# CS 3100 – Models of Computation – Fall 2011 – Notes for L25
# Cardinalities, Computability, Undecidability
**Last Updated 9:54am, 11/22/11**

# 1 Cardinalities of Infinite Sets

In these notes, we discuss the important idea of measuring sizes of infinite sets. To motivate this problem, consider the following:

$$S_1 = \{0, 2, 4, 8, \ldots\} \subset Nat = \{0, 1, 2, 3, \ldots\}$$

and yet by dividing every element of $S_1$ by 2, we get the set of $Nat$; thus, one can put $S_1$ in 1-1 correspondence with $Nat$ through the 1-1, onto, and total function $\lambda x.(x/2)$. In effect, comparing the sizes of infinite sets is like the "barter method" (try and match up 1-1). We have these facts:

- Even though one set may be a *proper* subset of another set, so long as there is a 1-1 correspondence (1-1, onto, and total) map from one set to the other, *the sets are of the same size or cardinality.*
- If one cannot have such a correspondence, then the sets are of different cardinalities.

The first cardinal number is $\aleph_0$ ("aleph 0"). This is taken to be the "size" of $Nat$. Thus, the "size" (or *cardinality*) of $S_1$, above, is $\aleph_0$. The second cardinal number is $\aleph_1$ ("aleph 1")—same as the size of $Reals$, etc.

## 1.1 Correspondence: Total Bijection $f : A \to B$

A correspondence is a total bijection.

- A *total* function is defined everywhere in its domain.
- A *bijection* is a 1-1 and onto function.
- A *1-1* ensures that each element in the range (or co-domain) comes from at most one domain element.
- An *onto* map ensures that every element of the range is covered.

## 1.2 Schröder-Bernstein (SB) Theorem

Finding a correspondence is not that easy; finding 1-1 and *into* maps is often easier. The Schröder-Bernstein (SB) theorem is as follows:

> Suppose for two sets $A$ and $B$, there exists a 1-1, total, and into map $f : A \to B$, and another total into map $g : B \to A$. Then there is a total bijection $h : A \to B$.

Of course, $h : A \to B$ is a total bijection from $A$ to $B$ if and only if $h^{-1}$ is a total bijection from $B$ to $A$.

## 1.3 Counting the number of points in a 4-dimensional grid

Let us try and count the number of points in a 4-dimensional grid

$$4dGrid = \{\langle x, y, z, w\rangle \ \mid \ x, y, z, w \in Nat\}.$$

Normally, one would have to find a total bijection $h$ of the form
$\langle 0,0,0,0\rangle \rightarrow n_0$
$\langle 0,0,0,1\rangle \rightarrow n_1$
$\langle 0,0,0,2\rangle \rightarrow n_2$
...
$\langle 0,0,1,0\rangle \rightarrow n_k$
...
Finding these bijections is possible, but a bit tedious.

Instead, consider this $f$:
$\langle x,y,z,w\rangle \rightarrow 2^x \times 3^y \times 5^z \times 7^w$

Similarly, consider this $g$:
$\langle x\rangle \rightarrow \langle x,0,0,0\rangle$

Since $f$ and $g$ are total, 1-1, and into maps from $4dGrid$ to $Nat$ and from $Nat$ to $4dGrid$ respectively, there does exist a total bijection $h$ from $4dGrid$ to $Nat$ and vice-versa. Hence these sets have the same cardinality.

## 1.4 Counting the number of C programs

Let $CP$ be the set of all legal C programs where each program is viewed as a string. Then, define $f : CP \rightarrow Nat$ to be the natural number encoded by the ASCII bits of the C program characters, strung end to end.

There are thousands of ways to define such an $f$. One can, for instance, obtain this from the `od` program run with a `-H` option:

```
od -H << END
ABCDEFGH
END
44434241        48474645        0000000a

This means: "A" got packed as 41 in hex; then "B" as "42 in hex, etc.

-- another example --

od -H << END
main(){}
END
0000000         6e69616d        7d7b2928        0000000a
0000011

Excluding CR and LF, we have 6e69616d7d7b2928, which is some natural number.
```

Now, one can define function $g$ as follows:

```
0 -> main(){}
1 -> main(){;}
2 -> main(){;;}
...
10 -> 2 -> main(){;;;;;;;;;;}
...
```

Given $f$ and $g$, there is a correspondence between $CP$ and $Nat$, showing that the cardinality of $CP$ is $\aleph_0$.

## 1.5 Counting the Powerset of Nat

The powerset of $Nat$ has cardinal number $\aleph_1$. Each element of the powerset of $Nat$ is a bit-vector of infinite length, each telling whether a given number is there or not. For example, $\{2, 4\}$ is denoted by

```
001010...
```

which tells that 2 and 4 are "on" (there), and all other natural numbers aren't in that set. These vectors are called *characteristic vectors* ($cv$).
Suppose we claim a correspondence:

$0 \to cv_0$
$1 \to cv_1$
$\ldots$
$i \to cv_i$

Then we can find a $cv$ that evades mapping. Take the diagonal entry of the infinite matrix defined by $cv_0$, $cv_1$, etc. Invert this diagonal. This gives us a $cv$ that can't be the image of any natural number $i$ (it differs from what each $Nat$ element already maps to). Thus, in the absence of a correspondence, we give the cardinal number $\aleph_1$ to the powerset of $Nat$.

