps of the 1-tape machine. We are really proving that we can simulate a bounded concurrent system on a sequential machine, albeit slowly.

3.3.5. Theorem

Let n be any whole number, with n≥2. Then:

1. For any ordinary, 1-tape Turing machine M, there is an n-tape Turing machine M_n that is equivalent to M.

2. For any n-tape Turing machine M_n, there is an ordinary, 1-tape Turing machine M that is equivalent to M_n.

Proof: To show (1) is easy (because expensive is 'obviously better' than cheap). Given an ordinary 1-tape Turing machine M, we can make it into an n-tape Turing machine by adding extra tapes and heads but telling it not to use them. In short, it ignores the extra tapes and goes on ignoring them!

Formally, if M = $(Q,\Sigma,I,q_0,\delta,F)$, we define $M_n = (Q,\Sigma,I,q_0,\delta',F)$ by:

$\delta' : Q \times \Sigma^n \to Q \times \Sigma^n \times \{-1,0,1\}^n$

$\delta'(q, a_1,\ldots,a_n) = (q', b_1,\wedge,\wedge,\ldots,\wedge, d_1,0,0,\ldots,0)$

where $\delta(q,a_1) = (q',b_1,d_1)$.

(Recall that δ is the only formal difference between TMs with different numbers of tapes.) Clearly M_n computes the same function as M, so it's equivalent to M.

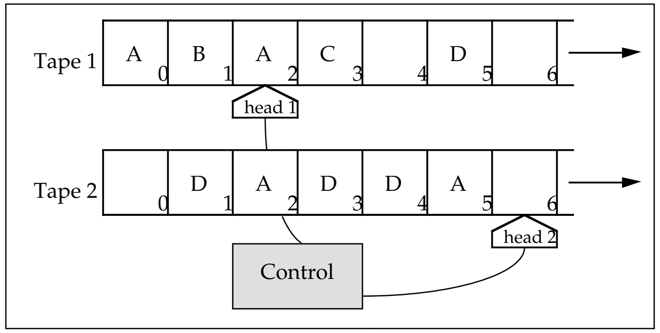
The converse (2), showing that cheap is really just as good as expensive, is of course harder to prove. For simplicity we only do it for n=2, but the idea for larger n is the same.

So let M_2 be a 2-tape Turing machine. We will construct an 1-tape Turing machine M that *simulates* M_2. As we said, each M_2-instruction will correspond to an entire subroutine for M.

M has a single 4-track tape (cf. §2.4.2). At each stage, track 1 will have the same contents as tape 1 of M_2. Track 2 will show the location of head 1 of M_2, by having an X in the current location of head 1 and a blank in the other squares. Tracks 3 and 4 will do the same for tape 2 and head 2 of M_2.

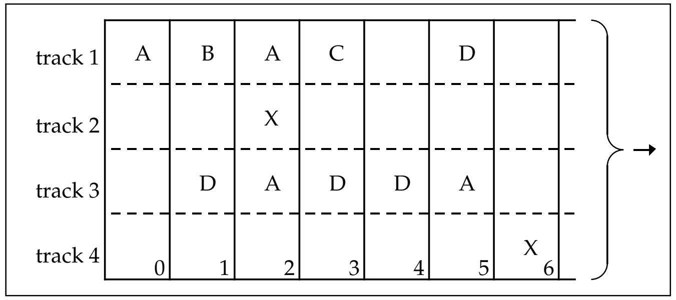
3.3.5.1. Example

Suppose that at some point of execution, the tapes and heads of M_2 look like this:



3.6 M_2, with 2 tapes

Then the tape of M will currently be looking like this:



3.7 a single 4-track tape with the same data[1]

So the tape of M will always give the current layout of M_2's tapes and heads. We have to show how M can update its tape to keep track of M_2. Let us describe M's operation from start to finish, beginning with the setting-up of the tracks.

[1] If we want to do this without mentioning tracks at all (cf. §2.3.5) then formally the alphabet Σ' of M will be: $\Sigma' = \Sigma \cup (\Sigma \times \{\wedge,X\} \times \Sigma \times \{\wedge,X\})$. That is, Σ' has as symbols those of Σ, plus 4-tuples (a_1,a_2,a_3,a_4); a_i will represent the contents of the i^{th} track.

