Uncomputability

Robert M. Keller
Harvey Mudd College
April 2013

Generality of Decision Problems

- Decision problems are about existence of algorithms to test membership in a language that is the subset of all problem encodings.
- Because Turing machines accept languages, it is possible to ask the question about whether a TM exists that does a certain **algorithm**.
- The fundamental question is whether there is an **algorithm**. We are not generally interested in answering the question for just one specific case.

Decidability

- A problem is called **decidable** if there is an **algorithm** that will determine a **yes/no** answer for every instance of the problem.
- Equivalently, is there an algorithm that determines membership in the **language** of instances for which the answer is yes.
- The language is said to be a **decidable language**.

There Are Undecidable Languages

- Every decidable language corresponds to a Turing machine that always halts and gives a yes or no answer, per the Church-Turing thesis.
- The set of **encodings** is countably infinite.
- The set of **languages** is equivalent to the power set of Σ^*, which is **uncountable**.
- So there are languages L for which there is no Turing machine that accepts L.

Meaning of “Enumerate”

- A set is **enumerable** (or “denumerable”) iff it can be put into one-to-one correspondence with the natural numbers or a subset thereof.

- Examples:
  - The set of all **pairs** of natural numbers **is** enumerable.
  - The set of all **subsets** of natural numbers **is not** enumerable.
  - The set of all **finite subsets** of natural numbers **is** enumerable.

Cantor's Diagonal Argument

- Suppose the set of all languages can be enumerated.

Enumeration of Σ^*

$x_0 \quad x_1 \quad x_2 \quad x_3 \ . \ . \ .$

Supposed enumeration of ***all*** languages

L_0

L_1

L_2

L_3

.
.
.

Have a 1 in row i column j if $x_j \in L_i$. Have a 0 otherwise.

Diagonal Argument

- Have a 1 in row i column j if $x_j \in L_i$. Have a 0 otherwise.

Enumeration of Σ^*

Supposed enumeration of ***all*** languages

	x_0	x_1	x_2	x_3	. . .		
L_0	0	1	0	1	0	1	
L_1	1	0	1	0	1	0	
L_2	0	0	1	1	0	1	…
L_3							
⋮							

Diagonal Argument

- Where is the flattened flipped diagonal in the enumeration of languages?

Enumeration of Σ^*

	x_0	x_1	x_2	x_3	. . .		
L_0	0	1	0	1	0	1	
L_1	1	0	1	0	1	0	
L_2	0	0	1	1	0	1	...
L_3							
. . .							

Supposed enumeration of ***all*** languages

flattened flipped diagonal

1 1 0 ...

Diagonal Argument

- The flattened flipped diagonal can't be anywhere in the enumeration. It disagrees with each language in at least one bit.

Supposed enumeration of ***all*** languages

Busted!

	x_0	x_1	x_2	x_3	. . .		
L_0	0	1	0	1	0	1	
L_1	1	0	1	0	1	0	
L_2	0	0	1	1	0	1	…
L_3							
⋮							

flattened flipped diagonal

1 1 0 …

Diagonal argument in symbols

- From the supposed enumeration of all languages L_0 , L_1 , L_2 , … and the known enumeration of Σ^*, x_0 , x_1 , x_2 , …, we constructed a new language not in the enumeration after all:

  $$D = \{x_i \mid x_i \notin L_i\}$$

  For, if D were in the enumeration it would be L_d for some d, thus:

  $$L_d = \{x_i \mid x_i \notin L_i\}$$

  which is absurd,
  because it implies x_d is in L_d iff it isn't in L_d.

Summary so far, and direction

- The set of all languages cannot be enumerated. There are "too many".
- The set of all Turing machines *can* be enumerated. Each one can be encoded into a *single element* of Σ^* (the machine's description).
- Thus there must be *some* languages (many, in fact) for which there is no Turing machine.
- That is, *some* languages are not effectively computable.

Can't we simply add D to the list?

- We could add D to the list, in principle.
- This would yield a new list.
- The diagonal argument can then be repeated on the new list

and so on, until we get tired.

A Specific Undecidable Language

- <M> is the encoding of machine TM M, for every M.
- Consider language K = {<M> | M does **not** accept <M>}.
- **There is no Turing machine that accepts K, i.e. K is undecidable.**
- **Proof**: **Suppose some machine, N, *does* accept K**.
- Ask the question whether <N> ∈ K or not.
- Then the following are equivalent:
  - <N> ∈ K ← iff, by definition of K
  - N does not accept <N>
  - <N> ∉ K ← iff, by definition of N as the machine for K
- Obviously, we have a **contradiction**.

K as the Mother Lode

- K is the “fountain” from which all other undecidable languages spring forth.

Sipser's Notation

- Let $\mathscr{F}$ be a family of machines (such as DFA, PDA, TM, …) or grammars (such as CFG, CSG, …).
- $A_{\mathscr{F}}$ is the language consisting of an encoding of:
  - a member M of $\mathscr{F}$
  - a string x over the alphabet of M

  such that M accepts x (or generates x in the case of a grammar).

Sipser’s Notation Example

- A_{DFA} =
  $\{<M, x> \mid M \text{ is a DFA and } x \in L(M)\}$

- A_{TM} =
  $\{<M, x> \mid M \text{ is a TM and } x \in L(M)\}$

- A_{CFG} =
  $\{<G, x> \mid G \text{ is a CFG and } x \in L(G)\}$

etc.

Some Decidable Languages

Why are these decidable?

- A_{DFA}
- A_{PDA}
- A_{CFG}
- A_{REGEX}

Accept vs. Recognize

- A TM **accepts** (or “**decides**”) a language iff it always halts, and indicates whether its original input is in the language or not.

- A TM M **recognizes** a language L iff for each input x:
  - If $x \in L$ then M will halt and indicate acceptance.
  - If $x \notin L$ then M does **not** indicate acceptance.
    (M **may halt** and reject, or it **may diverge**, i.e. go on forever without halting.)

Notation Refinement

- Let M be a Turing machine.
- By L(M) we will mean the language **recognized** by M.
- **If** M *always halts*, L(M) is **also** the language **accepted** by M.

Recognizability vs. Decidability

- A language is (Turing-) **recognizable** iff it is recognized by some TM.
- A language is (Turing-) **decidable** iff it is accepted by some TM.
- Decidable $\rightarrow$ recognizable.
- But is the converse true?

K is not even recognizable.

- We claim the language K defined as
  K = {<M> | <M> encodes a TM and <M> ∉ L(M)} (1)
  is not recognizable by a Turing Machine.

- **Proof:** Suppose K were recognized by some TM, say N,
  In other words, L(N) = K. (2)

  Then explore whether <N> ∈ K.

- <N> ∈ K iff <N> ∉ L(N) by the definition of K (1).
  and
- <N> ∈ K iff <N> ∈ L(N) by the definition of N (2).

  So our supposition is invalid. K is not recognizable.

The **complement** of K **is** recognizable.

- The **complement** $K^c = \Sigma^* - K$ is recognizable.
- $K^c = \{<M> \mid <M> \in L(M)\}$

Proof:

- Adapt a UTM U such that, with input <M>, it will simulate M acting on <M>.

- If and when U halts, if M accepts, then U will accept.

Complementarity Theorem

- A language L is decidable iff both it and its complement are recognizable.

Proof:

- (→) If L is decidable, it is recognizable. Furthermore, its complement is decidable by the same machine with accept and reject states interchanged. Hence the complement is decidable.

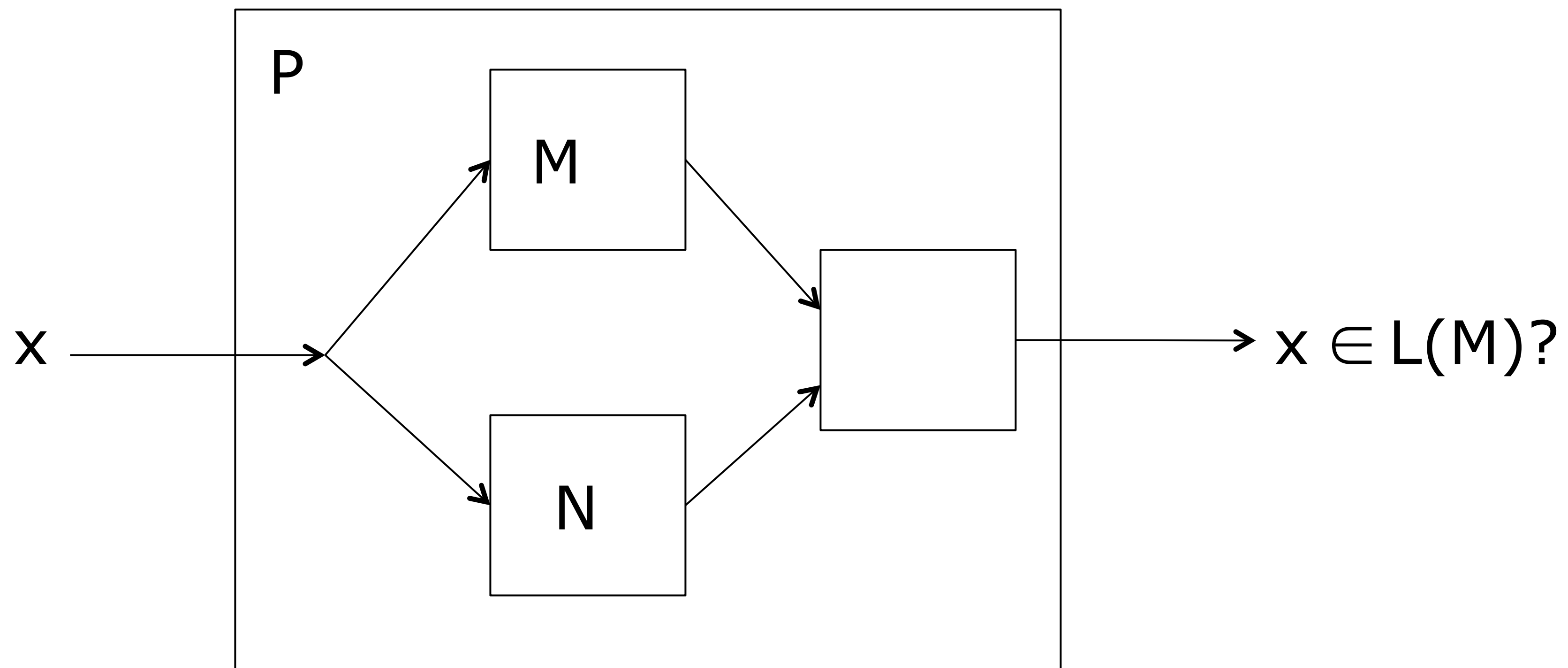
Complementarity Theorem

- (←) Suppose L and its complement are recognizable. Let M recognize L and
  N recognize its complement.