## 1.6 Counting Reals

The argument for *Real*s goes quite similarly. For the simplicity of exposition, we first present a proof that is "nearly right," and much simpler than the actual proof. In the next section, we repair this proof, giving us the actual proof. Suppose there is a bijection $f$ that puts $Nat$ and $[0, 1)$ in correspondence C1 as follows:

$0 \to .b_{00}b_{01}b_{02}b_{03}\ldots$
$1 \to .b_{10}b_{11}b_{12}b_{13}\ldots$
$\ldots$
$n \to .b_{n0}b_{n1}b_{n2}b_{n3}\ldots$
$\ldots$

where each $b_{ij}$ is 0 or 1.

Now, consider the real number

$$D = 0.\neg b_{00}\ \neg b_{11}\ \neg b_{22}\ \neg b_{33}\ldots.$$

This number is *not* in the above listing, because it differs from the $i$-th number in bit-position $b_{ii}$ **for every** $i$. Since this number $D$ is not represented, $f$ cannot be a bijection as claimed. Hence such an $f$ does not exist.

## 1.7 'Fixing' the proof a little bit

Actually the above proof needs a small "fix"; *what if the complement of the diagonal happens to involve a* $\overline{1}$*?* The danger then is that we *cannot* claim that a number equal to the complemented diagonal does not appear in our listing. It might then end up existing in our listing of Reals in a "non $\overline{1}$ form."

We overcome this problem through a simple correction. This correction ensures that the complemented diagonal will never contain a $\overline{1}$. In fact, we arrange things so that the complemented diagonal will contain zeros infinitely often. This is achieved by placing a 1 in the uncomplemented diagonal every so often; we choose to do so for all *even* positions, by listing the *Real* number $.1^{2n+1}\overline{0}\ldots$ ($2n+1$ 1s followed by $\overline{0}$) at position $2n$, for all $n$. Consider the following correspondence, for example:

$0 \rightarrow .1\overline{0}$
$1 \rightarrow .c_{00}c_{01}c_{02}c_{03}\ldots$
$2 \rightarrow .111\overline{0}$
$3 \rightarrow .c_{10}c_{11}c_{12}c_{13}\ldots$
$4 \rightarrow .11111\overline{0}$
$5 \rightarrow .c_{20}c_{21}c_{22}c_{23}\ldots$
$6 \rightarrow .1111111\overline{0}$
$\ldots$
$2n \rightarrow .1^{2n+1}\overline{0}\ldots$
$2n+1 \rightarrow .c_{n0}c_{n1}c_{n2}c_{n3}\ldots$
$\ldots$

Call this correspondence C2. We obtain C2 as follows. We know that the numbers $.1\overline{0}$, $.111\overline{0}$, $.11111\overline{0}$, etc., exist in the original correspondence C1. C2 is obtained from C1 by first permuting it so that the above elements are moved to the *even positions* within C2 (they may exist arbitrarily scattered or grouped, within C1). We then go through C1, strike out the above-listed elements, and list its remaining elements in the odd positions within C2. We represent C2 using rows of $.c_{ij}$, as above.

We can now finish our argument as follows. The complemented diagonal doesn't contain a $\overline{1}$, because it contains 0 occurring in it infinitely often. Now, this complemented diagonal cannot exist anywhere in our $.c_{ij}$ listing. The complemented diagonal is certainly a Real number missed by the original correspondence C1 (and hence, also missed by C2). Hence, we arrive at a contradiction that we have a *correspondence*, and therefore, we cannot assign the same cardinal number to the set $[0, 1) \subseteq Real$. It is therefore of higher cardinality.

The conclusion we draw from the above proof is that *Real* and *Nat* have different cardinalities. Are there any cardinalities "in between" that of *Real* and *Nat*? Loosely speaking, "is there a $\aleph_{0.5}$?!" The hypothesis that states "no there isn't a cardinality between $\aleph_0$ and $\aleph_1$," or in other words, "there isn't a $\aleph_{0.5}$," is known as the *Continuum Hypothesis*. It has been a problem of intense study over the last 120 years, and in fact is the *first* of Hilbert's 23 challenges to computer science. These challenges helped spur considerable amounts of research in Computer Science, and contributed to much of the foundational knowledge of the subject area (e.g., as covered in this book). We shall use cardinality arguments when comparing the set of all functions and the set of all *computable* functions.

