Turing Machines

Robert M. Keller
Harvey Mudd College
15 October 2003

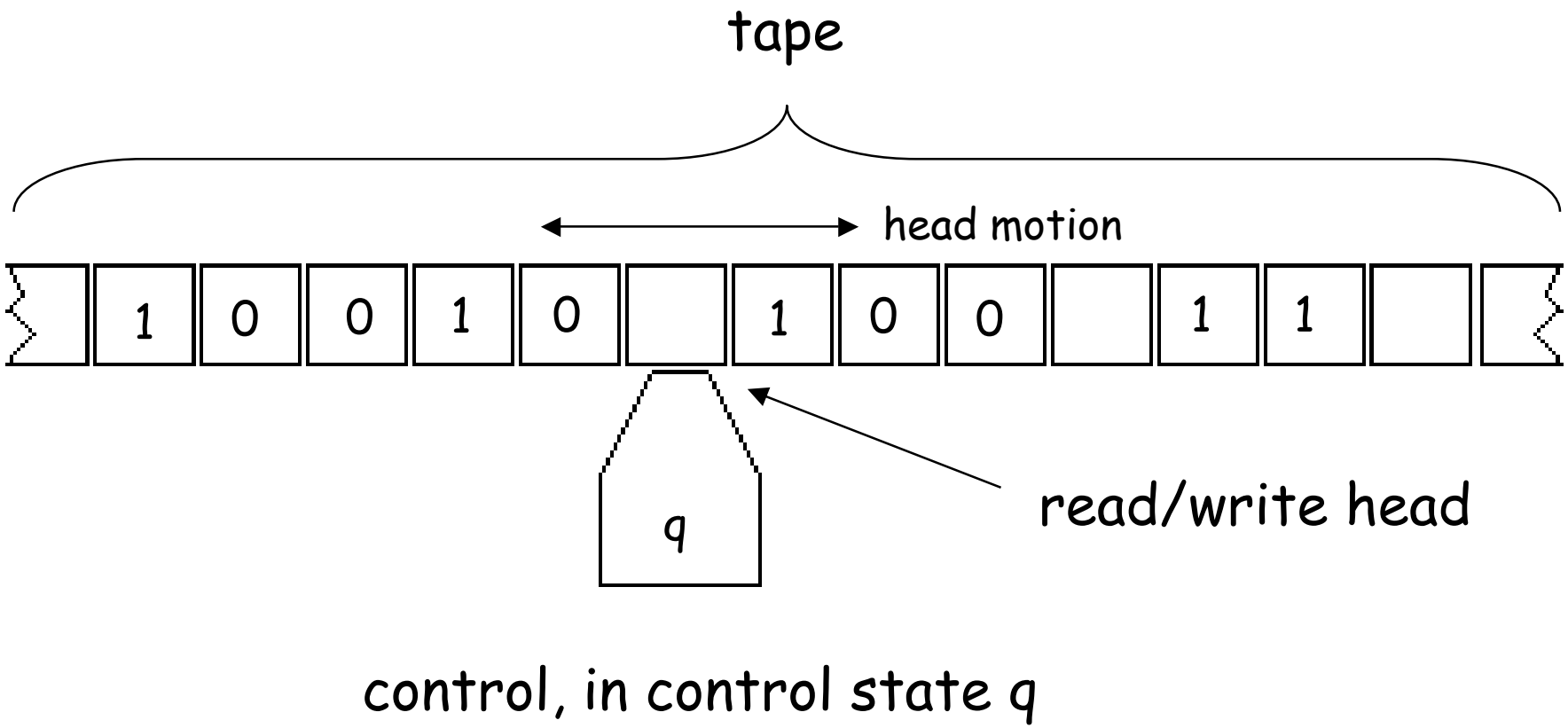
What is a Turing Machine?

- Named after (not by) Alan M. Turing.
- Perhaps the most important computational model, from a theoretical viewpoint.
- Simplistic, yet universal.

Relation to Previous Machines

- Like a FSA but
  - can **write** as well as read its tape
  - can move in **both directions**
- Like a PDA with 2 stacks
- More powerful than a locomotive, able to …

Turing Machine Diagram



Turing Machine (tm) Components (a la John Martin's book)

- $(Q, \Sigma, \Gamma, q_0, \delta)$
- Q: finite set of control states
- Σ: input alphabet
- Γ: tape alphabet ($\Sigma \subseteq \Gamma$)
- q_0: initial control state
- δ: transition function (next page)

Turing Machine (TM) Components (a la John Martin's book)

- Let □ represent "blank", which is not in Σ.
- Let Γ′ = Γ ∪ {□}.
- Let h_a, h_r be states not in Q (halting-accepting and halting-rejecting respectively)
- Let Q′ = Q ∪ {h_a, h_r}
- δ: transition function

  δ: Q x Γ′ → Q′ x Γ′ x {R, L, S}
- R, L, and S represent head motion (right, left, and stationary, respectively)

TM Transition Function

- For each q □ Q and X □ □’,

  □(q, X) = (q’, Y, M)

  where q’ □ Q’, Y □ □’, and M □ {R, L, S}

- means
  - the machine is in state q and reading X
  - the machine goes to state q’, writes Y, and moves
    - Left (L), Right(R), or Not at all (S)

Abbreviations

- No transitions are specified from the two halting states: h_a, h_r
- To reduce the size of a specification, if a transition is unspecified for a given (q, X), assume it goes to h_r, rewrites X, and does not move.

TM Rule Notation

- Instead of

  $\delta(q, X) = (q', Y, M)$

  we may choose to write

  $q, X \rightarrow q', Y, M$

- This is called “5-tuple” notation.

TM Configuration (i.e. complete state)

- We show the control state q ∈ Q’ and the tape ∈ Γ’*.
- All but a finite number of symbols are non-blank (□)
- We underline the symbol over which the tape is positioned, e.g.

$$(q_2, 011\underline{0}1\square110)$$

- Outside this string, all tape symbols are □.
- Hence the tape may be represented by two finite strings, both elements of Γ’*:
  - The symbols to the left of the head.
  - The symbols to the right of, and including the head.

General Moves

- A machine makes one of the following transitions
  - (q, w$\underline{X}$Yz) |- (q', wX'$\underline{Y}$z) if □(q, X) = (q', X', **R**)
  - (q, wX$\underline{Y}$z) |- (q', w$\underline{X}$Y'z) if □(q, Y) = (q', Y', **L**)
  - (q, w$\underline{X}$Yz) |- (q', w$\underline{X}$'Yz) if □(q, X) = (q', X', **S**)

Standard Starting Configuration

- The machine starts in q_0 .
- The head is positioned at the over a blank just to the left of the input, which consists of symbols in Σ (input alphabet) only.

$(q_0, \sqcup x)$ where $x \in \Sigma^*$

Halting Configuration

- The machine halts in one of h_a or h_r, indicating acceptance or rejection of the original input respectively. No transitions are defined from these states.

- It is possible that the machine never halts, in which case we say it "diverges".

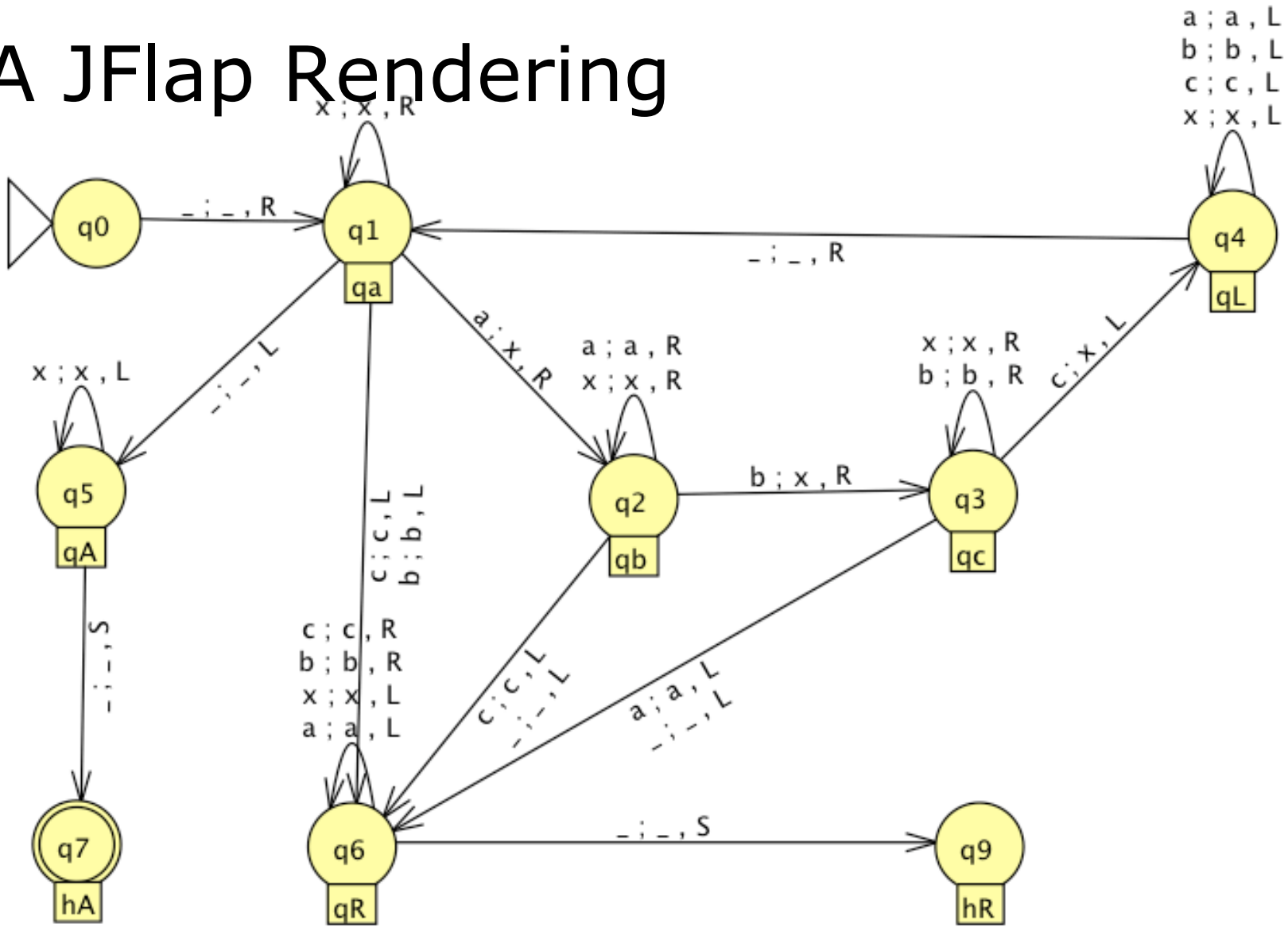
A TM accepting $\{a^n b^n c^n \mid n \in \mathbb{N}\}$

- $q_0, \sqcup \rightarrow q_a, \sqcup, R$
- $q_a, \sqcup \rightarrow q_A, \sqcup, L$
- $q_a, x \rightarrow q_a, x, R$
- $q_a, a \rightarrow q_b, x, R$
- $q_b, b \rightarrow q_c, x, R$
- $q_b, x \rightarrow q_b, x, R$
- $q_c, c \rightarrow q_L, x, L$
- $q_L, c \rightarrow q_L, c, L$
- $q_L, b \rightarrow q_L, b, L$
- $q_L, a \rightarrow q_L, a, L$
- $q_L, x \rightarrow q_L, x, L$
- $q_L, \sqcup \rightarrow q_a, \sqcup, R$
- $q_A, x \rightarrow q_A, x, L$
- $q_A, \sqcup \rightarrow h_a, \sqcup, S$
- $q_a, b \rightarrow q_R, b, L$
- $q_a, c \rightarrow q_R, c, L$
- $q_b, a \rightarrow q_b, a, R$
- $q_b, c \rightarrow q_R, c, L$
- $q_c, b \rightarrow q_c, b, R$
- $q_c, a \rightarrow q_R, c, L$
- $q_c, x \rightarrow q_c, x, R$
- $q_b, \sqcup \rightarrow q_R, \sqcup, L$
- $q_c, \sqcup \rightarrow q_R, \sqcup, L$
- $q_R, \sqcup \rightarrow h_R, \sqcup, S$
- $q_R, a \rightarrow q_R, a, L$
- $q_R, b \rightarrow q_R, b, L$
- $q_R, c \rightarrow q_R, c, L$
- Any other combinations go to q_R with no change in symbol or direction.