- Construct a deterministic TM P that decides L, as follows: For a given input, P alternates simulating a step of M on x with simulating N on x. If M accepts x, then P accepts x. If N accepts x, then P rejects x.

- One of M or N will accept x eventually (by definition of recognition), so P will decide x. Not both of M or N will accept x, because one recognizes the complement of what the other recognizes.

Structure of P in the Proof of the Complementarity Theorem



Note: M and N must run in parallel or interleaved, not sequentially.

co-recognizability

- A language is called **co-recognizable** iff its complement is recognizable.
- Example: K is co-recognizable, but not recognizable.
- Thus, from the Complementarity Theorem:
  - L decidable iff (L recognizable *and* L co-recognizable)
  - (L recognizable but not co-recognizable) implies L not decidable
  - (L co-recognizable but not recognizable) implies L not decidable
- There are languages that are **neither** recognizable nor co-recognizable, as we shall see.

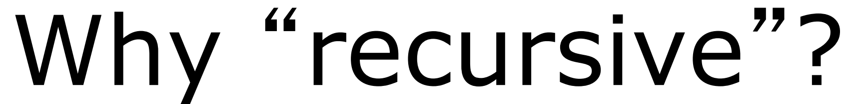
More Standard Terminology

- Recall: A language is (Turing-) **decidable** iff it is accepted by some TM.

  Much literature uses the term **“recursive”** *as a synonym for* **“decidable”.**

- Recall: A language is (Turing-) **recognizable** iff it is recognized by some TM:

  Most literature uses the term
  “recursively-enumerable” (r.e. or RE)
  instead of **“recognizable.**
  The term **“semi-decidable”** is also descriptive for this case.



Why “recursive”?

- It has to do with the Gödel/Kleene notion of **recursive functions** as being the equivalent of the effectively computable functions.

- A recursive language has a recursive **characteristic function**.

- Note: Do not read into this anything about the function calling itself, etc. which deals with the way in which the function can be *expressed*.

Why “recursively enumerable”?

- It means there is a recursive function that **enumerates** the language. We’ll soon see that this is equivalent to the language being recognizable by a Turing machine.

Computable Function
(vs. decidable language)

A ***function*** f: $\Sigma^* \rightarrow \Sigma^*$ is ***computable*** if there is a TM such that if M is started with x on its tape, M will eventually halt with f(x) on its tape.

Computable functions are also called **recursive functions** in the computability literature.

Partial Functions

A ***partial function*** is like a function, except that it can be **undefined** for some or all values of arguments.

So it has the **uniqueness** property of a function: $x = y$ implies $f(x) = f(y)$, but may lack the **definedness** property, that $f(x)$ is defined for all x in the domain.

The **same notation** is usually used for function and partial function, relying on context to resolve the distinction.

Sometimes we write $f(x) = \perp$ to designate "$f(x)$ is undefined". But be aware that

$\perp$ **is no ordinary value**.

Computable Partial Function

A ***partial function*** $f: \Sigma^* \rightarrow \Sigma^*$ is ***computable*** if there is a TM such that if M is started with x on its tape, M will **eventually halt** with f(x) on its tape,
or **diverge** (never halt).

Common notation:

$f(x)\downarrow$ means f(x) is defined.

$f(x)\uparrow$ means f(x) diverges ($f(x) = \perp$).

In general, there is **no computable test for $f(x) = \perp$**.
It is just a notational convenience.

Computable partial functions are also called ***partial recursive functions*** in the literature.

Characteristic Functions

- The **characteristic function** of a language $L \subseteq \Sigma^*$ is a function $ch_L: \Sigma^* \rightarrow \{0, 1\}$ defined by

  $\forall x \in \Sigma^*$ $\quad ch_L(x) = 1$ if $x \in L$
  $\qquad\qquad\qquad\qquad 0$ otherwise

- Every language has one, and every function of this form determines a language.

Observations

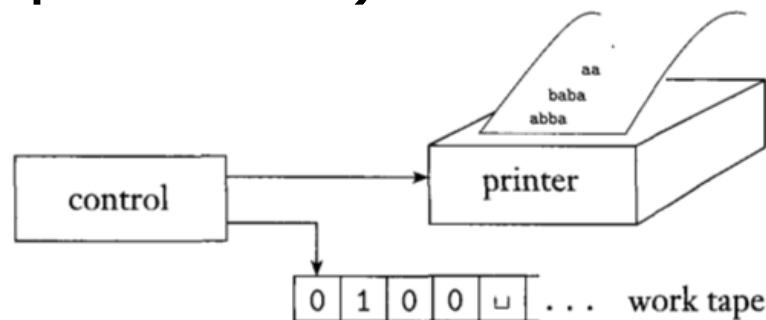
- A language is **decidable** iff it has a computable characteristic function.
- A language is **recognizable** iff it has a computable "partial characteristic function":

  $\forall x \in \Sigma^* \; pch_L(x) = 1$ if $x \in L$

  undefined otherwise

Enumerability

- The Sipser definition of an Enumerator:
- A Turing machine with a metaphorical "printer" that prints out all the elements of a language over time (possibly with repetitions).



This could be replaced with a linear "output tape", or *interleaved* with the squares of the work tape.

The Language Enumerated

- An enumerator **enumerates** a language if, running autonomously, *each* element of the language will get printed at some step ("in finite time").

- If the language enumerated is **infinite**, then the enumerator can never halt.

- If the language enumerated is **finite**, the enumerator may or may not halt (it could print the same element multiple times).

An Alternate Definition of **Enumerator**

- A Turing machine that interprets its input as an encoded natural number <i>, and

- for each value of i, the machine **halts** with an element of Σ^* in a designated area of its tape.
  This element is declared to be the **i^{th} element** x_i of a language L.

- The language enumerated is $L = \{x_0, x_1, x_2, \ldots\}$ the set of strings produced by halting computations.

- (If L is finite, there will be repetitions in the sequence.)

Why is the Alternate Definition Equivalent to Sipser's?

- Given a Sipser enumerator S, we can construct an Alternate enumerator A:
  - With input <i>, where $i \geq 0$, A simulates S up until i+1 strings have been "printed". Then A outputs the last string printed as x_i.

- Given an alternate enumerator A, we can construct a Sipser enumerator S: S simulates calls A on different arguments: A(<0>), A (<1>), …, to produce the elements of the set being enumerated.

Dovetailing

- If Sipser enumerator were to call A(<0>), then A (<1>), then A(<2>), … and wait until the previous computation succeeded before going on, there could be a problem: A(i) might **diverge**.
- To avoid this issue, S simulates:
  - 0 steps of A(<0>),
  - 1 step of A(<0>) then 1 step of A(<1>),

    . . .
  - i steps of A(<0>), A(<1>), …., A(<i>)

    . . .
- This type of scheme is called "**dovetailing**".

- Whenever one of the A(<j>) halts, the value on the tape is the next string in the enumeration.

Acceptance Languages

- Sipser's notation includes languages of the form A_F where F is some formalism, such as DFA, Regular Expressions, Grammars, etc.

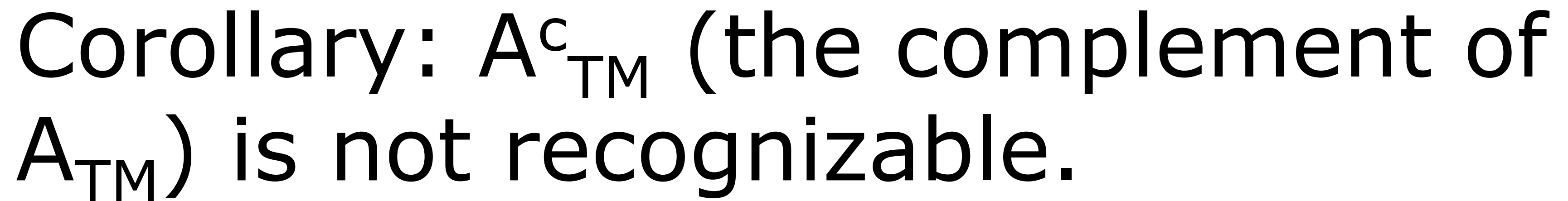
- Example: A_{DFA} is the language of encodings of DFAs with a string the DFA accepts.
  <B, w> is in the language iff:
  - <B> encodes a DFA
  - <w> encodes an input to the DFA
  - B accepts w
- If <B> or <w> are malformed in some way, then <B, w> is simply not in the language.

Why A_{TM} is not decidable.

- Claim: A_{TM} decidable → K decidable.
- But we know K is not decidable.
- If A_{TM} were decidable, we could create an algorithm for deciding K as well:
  - With input <M>, construct <M, <M>> (the description of M together with its own description) and pass it to the algorithm for A_{TM}. <M, <M>> $\notin A_{TM}$ iff M does not accept <M> iff <M> $\in$ K.
- Thus an algorithm for A_{TM} can be used to construct an algorithm for K, which we previously showed to be impossible.

Note on Sipser's Proof of A_{TM} Undecidable

- Sipser does not first identify K.
- Instead, he constructs D which is similar to K from A_{TM} in the proof.
- $$D(<M>) = \begin{cases} \text{accept if M does not accept <M>} \\ \text{reject if M accepts <M>} \end{cases}$$
- He then observes that D(<D>) yields a contradiction, and notes that this is a diagonalization.