Initialisation

Recall that initially both heads of M_2 are over square 0; tape 1 carries the input of M_2, and tape 2 of M_2 is blank. M is trying to compute the same function as M_2, so we can assume its input is the same. That is, initially M's (single) tape is the same as tape 1 of M_2.

First, M sets up square 0. Suppose that tape 1 of M_2 has the symbol a in square 0. Then M writes $(a,X,\wedge,X)$ in square 0. This is because it knows that both heads of M_2 start in square 0 — so that's where the X's should be! And it knows tape 2 of M_2 is blank.

But these X's will move around later, with the heads of M_2, so also, square 0 should be marked. M can mark square 0 with an extra track — cf. §2.4.3. So really M's tape has five tracks, but we agreed in §2.4.3.1 not to mention this track, for simplicity.

We'll assume *dynamic track set-up,* as in §3.2.3. So whenever M moves into a square whose contents are not of the form (a,b,c,d), but just a, it immediately overwrites the square with $(a,\wedge,\wedge,\wedge)$, and then continues. This is because to begin with, M's tape is the same as tape 1 of M_2, and tape 2 of M_2 is blank. So $(a,\wedge,\wedge,\wedge)$ is the right thing to write. We assume also that M always knows the current state q of M_2 (initially q_0). It can keep this information in its state set as well, because the state set of M_2 is also finite.

M must now update the tape after each move of M_2, repeating the process until M_2 halts. Suppose that M_2 is about to execute an instruction (i.e. to read from and write to the tapes, move the heads, and change state). When M_2 has done this, its head positions and tape contents may be different. M updates its own tape to reflect this, in two stages:

Stage 1: Finding out what M_2 knows

First M's head sweeps from square 0 to the right. As it does so, it will come across the X's in tracks 2 and 4. When it hits the X in track 2, it looks at the symbol in track 1 of the same square. This is the symbol that head 1 of M_2 is currently scanning. Suppose it is a_1, say. M remembers this symbol 'a_1' in its own internal states — cf. §2.3.4. It can do this because there are only finitely many possible symbols that a_1 could be (Σ is finite).

Similarly, M will eventually find the X in track 4, and then it also remembers the symbol — a_2, say — in track 3 of the same square. a_2 is the symbol that head 2 of M_2 is currently scanning.

So when it has found both X's, M knows both the current symbols a_1, a_2 that M_2 is scanning.

Stage 2: Updating the tape

We assume that M knows the <u>instruction table</u> δ of M_2. This never changes so can be 'hard-coded' in the instruction table of M. As with the 2-way-infinite tape simulation (§3.2.3), M does not have to <u>compute</u> δ — δ is built into M in the sense that the instruction table of M is based on it. Eg., a pseudo-code representation of M would involve a long *case* statement, one case for each line of the instruction table of M_2.

M now has enough information to work out what M_2 will do next. For once M knows a_1, a_2 and q, then since it knows δ, it also knows the value

$\delta(q,a_1,a_2) = (q',b_1,b_2,d_1,d_2) \in Q \times \Sigma^2 \times \{-1,0,1\}^2$, if defined.

If it is not defined, M halts and fails (as M_2 does). Assume it is defined. Then, remembering the value (q',b_1,b_2,d_1,d_2) in its states, M's head sweeps back leftwards, updating the tape to reflect what M_2 does. That is:

```
Procedure Sweep_Left [assuming head starts off just to the right of the rightmost X]
put false into DONE1 and into DONE2
repeat until square 0 reached
    if not in square 0 then move left
    if track 2 has X and not DONE1 then
        put true into DONE1
        write $b_1$ in track 1 and ∧ in track 2            /* so erasing the X
        move in direction $d_1$                            /* move of h1
        write X in track 2 (leaving the other tracks alone) /* new position of h1
        move in direction $-d_1$
    end if
    <a similar routine for head 2 of M, using tracks 3 and 4 and variable DONE2>
end repeat
```

M ends up in square 0 with the correct tape reflecting M_2's new pattern. If q' is <u>not</u> a halting state for M_2, M now forgets M_2's old state q, remembers the new state q', and begins the next sweep at Stage 1 above.

The Output

Suppose then that q' is a halting state for M_2. So at this point M_2 will halt and succeed with the output on tape 1. As track 1 of M's tape always looks the same as tape 1 of M_2, this same output word must now be on track 1 of M's tape. M now demolishes the other three tracks in the usual way, leaving a single track tape containing the contents of the old track 1, up to the first blank.