Legend for the Previous TM

- q_0: Just Starting
- q_a: Looking for a, reject anything else.
- q_b: Looking for b, skipping over a's.
- q_c: Looking for c, skipping over b's, reject anything else.
- q_L: Moving left after matching one a, b, and c, looking for blank.
- q_R: Moving left after deciding rejection.
- q_A: Moving left after deciding rejection.
- h_a: Halt and accept.
- h_r: Halt and reject.

A JFlap Rendering



A rex Rendering (see /cs/cs81/abc.rex)

```
f("q0", " ") => ["qa", " ", "R"];
f("qa", " ") => ["qA", " ", "L"];
f("qa", "a") => ["qb", "x", "R"];
f("qa", "b") => ["qR", "b", "L"];
f("qa", "c") => ["qR", "c", "L"];
f("qa", "x") => ["qa", "x", "R"];
f("qb", " ") => ["qR", " ", "L"];
f("qb", "a") => ["qb", "a", "R"];
f("qb", "b") => ["qc", "x", "R"];
f("qb", "c") => ["qR", "c", "L"];
f("qb", "x") => ["qb", "x", "R"];
f("qc", " ") => ["qR", " ", "L"];
f("qc", "a") => ["qR", "c", "L"];
f("qc", "b") => ["qc", "b", "R"];
f("qc", "c") => ["qL", "x", "L"];
f("qc", "x") => ["qc", "x", "R"];
```

```
f("qA", " ") => ["ha", " ", "S"];
f("qA", "x") => ["qA", "x", "L"];
f("qL", " ") => ["qa", " ", "R"];
f("qL", "a") => ["qL", "a", "L"];
f("qL", "b") => ["qL", "b", "L"];
f("qL", "c") => ["qL", "c", "L"];
f("qL", "x") => ["qL", "x", "L"];
f("qR", " ") => ["hr", " ", "S"];
f("qR", "a") => ["qR", "a", "L"];
f("qR", "b") => ["qR", "b", "L"];
f("qR", "c") => ["qR", "c", "L"];
f("qR", "x") => ["qR", "x", "L"];
```

The rex TM Infrastructure

```
// TM single-step function

step([L, Q, [X | M]]) => step1(L, f(Q, X), [X | M]);

step([L, Q, []])      => step([L, Q, [" "]]);   // Add more tape on right.


// This auxiliary step function discriminates based on the "move" symbol.
// The left tape is reversed for convenient access to its rightmost symbol.

step1(L,       [P, Y, "R"], [_ | M]) => [[Y | L], P, M];            // Move right

step1(L,       [P, Y, "S"], [_ | M]) => [L,       P, [Y | M]];      // Stationary

step1([Z | L], [P, Y, "L"], [_ | M]) => [L,       P, [Z, Y | M]];   // Move left

step1([],      [P, Y, "L"], [_ | M]) => [L,       P, [" ", Y | M]]; // More tape on left


// TM run Function

run([L, "ha", M]) => [[L, "ha", M]];           // Halt and accept

run([L, "hr", M]) => [[L, "hr", M]];           // Halt and reject

run(State) => [State | run(step(State))];      // Keep running
```

rex simulation (slightly edited)

```
[], q0, [ , a, a, b, b, c, c]
[ ], qa, [a, a, b, b, c, c]
[ , x], qb, [a, b, b, c, c]
[ , x, a], qb, [b, b, c, c]
[ , x, a, x], qc, [b, c, c]
[ , x, a, x, b], qc, [c, c]
[ , x, a, x], qL, [b, x, c]
[ , x, a], qL, [x, b, x, c]
[ , x], qL, [a, x, b, x, c]
[ ], qL, [x, a, x, b, x, c]
[], qL, [ , x, a, x, b, x, c]
[ ], qa, [x, a, x, b, x, c]
[ , x], qa, [a, x, b, x, c]
[ , x, x], qb, [x, b, x, c]
[ , x, x, x], qb, [b, x, c]
[ , x, x, x, x], qc, [x, c]
[ , x, x, x, x, x], qc, [c]
[ , x, x, x, x], qL, [x, x]
[ , x, x, x], qL, [x, x, x]
[ , x, x], qL, [x, x, x, x]
[ , x], qL, [x, x, x, x, x]
[ ], qL, [x, x, x, x, x, x]
[], qL, [ , x, x, x, x, x, x]
[ ], qa, [x, x, x, x, x, x]
[ , x], qa, [x, x, x, x, x]
[ , x, x], qa, [x, x, x, x]
[ , x, x, x], qa, [x, x, x]
[ , x, x, x, x], qa, [x, x]
[ , x, x, x, x, x], qa, [x]
[ , x, x, x, x, x, x], qa, []
[ , x, x, x, x, x], qA, [x,  ]
[ , x, x, x, x], qA, [x, x,  ]
[ , x, x, x], qA, [x, x, x,  ]
[ , x, x], qA, [x, x, x, x,  ]
[ , x], qA, [x, x, x, x, x,  ]
[ ], qA, [x, x, x, x, x, x,  ]
[], qA, [ , x, x, x, x, x, x,  ]
[], ha, [ , x, x, x, x, x, x,  ]
```

Another Possible Rendering

/cs/cs81/tm provides a TM simulator

There is documentation (tm.doc) and examples:

add1.tm	Adds 1 to a binary numeral
add.tm	Adds two binary numerals

add1.tm:

start	_	_	left	add1
add1	0	1	right	end
add1	_	1	right	end
add1	1	0	left	add1
end	0	0	right	end
end	1	1	right	end

Conclusion:

- There is a Turing machine that accepts a non-context free language.

How to use or get JFlap.

- Go to Susan Rodger's (the author's) page:

  www.cs.duke.edu/~rodger/tools/jflap/

- Either:
  - Download the .jar file and run it on your machine, or
  - Use the web applet provided.

- JFlap simulates: FA, NFA, PDA, TM, etc.

Fact:

- For every *context-sensitive* language L, there is a Turing Machine that accepts L.

- Proof: A TM can be programmed to generate all derivable strings from a type 1 grammar, up to the length of the input. Those strings are strictly increasing in length.

  At some determinable point, no new strings of length less-than-or-equal-to the length of the input will be generated. The TM can determine acceptance or rejection by then.

Possible TM behaviors on a given input

- Halt in the accepting state.
- Halt in the rejecting state.
- Fail to halt (**diverge**).

Acceptance vs. Recognition

- This is technical (but very important) distinction, and the terminology is not universal.
- We will use that of John Martin:
  - A TM **accepts** a language L if for every string in the language, the machine halts in the accepting state.
    - If L is not in the language, it may halt in the rejecting state, or may diverge.
  - A TM **recognizes** the language if it accepts the language and **always halts** (never diverges).

Acceptance vs. Recognition

- One reason the distinction is important is that there are languages that are:
  - accepted by some TM
  - not recognized by any TM

Terminology

- A language that is recognized by some TM is called "**recursive**".
- A language that is accepted by some TM is called "**recursively enumerable**" (R.E.).
- So recursive ⊆ recursively-enumerable, by definition (but not conversely).

Note on Terminology

- "**recursive**" doesn't mean that the language has some kind of self-referential structure.

- Rather this term comes from the "recursive functions" model of Godel and Kleene. These functions can be shown to be equivalent to Turing machines.

More on the Recursive vs. R.E. Distinction

- If a language is recursive, then its complement is also.
- This can be seen by simply swapping the accepting and rejecting halting states in a TM that recognizes the language.

More on the Recursive vs. R.E. Distinction

- If a language is R.E. and its complement is R.E., then the language is recursive.

- How to show this?

TMs Computing Functions

- Rather than accepting a language, we can use a TM to **compute a function**:
  - The machine starts with some string x on its tape initially.
  - The machine halts with some string y on its tape finally.
  - Then the corresponding function f would have

$$f(x) = y$$

Characteristic Functions

- A characteristic function is just a way to represent a subset of a given universe.
- If W is the universe, and S $\subseteq$ W, then

$$c_S(x) = \begin{cases} 1 \text{ if } x \in S \\ 0 \text{ otherwise} \end{cases}$$

- c_S: W $\rightarrow$ {0, 1} is the **characteristic function** of S.
- So sets are equivalent to these special kind of functions.

TMs Computing Partial Functions

- For some initial strings x the TM might **diverge**.
- More generally, we say the machine computes a **partial function**, and f(x) is undefined for such x.
- An ordinary function is a special case of a partial function, one which is nowhere undefined. We call such a partial function a **total function** to emphasize the distinction.

Strings as Natural Numbers

- We can view any string over a fixed finite alphabet as an **encoding** of a natural number.
- The encoding is a 1-1 correspondence.
- **Example**: Consider the alphabet {a, b}.
  - 0 ↔ ε
  - 1 ↔ a
  - 2 ↔ b
  - 3 ↔ aa
  - 4 ↔ ab
  - 5 ↔ ba
  - 6 ↔ bb
  - 7 ↔ aaa
  - . . .

(called the "2-adic" encoding)

Consequence

- Turing machines can be viewed as computing with (natural) **numbers** just as well as they can with strings.

Turing Machines as Enumerators

- Suppose M is a Turing Machine where we view the input as a natural number and the output as a string over alphabet □.
- Then M **enumerates** a subset of □*, in the sense that it computes a function

  f(i) = some element of □*, for each i □ □

- The set enumerated is {f(0), f(1), f(2), . . .}, which of course is a **language**.

We will eventually see:

- A language is recursively enumerable iff there is a TM that enumerates it.

- (Recall that the definition of R.E. is that there is a TM that **accepts** it.)

Encoding TMs as Strings

- We know that a TM can be described fully by a list of transition rules, of the form:
  - $q_0, \square \rightarrow q_a, \square, R$
  - $q_a, \square \rightarrow q_A, \square, L$
  - $q_a, x \rightarrow q_a, x, L$
  - $q_a, a \rightarrow q_b, x, R$
    etc.
- from earlier examples.
- We can devise a way to encode an **arbitrary** set of such rules into a **fixed alphabet**.
- Although the number of states and tape symbols can be arbitrary, we can **encode** these by using strings of symbols, say {a, b}, and concatenate the symbols to get rules, and concatenate rules to get machines.