## 1.8 Cardinality of $2^{Nat}$ and $Nat \rightarrow Bool$

In this section, we argue that the sets $2^{Nat}$ (the powerset of $Nat$) and $Nat \rightarrow Bool$ (the set of functions from $Nat$ to $Bool$) have the same cardinality as $Real$. Notice that each set within $2^{Nat}$ can be represented by an infinitely long characteristic sequence. For instance, the sequence $1001010\overline{0}$ represents the set $\{0, 3, 5\}$; the sequence $101010\ldots$ represents the set $Even$; the sequence $010101\ldots$ represents the set $Odd$; and so on. Notice that the very same characteristic sequences also represent functions from $Nat$ to $Bool$. For instance, the sequence $1001010\overline{0}$ represents the function that maps 0, 3, and 5 to $true$, and the rest of $Nat$ to $false$; the sequence $101010\ldots$ represents the function $\lambda x.even(x)$; and the sequence $010101\ldots$ represents the function $\lambda x.odd(x)$. Hence, the above two sets have the same cardinality as the set of all infinitely long bit-sequences. How many such sequences are there? By putting a "0." before each such sequence, it appears that we can define the $Real$s in the range $[0, 1]$. However, we face the difficulty caused by infinite 1s, i.e., we will end up having $\overline{1}$ occurring within an infinite number of infinite sequences. However, all these have been addressed cleanly in the construction we just now illustrated in § 1.7.

# 2 Undecidability of the Halting Problem

We define the notion of *decidability*, *semi-decidability*, and *undecidability*. These notions pertain to *degrees of solvability* of problems by Turing machines. We will present three proof methods in this chapter: (i) through contradiction (Section 4.3), (ii) through reductions from languages unknown to be decidable (Section 4.4), and (iii) through mapping reductions (Section 4.5).

Methods (ii), (iii), and (iv) are strongly related to each other, in the following sense:

- Applications of method (ii), namely reduction *from* a known undecidable language, $A$, *to* the language in question, $B$, employs an 'if and only if' argument of the form $x \in A \Leftrightarrow f(x) \in B$.
- Method (iii), namely *mapping reductions*, isolates this 'if and only if' argument into a *mapping reduction* principle which is quite powerful, and also applicable in other contexts (e.g., in our study of NP-completeness later).

# 3 Some Decidable and Undecidable Problems

## 3.1 An assortment of decidable problems

In all descriptions below, we use $\langle\rangle$ to indicate the *code* or *description*; for instance, $\langle G\rangle$ stands for a grammar $G$'s description as a character- or bit-string. Also, for a Turing machine $M$, $\langle M\rangle$ will mean its description, say in the form of a table such as JFLAP's tabular coding of TM transitions. Sometimes, we omit $\langle\ldots\rangle$ if the intent is clear from the context.

$\diamond$ $ALL_{DFA}$**:** Given a DFA, is its language $\Sigma^*$?
This problem is modeled as a language membership question in the language:

$$ALL_{DFA} = \{\langle A\rangle \mid A \text{ is a DFA that recognizes } \Sigma^*\}.$$

$ALL_{DFA}$ can be shown to be decidable by minimizing the given DFA and examining the result.

$\diamond$ $A\varepsilon_{CFG}$**:** Given a CFG, does it generate $\varepsilon$?

$$A\varepsilon_{CFG} = \{\langle G\rangle \ \mid\ G \text{ is a CFG that generates } \varepsilon\}.$$

One approach is to trace all $\varepsilon$-generating productions using a bottom-up marking algorithm similar to how we found generating non-terminals—except we focus on which non-terminals generate $\varepsilon$. Call this notion $\varepsilon$-*generating.* Now, for any $a \in \Sigma$, $a$ is not $\varepsilon$-generating. If $A \rightarrow \varepsilon$, then $A$ is $\varepsilon$-generating. If $A \rightarrow B_1 \dots B_n$ and if all of $B_i$ are $\varepsilon$-generating, then so is $A$. Finally, check whether the start symbol $S$ is $\varepsilon$-generating.

$\diamond$ $INFINITE_{DFA}$**:** Given a DFA, does it have an infinite language?

$$INFINITE_{DFA} = \{\langle A\rangle \ \mid\ A \text{ is a DFA and } L(A) \text{ is Infinite}\}.$$

$\diamond$ $NOODD_{DFA} =$

$$\{\langle A\rangle \ \mid\ A \text{ is a DFA that does not accept any string with odd 1s}\}.$$

$\diamond$ $ONESTAR_{CFG}$**:** Given a CFG, does it include *some* strings from $1^*$?

$$ONESTAR_{CFG} = \{\langle G\rangle \ \mid\ G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

One algorithm is to build the product machine of a DFA for $1^*$ and a PDA for $G$ with a view to obtain the intersection of their languages. The result will be a PDA. We can then check the language emptiness of this PDA, which is decidable (e.g., by converting the resulting PDA to a CFG and running the bottom-up marking algorithm on the CFG to see if the start symbol, $S$, of the CFG, is generating). Details of this product construction algorithm are left to the reader.

$\diamond$ $EMPTY_{CFG}$**:**

$$EMPTY_{CFG} = \{\langle G\rangle \ \mid\ \text{is a CFG and L(G)} = \emptyset\}.$$

One can employ a marking-based algorithm to decide whether starting from the initial non-terminal, $S$, one can generate a terminal-only string.

#### Main Take-away Message

The main take-away message is that we are often asked to answer queries about *languages* accepted by specific machines. These queries can often be answered algorithmically; but beginning with PDAs, many questions, we have limited luck. We can decide $EMPTY_{CFG}$. With TMs, all but the trivial questions will prove to have associated *undecidable* languages—meaning, these questions can't be algorithmically answered.