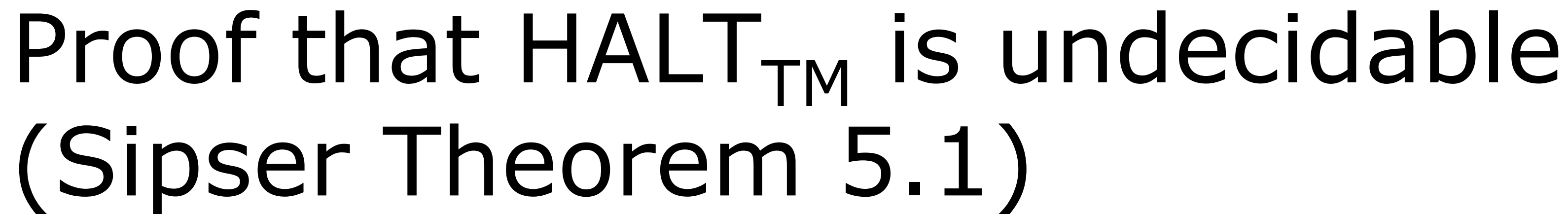
Corollary: A^c_{TM} (the complement of A_{TM}) is not recognizable.

- Why?
- A^c_{TM} = {<M, x> | M diverges on x}
- If A^c_{TM} were recognizable, so would K be recognizable:
  K = {<M> | M diverges on <M>}

The "Halting Problem"

- $HALT_{TM}$ =
  {<M, w> | M is a TM and M halts on input w}

- $HALT_{TM}$ is undecidable.



Proof that $HALT_{TM}$ is undecidable (Sipser Theorem 5.1)

- Suppose $HALT_{TM}$ were decidable. That means there is an algorithm for it. We'll use that algorithm to **construct an algorithm for A_{TM}**, which we previously showed to be impossible.

- Algorithm: With input <M, w>, first check if <M, w> $\in HALT_{TM}$.
  - **If not**, then **reject**, as M cannot accept w in this case.
  - **If so**, then simulate M on w (e.g. using a universal TM), and accept iff M accepts w.

Summary

- An algorithm for $HALT_{TM}$ could be used to construct an algorithm for A_{TM}

  thus

- $HALT_{TM}$ decidable → A_{TM} decidable

  equivalently (**contrapositive**):

- A_{TM} **un**decidable → $HALT_{TM}$ **un**decidable

Terminology

- On the previous slide, we have

  reduced A_{TM} to the **halting problem**,

  in the sense that **if** the halting problem were solvable, so would A_{TM} be.

  But A_{TM} was already established to be **unsolvable**.

- Note: It is **not correct** to say the **opposite**, that we have reduced the halting problem to A_{TM} (although this can also be done as well).

Notation for Reduction

- If a problem (language) A can be reduced to a problem (language) B we write:

  $A \leq B$

  suggesting that
  B is "at least as difficult" as A.

  So if A is unsolvable, so must be B.

Reduction in Academic Life

A physicist and a mathematician setting in a faculty lounge. Suddenly, the coffee machine catches on fire. The physicist grabs a bucket and leaps towards the sink, fills the bucket with water and puts out the fire.

The second day, the same two sit in the same lounge. Again, the coffee machine catches on fire. This time, the mathematician stands up, gets a bucket, hands the bucket to the physicist, thus reducing the problem to a previously solved one.

Reduction in Tropical Life

A mathematician and an physicist are on desert island and develop hunger. They find two palm trees with one coconut each. The physicist shins up the first tree, gets the coconut, eats.

The mathematician shins up the second tree, gets the coconut, climbs the first tree and puts it there. “Now we’ve reduced it to a problem we know how to solve.”

Proved So Far

- $K \leq A_{TM}$
- $A_{TM} \leq HALT_{TM}$
- $K \leq A^c_{TM}$

Transitivity of Reduction

- If $A \leq B$ and $B \leq C$, then $A \leq C$.
- Example:

  $K \leq A_{TM}$ and $A_{TM} \leq HALT_{TM}$,

  therefore $K \leq HALT_{TM}$.

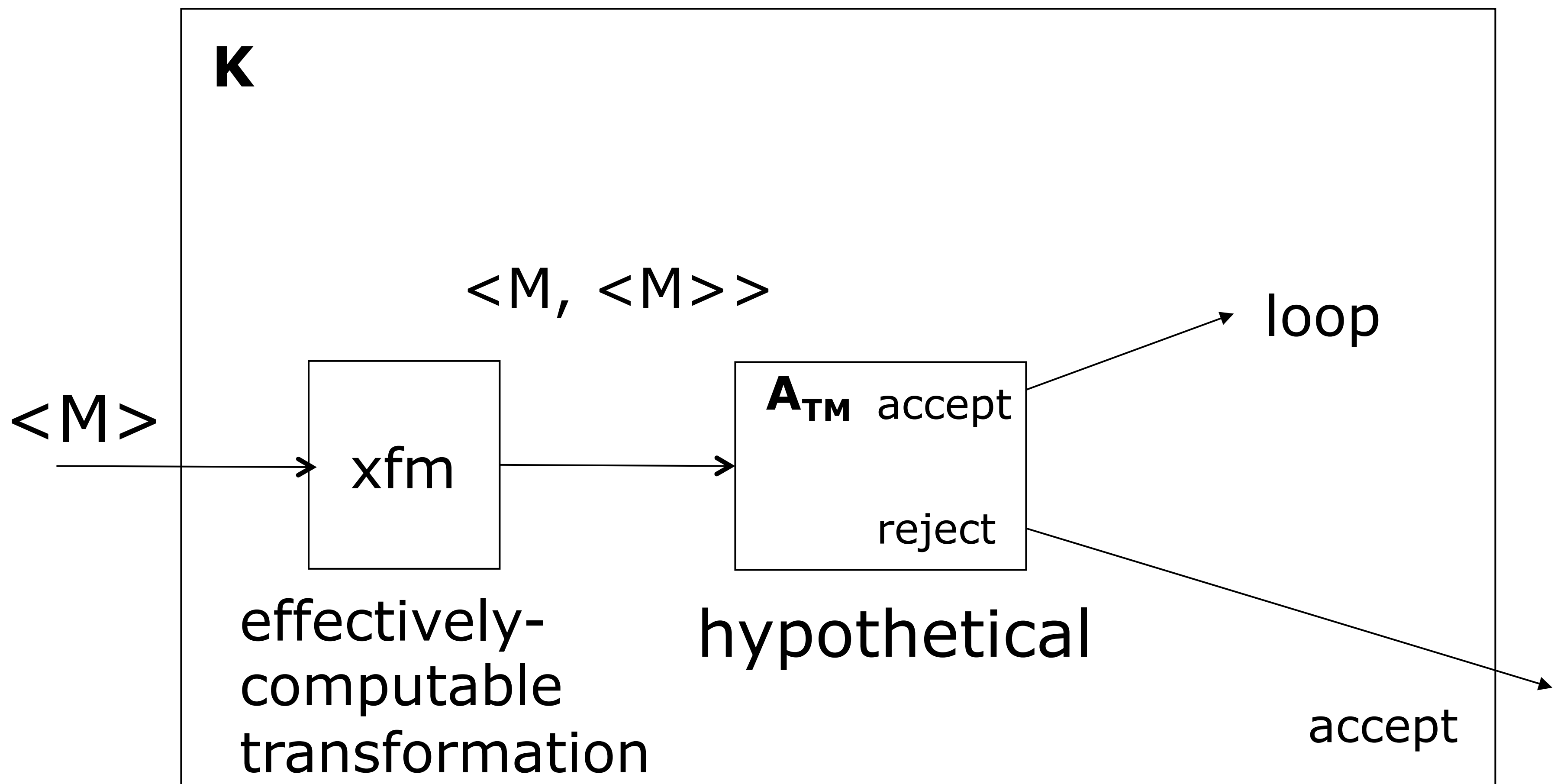
Reduction Diagrams

- It is sometimes helpful to depict reduction arguments in diagrammatic form, to permit comparison of different reductions for example.

- The general form has:
  - An **outer box**, representing the problem **known** to be undecidable.
  - An **inner box**, representing the problem we are trying to show undecidable.
  - A input transformation step, representing the creation of a new machine description from an existing one.
  - An output transformation step, representing how the output of the inner box is interpreted.

Example: K $\leq$ A_{TM}

(If were A_{TM} computable, K would be also.)