M has simulated every move of M_2. So for all inputs in I*, the output of M is the same as that of M_2. Thus $f_M = f_{M_2}$, and M is equivalent to M_2, as required. QED.

3.3.6. Summary

We showed that any algorithm implementable by a 2-tape machine M_2 is implementable by a 1-tape machine M.

3.3.6.1. Questions

1. Write out the pseudo-code routine to handle tracks 3 and 4 in Sweep_Left.
2. Draw flowcharts of the parts of M that handle stages 1 and 2 above. It's not too complicated if you use parameters to store a_1, a_2, q, q', b_1, b_2, d_1 and d_2, as in §2.3.4.
3. Why do we need the variable DONE1 in Sweep_Left? What might happen if we omitted it? Why do we 'move $-d_1$' after writing X in track 2? (Hint: what if the heads are both in square 6?)
4. What alterations would be needed to simulate an n-tape machine?
5. Suppose that at some point M_2 tries to move one of its heads left from square 0. M_2 halts and fails in this situation. What will M do?
6. Suppose that on some input, M_2 never halts. What will M do?
7. How could we make M more efficient?
8. (Quite long.) Let M_2 be the 'reverser' 2-tape Turing machine of §3.3.3.2. Suppose M_2 is given as input a word of length n. How many steps will M_2 take before it halts? If M is the 1-tape machine that simulates M_2, as above, how many steps (roughly!) will it take?

3.3.7. Exam questions on Turing machines

1. (a) Design a Turing machine M with input alphabet {a,b,c}, which, given as input a word w of this alphabet, outputs the word obtained from w by deleting all occurrences of 'a'. For example,

$$f_M(bcaba) = bcb$$

You may use pseudo-code or a flow-chart diagram; in the latter case you should explain your notation for instructions. You may use several tapes, and you can assume that square 0 of each tape is implicitly marked.

(b) *Briefly* explain how you would design a Turing machine N, with the same input alphabet as M, that moves all occurrences of 'a' in its input word to the front (left), leaving the order of the other characters unchanged. Thus,

$$f_N(bcaba) = aabcb$$

The two parts carry, respectively, 65% and 35% of the marks.

2. (a) Explain the difference between a *2-track* and a *2-tape* Turing machine.

*Below, the notation 1^n denotes a string 111…1 of n 1's. The symbol * is used as a delimiter. You may assume that square 0 of each Turing machine tape is implicitly marked.*

(b) Design a 2-tape Turing machine M with input alphabet {1}, such that if the initial contents of tape 1 are 1^n (for some n≥0) and the initial contents of tape 2 are 1^m (for some m>0), then M halts and succeeds if and only if m divides n without remainder. You may use pseudo-code or a flow-chart diagram; in the latter case you should explain your notation for instructions.

(c) By modifying M or otherwise, *briefly* explain how you would design a (2-tape) Turing machine M* with input alphabet {1,*}, such that for any n≥0 and m>0, $f_{M^*}(1^n{*}1^m) = 1^r$, where r is the remainder when dividing n by m.

[Note: approximately 45% of the marks for this question are awarded for part (b), and 35% for part (c).]

3a What is *Church's thesis?* Explain why it cannot be proved but could possibly be disproved. What kinds of evidence for the thesis are there?

b Design a 2-tape Turing machine M with input alphabet I, such that if w_1 and w_2 are words of I of equal length, the initial contents of tape 1 are w_1 and the initial contents of tape 2 are w_2, then M halts and succeeds if w_1 is an *anagram* (i.e., a rearrangement of the letters) of w_2, and halts and fails otherwise.

For example, if w_1 = abca and w_2 = caba, M halts and succeeds; if w_1 = abca and w_2 = cabb, M halts and fails.

You may use pseudo-code or a flow-chart diagram; in the latter case you should explain your notation for instructions. You may assume that square 0 of each Turing machine tape is implicitly marked.

The two parts carry, respectively, 40% and 60% of the marks. [1993]

3.4. OTHER VARIANTS

We briefly mention some other kinds of Turing machine, and how they are proved equivalent to the original version.

3.4.1. Two-dimensional tapes

We can have a Turing machine with a 2-dimensional 'tape' with squares labelled by pairs of whole numbers (n,m) (for all n,m≥0). Reading, writing and state changing are as before, but at each step the head can move left, right, up, or down. So δ is a partial function : $Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R,U,D,0\}$.

