Comp. Models - Lecture 6, Spring 2009

- Turing Machines
- Turing Machines
- Turing Machines

- Alternative Models of Computers
- Multitape TMs, RAMs, Non Deterministic TMs
- The Church-Turing Thesis
- The language classes $\mathcal{R} = \mathcal{RE} \cap \text{coRE}$
- David Hilbert’s Tenth Problem

- Sipser’s book, 3.1, 3.2, & 3.3
A Finite Automaton

01101 1001

read    unread
A Pushdown Automaton

[Diagram of a Pushdown Automaton]

01101 1001 $s$
read unread

aba

pop

push
A Turing Machine
Turing Machines

- Machines so far (DFA, PDA) read input only once
- Turing Machines
  - Can back up over the input
  - Can overwrite the input
  - Can write information on tape and come back to it later
Turing Machines

- Input string is written on a tape:

- At each step, machine reads a symbol, and then
  - writes a new symbol, and
  - either moves read/write head to right,
  - or moves read/write head to left
TM vs. PDA vs. DFA: Differences

- A Turing machine can both write on the tape and read from it.
- A PDA is restricted to reading from the stack in LIFO manner.
- A DFA has no media for writing anything – it must all be in its finite state.
- The TM read-write head can move both to the left and to the right.
- The TM read-write tape is infinite to the right.
- The special final (accepting/rejecting) states of TM take immediate effect (so the head need not be at some special position).
Effects of One Step

One step of computation changes

- current state,
- current head position,
- and tape contents at current position.

- Each step has very local, small effect.
- But many small effects can accumulate to a meaningful task.
Formal Definition

We start with the transition function

\[ \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} \]

\(\delta(q, a) = (r, b, L)\) means:

- in state \(q\) where head reads tape symbol \(a\),
- the machine writes \(b\), replacing the \(a\) (\(a = b\) is possible),
- enters state \(r\),
- and moves the head left
  (this is what the \(L\) stands for).
Formal Definition (2)

Now the transition function, with a move to the right

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$  

$$\delta(q, a) = (r, b, R)$$ means:

- in state $q$ where head reads tape symbol $a$,
- the machine writes $b$, replacing the $a$ ($a = b$ is possible),
- enters state $r$,
- and moves the head right (this is what the $R$ stands for).
Formal Definition (3)

A Turing machine (TM) is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)\), where

- \(Q\) is a finite set of states,
- \(\Sigma\) is the **input alphabet** not containing the blank symbol, \(
\)
- \(\Gamma\) is the **tape alphabet**, where \(\_\in\Gamma\) and \(\Sigma\subset\Gamma\).
- \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
- \(q_0 \in Q\) is the **start state**,
- \(q_a \in Q\) is the **accept state**, and
- \(q_r \in Q\) is the **reject state**.
Formal Definition (4)

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ computes as follows

- an input of length $n$, $w = w_1w_2 \ldots w_n \in \Sigma^*$
- is placed on $n$ leftmost tape squares, one tape square per input letter
- rest of tape contains blanks
- read/write head is on leftmost square of tape
- since $\bot \not\in \Sigma$, leftmost blank indicates end of input.
Formal Definition (5)

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$.

When computation starts,

- $M$ proceeds according to transition function $\delta$.
- If $M$ tries to move head beyond left-hand-end of tape, it doesn’t move (still $M$ does not crash).
- Computation continues until $q_a$ or $q_r$ is reached,
- otherwise $M$ runs forever.
TM Configurations

A TM configuration is a convenient notation for recording the state, head location, and tape contents of a TM in a given instant. Think of it as a **snapshot**.

- For example, configuration $1011q_70111$ means:
  - Current state is $q_7$,
  - left hand side of tape (to the left of the head) is $1011$,
  - right hand side of tape is $0111$,
  - and head is on 0 (leftmost entry of right hand side).
Configurations: The Yield Relation

- If $\delta(q_i, b) = (q_j, c, L)$ then configuration $u a q_i b v$ yields configuration $u q_j a c v$.
- If $\delta(q_i, b) = (q_j, c, R)$, then configuration $u a q_i b v$ yields configuration $u a c q_j v$.
- Special case (1): When head is at left end and tries to move left, it changes state and writes on tape but does not move, so if $\delta(q_i, b) = (q_j, c, L)$, configuration $q_i b v$ yields $q_j c v$.
- Special case (2): What happens when head is at right end? We let $w q_i$ and $w q_i \downarrow$ denote the same configuration, so moves to the right can now be accommodated.
Special case (2): What happens when head is at right end? We let $wq_i$ and $wq_{i-1}$ denote the same configuration, so moves to the right can now be accommodated.