Encoding TMs as Strings

- Not **every** string of symbols in the encoding alphabet will be a well-formed TM.
- But we can determine algorithmically which are and which aren't. For those that aren't, assume they default to a trivial TM: one that immediately halts in the rejecting state.
- We thus have an enumeration of **all the TM's**:

  $T_0, T_1, T_2, \ldots$

- We can do the same for encodings of **initial tapes**:

  $x_0, x_1, x_2, \ldots$

Fact:

- There are functions that cannot be computed by **any** TM.
- The simple way to see this is to use the concept of countability.

Definitions:

- Two sets are **equipotent** if there is a 1-1 correspondence between them. (Equipotence is sort of an equivalence relation.) Write |A| = |B| to denote that A and B are equipotent.
- A set is **infinite** if it is equipotent with a *proper subset* of itself, otherwise it is **finite**.
- A set is **countably-infinite** if it is equipotent with $\mathbb{N}$, the set of natural numbers.
- Example: The set of all TMs is countably infinite.
- A set is **countable** if it is finite or countably-infinite; otherwise is **uncountable**.

Further Definitions:

- Write $|A| \leq |B|$ to denote that there is a 1-1 mapping from A into B. This means that B is "at least as big" as A.
- It can be shown (Schröder-Bernstein theorem) that $|A| \leq |B|$ and $|B| \leq |A|$ together imply $|A| = |B|$.
- Write $|A| < |B|$ to mean $|A| \leq |B|$ but not $|B| \leq |A|$.
- Example: A is **countable** and B is **uncountable**, then $|A| < |B|$.

Theorem of Cantor

- The set of functions of the form
  $\square \rightarrow \square$ {0, 1}
  is uncountable.
- **Proof by contradiction:**
  Suppose that the set of such functions is countable.
  This means that there is an enumeration of all such functions:
  $f_0, f_1, f_2, \ldots$
- Define the function
  $f^*(i) = 1$ if $f_i(i) = 0$, and 0 otherwise.

Proof of Theorem of Cantor (cont'd)

- $f^*(i) = 1$ if $f_i(i) = 0$, and 0 otherwise.

- Clearly $f^*: \square \square \{0, 1\}$, so it must be in the list somewhere. That is, there is a j such that $f^* = f_j$.
- Then we have

$f^*(j) = 1$ if $f_j(j) = 0$, and 0 otherwise

which is to say

$f_j(j) = 1$ if $f_j(j) = 0$, and 0 otherwise

which is absurd.

Diagonalization Proofs

- The preceding type of proof is known as proof by “diagonalization”, because the function f* can be seen as being derived from the “diagonal” in a listing of all the characteristic functions (as if infinitely-long bit vectors).

Corollary

- There are functions of the form
  $\square \square \{0, 1\}$
  that are not computed by any Turing machine.

- Proof: The set of such functions was shown uncountable. But the set of Turing machines were shown *countable*.

  So there functions computable by Turing machines (interpreting acceptance as 1 and rejection as 0) form a **proper subset** of the functions of the indicated form.

Unsolvability of the **Halting Problem**

- We showed that there are non-Turing computable functions, but this was non-constructive; we didn't exhibit such a function.
- Now we will:

  The partial function h given by

$$h(x_j) = \begin{cases} 1 \text{ if } T_j \text{ diverges on } x_j \\ \text{undefined (diverges) if } T_j \text{ defined on } x_j \end{cases}$$

  is not computable by any Turing machine.

Proof of Unsolvability of the Halting Problem

- Suppose that h is computable by some TM, say T_k. Then from the definition of h:

$$T_k(x_i) = \begin{cases} 1 \text{ if } T_j \text{ diverges on } x_j \\ \text{undefined (diverges) if } T_j \text{ defined on } x_j \end{cases}$$

But if we supply the corresponding argument x_k :

$$T_k(x_k) = \begin{cases} 1 \text{ if } T_k \text{ diverges on } x_k \\ \text{undefined (diverges) if } T_k \text{ defined on } x_k \end{cases}$$

which is absurd, because no machine can both diverge and give answer 1.

Corollary

- These functions are not computable:

$$g(j) = \begin{cases} 1 \text{ if } T_j \text{ diverges on } x_j \\ 0 \text{ if } T_j \text{ defined on } x_j \end{cases}$$

$$f(j, k) = \begin{cases} 1 \text{ if } T_j \text{ diverges on } x_k \\ \text{undefined if } T_j \text{ defined on } x_k \end{cases}$$

- Proof: If these were computable, it can be argued that h would be computable as well, by a simple modification of the TM that computes g or f.

Corollary

- This language is not recursively-enumerable:

$$\{x_j \mid T_j \text{ diverges on } x_j\}$$

Variants on Turing Machines

- The next few slides present variations that, at first glance, might be either simpler or more powerful than our TM model.

- Upon further reflection, they can be seen to be equivalent to our model.

TM without S (stationary) Move

- This is simpler, but S can be simulated by a combination of L and R moves.

TM that can Write or Move but not both

- This is simpler, but the more complex original machine can be simulated by a finite sequence of steps of the restricted machine.

TM with Added Registers

- Some finite number of **registers** is added, each with a finite set of possible values.
- The transitions can query the registers for values and change their contents.
- Everything can be simulated by the enlarging control states.

TM with Non-recursive Subroutines

- As long as the subroutines are not recursive, they can be simulated with added control states to remember the **return state**.

- These subroutines can also take **arguments** provided the values are drawn from a finite set.

TM with Multi-track Tape

- This machine has the tape divided into separate **tracks**.
- Each transition specifies which track being read and written.
- Tracks can be simulated by taking the tape alphabet to be the **Cartesian product** of the track alphabets.
- Another approach is to **interleave** the cells of the tracks.

TM with Semi-infinite Tape

- The tape is extendible on one side only.
- This is simpler, but a two-infinite tape can be simulated by **folding** the latter at some arbitrary point on the tape into two “tracks”, each simulating one direction of tape expansion.

TM with Insertible Cells

- Rather than allowing extension only by means of writing on formerly blank cells at the tape extremities, this machine has a special move that inserts a new cell just to the right (say) of where the head is. The symbol written would go in that new cell and the head would stay put.
- Other variations are possible.

TM with Multiple Tapes

- This machine has multiple tapes, rather than a single tape. Each tape has its own head.

- An n-tape machine can be simulated by a 2n-track machine, where half of the tracks are devoted to keeping the head position for the corresponding tape.

TM with Recursive Subroutines

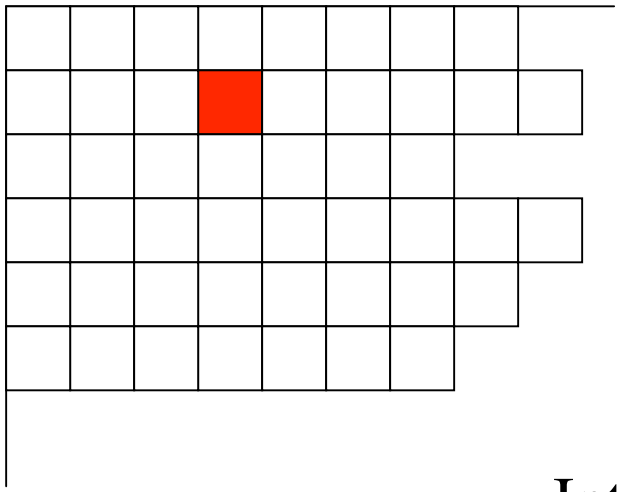
- As long as the return is to specific control states, recursion can be carried out by allocating a separate tape to implement a **stack** of return states.

TM with 2-Dimensional Tape

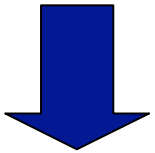
- This machine has a 2-D tape and moves can be Up or Down as well as L, R, S.
- To simulate a 2-D tape, divide the 2-D plane into 4 quadrants and use a separate track for each.
- Each track will contain strips representing the non-blank portions of the corresponding quadrant.
- The simulating machine would need to move cells to add to a strip.
- It would add cells to add a new strip.

TM with 2-Dimensional Tape

2-D Quadrant

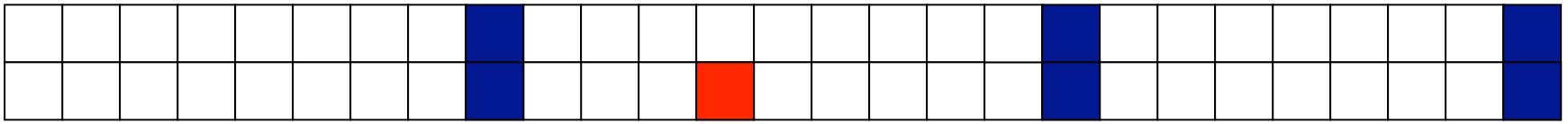


Strips



Inter-strip markers

1-D Track (1 of 4)



Simulated-head marker

Non-Deterministic Turing Machines

- A TM that specifies more than one transition for a given control state and read-symbol is non-deterministic (NTM).
- An NTM **accepts** an input x if **some** sequence of moves leads to the halting accepting state.
- An NTM **recognizes** an input x if it accepts x and every sequence of moves leads to the halting accepting state.

Non-Deterministic Turing Machines

- The **language** L(M) **accepted** by an NTM M is the set of all x that are accepted by M.

- If M halts on every input, and recognizes exactly the strings in L, then M is said to **recognize** L.

Non-Deterministic Turing Machines

- **Theorem**: If L is accepted by an NTM N, then L is also accepted by a TM M.

- To show this, we'd construct M to simulate all possible moves of N. It can do so by keeping on its tape a replica of the configurations of M. Generally, this set of configurations would increase in size. If one configuration became accepting, then M would accept.

Type 0 Languages are R.E.

- Recall the definition of type 0 grammar. Their productions have an unrestricted LHS and RHS.
- **Theorem**: If L is a Type 0 language, then there is a TM accepting L.
- Proof: Let G be a grammar for L. Construct an NTM that works as follows:
  - With input x, allocate the remainder of the tape to simulate a derivation in G. This part of the tapes starts with just S.
  - Non-deterministically choose a rule in G and rewrite the working area according to that rule.
  - After applying a rule compare with x. If x has been generated, then accept.

R.E. Languages are Type 0

- **Theorem**: If L is accepted by a TM, then L is a Type 0 language.
- Proof: Let M be a TM. We want to construct a grammar G that generates exactly the terminal strings that M accepts:
  - G will generate two adjacent copies of every string in □* non-deterministically.
  - G will have productions corresponding to each transition of M.
  - Once G generates a pair of strings in □*, G will simulate the behavior of M on one of the copies.
  - If M accepts the string, then M will erase all symbols but the other copy, leaving it as a terminal string.

Linear-Bounded Automata

- A Linear-Bounded Automaton (LBA) is a **non-deterministic** Turing machine with its tape a fixed length. There are end-marker symbols that it can use to detect the boundary.
- Analogous to the previous result can be proven:
- **Theorem**: A language L is accepted by an LBA iff L is context-sensitive (type 1).

Universal Turing Machines

- There are other uses of strings that describe Turing machines besides unsolvability arguments.
- Such a string is effectively a **program** for a Turing machine.
- A Turing machine that interprets such a program to carry out the actions specified is calleda **Universal Turing Machine** (UTM).