When the run starts the input word is on the 'x-axis' — in squares (0,0), (1,0),…, (k,0) for some k≥0 — and all other squares of the tape contain ∧. The machine must leave the output on the x-axis as well, but it can use the rest of the plane as work space. Thus it is like a 1-tape machine with an unbounded number of tracks on the tape, except that access to other tracks is not instantaneous, as the head must move there first. As the input word is finite and only 1 symbol is written at each step, at all times there will only be finitely many non-blank symbols on the 2-dimensional tape.

3.4.1.1. Simulating a 2-dimensional Turing machine

We will show how a 'big' 2-dimensional machine can be simulated by a 'little' 2-tape machine. Then we will know by §3.3.5 that the big machine can be simulated by a 1-tape machine too.

The contents of the big 2-dimensional tape are kept on little tape 1 in the following format. The data on tape 1 is divided into segments of equal length, separated by a marker, '*'. The segments list in order the non-blank rows of the big tape. For example, suppose the non-blank part of the big tape is as in the figure:

a	b		a		
1	1	1	0	a	
	1		a	1	

3.8 2-dimensional tape

Then tape 1 of the little machine will contain the three segments

∧1∧a1*1110a*ab∧a∧**

— or the same but with longer segments filled out by blanks. Note the double ** at the end. Tape 2 of the little machine is used for scratch work.

Head 1 of the little machine is over the symbol corresponding to where the big machine's head is. If big head moves left or right, so does little head 1. If however, big head moves up, little head 1 must move to the corresponding symbol in the next segment to the right. So the little machine must remember the offset of head 1 within its current segment. This offset is put on tape 2, eg. in unary notation. So in this case, little head 1 moves left until it sees '*'. For each move left, little head 2 writes a 1 to tape 2. When '*' is hit, head 1 moves right to the next '*'. Then for each further move right, head 2 deletes a 1 from tape 2. When all the 1's have gone, head 1 is over the correct square, and the next cycle commences.

Sometimes the little machine must add a segment to tape 1 (if big head moves higher on the big tape than ever before), or lengthen each segment (if big head moves further right than before). It is easy to add an extra segment of blanks on the end of tape 1 of the right length — tape 2 is used to count out the length. Adding a blank at the end of each segment can be done by shifting, as in §2.4.1. The little machine can do all this, return head 1 to the correct position (how?), and then implement the move of the big head as above — there is now room for it to do so.

This bears out Turing's remark in his pioneering paper that whilst people use paper to calculate, the 2-dimensional character of the paper is never strictly necessary. Again we have found evidence of type (b) for Church's thesis. A similar construction can be used to show that for any $n \geq 1$, n-dimensional Turing machines are equivalent to ordinary ones.

3.4.2. Turing machines with limited alphabet

We can imagine Turing machines with alphabet $\Sigma_0 = \{0,1,\wedge\}$ and $I = \{0,1\}$. Unlike the previous variants, these are seemingly less powerful (cheaper) than the basic model. But they can compute any function $f_M : I \to I$ for any Turing machine M. The idea is to simulate a given Turing machine $(Q,\Sigma,I,q_0,\delta,F)$ by <u>coding</u> its scratch characters (those of $\Sigma \setminus I$) as strings of 1's. Eg. we list Σ as $\{s_1,\ldots,s_n\}$ and represent s_i by a string 1^i of i 1's. Exercise: work out the details. We will develop this idea considerably in the next section.

3.4.3. Non-deterministic Turing machines

We will define these and show that they're equivalent to ordinary machines in Part III of the course.

3.4.4. Other machines and formalisms

Ordinary Turing machines have the same computational power as register machines, and also more abstract systems such as the lambda calculus and partial recursive functions. No-one has found a formalism that is intuitively algorithmic in nature but has more computational power. This fact provides further evidence for Church's thesis.

3.5. SUMMARY OF SECTION

We considered what it means for two different kinds of machine to have the same computational power, deciding that it meant that they could compute the same class of functions. We proved or indicated that the ordinary Turing machine has the same computational power as the variants: 2-way infinite tape machines, multi-tape machines, 2-dimensional tape machines, limited character machines, and non-deterministic machines. This provided evidence for Church's thesis.