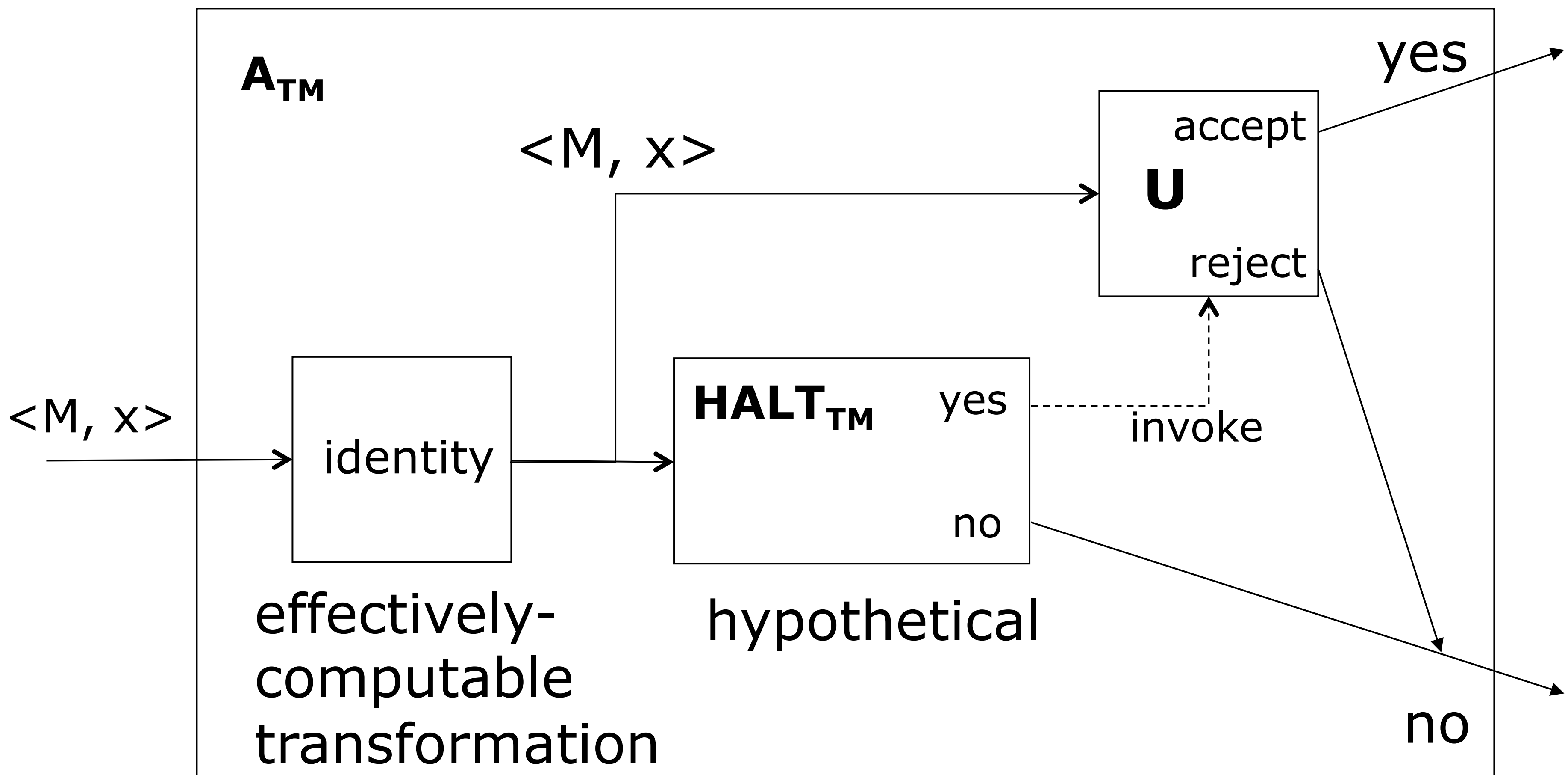


Important Notes

- Reductions **analyze** the **source code** of the machine <M>, using it to create source code of a new machine <M'>.
- **Analyzing** the machine, might entail running it, or it might not.
- **Running** the code tends to be the exception rather than the norm.
- The $A_{TM} \leq HALT_{TM}$ example runs the code.

déjà vu Example: $A_{TM} \leq HALT_{TM}$

(If were $HALT_{TM}$ computable, A_{TM} would be also.)



- This is the “other direction”.
- If A_{TM} were decidable, then so $HALT_{TM}$ would be.
- This problem is simpler to show.
- The reduction diagram is on the next page.
- But what goes in the box marked xfm?

Example: $HALT_{TM} \leq A_{TM}$

(If were $HALT_{TM}$ computable, A_{TM} would be also.)

$HALT_{TM}$

<M, x>

xfm?

<M', x>

A_{TM}

yes

no

hypothetical

yes

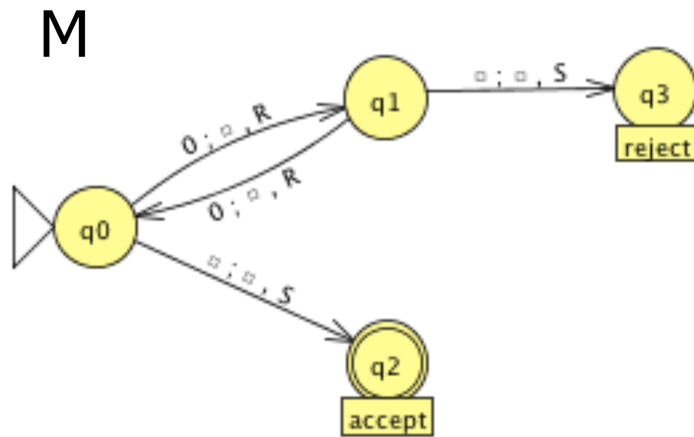
no

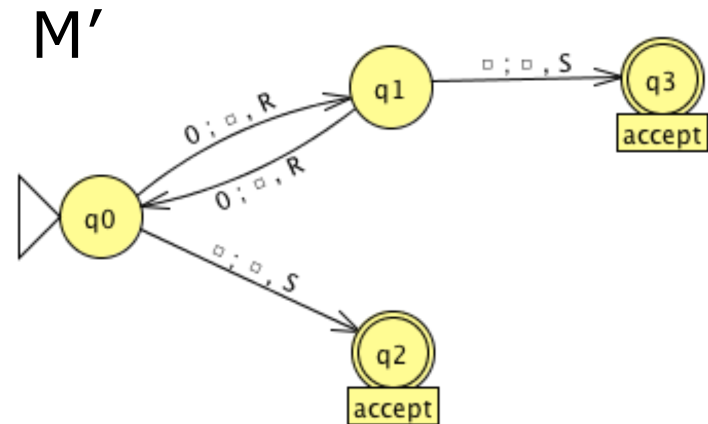
What transformation xfm?

- We want
  M halts on x
  iff
  M′ accepts x.

- So xfm has to transform M to M′. Any *halting* state of M is made into an *accepting* state of M′.

Example of Source xfm



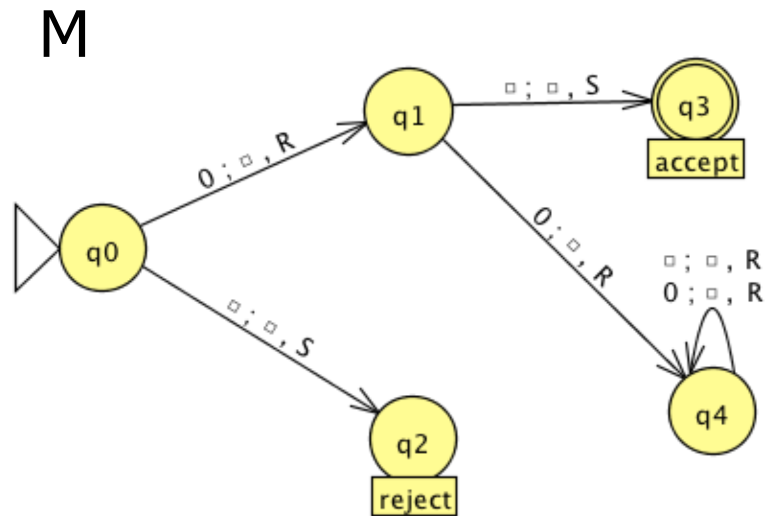


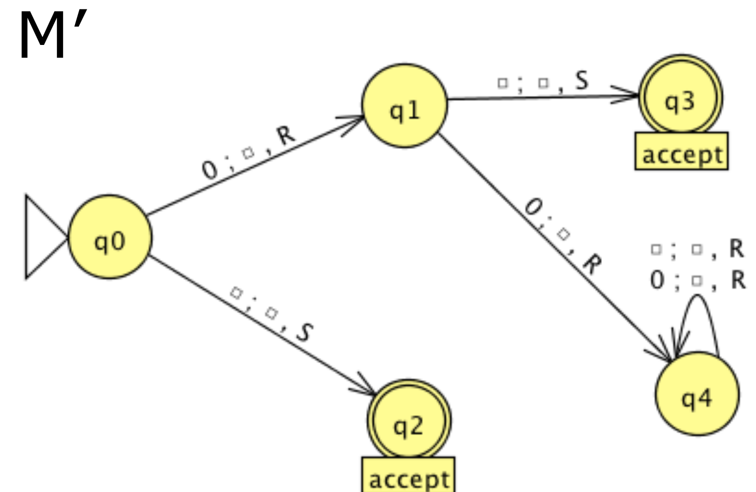
Any state from which there is no transition is rejecting if it is not accepting.

Input	Result
	Accept
0	Reject
00	Accept
000	Reject
0000	Accept
00000	Reject
000000	Accept

Input	Result
	Accept
0	Accept
00	Accept
000	Accept
0000	Accept
00000	Accept
000000	Accept

Example of Source xfm with Divergence





Input	Result
	Reject
0	Accept
00	Cancelled
000	Cancelled
0000	Cancelled
00000	Cancelled
000000	Cancelled

Input	Result
	Accept
0	Accept
00	Cancelled
000	Cancelled
0000	Cancelled
00000	Cancelled
000000	Cancelled

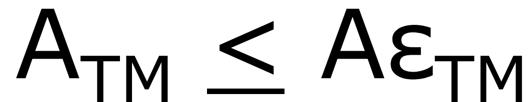
Cancelled suggests divergence

Example: Empty-Tape Halting Problem

- Is there an algorithm that will determine whether an arbitrary TM will accept an empty (i.e. all-blank) tape ε?

- This seems “easier” than A_{TM} (because there is one less variable: the tape) but it’s actually just as hard.

- Call this problem $A\varepsilon_{TM}$.

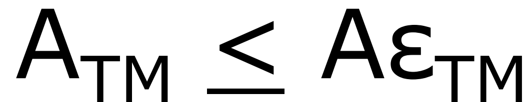


$A_{TM} \leq A\varepsilon_{TM}$

- Suppose we had an algorithm that would decide $A\varepsilon_{TM}$. We can use it to decide A_{TM}, as follows.

- Let <M, x> be an instance of the A_{TM} problem.

- From <M, x> construct M’: M’ writes x on its tape, then behaves just as M would have on the result.



- From <M, x> construct M’: M’ first writes x on its tape, then behaves just as M would have on the result. (So M’ has x “wired into” its program.)

- Therefore M’ on an empty tape will give the same result as M would on x.

- So determining whether M’ accepts ε is the same as determining whether M accepts x.