Universal Turing Machines

- UTMs can be shown to exist by constructing them.
- Think about what would be required.
  - The tape has to hold the tape of the machine being simulated.
  - The tape has to hold the program of the machine being simulated.
  - The program must be laid out in such a way that the necessary markers can be inserted to keep track of the current state, etc.
  - All this is possible, if somewhat laborious to construct.
- Whether a machine is universal will depend on the particular encoding used.

Specific Universal Turing Machines

- The first was constructed by Turing himself.
- Shannon showed how a UTM could be converted to a 2-symbol machine and to a 2-state machine (not at the same time), by augmenting states or symbols, respectively.
- Minsky (1960) gave a 7-state 6-symbol machine.
- Watanabe (1961) gave an 8-state 5-symbol machine.
- Minsky (1962) gave a 7-state 4-symbol machine.
- Rogozhin (1996) gave a 4-state 6-symbol machine, among others.
- Wolfram (2002) gives a 2-state 5-symbol machine.

A Specific Universal Turing Machine

- 8-state, 5 symbol UTM published by **Watanabe** in 1961 (see http://portal.acm.org/citation.cfm?id=321090&jmp=cit&dl=GUIDE&dl=ACM)
- To understand this, you would need to understand the encoding used.

TABLE 10

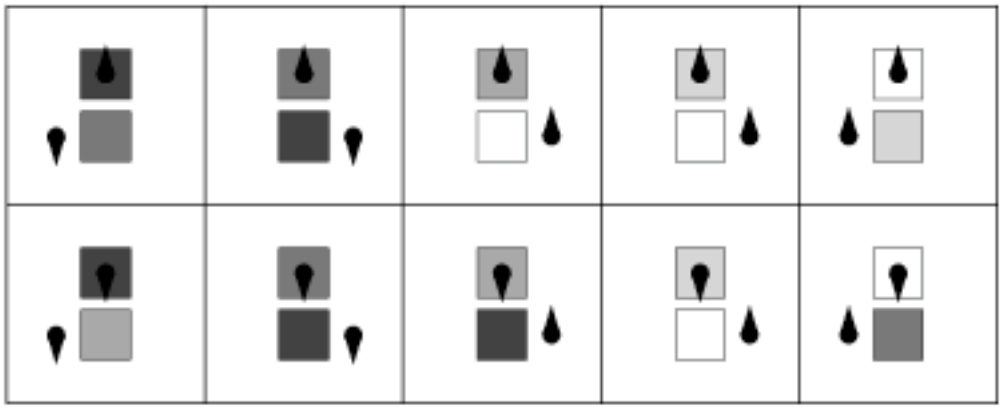
	0	1	*	0′	1′
A	* L B	* L C	R	0 L E	1 R
B	0′ L	1′ L	0′ L D	0 R G	1 R H
C	0′ L	1′ L	0′ L B	R G	R H
D	R C	1′ R E	R A	* L	L
E	R D	1′ L F	0′ R	* L D	R
F	0′ L	1′ L	0 R G	0 R B	1 R B
G	R A	R A	0 L F	0 R	1 R
H	L A	L A	1 L F	0 R	1 R

A Specific Universal Turing Machine

- 2-state, 5 symbol UTM published by **Wolfram** in 2002

states →

symbols ↓



adapted from Wolfram, S. *A New Kind of Science*. Wolfram Media, p. 707, 2002.

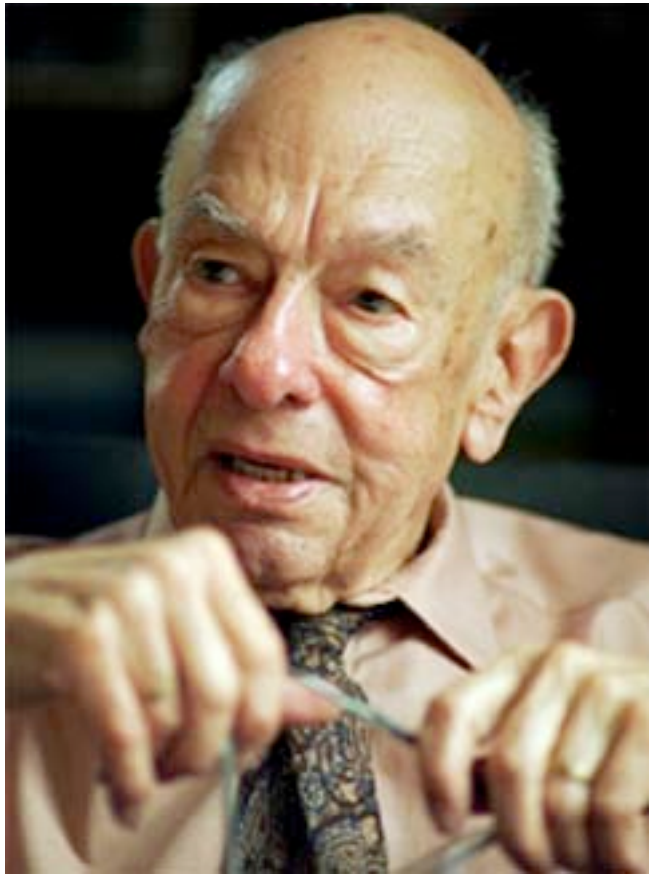
Wolfram expresses belief that there is a 2-state, 3-symbol UTM: http://mathworld.wolfram.com/UniversalTuringMachine.html

No 2-state, 2-symbol UTM exists.

Turing machines that can print their own description

- At first this might sound impossible.
- Machines that can print their own description have value in establishing certain theoretical results.
- Such machines (and programs in various languages) have been called "Quines" after Willard Van Orman Quine, a famous logician who wrote on self-referential paradoxes.

Willard Van Orman Quine (1908 - 2000)



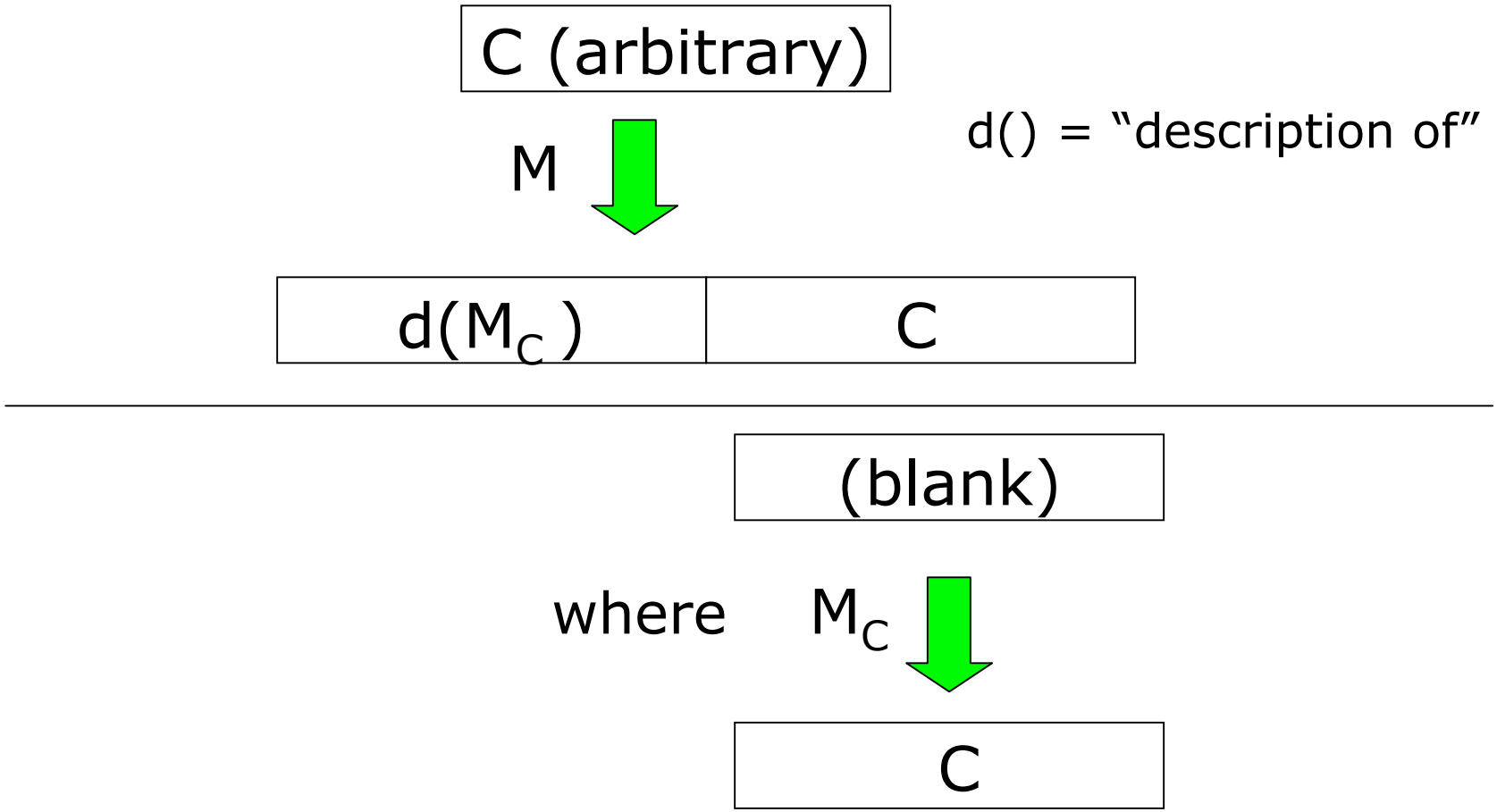
There is a TM that can print its own description

- **Proof**:
- Let C be the contents of an arbitrary region of tape (say one delimited by blanks).

- **First** construct M, a machine that can read the tape contents C and print the **description** of a machine M_C that will print C, followed by C itself. Let the machine M_C have initial state q_0.

- [Note that M does not attempt to "understand" the meaning of C. It simply creates a machine that prints a sequence.]

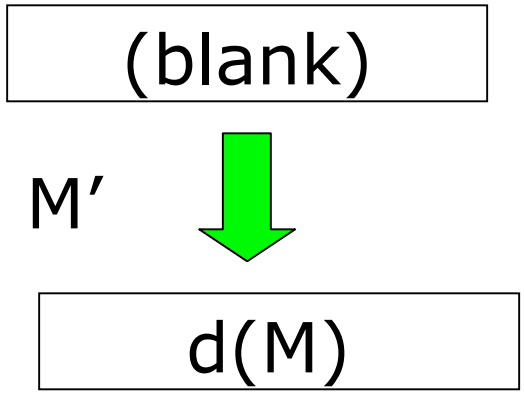
Behavior of M



A TM that can print its own description, continued

- Next construct M’ a machine that can print the description of the fixed machine M on an initially blank tape, then transfer control to a state named q_0 which is not otherwise in M’ (but as we know, is the name of the initial state of M_C).

Behavior of M’



where M is as on the previous diagram.