## 3.2 Assorted undecidable problems

We now present a list of undecidable problems and sketch reasons for them to be undecidable. In Section 4, we present the actual undecidability proofs, after introducing basic notions such as recursive enumerability.

- The universality of the language of a CFG is undecidable. A language $L$ is universal if $L = \Sigma^*$.
- The equivalence of two CFGs is undecidable.

- In-equivalence of two CFGs is undecidable.
- Whether a given Turing machine accepts string $w$ is undecidable.
- Whether a given Turing machine halts on string $w$ is undecidable.
- The emptiness of the language of a Turing machine is undecidable.
- Whether a given Turing machine's language is context-free is undecidable.

Here are intuitive arguments that support the above claims:

- $L_G = \Sigma^*$: Informally, given a CFG $G$, it seems one must find a string that $G$ cannot generate. One can, of course, keep checking the strings within $\Sigma^*$ in an ascending order of lengths, with each check taking a finite amount of time; however, this process does not have a definite stopping criterion. *A formal proof will be given later,* but intuitively it does appear that this is undecidable – and let us assume so for the purpose of supporting the following discussions.
- $L_{G_1} = L_{G_2}$: If this were to be decidable, we would be able to decide $L_G = \Sigma^*$.
- $L_{G_1} \neq L_{G_2}$: We observe that we must examine whether every string generated by $G_1$ is generated by $G_2$, and vice versa. This appears to be a search with no definite stopping criterion. However, since $L_{G_1} = L_{G_2}$ is not decidable, $L_{G_1} \neq L_{G_2}$ cannot be decidable (if set $S$ is decidable, so is $\overline{S}$; otherwise, we can employ the algorithm to decide $\overline{S}$ as an algorithm to decide $S$. Please think why).
- We will state and prove results about the halting and acceptance of Turing machines. It also will turn out that we can find similar proofs for the emptiness of the language of a Turing machine being undecidable, and for whether a given Turing machine's language is context-free being undecidable. This series of undecidability results about Turing machines is usually captured by one "master theorem" called Rice's Theorem.

In Section 4, we will motivate the important concept of *recursive enumerability.* We will show that all pairs $\langle G_1, G_2\rangle$ such that $L_{G_1} \neq L_{G_2}$ are enumerable, in the sense that every such pair can be found in a finite amount of time and printed out. This will mean that the language of pairs $\langle G_1, G_2\rangle$ such that $L_{G_1} = L_{G_2}$ is *not* enumerable (if a set $S$ and its complement are enumerable, then membership in $S$ becomes decidable, as will be re-iterated soon). This is another reason why the language of pairs $\langle G_1, G_2\rangle$ such that $L_{G_1} \neq L_{G_2}$ is not decidable.

## 4 Undecidability Proofs

### 4.1 Turing recognizable (or recursively enumerable) sets

The terms Turing recognizable (TR) and recursively enumerable (RE) will be used interchangeably as they essentially mean the same thing, but from two different perspectives. A language $L$ is TR if it is the language of *some* Turing machine $M$. We write $L_M$ for emphasis. A language $L$ is RE if there exists a Turing machine $M$ that can enumerate the strings in $L$ (say, on an "output tape") such that any member $x \in L$ is guaranteed to appear in a finite amount of time.
Some clarifications:

- Suppose we are required to show that $L$ is TR. We must seek a Turing machine that
  - Accepts a candidate string $x$ in $\Sigma^*$
  - Engages in the process of determining whether $x \in L$

  - Halts in the "Accept" state if $x \in L$
  - Is not required to behave in any specific way (may halt in a non-accepting state—*or in other words, reject* $x$—if $x \notin L$. Or it may loop forever.

- Suppose we are required to show that $L$ is RE. We must seek a Turing machine that
  - After being fired up, may never stop
  - The TM is expected to print out on the tape each member of $L$ in a finite amount of time, typically one string after the other.

For instance,
$$NEQ_{CFG} = \{\langle G_1, G_2\rangle \ \mid \ G_1, G_2 \text{ are CFGs and } L(G_1) \neq L(G_2)\}$$
is TR, with a candidate Turing machine being the following:

- Input: $\langle G_1, G_2\rangle$.
- Output: If $\langle G_1, G_2\rangle \in NEQ_{CFG}$, then "accept," else "loop."
- Method:
  - Generate strings $x \in \Sigma^*$ in numeric order, feeding $x$ to a parser (PDA) for $G_1$ and another for $G_2$.
  - For each such $x$, if $x$ can be parsed by $G_1$ and not by $G_2$ (or vice versa), go to *accept*.

This process can keep enumerating unequal CFG pairs; but if two CFGs $P$ and $Q$ happen to denote the same language, this process will *not* be able to list $\langle P, Q\rangle$. This is typical of problems that are *semi decidable*.

**Many TMs may be "up to the job":** Notice that if $L$ is $L_M$ for one TM $M$, then it is the language of an infinite number ($\aleph_0$) of other machines, $M^{'}$, as we can simply pad $M$ with $i$ "no op" instructions, for every $i \in Nat$. Each such artificially bloated TM is indeed considered a distinct TM!