$A_{TM} \leq A\varepsilon_{TM}$ Diagram

A_{TM}

yes

<M, x>

xfm

<M’>

$A\varepsilon_{TM}$

yes

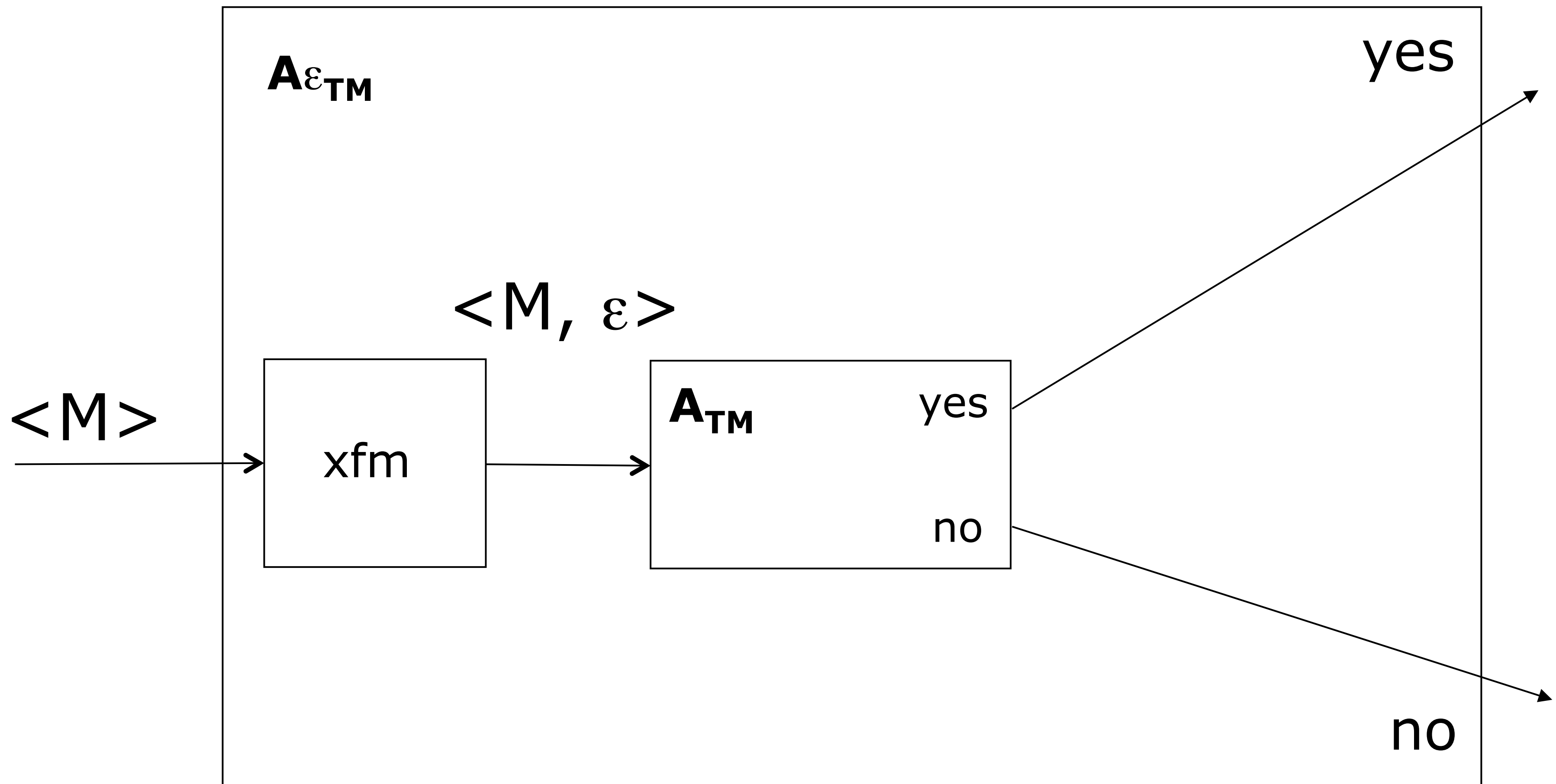
no

no

- This is true because ε is a special case of a general input.

- (Neither problem is decidable.)

$A\varepsilon_{TM} \leq A_{TM}$ Diagram



Example: Fixed-Tape Halting Problem

- Is there an algorithm that will determine whether an arbitrary TM will accept a fixed tape z?
  (such as z = 1101101)

- We showed this for the special case
  $z = \varepsilon$.

- Call this problem Az_{TM}.

Az_{TM} is undecidable for any fixed z.

- Let <M, x> be an instance of A_{TM}.
- We show how to convert <M, x> to an equivalent instance <M'> of Az_{TM}.
  - M' with arbitrary input y:
    - Compare y to z. If y = z, then behave just as M would have on x.
    - But if y ≠ z, diverge.
- So M' accepts y iff M accepts x.

$A_{TM} \leq Az_{TM}$ Diagram

Emptiness Problems

- E_{DFA} = {<B> | B is a DFA accepting no strings}
- E_{REX} = {<R> | R is a regular expression and L(R) = ∅}
- E_{CFG} = {<G> | G is a context-free grammar and L(G) = ∅}.
- E_{CSG} = {<G> | G is a context-sensitive grammar and L(G) = ∅}.
- E_{TM} = {<M> | M is a Turing machine and L(M) = ∅}
- Which of these are decidable? (Not necessarily obvious).

E_{TM} is not decidable

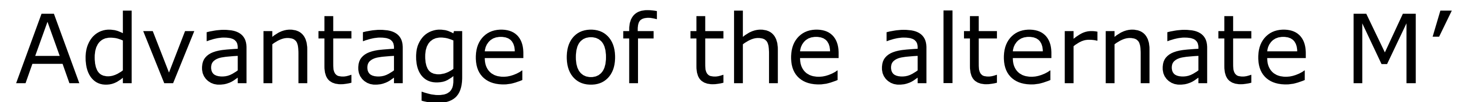
- For problems that follow the usual pattern, we can just give the transformation.
- To reduce A_{TM} to E_{TM}, the transformation is: Given <M, x> as an instance of A_{TM}, construct instance M’ of E_{TM}.

- M’ on input w behaves as follows:
  - If w = x, then behave as M on x.
  - If w ≠ x, then diverge.

- Thus M’ accepts *some* string (namely x, and L(M)≠ ∅) iff M accepts x.

Alternate Transformation for $A_{TM} \leq Az_{TM}$, etc.

- This handles a number of problems with a single construction.
- Given <M, x>, an instance of , construct M':
  - M' first **erases** whatever is on its tape.
  - M' then writes x on its tape.
  - Then M' behaves as M would.
- So M' accepts any and all tapes, including z, iff M accepts x.



Advantage of the alternate M′

- M′ shows that $A_{TM} \leq Az_{TM}$.
- But it can also be used in other reductions:
  - Emptiness: $A_{TM} \leq E_{TM}$ Does M′ accept anything?
  - Fullness: $A_{TM} \leq ALL_{TM}$ Does M′ accept everything?
  - Infinite: $A_{TM} \leq Infinite_{TM}$ Does M′ accept an infinite set?
- Where can't it be used?
  - Regularity: $A_{TM} \leq Regularity_{TM}$
    Does M′ accept an regular language?
  - Why not? Because both $\emptyset$ and Σ^* are regular.

Regular_{TM} is not decidable (Sipser proof)

- Regular_{TM} = {<M> | L(M) is regular}

- Given <M, x> as an instance of A_{TM}, construct instance M′ of Regular_{TM}.

- M′ on input w behaves as follows:
  - If w is of the form 0^n1^n, then accept w.
  - Otherwise, behave as M on x, and if M(x) accepts, accept w.

- Thus M′ accepts Σ* (regular) if M accepts x, and M′ recognizes $\{0^n1^n \mid n \in N\}$ (non-regular) if M does *not* accept x.

Rice’s Theorem

- One pattern in proving languages undecidable has been generalized to a meta-theorem by H.G. Rice, 1953.
- We call it **meta-** because the properties it shows undecidable are very general.

Rice's Theorem, Informally

- Practically **no** functional properties of TM recognizable languages are decidable.
- The caveat here is that the theorem only applies to ***functional*** properties, not structural ones.
- The concept of structural vs. functional is used in software development, e.g. in software testing (white-box vs. black-box testing).

Functional vs. Structural Properties

- We have been using TM's to represent languages.
- When a property of a machine depends only on the **language** and not the ***specific*** TM used to represent the language, we call this a **functional property**.
- When a property depends on the **specific** details of a **machine**, it is called a **structural property**.

Functional vs. Structural Examples

- Functional properties of the language of a machine M:
  - L(M) = ∅
  - L(M) is regular
  - L(M) is infinite
  - L(M) is decidable
  - L(M) is recognizable (always true, by definition, since M is a TM)
- Structural properties of a machine M:
  - M has more than 100 reachable control states.
  - M writes a non-blank character on its tape when started on an empty tape.
  - M reverses its direction of head travel at least once for every input.

Trivial Functional Properties

- A functional property of a recognizable language is called **trivial** if it is either:
  - true for *all* recognizable languages, or
  - true for *no* recognizable language.
- A **non-trivial property**, then, holds for **some** recognizable languages, but **not all**.

Rice's Theorem

- **Any non-trivial functional property of the recognizable languages is not decidable.**

- Put another way:

  For any non-trivial functional property, there is no algorithm that will determine whether or not the language recognized by a given TM has the property.

Proof of Rice's Theorem (1 of 4)

- Suppose **P** is a non-trivial functional property. We are going to draw a contradiction.

- **Critical assumption**: **The empty language ∅ does *not* have property P**.

  If the **opposite** is true, then **interchange** P and ¬P so that the assumption is true.

  [This would be necessary in the case of P = Regular, for example, as ∅ is regular.]

Proof of Rice's Theorem (2 of 4)

- Let $\mathbf{L_P}$ be some **arbitrary** recognizable language **with** property P (which must exist, because P is non-trivial).
- We know that $\mathbf{L_P}$ is distinct from $\emptyset$, by the assumption above.
- Let $\mathbf{M_P}$ be a machine recognizing the chosen language $\mathbf{L_P}$.

Proof so far (3 of 4)

- ∅ does *not* have property P.
- L_P has property P.
- M_P recognizes L_P.
- **Plan**: Reduce A_{TM} to deciding property P.
- This will imply that there is no algorithm for deciding P. This will hinge on having the P decider **differentiate** between $\mathbf{L_P}$, which has property P, and ∅, which does not.