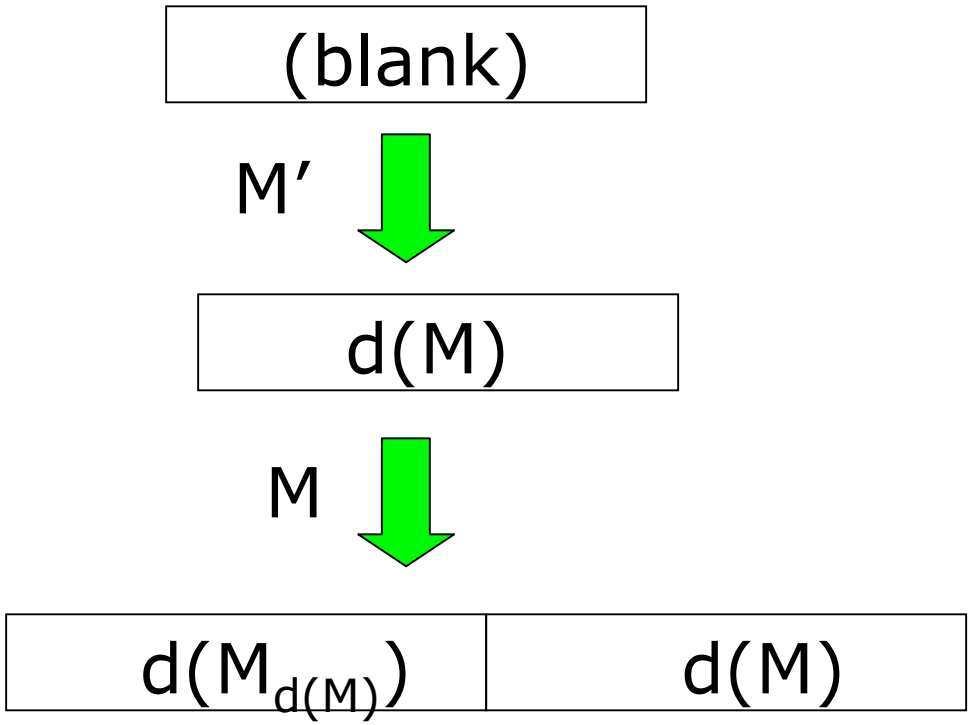
There is a TM that can print its own description

- Consider the combined program M';M.
- This sequence will:
  - Print a description of M.
  - Print the description of
    - (a machine that prints the description of M)
    - along with the description of M.

- So the description written will be that of M';M.

- Hence this machine prints its own description.

Summary of M’; M



But $d(M_{d(M)})$ ***is*** d(M’), so the final tape is d(M’;M).

Difficulties Constructing Self-printing Programs

- One difficulty is quoting. To get a literal symbol to print, the normal way is to quote it. But that means that the quote marks have to be printed too. And of course, they will have to be quoted, which means that escape characters may be needed. And the escape characters themselves will have to be printed.

- An escape for this is to use numeric codes that print as characters in lieu of the actual literal symbols. The numeric codes don’t normally have to be quoted.

A C Quine (author unknown)

```
char f[] =
"char f[] =%c%c%s%c;%cmain()
{printf(f,10,34,f,34,10,10);}%c";
main() {printf(f,10,34,f,34,10,10);}
```

Example: Prolog Quine

Author: Pekka P. Pirinen (pekka@harlequin.co.uk)

```
quine:–Q="write(quine),write((:–)),put(81),put(61),put(34),writes(Q),put(34),put(44),writes(Q),put(
46),put(13),put(10),write(writes),put(40),put(91),put(72),put(124),put(84),put(93),put(41),write((:–)
),write(put),put(40),put(72),put(41),put(44),write(writes),put(40),put(84),put(41),put(46),put(13),put
(10),write(writes),put(40),put(91),put(93),put(41),put(46)",write(quine),write((:–)),put(81),put(61),p
ut(34),writes(Q),put(34),put(44),writes(Q),put(46),put(13),put(10),write(writes),put(40),put(91),put(
72),put(124),put(84),put(93),put(41),write((:–)),write(put),put(40),put(72),put(41),put(44),write(writ
es),put(40),put(84),put(41),put(46),put(13),put(10),write(writes),put(40),put(91),put(93),put(41),put
(46).
writes([H|T]):-put(H),writes(T).
writes([]).
```

This is 3-lines; the first line is one really long one.
put(X) outputs the character corresponding to numeric code X.
For example, 81 is the code for Q (a Prolog variable), 34 is the code for “, etc..
“…” creates a list of character codes for the character string.

Prolog Quine Execution Log

```
turing ~:3> prolog +l quine.pl
% compiling file /home/keller/quine.pl
% quine.pl compiled in module user, 0.080 sec 3,648 bytes
Quintus Prolog Release 3.2 (Sun 4, SunOS 5.3)
Copyright (C) 1994, Quintus Corporation.  All rights reserved.
301 East Evelyn Ave, Mountain View, California U.S.A. (415) 254-2800
Licensed to Harvey Mudd College, CS Dept.

| ?- quine.
quine:–Q="write(quine),write((:–)),put(81),put(61),put(34),writes(Q),put(34),put(44),writes(Q),put(
46),put(13),put(10),write(writes),put(40),put(91),put(72),put(124),put(84),put(93),put(41),write((:–)
),write(put),put(40),put(72),put(41),put(44),write(writes),put(40),put(84),put(41),put(46),put(13),put
(10),write(writes),put(40),put(91),put(93),put(41),put(46)",write(quine),write((:–)),put(81),put(61),p
ut(34),writes(Q),put(34),put(44),writes(Q),put(46),put(13),put(10),write(writes),put(40),put(91),put(
72),put(124),put(84),put(93),put(41),write((:–)),write(put),put(40),put(72),put(41),put(44),write(writ
es),put(40),put(84),put(41),put(46),put(13),put(10),write(writes),put(40),put(91),put(93),put(41),put
(46).
writes([H|T]):-put(H),writes(T).
writes([]).
yes
| ?-
```

Example: A rex Quine
constructed by a Pomona College Student

```
a="\\";aa="a";
b="\"";bb="b";
c="=";cc="c";
d="print(

                aa,c,      b,    a,a,b,     f,
               aa,aa,     c,b,aa,b,f,g,bb,c,b,
              a,b,b,f    ,bb,bb,c,b,bb,b,f,g,
             cc,c,b,c   ,b,f,   cc,cc,c,b,cc,
            b,    f,g  ,dd            ,c,b,
           d,     b,f   ,              g,g,
          dd,     dd,                  c,b,
         dd,      b,f                 ,g,ee
        ,c,b      ,e,                 b,f,
        ee,       ee,                 c,b,
       ee,b,f,     g,ff,c,            b,f,
       b,f,ff,ff,c,b,ff,b,f,      g,gg,
                c,b,a          ,nn,b,
                 f,gg         ,gg,c,b,
                 gg,b,f,       g,nn,nn,c
                ,b,nn,b,      f,g,g,d,g);";
```

```
dd="d";
e=")";ee="e";
f=";";ff="f";
g="\n";gg="g";
nn="n";

print(

                aa,c,      b,    a,a,b,     f,
               aa,aa,     c,b,aa,b,f,g,bb,c,b,
              a,b,b,f    ,bb,bb,c,b,bb,b,f,g,
             cc,c,b,c   ,b,f,   cc,cc,c,b,cc,
            b,    f,g  ,dd            ,c,b,
           d,     b,f   ,              g,g,
          dd,     dd,                  c,b,
         dd,      b,f                 ,g,ee
        ,c,b      ,e,                 b,f,
        ee,       ee,                 c,b,
       ee,b,f,     g,ff,c,            b,f,
       b,f,ff,ff,c,b,ff,b,f,      g,gg,
                c,b,a          ,nn,b,
                 f,gg         ,gg,c,b,
                 gg,b,f,       g,nn,nn,c
                ,b,nn,b,      f,g,g,d,g);
```

For many Quines in many diverse languages, see:

http://www.nyx.net/~gthompso/quine.htm

For other varieties of “signature” programs, see:

http://home.planet.nl/~faase009/Signindex.html

Extended Quines

- A Quine can be extended to have arbitrary other functionality that could be computed in the given framework.

- For example, suppose f: □ □ □ is a function in such a framework. We could define f'(n) = f(n-1) for n > 0 and have f'(0) return ("print") the description of f' itself.

Other Uses of Self-Reference

- Programs, e.g. Turing machine programs, can be assumed to have reference to their own source code (as a built-in constant).

- Equivalently, in an enumeration of the Turing computable functions

  $T_0, T_1, T_2, \ldots$

  the definition of a function T_j can involve use of its own index j.

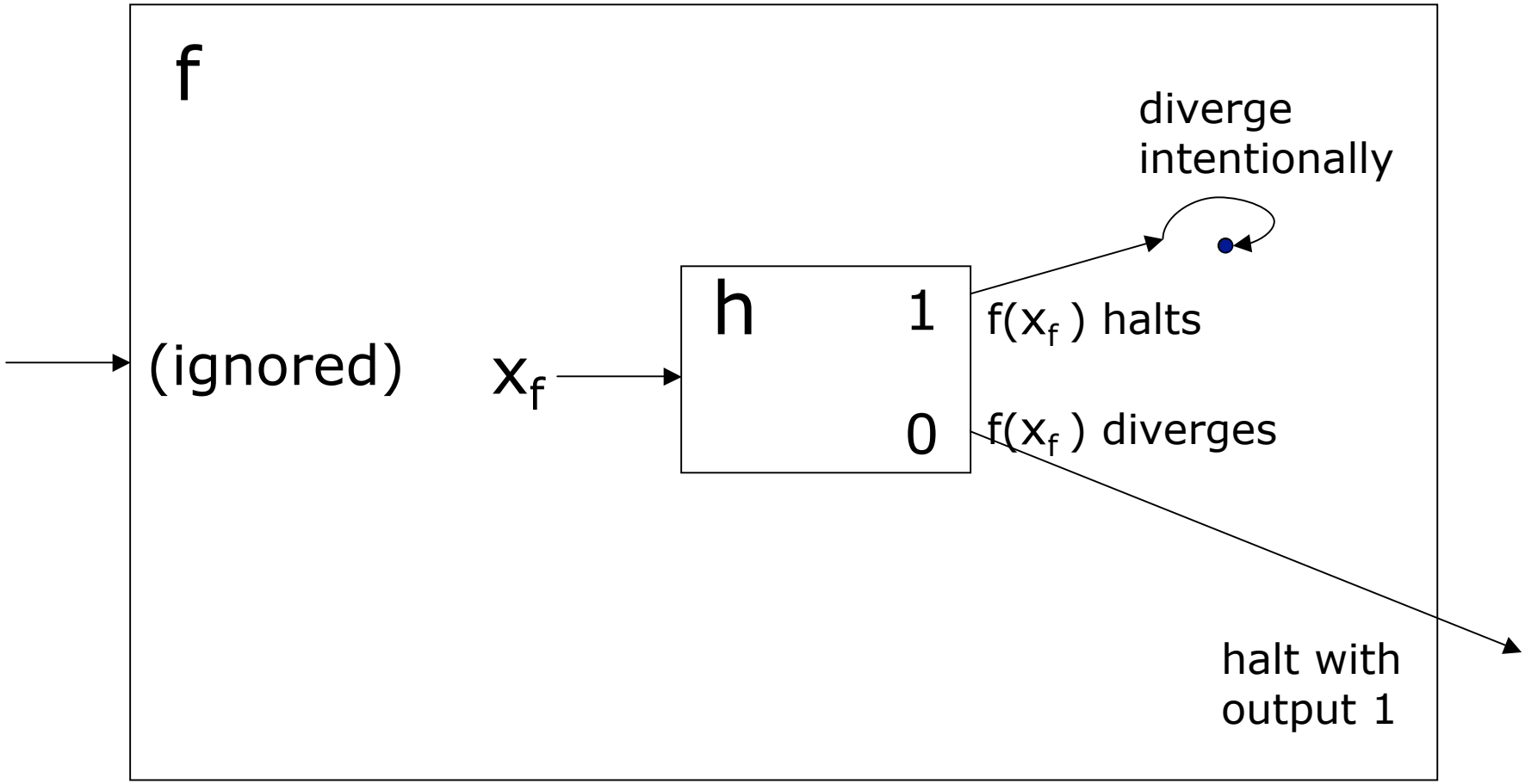
Using Self-Reference to Establish the Unsolvability of the Halting Problem

- Consider the function

$$h(x_j) = \begin{cases} 1 \text{ if } T_j \text{ halts on input } x_j \\ 0 \text{ otherwise} \end{cases}$$

- Suppose h were computable.
- Let **f** be the machine that, with input x_j runs $h(x_f)$, where x_f is the description of f, then if 1 is returned **diverges** intentionally, otherwise returns 1.
- If h were computable, so would f be. Now if we give x_f as the argument to f, then we get a contradiction.