### Recursively Enumerable Languages

A recursively enumerable (RE) language is the language that an *enumerator* Turing machine can enumerate. An enumerator Turing machine is a Turing machine that has no input tape, but has an output tape. In addition, it may employ a working tape. It keeps generating (finite) strings, and appends each generated string to the output tape. Here is an enumerator for $NEQ_{CFG}$:

- Keep enumerating all possible pairs of grammars $\langle G_1, G_2\rangle$ over the given alphabet, on a working tape. This is possible because the *syntax* of any context-free grammar over a given set of terminals and non-terminals is expressible as a regular expression, and one can generate random strings and filter those passing the regular expression as a legal CFG.
- Keep enumerating strings $x \in \Sigma^*$ in enumeration order, also on the working tape.
- Run one additional step of a parsing algorithm for *all* the grammar pairs $\langle G_1, G_2\rangle$ generated so far acting on *all* the inputs $x$ generated so far (these are called the 'simulations in progress').
- If one of the simulations in progress reports that the parser for grammar $G_1$ accepted an $x$ while the parser for the corresponding grammar $G_2$ rejected $x$ (or vice versa), then write $\langle G_1, G_2\rangle$ on the output tape.

The above enumerator guarantees that every pair of nonequivalent grammars will, eventually, be enumerated. From the above constructions, it is an easy exercise to conclude the following theorem.

**Theorem 4.1** A language is TR if and only if it is RE.

The main argument is that given an enumerator (in the sense of RE), we can build a recognizer (in the sense of TR), and vice versa.

**Why study both definitions, namely TR and RE?:** The main reason one should study both these definitions is to gain mathematical maturity as well as gain experience building various types of TMs. It also improves one's ability to come up with innovative proofs either using the TR type of TMs or the RE types of TMs.

### 4.1.1 Dovetailing and systematic enumeration methods

In many of our Turing machine constructions, we face the situation of enumerating the Cartesian product of a collection of sets $S_i,\ i \in k$ for some $k \in N$. For example, we may have to enumerate pairs of grammars and strings, thus effectively enumerating triples from sets that, individually, have $\aleph_0$ cardinality. There are many systematic approaches for achieving this end; we now summarize a standard approach, taking triples of $Nat$ as an example:

- Enumerate all the triples that add up to $i$ before enumerating any triple that adds up to $i+1$. Within each group that adds up to $i$, employ a lexicographic order.
- As an example, enumerate $\langle 0,0,0\rangle$, followed by $\langle 0,0,1\rangle$, $\langle 0,1,0\rangle$, $\langle 1,0,0\rangle$, followed by $\langle 0,1,1\rangle$, $\langle 1,0,1\rangle$, $\langle 1,1,0\rangle$, $\langle 0,0,2\rangle$, $\langle 0,2,0\rangle$, $\langle 2,0,0\rangle$, etc.

## 4.2 Recursive (or decidable) languages

A recursive language $L_M$ is the language of a Turing machine $M$ that, given any $x \in L_M$, accepts $x$, and given any $y \notin L_M$, rejects $y$. In other words, $M$ does not loop on any input. Note that it is possible to have another machine $N$ such that $L(M) = L(N)$ and $N$ loops on inputs $y \notin L_M$; however, so long as there exists *one* decider $M$, we can conclude that $L_M$ is recursive (or decidable).
Another *very important* characterization of recursive languages is this:

**Theorem 4.2** $L$ is recursive if and only if $L$ and $\overline{L}$ are RE (equivalently, are TR).

Imagine an enumerator enumerating $L$ and another enumerating $\overline{L}$. To decide whether some $x \in \Sigma^*$ is in $L$, all we need to do is watch which enumerator outputs $x$.[1] Decidable languages correspond to algorithmically solvable problems.

### 4.2.1 Non-RE languages

The cardinality of the set of all Turing machine descriptions is $\aleph_0$. This is because a Turing machine can be described through a finite number of bits that model its states and its transitions, and such a description can be read as a natural number (these numbers are known as Gödel numbers). On the other hand, there are $\aleph_1$ languages over $\Sigma$. Therefore, there are non-TR (non-RE) languages.

[1]Imagine two big 'spigots,' one pouring out the contents of $L$ and another pouring out the contents of $\overline{L}$. One can decide membership of $x \in L$ by watching which spigot emits $x$.

This means that there exist languages in which membership testing cannot be carried out by *any* Turing machine.

## 4.3 Acceptance ($A_{TM}$) is undecidable (important!)

This is one of the most fundamental results that we shall encounter in our study of Turing machines and computation. Define

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts string } w\}.$$

Deciding membership in $A_{TM}$ is tantamount to asking "does a given Turing machine $M$ accept a string $w$?" We prove this set to be *undecidable* through contradiction, as follows:

- **Suppose** there exists a decider $H$ for $A_{TM}$. $H$ expects to be given a Turing machine $M$ and a string $w$. Notice that "giving $H$ a Turing machine" means "giving it a character string representing a Turing machine program." Hence, in reality, we will be feeding it $\langle M, w\rangle$ as mentioned in Section 3.1.
- Build a program called $D$ as follows:
  - $D$ takes a single argument $M$.
  - As its first step, $D$ invokes $H$ on $\langle M, M\rangle$.[2]
  - If $H(\langle M, M\rangle)$ rejects, $D(\langle M\rangle)$ accepts.
  - If $H(\langle M, M\rangle)$ accepts, $D(\langle M\rangle)$ rejects.
- Now we can ask what $D(\langle D\rangle)$ will result in (to preserve the clarity of our arguments, the reader is invited to suppress any occurrence of $\langle\ldots\rangle$ in the text below):
  - The $D(\langle D\rangle)$ "call" turns into an $H(\langle D, D\rangle)$ call.
  - Suppose $H(\langle D, D\rangle)$ rejects. In that case, $\boxed{D(\langle D\rangle) \text{ accepts.}}$
  - But, according to the advertised behavior of $H$ — which is that it is a decider for $A_{TM}$ — the fact that $H(\langle D, D\rangle)$ rejects means that $D$ is *not* a Turing machine that will accept $\langle D\rangle$, or that $\boxed{D(\langle D\rangle) \text{ rejects!}}$
  - Suppose $H(\langle D, D\rangle)$ accepts. In that case, $\boxed{D(\langle D\rangle) \text{ rejects.}}$
  - But, according to the advertised behavior of $H$ — which is that it is a decider for $A_{TM}$ — the fact that $H(\langle D, D\rangle)$ accepts means that $D$ *is* a Turing machine that accepts $\langle D\rangle$, or that $\boxed{D(\langle D\rangle) \text{ accepts!}}$

Therefore, we obtain a contradiction *in both cases.*

Hence, the claimed decider $H$ for $A_{TM}$ cannot exist, or

The *acceptance problem* for Turing machines is undecidable.

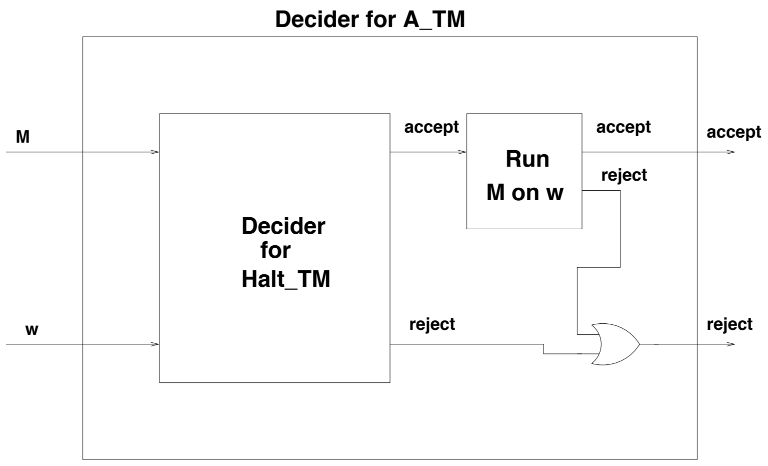

Figure 1: $A_{TM}$ to $Halt_{TM}$ reduction. Notice that *if* we assume that the inner components – namely the OR-gate, the ability to run $M$ on $w$, and $D_{Halt_{TM}}$ exist, then $D_{ATM}$ can be constructed; and hence, $D_{Halt_{TM}}$ cannot exist!

## 4.4 Halting ($Halt_{TM}$) is undecidable (important!)

The golden rule of reduction is: **Reduce an existing ("old") undecidable problem to the given ("new") problem.** This way, if we assume that the new problem is decidable, we would be forced to conclude that the existing undecidable problem is decidable — a contradiction. See Figure 1 where the "*new problem*" is $Halt_{TM}$, and by assuming that it is decidable, we can assume the existence of the decider $D_{Halt_{TM}}$, and using it, build a decider for $A_{TM}$, the "*old problem*" already shown to be undecidable; this obtains a contradiction. Hence, $D_{Halt_{TM}}$ cannot exist.

### 4.4.1 Don't get reduction backwards!

We should not go by the English language meaning of the term "reduction" that can lead us astray. Getting the meaning of 'reduction' backwards means trying to reduce the new problem to an existing ("old") problem. This does not help us. To see why, assume this to be the right direction. Then we will be trying to employ proof by contradiction (following reduction) in the following futile way: "IF the existing undecidable problem is decidable, THEN we would have shown the new problem to be decidable." However, this is a statement of the form "IF false, THEN assert X." Clearly, in this case, we cannot assert X.

### 4.4.2 An introduction to mapping reduction

*Reduction* is an abbreviation for *mapping reduction* — a concept fully explored in Section 4.5. In this section, we informally apply the (mapping) reduction idea to show that $Halt_{TM}$ is undecidable. Define

$$Halt_{TM} = \{\langle M, w\rangle \ \mid \ M \text{ is a Turing machine that halts on string } w\}.$$

We show $Halt_{TM}$ to be undecidable as follows:

[2]Basically, we feed $\langle M\rangle$ twice over, just to 'please' $H$ that expects two arguments.