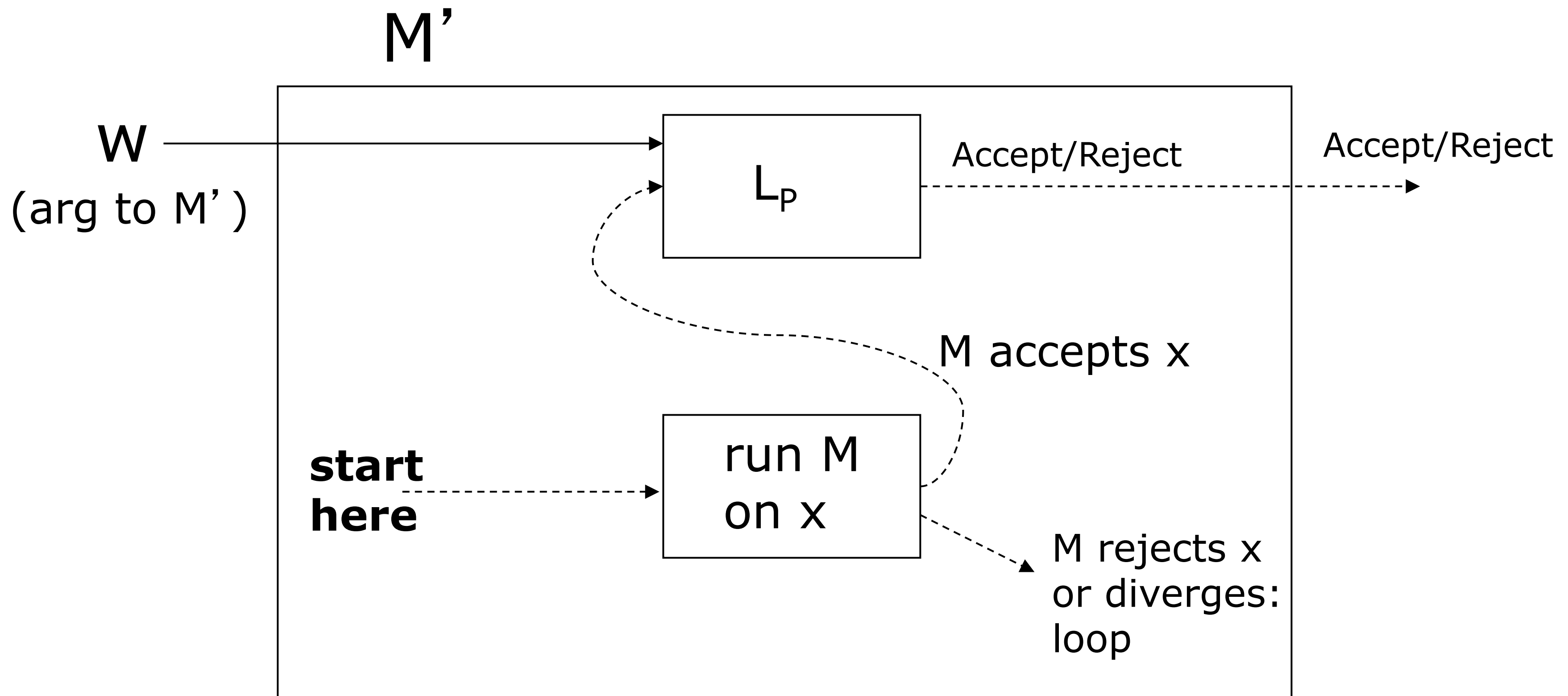
Proof of Rice's Theorem (4 of 4)

- Transform <M, x> to M'.
- M' : with input **w**, (temporarily set aside **w** and) start **behaving as** M on x.
- **If** M on x **accepts**, then continue by behaving as **M_P** on the original input w. So in this case, L(M') has property P.
- If M on x rejects or does not terminate, purposely diverge. So in this case, L(M') does not have property P.
- Thus if there were an algorithm that tested for property P of an arbitrary machine, including M', there is also one that can test whether M accepts x. But we know this is impossible.

Diagram of M' as constructed

(dashed lines = control, solid = data)

Note: This shows the transformation; it is **not** a reduction diagram.



Note

- Although P was discussed as a property of **language** recognized by a Turing machine, the same proof works in the case that:

  P is a property of the **partial function** computed by a Turing machine.

- This means, for example, there is no algorithm that will decide equivalence of an arbitrary machine's partial function to that of a given machine.

Decidable vs. Undecidable Structural Properties

- M has more than 100 control states.
- M reaches a specific control state when started on an empty tape.
- M writes a non-blank character on its tape when started on an empty tape.
- M uses more than a specified amount of tape when started on a certain input.

- Which of these properties is decidable?

- Language $A \subseteq \Sigma^*$ is ***Turing reducible*** to $B \subseteq \Delta^*$, notated $A \leq_T B$ provided:

  An algorithm for deciding A can be implemented by calling an algorithm for deciding B (querying B as if it were an "***oracle***").

  Any number of such calls can be used in general.

More varieties: http://en.wikipedia.org/wiki/Reduction_(recursion_theory)

Mapping Reducibility $\leq_m$

- Language $A \subseteq \Sigma^*$ is ***mapping reducible*** to $B \subseteq \Sigma^*$, notated $A \leq_m B$ provided:

  there is some *computable* $f: \Sigma^* \rightarrow \Delta^*$ such that

  $$\forall x \in \Sigma^* \quad x \in A \text{ iff } f(x) \in B.$$

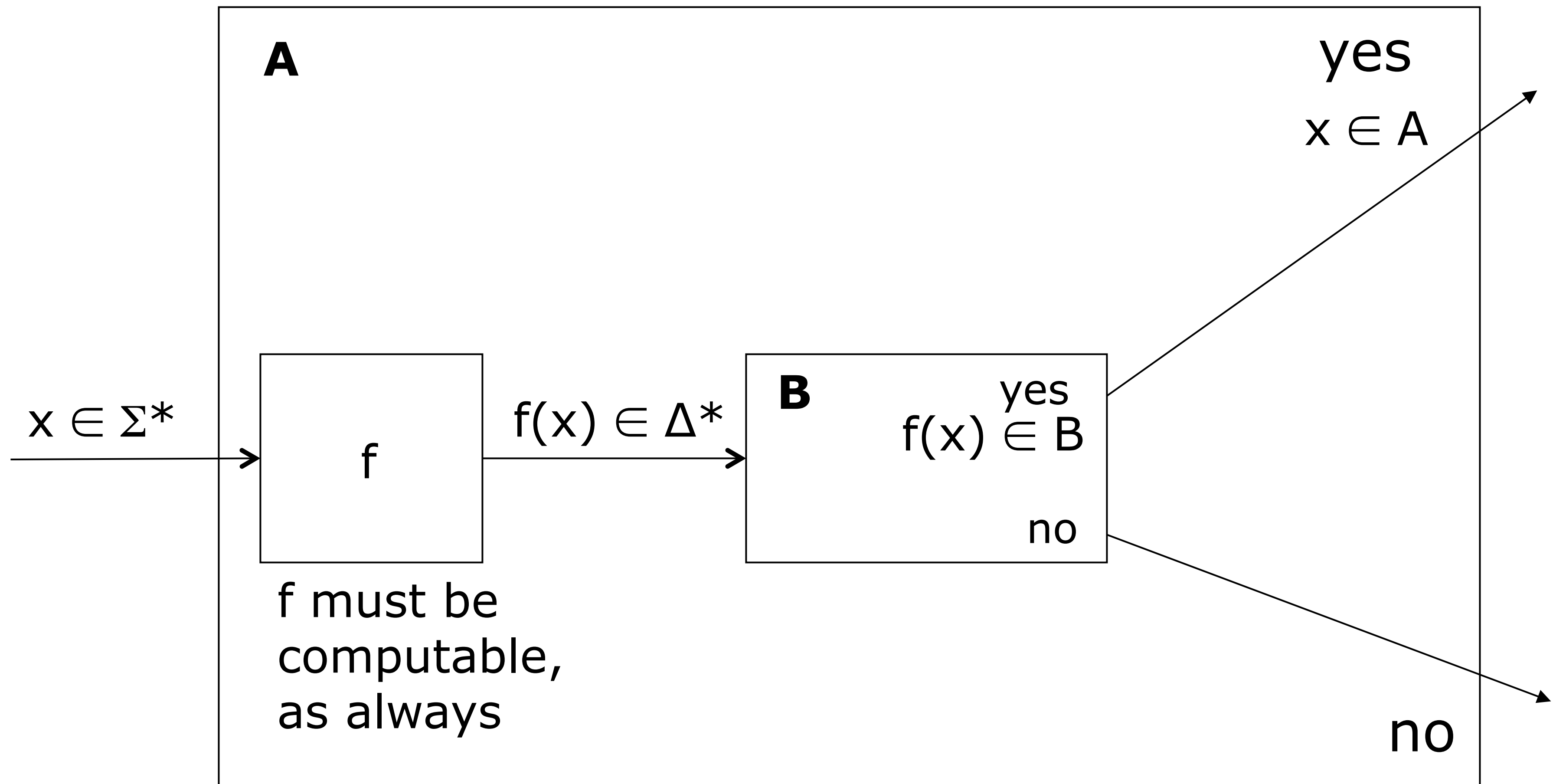
- In this case, function f is called a **reduction** of A to B.

- Mapping reductions are also called “many-to-one” reductions.

- They are a special case of Turing reductions, in which the oracle is only called once, to give the final answer. (Sort of like a **tail-recursive** version of $\leq_T$.)

Reduction Diagram for Mapping Reductions

Mapping reductions are a simple specialized case of what we've seen before.



$A \leq_m B$ and computability

- As with any reduction, $A \leq_m B$ implies if B were computable, so would A be:

  To determine if x is in A: compute f(x) by machine, determine whether f(x) is in B, to get your answer.