The machine f always gets it wrong.
This is because h can't really exist.



The Blank-Tape Halting Problem

- This problem is *ostensibly* simpler to solve than the halting problem:

$$b(x_j) = \begin{cases} 1 \text{ if } T_j \text{ halts on the all-blank tape} \\ 0 \text{ if } T_j \text{ diverges on the all-blank tape} \end{cases}$$

vs.

$$h(x_j) = \begin{cases} 1 \text{ if } T_j \text{ halts on input } x_j \\ 0 \text{ if } T_j \text{ diverges on input } x_j \end{cases}$$

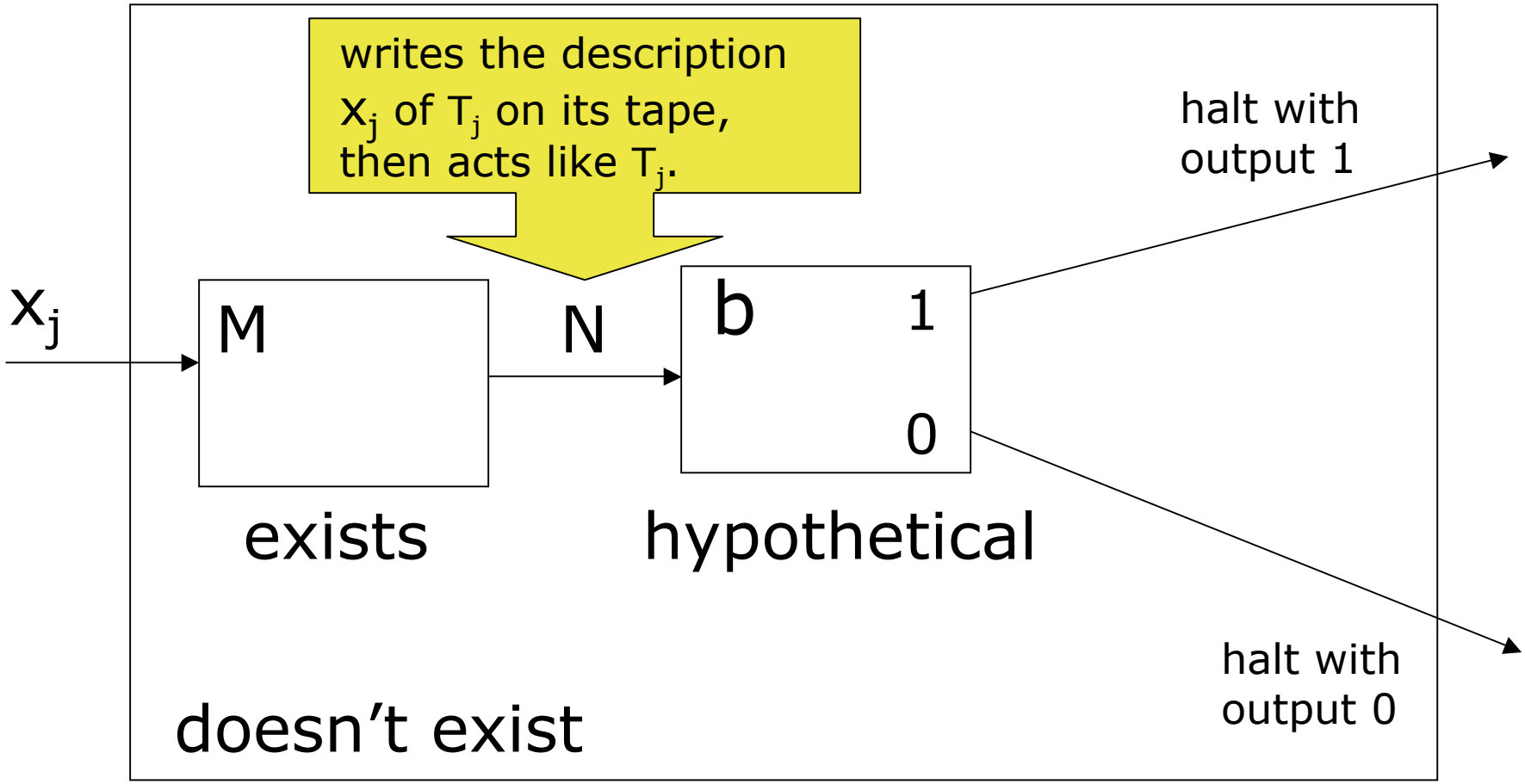
Blank-Tape Halting is Unsolvable

- It turns out that blank-tape halting is just different, not really simpler, as the following argument shows.
- Suppose that there is a Turing machine B that implements function b.
- Try to use B to solve the original halting problem.

Blank-Tape Halting is Unsolvable

- Let M be a machine such that, with any input x_j, M **constructs the description** of a machine N that
  - writes the description x_j of T_j on its tape, then moves its head to the left end of what it wrote and acts like T_j.
- So N on a *blank* tape is equivalent to T_j on x_j.
- So $b(N) = 1$ iff T_j halts on x_j .
- Therefore $b(M(x_j)) = h(x_j)$, i.e. if a TM for b existed, then so does a TM for h (since a TM for M obviously exists). But the latter can't be.

The Outer Box Solves the Halting Problem



Key Ideas in the Blank-Tape Problem

- Another problem (the halting problem) was established as unsolvable.
- TM's can be composed like functions.
- TM's can manipulate descriptions of other TM's.

Perspective

- Let F be a family of machines.
- Is there an algorithm that determines, for any machine M ∈ F, whether or not ∅ = L(M)?
- For finite-state machines: Yes.
- For pushdown automata: Yes.
- For Turing machines: **No**.

The Some-Tape Halting Problem

- Is there an algorithm for determining whether any TM halts on **some** input tape?
- Try to use the **Key Ideas** stated earlier.
- Note that it is enough to show that some-tape halting **can't** be determined for **some** TM's (It definitely can for others.)

The Some-Tape Halting Problem

- $s(x_j) = \begin{cases} 1 \text{ if } T_j \text{ halts on some tape} \\ 0 \text{ if } T_j \text{ diverges on every tape} \end{cases}$

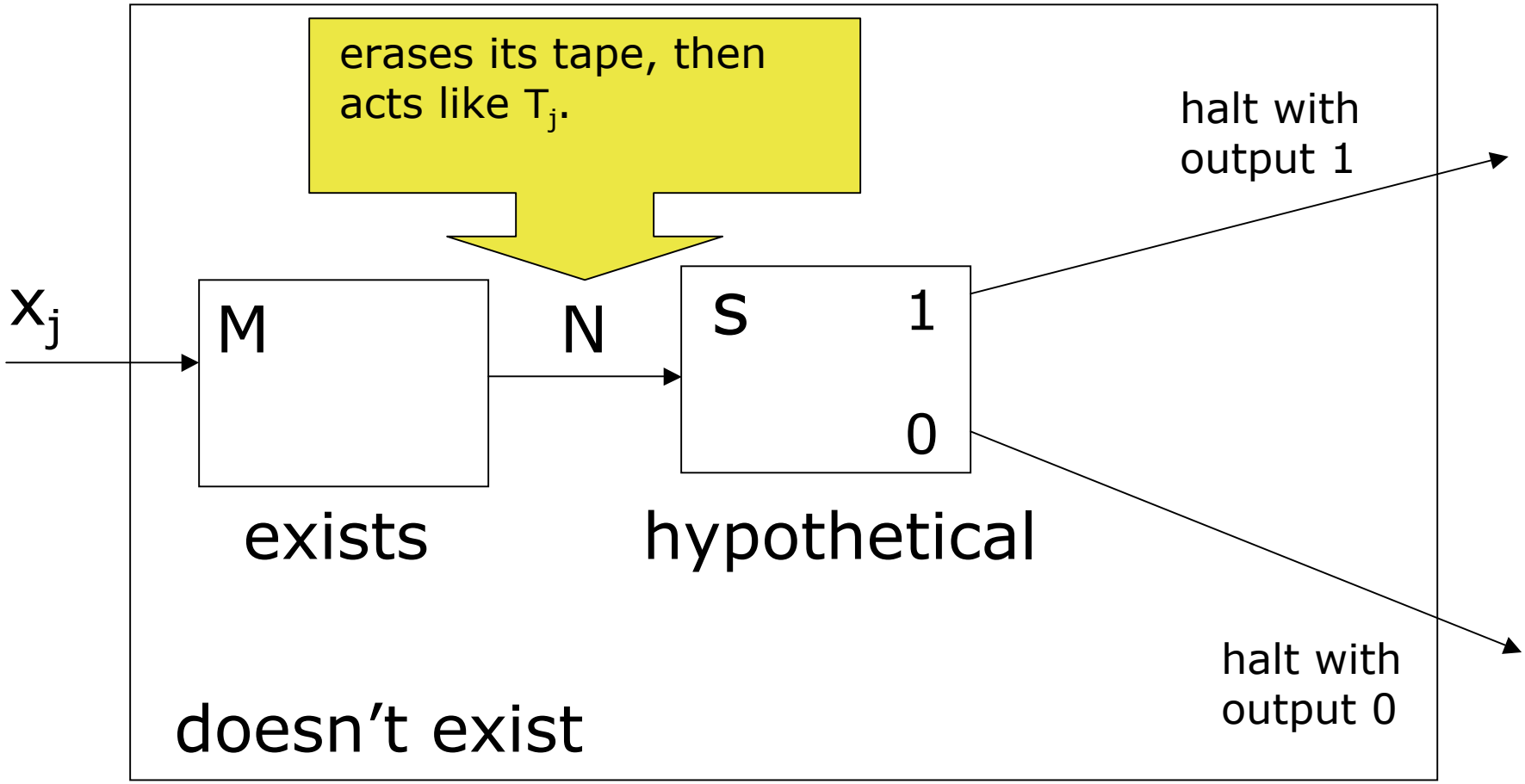
  This problem is ostensibly less restrictive than the general halting problem, because the algorithm only needs to find some tape on which the machine in question halts; we are not restricted to a particular tape in any way.