Suppose not; i.e., there is a decider for $Halt_{TM}$. Let's now build a decider for $A_{TM}$ (call it $D_{A_{TM}}$). $D_{A_{TM}}$'s design will be as follows:

- $D_{A_{TM}}$ will first feed $M$ and $w$ to $D_{Halt_{TM}}$, the claimed decider for $Halt_{TM}$.
- If $D_{Halt_{TM}}$ goes to $accept_{D_{Halt_{TM}}}$, $D_{A_{TM}}$ knows that it can safely run $M$ on $w$, which it does.
- If $M$ goes to $accept_{M}$, $D_{A_{TM}}$ will go to $accept_{D_{A_{TM}}}$.
- If $M$ goes to $reject_{M}$, or if $D_{Halt_{TM}}$ goes to $reject_{D_{Halt_{TM}}}$, $D_{A_{TM}}$ will go to $reject_{D_{A_{TM}}}$.

Notice that we have labeled the accept and reject states of the two machines $D_{Halt_{TM}}$ and $D_{A_{TM}}$. After one becomes familiar with these kinds of proofs, higher-level proof sketches are preferred. Here is such a higher-level proof sketch:

> Build a decider for $A_{TM}$. This decider accepts input $\langle M, w\rangle$ and runs *Halt_decider* (if it exists) on it. If this run accepts, then we can safely (without the fear of looping) run $M$ on $w$, and return the accept/reject result that this run returns; else return "reject."

A diagram that illustrates this construction is in Figure 1. Therefore, we conclude that

> the *Halting problem* for Turing machines is undecidable.

Two observations that the reader can make after seeing many such proofs (to follow) are the following:

- One *cannot* write statements of the form "if $f(x)$ loops, then ..." in any algorithm, because termination is not detectable. Of course, one *can* write "if $f(x)$ halts, then ... ." This asymmetry is quite fundamental, and underlies all the results pertaining to halting / acceptance.
- One cannot examine the code ("program") of a Turing machine and decide what its language is. More precisely, one cannot build a classifier program $Q$ that, given access only to Turing machine programs $P_m$ (which encode Turing machines $m$), classify the $m$'s into two bins (say "good" and "bad") according to the language of $m$. Any such classifier will have to classify all Turing machines as "good" or all as "bad, " or itself be incapable of handling all Turing machine codes (not be *total*).

## 4.5 Mapping reductions

**Definition:** A *computable* function $f : \Sigma^* \to \Sigma^*$ is a mapping reduction from $A \subseteq \Sigma^*$ into $B \subseteq \Sigma^*$ if for all $x \in \Sigma^*$, $x \in A \Leftrightarrow f(x) \in B$.
**Definition:** A polynomial-time mapping reduction $\leq_P$ is a mapping reduction where the reduction function $f$ has polynomial-time asymptotic upper-bound time complexity.[3]

See Figure 2 which illustrates the general situation that $A$ maps into a subset denoted by $f(A)$ of $B$, and members of $A$ map into $f(A)$ while non-members of $A$ map outside of $B$ (that means they map outside of even $B \setminus f(A)$). Also note that $A$ and $B$ need not be disjoint sets, although they often are. A mapping reduction can be (and usually is) a non-injection and non-surjection; *i.e.*, it can be many to one and not necessarily onto. It is denoted by $\leq_m$. By asserting $A \leq_m B$, the existence of an $f$ as described above is also being asserted. Typically mapping reductions are used as follows:

- Let $A$ be a language known to be undecidable ("old" or "existing" language).
- Let $B$ be the language that must be shown to be undecidable ("new" language).

[3] Using the familiar notation $\mathcal{O}(\ldots)$ for asymptotic upper-bounds, polynomial-time means $\mathcal{O}(n^k)$ for an input of length $n$, and $k > 1$.

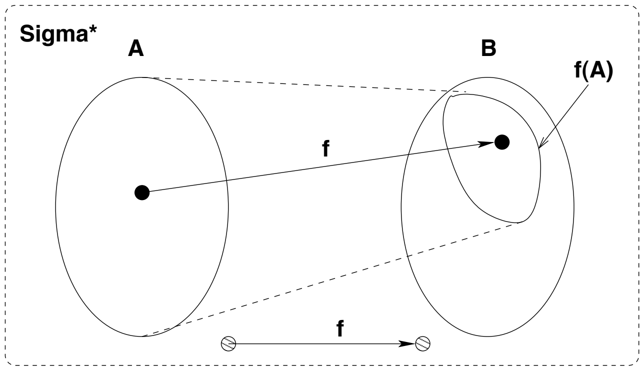

Figure 2: Illustration of mapping reduction $A \leq_M B$

- Find a mapping reduction $f$ from $A$ into $B$.
- Now, if $B$ has a decider $D_B$, then we can decide membership in $A$ as follows:
  - On input $z$, in order to check if $z \in A$, find out if $D_B(f(z))$ accepts or not. If it accepts, then $z \in A$, and if it rejects, then $z \notin A$.