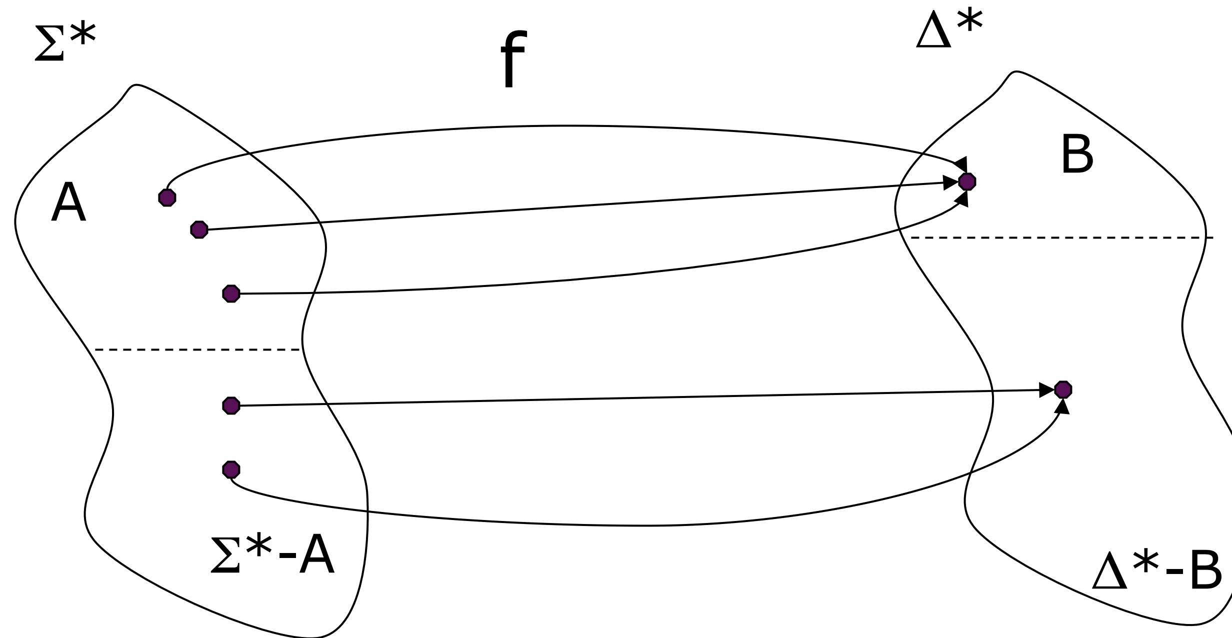
- It also means the **contrapositive**, that if A is uncomputable, so is B.

Example

- $K \leq_m A^c_{TM}$ (the *non*-acceptance language for TM)
- $K = \{<M> \mid <M> \notin L(M)\}$
- $A^c_{TM} = \{<M, w> \mid w \notin L(M)\}$
- The reduction mapping in this case is:

$$f(<M>) = <M, <M>>$$

$\leq_m$ Mapping need not be 1-1



- What function maps Map $HALT_{TM}$ into A_{TM}.
- The mapping reduction f does this transformation:

  f(<M, x>) = <M', x>.

  where M' accepts x iff M halts on x.

  The transformation from M to M′ changes all halting states into accepting states.

- Use a mapping reduction $A_{TM} \leq_m E^c_{TM}$, the latter accepting those <M> where L(M) is **non-empty**, in other words M accepts ***some*** input.

- Transformation: With input <M, x>, construct machine <M'> such that:
  - <M'> $\in E^c_{TM}$

  iff
  - <M, x> $\in A_{TM}$

- Based on <M, x>, M' with input y compares y to x and if equal, behave as M on x, otherwise diverge. M' halts on *some* input iff M accepts x.

Is either E_{TM} or its complement recognizable?

- We just showed that E_{TM}, the emptiness problem for TM's is undecidable.

- But maybe E_{TM} or its complement is recognizable.

- What do you think?

$A \leq_m B$ and recognizability

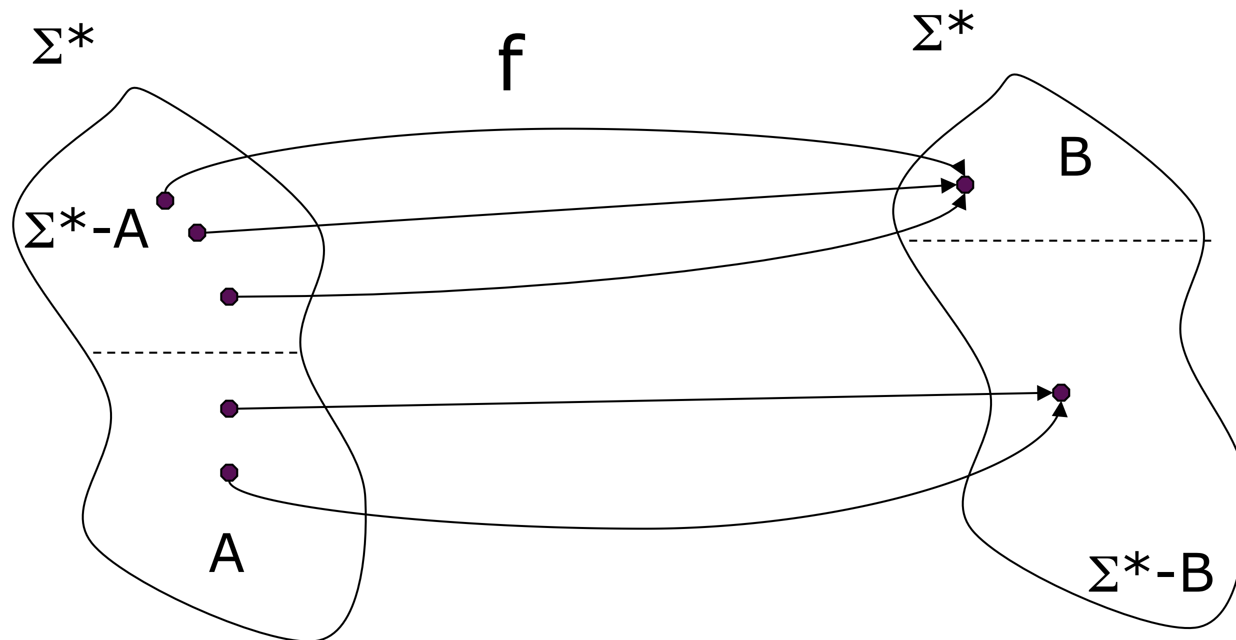
- $A \leq_m B$ also implies that if B is recognizable, so is A.
- To recognize if x is in A: compute f(x), then ask whether f(x) is in B.
  - If the answer is affirmative, then the original x was in A.
  - If we never get an answer, then we don't get answer for whether the original x was in A either.
- Thus if A is *not* recognizable, neither is B.

$A \leq_m B$ and co-recognizability

- $A \leq_m B$ similarly implies that if B were co-recognizable, so would A be.
- $A \leq_m B$ implies if A is *not* co-recognizable, neither is B.

Mixture: $A^c \leq_m B$

- $A^c \leq_m B$ means the *complement* of A reduces to B.
- This is **equivalent** to $A \leq_m B^c$.



Sipser Exercise 5.7

- **Show**: If A is recognizable, and $A \leq_m A^c$, then A is decidable.

- **Proof**: Suppose A is recognizable and $A \leq_m A^c$.

- By the previous slide, also $A^c \leq_m A$.

- Thus A^c is recognizable, from mapping reduction.

- As both A and are A^c recognizable, A is decidable by the complementarity lemma.

Example

- We already showed $A_{TM} \leq_m E^c_{TM}$ (non-emptiness is undecidable).
- In effect, the same mapping gives $A^c_{TM} \leq_m E_{TM}$.
- But A^c_{TM} is not recognizable.
- Therefore E_{TM} is not recognizable.
- However, E^c_{TM} is co-recognizable. Why?

- ALL_{DFA} = {<B> | B is a DFA accepting all strings}

  ("all" means all of Σ^*, where Σ is the alphabet of the DFA)

- ALL_{REX} = {<R> | R is a regular expression and $L(R) = \Sigma^*$}
- ALL_{CFG} = {<G> | G is a context-free grammar and $L(G) = \Sigma^*$}.
- ALL_{CSG} = {<G> | G is a context-sensitive grammar and $L(G) = \Sigma^*$}.
- ALL_{TM} = {<M> | M is a Turing machine and $L(M) = \Sigma^*$}
- Which of these are decidable? (Not necessarily obvious).

ALL_{TM}

- Is ALL_{TM} decidable?
- What is your intuition?
- How would you prove it?
- What about recognizable or co-recognizable, if not decidable?

Languages neither recognizable nor co-recognizable.

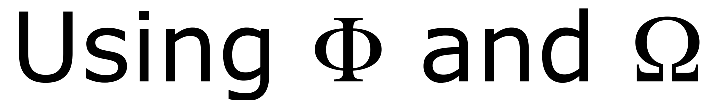
- $EQ_{TM} = \{<M, N> \mid L(M) = L(N)\}$ is neither.
- To explore this kind of problem, try letting N be a fixed machine and see if another unrecognizable language can be reduced to it.
- Candidates for N:
    - Φ, a machine that accepts nothing.
    - Ω, a machine that accepts everything.

$EQ_{TM} = \{<M, N> \mid L(M) = L(N)\}$ is not recognizable

- We know A^c_{TM} is not recognizable.
- So try to reduce A_{TM} to EQ^c_{TM}
  (which is equivalent to $A^c_{TM} \leq_m EQ_{TM}$).
- Consider the mapping $f(<M, x>) = <M', \Phi>$,
  where M' accepts its input iff M accepts x.
- Then $<M, x> \in A_{TM}$ iff $<M', \Phi> \notin EQ_{TM}$
  (i.e. M' accepts something).

EQ_{TM} = {<M, N> | L(M) = L(N)} is not co-recognizable

- We know A_{TM} is not co-recognizable.
- So try to reduce A_{TM} to EQ_{TM}.
- Consider the mapping f(<M, x>) = <M'', Ω>, where M' accepts its input iff M accepts x.
- Then <M, x> $\in A_{TM}$ iff <M'', Φ> $\in EQ_{TM}$ (i.e. M'' accepts everything).



- EQ_{TM} is not recognizable:
- These statements are equivalent for any M:
  - $<M, \Phi> \in EQ_{TM}$
  - M accepts nothing.
  - $M \in E_{TM}$
- But E_{TM} is not recognizable.

- So we have a mapping reduction
  $f(<M>) = <M, \Phi>$ that reduces E_{TM} to EQ_{TM}.

- Therefore EQ_{TM} is not recognizable.

Other languages neither recognizable nor co-recognizable.

- ALL_{TM} is neither.

- To show ALL_{TM} is **not co-recognizable**, give a mapping reduction from A_{TM} (which is not co–recognizable), to ALL_{TM}.

  f(<M, w>) = M’, where M’(x) = M(w).

  Thus M’ accepts all iff M accepts w.

- But how to show ALL_{TM} is not recognizable??

ALL_{TM} is not recognizable (New Technique)

- We prove this by $HALT^c_{TM} \leq_m ALL_{TM}$.
- Let <M, x> be an arbitrary machine with input. The reduction mapping constructs M' to behave as follows on input w:
  - M'(w) simulates M(x) for **|w| steps**.
  - M' accepts w iff M(x) ***fails* to halt** within |w| steps.
  - Otherwise M' rejects w.
- Hence M' accepts all inputs iff M(x) does not halt.