Some-Tape Halting is Unsolvable

- Let M be a machine such that, with any input x_j, M **constructs the description** of a machine N that
  - erases whatever is on its tape*, then acts like T_j (on a blank tape).
- N on *any* tape is equivalent to T_j on a blank tape.
- So s(N) = 1 iff T_j halts on a blank tape.
- Therefore $s(M(x_j)) = b(x_j)$, i.e. if a TM for s existed, then so does a TM for b (since a TM for M obviously exists). But the latter can't be, as we already showed.

(* see Incremental Erasure)

The Outer Box Solves the Blank-Tape Problem



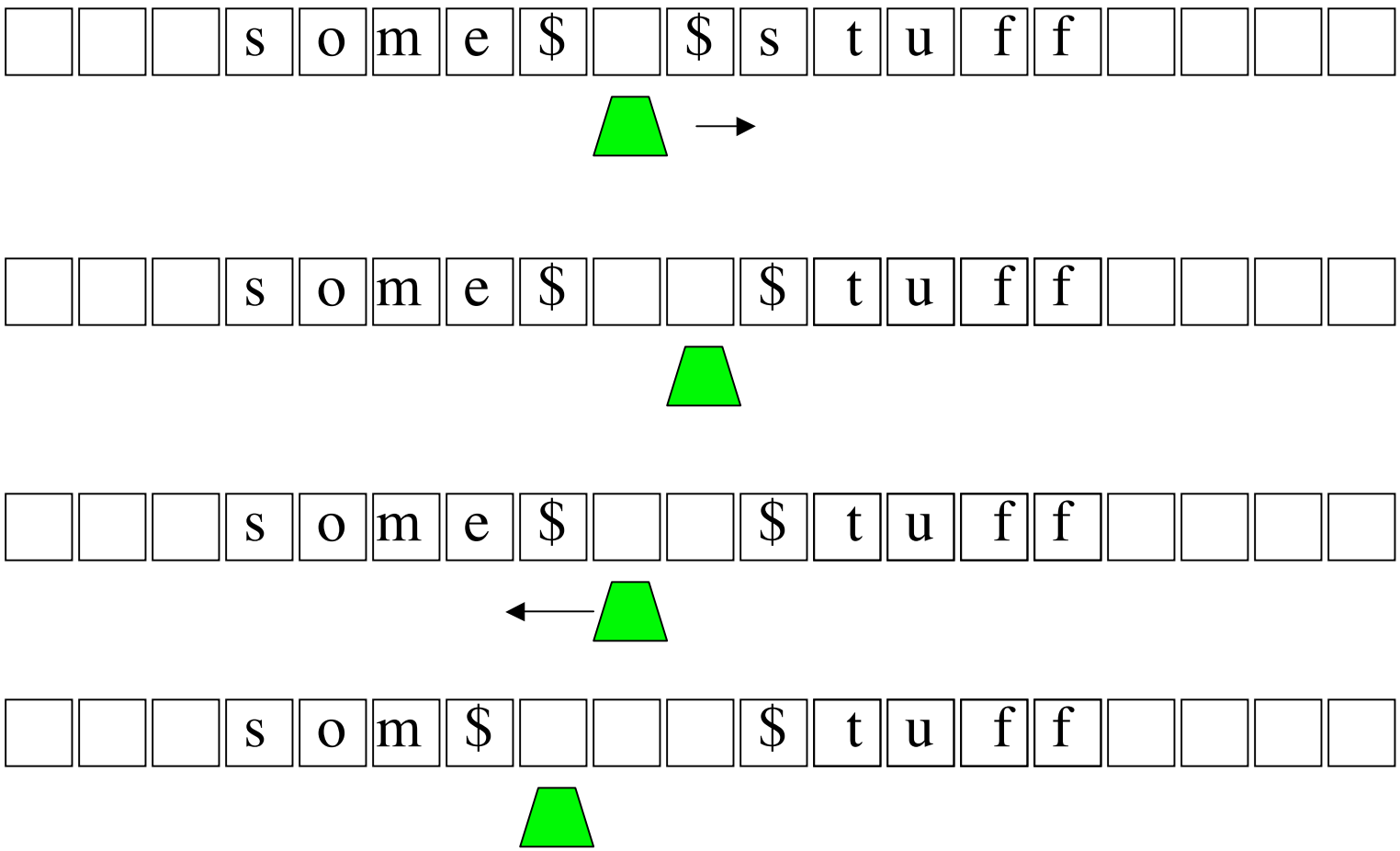
Incremental Erasure

- The idea of a machine erasing **whatever** is on its tape may seem like an impossibility.
- Consider the technique of incremental erasure:
  - Instead of erasing everything at the very beginning, the machine only erases when it would have have encountered a non-blank square for the first time.

Incremental Erasure

- Overwrite the cell under the head with a blank.
- Introduce a **new symbol**, say $, and overwrite the cells on either side of the head with it.
- Introduce **new rules** that behave as follows: Whenever a cell x with $ is moved onto from the left for the first time, write a blank in its place and a $ to its right, then move back and behave as the original machine would at cell x (which now contains a blank). Do the same for when cells containing $ are moved to from the right, except place a $ to their left.
- This modified machine's behavior, in the long run, behaves as the original machine would have on a blank tape.

Incremental Erasure



Gordian-Knot Erasure

- It would not be a violation of the spirit of Turing's argument to endow the machine with a special "erase" action.
- Add E to the repertoire of moves {L, R, S}.
- Doing E erases the entire tape.

Problem Reduction Formalized

- In the preceding couple of proofs we implicitly used the idea of "problem reduction".
- Let P and Q be problems to be solved.
- Write $P \leq Q$ (P is **reducible to** Q) if there is an *algorithm* that will transform any *instance* of P into an *instance* of Q in such a way that the answer to the instance of P is the answer to Q.

Examples of Reductions

- Halting Problem $\leq$ Blank-Tape Problem
  - If the blank-tape problem were solvable, then the halting problem would be solvable.
- Blank-Tape Problem $\leq$ Some-Tape Problem
  - If the some-tape problem were solvable, then the blank-tape problem would be solvable.
- Note that this may seem somewhat "backward": we are reducing the problem that **can't** be solved to the problem in question.

Notes on Reductions

- We typically use reductions in the "contrapositive direction": to show that a problem is **un**solvable.
  - If the blank-tape problem were solvable, then the halting problem would be solvable.
  - The halting problem is **un**solvable, therefore the blank-tape problem is **un**solvable.
  - The second statement is the **contrapositive** of the first.
- Reduction is transitive:
  - If $P \leq Q$ and $Q \leq R$, then $P \leq R$.

Reduction Formalized

- Suppose P and Q are language recognition problems.
- P is the problem: Give an algorithm (TM) that will determine, for any input $x \in \Sigma^*$, whether or not $x \in L_P$.
- Q is the problem: Give an algorithm (TM) that will determine, for any input $y \in \Sigma^*$, whether or not $y \in L_Q$.
- $P \leq Q$ provided there is an algorithm (TM) $f: \Sigma^* \rightarrow \Sigma^*$ such that

$$x \in L_P \text{ iff } f(x) \in L_Q$$

Double Utility of Reduction

- Reduction can apply to language **acceptance** problems as well as language **recognition** problems.

- **Recognition**: P $\leq$ Q implies L_P is recognized by a TM provided L_Q is recognized by a TM.
  - i.e. L_Q recursive $\Rightarrow$ L_P recursive

- **Acceptance**: P $\leq$ Q implies L_P is accepted by a TM provided L_Q is accepted by a TM.
  - i.e. L_Q recursively-enumerable $\Rightarrow$ L_P recursively-enumerable

Reduction Examples

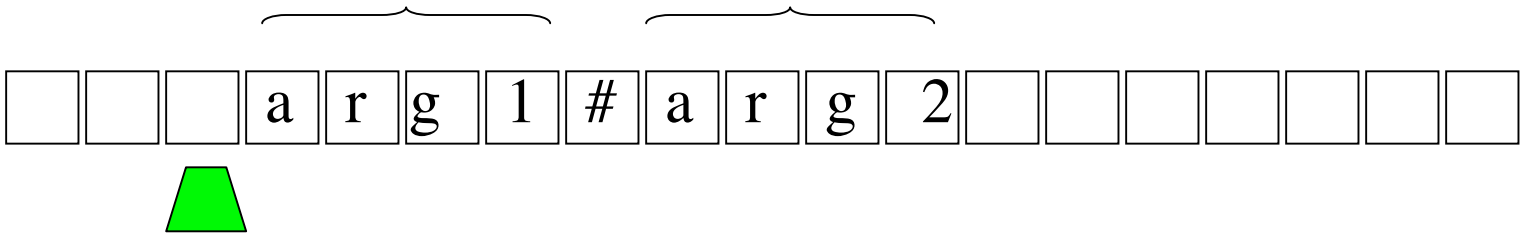
- P is the halting problem,
  Q is the blank-tape halting problem:
  - $L_P = \{x_j \mid T_j \text{ halts on } x_j\}$
  - $L_Q = \{x_j \mid T_j \text{ halts on } \square\}$
  - $f(x_j)$ = description of the machine such that writes x_f on its tape and then behaves like T_j.
  - $P \leq Q$
  - Q solvable $\Rightarrow$ P solvable
  - P not solvable $\Rightarrow$ Q not solvable

Reduction Examples

- Q is the blank-tape halting problem,
  R is the some-tape halting problem:
  - $L_Q = \{x_j \mid T_j \text{ halts on } \square\}$
  - $L_R = \{x_j \mid T_j \text{ halts on some tape}\}$
  - $f(x_j)$ = description of the machine such that erases its tape, then x_f on its tape and then behaves like T_Q.
  - $Q \leq R$
  - R solvable $\Rightarrow$ Q solvable
  - Q not solvable $\Rightarrow$ R not solvable

Pairing Functions

- Often we need to be able to talk about TMs implementing functions of more than one argument.
- An easy way to handle this is to require that the arguments be adjacent on the input tape, separated by a special symbol, say # (blank would work too, but could cause confusion).



Pairing Functions

- We will sometimes use <x, y> to mean a single input representing the pair of inputs.
- It is required that we be able to compute <x, y> from x and y and also be able to compute both x and y from <x, y>.
- Therefore <> as a function must be 1-1, and ideally onto.

The “Recognizes” language

- **Recognizes** = {<T_i, x_j> | T_i recognizes x_j}
- Both of the following problems are reducible to the **Recognizes** language:
  - The halting problem.
  - The blank-tape halting problem.
- Therefore the **Recognizes** language is not recursive.

The “Diverges” language

- **Diverges** = {<T_i, x_j> | T_i diverges on x_j}
- Both of the following problems are reducible to the **Diverges** language:
  - {x_j | T_j diverges on x_j} (= “**SelfDiverge**”)
  - {x_j | T_j diverges on □}.
- **SelfDiverge** was proved to be not recursively-enumerable.
- Therefore **Diverges** is not recursively enumerable.

The “Accepts” language

- **Accepts** = {<T_i, x_j> | T_i accepts on x_j}
- **Accepts *is*** recursively enumerable.
- **Accepts** is the language accepted by any Universal Turing Machine.
- **Accepts** is not recursive; its complement is not recursively enumerable.