### 4.5.1 Mapping Reduction From $A_{TM}$ to $Halt_{TM}$

We first illustrate mapping reductions by taking $A = A_{TM}$ and $B = Halt_{TM}$ with respect to Figure 2. Function $f$ takes a member of $A_{TM}$, namely a pair $\langle M, w\rangle$, as input, and prints out $\langle M^{'}, w\rangle$ on the tape as its output. Function $f$, in effect, generates the *text* of the program $M^{'}$ from the text of the program $M$. Here is the makeup of $M^{'}$:

```
M'(x) =

    Run M on x
    If the result is "accept," then "accept"
    If the result is "reject," then loop
```

Notice that the text of $M^{'}$ has "spliced" within itself a copy of the text of program $M$ that was input. Mapping reductions such as $f$ illustrated here need not "run" the program they manufacture; they simply accept a program such as $M$, and a possible second input, such as $w$, and manufacture another program $M^{'}$ (and also copy over $w$) and then consider their task done! The reason such a process turns out to be useful is for the following reasons:

> Suppose someone were to provide a decider for $Halt_{TM}$. The mapping reduction $f$ then makes it possible to obtain a decider for $A_{TM}$. When given $\langle M, w\rangle$, this decider will obtain $\langle M^{'}, w\rangle = f(\langle M, w\rangle)$, and then feed it to the decider for $Halt_{TM}$.

We have to carefully argue that $f$ is a mapping reduction. We will be quite loose about the argument types of $f$ (namely that it maps $\Sigma^*$ to $\Sigma^*$; we will assume that any $\langle M, w\rangle$ pair can be thought to be a string, and hence a member of a suitable $\Sigma^*$. The proof itself is depicted in Figure 3.

```
How a decider for A_TM is obtained:

Step 1: Here is the initial tape.
-------------------------------------------------------------------
| M | w |
-------------------------------------------------------------------

Step 2. Build M' and put it on the tape
-------------------------------------------------------------------
| M | w | ..build M' that incorporates M here.. |
-------------------------------------------------------------------

Step 3. Put w on the tape.
----------------------------------------------------------------------
| M | w | ..build M' that incorporates M here.. | ..put w here.. |
----------------------------------------------------------------------

Step 4. Run Halt_TM_decider on M' and w  and return its decision
----------------------------------------------------------------------
| M | w | ..build M' that incorporates M here.. | ..put w here.. |
----------------------------------------------------------------------
```

$$D_{Halt_{TM}}(M^{'}, w) = \begin{cases} accepts & \Rightarrow\ M^{'}\ halts\ on\ w\ \Rightarrow \quad M\ accepts\ w \\ rejects & \Rightarrow\ M^{'}\ doesn't\ halt\ on\ w\ \Rightarrow \quad M\ doesn't\ accept\ w \end{cases}$$

Figure 3: How the mapping reduction from $A_{TM}$ to $Halt_{TM}$ works

### 4.5.2 Mapping reduction From $A_{TM}$ to $E_{TM}$

We show that

$$E_{TM} = \{\langle M\rangle \ \mid \ \text{M is a TM and } L(M) = \emptyset\}$$

is undecidable through a mapping reduction that maps $\langle M, w\rangle$ into $\langle M^{'}\rangle$, as explained in Figure 4.

### 4.5.3 Mapping reduction from $A_{TM}$ to $Regular_{TM}$

Similarly, we can prove $Regular_{TM}$ to be undecidable by building the $M^{'}$ shown in Figure 5.

## 4.6 Undecidable problems are "$A_{TM}$ in disguise"

The techniques discussed here lie at the core of the notion of "problem solving" in that they help identify which problems possess algorithms and which do not.

Undecidable problems are $A_{TM}$ in disguise. We leave you with this thought, hoping that it will provide you with useful intuitions.

```
    M'(x) {
      if x <> w then loop ; // could also goto reject_M' here
      Run M on w ;
      If M accepts w, goto accept_M' ;
      If M rejects w, goto reject_M' ; }

How a decider for E_TM is obtained:

Step 1: Build above M' and put it on the tape
------------------------------------------------------------------
| M | w | ..build M' that incorporates M and w here.. |
------------------------------------------------------------------

Step 2: Run E_TM_decider on M' and return its decision
------------------------------------------------------------------
| M | w | ..build M' that incorporates M and w here.. |
------------------------------------------------------------------
```

$$Decider_{E_{TM}}(M^{'}) = \begin{cases} accepts & \Rightarrow L(M^{'})\ is\ empty\ \Rightarrow & M\ does\ not\ accept\ w \\ rejects & \Rightarrow L(M^{'})\ is\ not\ empty\ \Rightarrow & M\ accepts\ w \end{cases}$$

Figure 4: Mapping reduction from $A_{TM}$ to $\overline{E_{TM}}$

```
    M'(x) {
      if x is of the form 0^n 1^n then goto accept_M' ;
      Run M on w ;
      If M accepts w, goto accept_M' ;
      If M rejects w, goto reject_M' ; }
```

$$Decider_{Regular_{TM}}(M^{'}) = \begin{cases} accepts & \Rightarrow L(M^{'})\ is\ regular \\ \quad \Rightarrow\ Language\ is \Sigma^{'} & \Rightarrow M\ accepts\ w \\ rejects & \Rightarrow L(M^{'})\ is\ not\ regular \\ \quad \Rightarrow\ Language\ is\ 0^n 1^n & \Rightarrow M\ does\ not\ accept\ w \end{cases}$$

Figure 5: Mapping reduction from $A_{TM}$ to $Regular_{TM}$