Post's Correspondence Problem (PCP)

- This is another unsolvable problem that might be unexpected due to its simplicity.
- $HALT_{TM} \leq_m PCP$
- Please read about it in Sipser 5.2 and/or Huth&Ryan.
- Huth & Ryan give a proof of Church's theorem using it.

Using Computation Histories

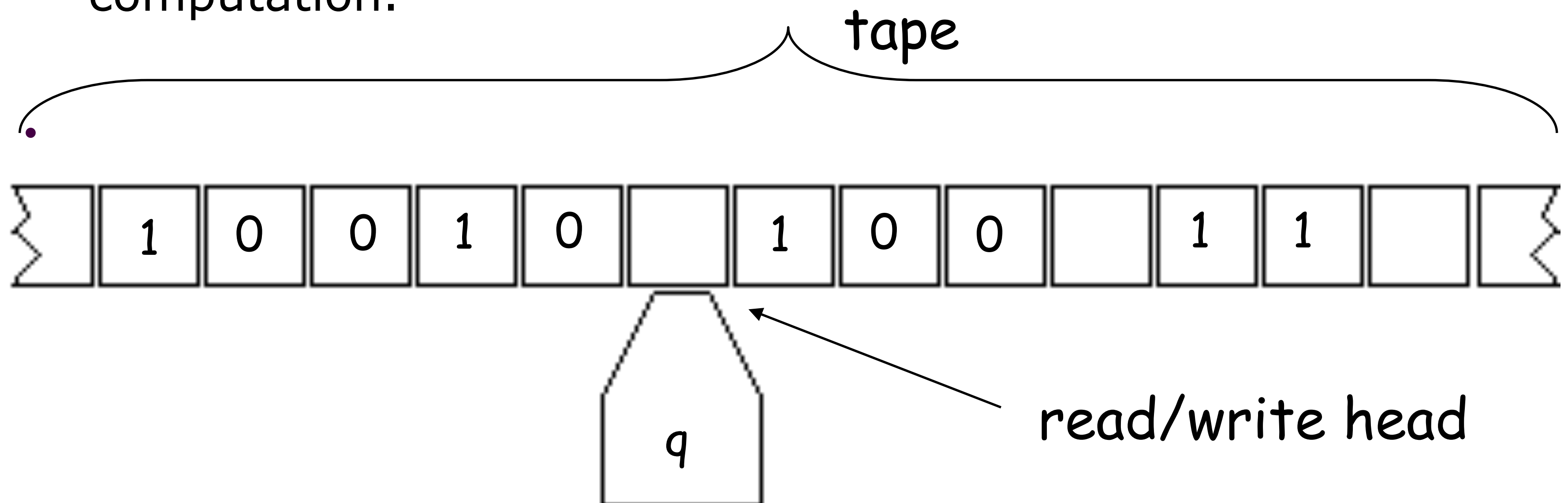
- A computation history (CH) is a recording of the history of a Turing-machine computation.

Of what use is it?

- Using CH's we can demonstrate that various problems are undecidable.
- These problems would be difficult to show using, say, mapping reduction.

State of a Turing Machine

- The total **state** (often called a “configuration” to distinguish it from the control state) represents everything that needs to be known about the machine to continue the computation.



Capturing States as Strings

- We have seen this before:
- A state can be represented as a string $xqy \in \Gamma^*$ where
  - Γ is the tape alphabet
  - x is the tape to the left of the head
  - q is the control state
  - y is the tape under and to the right of the head

Computation History

- As the TM computes, the states make transitions:

  $$x_1q_1y_1 \Rightarrow x_2q_2y_2 \Rightarrow x_3q_3y_3 \Rightarrow \ldots$$

- A deterministic machine will eventually either:
  - **Halt**: Reach a state xq_hy where no further transition is defined, or
  - **Diverge**: Never reach such a state, i.e. it goes on forever.

Encoding An Entire History

- **In the halting case**, the entire history is encodable as a **single string**:

  $x_1q_1y_1 \Rightarrow x_2q_2y_2 \Rightarrow x_3q_3y_3 \Rightarrow \ldots x_hq_hy_h$

  where here we consider $\Rightarrow$ to be a symbol in a larger alphabet.

- Usually # is used in place of $\Rightarrow$, maybe because its easier to type.

Checking a History with a TM

- There is a Turing machine C (checker) that, with input <M, h>, where <M> is an encoding of an arbitrary TM, can check whether h is a history of M.

- All C needs to do is see if the symbols around the head of each state change correspond to the transition table of M.

- Further more, C can check whether h is a halting (or an accepting) history by looking at the last control state.

Checking a History with a TM

- $x_1a_1q_1b_1y_1 \Rightarrow x_2a_2q_2b_2y_2 \Rightarrow x_3a_3q_3b_3y_3 \Rightarrow \ldots$ where each $a_i, b_i \in \Gamma$

  Do the local transitions $a_iq_ib_i \rightarrow a_{i+1}q_{i+1}b_{i+1}$ correspond to the rules in the description?

  Do the other parts of the string match?

  A TM can check these things.

LBAs

- In fact, the CH can be checked for validity by a less-than-general TM called an ***LBA***.
- LBA = "Linear Bounded Automaton", a TM that can never exceed the amount of tape on which the input is written.
- An LBA can have a tape alphabet that is bigger than the input alphabet, so it can effectively "mark" tape cells, etc. It just can't **grow** its tape.

LBA Languages

- Languages accepted by LBA’s coincide with the languages generated by **context-sensitive grammars**.

LBA vs. DFA

- At first glance, it might appear that an LBA is no more powerful than a DFA, since it cannot add new states.

- The difference is, however, that an LBA's total state set is a (linear) **function of the input string size**, which is not true for a DFA.

Aside: 2-way DFAs

- If the LBA never writes on its tape, then it becomes a 2-way DFA.
- In this case, the power is reduced to that of a DFA: 2-way DFA's only accept regular languages.
- The proof of this is non-trivial, and surprising. It involves the Myhill-Nerode theorem.

A_{LBA} is the Acceptance Language for LBAs

- Define A_{LBA} =

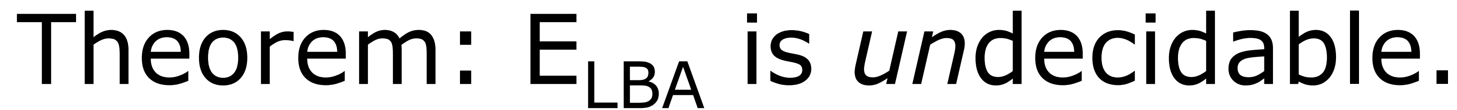
  $\{<M, w> \mid M \text{ is an LBA} \wedge M \text{ accepts } w\}$

Theorem: A_{LBA} is decidable.

- Proof: Given an LBA encoding with tape <M, w>, we can simulate M on w.
- The maximum number of distinct tape states is $n^{|w|}$, where n is tape alphabet size and |w| means the length of w.
- The number of different head positions is 1+ |w|.
- The number of control states is m, say.
- So the total number of different states of M for an input w is $mn^{|w|}(1+ |w|)$.

Proof that A_{LBA} is decidable (cont’d)

- For each step in the simulation of M on w, we increment the count, having started at 0.
- If M on w is still computing after $mn^{|w|}(1+ |w|)$ steps, we know that it is in a loop. So the simulating machine can **reject** <M, w> if this happens.
- The simulating machine will otherwise accept or reject <M, w>, depending on what happens with M on w.
- Hence there is a TM that can decide A_{LBA}.



- Define the Emptiness Language E_{LBA} =

  {<M> | M is an LBA ∧ L(M) = ∅}

  Proof: Show $A_{TM} \leq_m E^c_{LBA}$.

  Define f(<M, w>) = M', where M' is an LBA that accepts the **accepting computation histories** for M(w). Then L(M') ≠ ∅ iff <M, w> ∈ A_{TM}.

- Define ALL_{CFG} =

  $\{<G> \mid G \text{ is a CFG} \land L(G) = \Sigma^*\}$

  We know that for every CFG, there is a corresponding PDA that accepts the same language.

  We show that PDA's can also, in a sense, check computation histories.

  The halting computation of a TM will be identified with the **absence** of a string accepted by the PDA.

Proof of ALL_{CFG} is undecidable

- For any TM and input <M, w> we construct a PDA that accepts all strings that are **not** valid computation histories.

- Thus, if the PDA accepts all strings, then M does not halt on w.

Proof of ALL_{CFG} is undecidable

- We represent the computation histories slightly differently in this case:
  Every other state is **reversed**.

- Originally: $x_1a_1q_1b_1y_1 \Rightarrow x_2a_2q_2b_2y_2 \Rightarrow x_3a_3q_3b_3y_3 \Rightarrow \ldots$

- Now $x_1a_1q_1b_1y_1$ # $\overline{x_2a_2q_2b_2y_2}$ # $x_3a_3q_3b_3y_3$ # …

- where overbar represents the reversal of the string below.

Why reverse?

- Reversing every other state enables two successive states to be checked by a PDA.
- The PDA, given an input, checks one of these (non-deterministically):
  - Does the input ***not*** begin with the initial state of <M, w>? If not, **accept**.
  - Does the input ***not*** end with an accepting state of <M, w>? If not, **accept**.
  - Starting at any one of the sections $C_i\#\overline{C_{i+1}}$ or $\overline{C_i}\#C_{i+1}$, do the two sections ***not*** represent consecutive states. If not, **accept**.
- So a string is **accepted** iff it does **not** represent a valid computation history.
- Thus the PDA **accepts** Σ^* iff M(w) does not have an accepting computation history.

Corollary: EQ_{CFG} is undecidable

- Define EQ_{CFG} =

  $\{<G, H> \mid G \text{ and } H \text{ are CFG's} \wedge L(G) = L(H)\}$

  Proof: Show $ALL_{CFG} \leq_m EQ_{CFG}$.

Other Connections Between Computability & Logic

- Decidability of a certain logic theory (N, +) is reducible to DFA equivalence.

- Other theories (N, +, X) can be shown undecidable using Turing machines or their equivalent.

- Satisfiability of propositional logic plays a key role in polynomial-time reducibility (P vs. NP).