Table of "Solvabilities"

Language	Recursive?	R.E.?	Reason
SelfDiverges	No	No	diagonalization
SelfAccepts	No	Yes	red to Accepts
Accepts	No	Yes	UTM, comp
Diverges	No	No	red from SelfDiverges
Halts	No	Yes	red from Accepts
AcceptsEmpty			
AcceptsSome			
AcceptsAll			
AcceptsNone			

Rice's Theorem

- Could be considered a "meta-theorem" due to its sweeping character.

- Applies to ***functional*** questions: languages and functions in the abstract; not to TM ***structural*** questions.

Functional vs. Structural

- Structural questions:
  - Is there an algorithm for determining whether a TM has **more than 500 control states**?
  - Is there an algorithm for determining whether a TM ever uses **more than 500 cells of tape on a given input**?
- Functional questions:
  - Is there an algorithm for determining whether a TM accepts **the empty language**?
  - Is there an algorithm for determining whether a TM accepts **a finite language**?
  - Is there an algorithm for determining whether a TM accepts **a regular language**?

Encoding Functional Properties as Languages

- Consider some functional property P of Turing machines.
- We can define a property precisely as the language of descriptions (or indices) of TMs that have the property:
- Define $L_P = \{x_j \mid L(T_j) \text{ has property } P\}$.
- Example: $L_{AcceptsEmpty} = \{x_j \mid \square \in L(T_j)\}$.

Perspective on Functional Properties

- If a language is accepted by some TM, then it is accepted by infinitely many TMs. (Why?)
- For a given functional property, we must get the same categorization of an input, regardless of which of the many **equivalent** TM's is used.
- So a functional property truly is a property of a language or function, rather than just a particular **implementation** of the language or function.
- However, the only ways we know (so far) to **represent** some languages constructively is by **machines** or **grammars**.

Trivial Functional Properties

- A functional property is called “trivial” if its language is either ∅ or Σ*.
- In other words, a trivial property holds for no Turing machine or it holds for all of them.
- A **non-trivial property**, then, holds for some machines, but not all.

Rice's Theorem

- Any non-trivial functional property of Turing machines is not recursive.
- Put another way:

  No non-trivial functional property of Turing machines is recursive.
- In other words, there is no TM that will determine whether an arbitrary TM:
  - Accepts □.
  - Accepts a finite set.
  - Accepts an empty set.
  - Accepts a regular set.
  - etc.

Proof of Rice's Theorem (1 of 3)

- Suppose **P** is a non-trivial property.
- The empty language ∅ either has property P or does not.
- Proceed assuming that ∅ **does not have property P**. **This assumption is critical.**
- If instead ∅ does have property P, use the following argument on the **complementary** property, which ∅ does not have.
- Let $\mathbf{L_P}$ be the language of descriptions of TMs that have property P.

Proof of Rice's Theorem (2 of 3)

- The plan is to reduce **a non-recursive language L_u to L_P,** which will imply that **L_P** is also non-recursive**.**
- **L_u** is the language $\{<x_i, x_j> \mid T_i \text{ halts on } x_j\}$, essentially the language accepted by the universal TM (**L_u** is R.E. but not recursive).
- Let **L** be an *arbitrary* language with property P (which must exist, because P is non-trivial).
- Let **M_L** be a machine accepting L.
- The reduction works as follows: For any $<x_i, x_j>$, construct a machine $M'(x_i, x_j)$ with the specifications on the following page:

Proof of Rice's Theorem (3 of 3)

- M'(x_i, x_j): with input w, temporarily set aside w and simulate T_i on x_j.
- If the above simulation does not terminate, then L(M') = ∅, in which case M' **does not** have property P.
- If the above simulation terminates, then simulate $\mathbf{M_L}$ on the original input w. If and when $\mathbf{M_L}$ terminates, M' accepts w iff $\mathbf{M_L}$ accepts w. In this case, M' **does** have property P, because $\mathbf{M_L}$ was selected to have it.
- So $\mathbf{L_u}$ has been reduced to $\mathbf{L_P}$: T_i halts on x_j iff M' does not have property P.

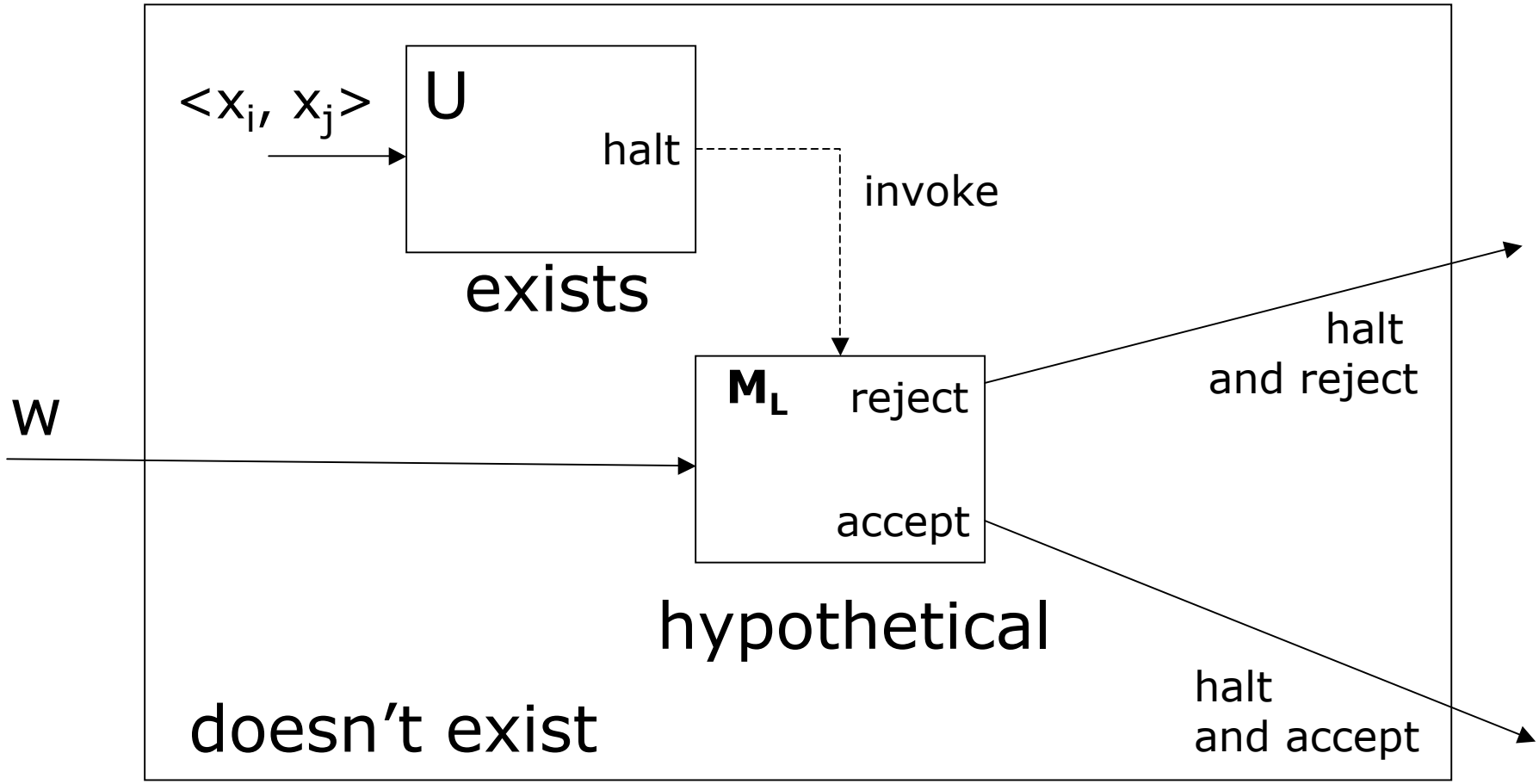
Practice expressing the proof as a reduction

- We want a function f:Tapes $\rightarrow$ Tapes such that

$$\langle x_i, x_j \rangle \in \mathbf{L_u} \text{ iff } f(\langle x_i, x_j \rangle) \in \mathbf{L_P}$$

- f is the function that with input $\langle x_i, x_j \rangle$ constructs machine M'(x_i, x_j) described in the previous slide (M' with input w …).

The Machine $M'(x_i, x_j)$ constructed by f:
M' accepts w iff T_i accepts x_j



Reflection on Rice's Theorem

- We were insistent on P being a **functional** property.
- Where did this assumption get used in the proof?
- Could we have used a different starting point for reduction, e.g. the blank-tape halting problem?

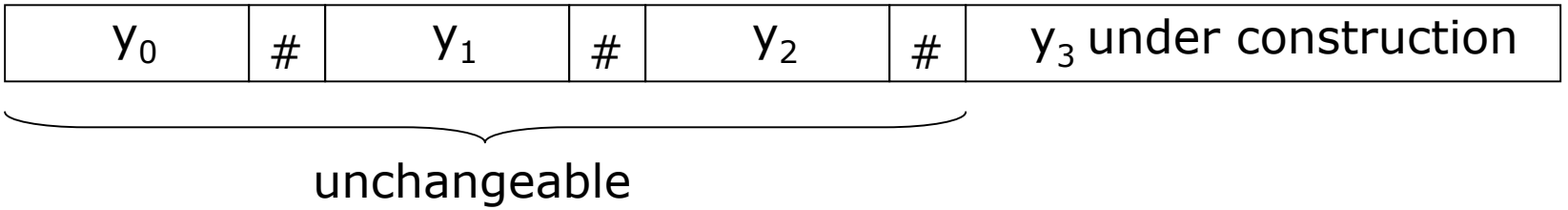
Note on the Definition of R.E.

- Various equivalent definitions are possible for a language L being R.E.
- The **original** definition given is that L is the **domain** of a **partial** recursive function (i.e. that of the TM that accepts it).
- The most suggestive definition is that L is either $\varnothing$ or is the **range** of a **total** recursive function (the function that enumerates it).
- The Martin book uses another style of enumeration, which I find less elegant than the preceding.

Martin's (and others) version of Enumeration

- To enumerate an infinite set, a TM must continue forever, writing elements of the set on its tape one after the other, separated by markers.

- When a marker is written, the TM is not allowed to go back and change any string to the left of the marker.



The Uniform Halting Problem

- Consider the set
  HaltsAll = $\{x_i \mid T_i$ halts on *all* inputs$\}$
- By now we can surmise that this set is **not recursive**.
- But is it even recursively-enumerable?

HaltsAll is Not R.E.

- If HaltsAll = $\{x_i \mid T_i$ halts on *all* inputs$\}$ were R.E. then there would be an always-halting Turing machine T* that enumerates it.
- That is $\{T^*(x_0), T^*(x_1), T^*(x_2), \ldots\}$ = HaltsAll.
- Now define

$$T^{**}(x_j) = \begin{cases} \square \text{ if } T^*(x_j) \neq \square \\ 1 \text{ if } T^*(x_j) = \square \end{cases}$$

(1 can be any string not $\square$)

- Since T* always halts, so does T**. But the description of T** can't be HaltsAll, for if it is in the list as $T^*(x_k)$, then by the definition above, $T^{**}(x_j)$ gives a value other than $T^*(x_k)$, which is absurd.