In special case (2), the new configuration is longer as it “annexed” one blank. This allows configurations to grow in length with computation. Yet at any given moment, they are finitely long.
More on Configurations

We have

- starting configuration \( q_0w \)
- accepting configuration \( w_0q_1w_1 \)
- rejecting configuration \( w_0q_rw_1 \)
- halting configurations \( w_0q_1w_1 \) and \( w_0q_rw_1 \)
Accepting a Language

A Turing machine $M$ accepts an input $w$ if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that

- $C_1$ is start configuration of $M$ on $w$,
- each $C_i$ yields $C_{i+1}$,
- $C_k$ is an accepting configuration.

The collection of strings accepted by $M$ is called the language of $M$, and is denoted $L(M)$.
Enumerable Languages

Definition: A language $L$ is called (recursively) enumerable (RE) if some Turing machine accepts $L$. 
Enumerable Languages (2)

On an input, $w$, a TM may

- accept
- reject
- loop (run forever)

Major concern: In general, we never know if the TM will halt.
Decidable Languages

**Definition:** A TM decides a language if for every input $w \in \Sigma^*$, the TM halts.

Namely the TM either reaches state $q_a$ (in case $w \in L(M)$) or it reaches state $q_r$ (in case $w \notin L(M)$), but it does not loop.

**Definition:** A language $L$ is decidable if some Turing machine decides it.
Example

Here is a high level description of a TM that decides the (non context free) language

\[ L = \{ a^i b^j c^k \mid i \times j = k \text{ where } i, j, k \geq 1 \} \]

- scan from left to right to check that input is \( a^* b^* c^* \)
- return to start of tape
- cross off one \( a \) and scan right until \( b \) occurs. Shuttle between \( b \)’s and \( c \)’s, crossing off one of each, until all \( b \)’s are gone.
- **Restore** the crossed-off \( b \)’s and repeat previous step if more \( a \)’s exist. If all \( a \)’s crossed off, check if all \( c \)’s crossed off. If yes, **accept**, otherwise **reject**.
A Minor, Technical Point

**Question:** To implement algorithm, should be able to tell when a TM is at the left end of the tape.

**Answer** Mark it with a special symbol. When head reads this symbol, TM “knows” it is on the leftmost tape square.
A Second Example

Consider the element distinctness problem

\[ E = \{ \#x_1\#x_2\# \ldots \#x_\ell \mid \text{each } x_i \in \{0, 1\}^* \text{ and for each } i \neq j, x_i \neq x_j \} . \]

Verbally,

- List of strings in \( \{0, 1\}^* \) separated by \#’s.
- List is in language (& machine should accept) if all strings are different.
Element Distinctness – Solution

On input \( w \)

- place a mark on leftmost tape symbol. If symbol not \#\, reject.
Element Distinctness – Solution (2)

- scan right to next # and place mark on top. If none encountered, reject

#011#00#1111
Element Distinctness – Solution (3)

By zig-zagging, compare the two strings to the right of the marked #’s. If equal, reject.
Element Distinctness – Solution (4)

- Iterate:
  - Move rightmost mark to next # on right, if any.
  - Compare two strings to the right of the marked #’s, as before.
  - Otherwise move leftmost mark to next # on right and rightmost mark to # after that.
  - If not possible, accept.
Element Distinctness – Solution (end)

**Question:** How do we “mark” a symbol?

**Answer:** For each tape symbol #, add tape symbol • # to the tape alphabet Γ.
TMs Variants

Alternative Turing machine definitions abound.

For example, suppose the Turing machine head is allowed to stay put.

\[ \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\} \]

**Question:** Does this add any power?

**Answer:** No. Replace each \( S \) transition with two transitions: \( R \) then \( L \). (ahmm ... why not vice-versa?)

Important notion here: Two-way simulation (model \( A \) capable of simulating model \( B \); model \( B \) capable of simulating model \( A \)).
Multitape Turing Machines

- each tape has its own head
- initially, input string on tape 1 and rest blank

For the transition function:
\[ \delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k, \]

the expression
\[ \delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L) \]

means

- machine starts in state \( q_i \)
- if heads 1 through \( k \) reading \( a_1, \ldots, a_k \),
- then machine goes to state \( q_j \),
- heads 1 through \( k \) write \( b_1, \ldots, b_k \),
- and moves each head right or left as specified.
Equivalence

**Theorem:** A language is enumerable if and only if there is some multitape Turing machine that accepts it.

One direction is trivial.

To prove the other direction, we will show how to convert a multitape TM, $M$, into an equivalent single-tape TM, $S$. 
Simulation

- $S$ simulates $k$ tapes of $M$ by storing them all on a single tape with delimiter $\#$.
- $S$ marks the current positions of the $k$ heads by placing • “above” the letters in current positions. It “knows” which tape the mark belongs to by counting (up to $k$) from the $\#$’s to the left.
On input $w = w_1 \cdots w_n$, $S$:

- writes on its tape $\# w_1 w_2 \cdot w_n \# \underline{\#} \ # \underline{\#} \ # \cdots \ #$
- scans its tape from first $\#$ to $k + 1$-st $\#$ to read symbols under “virtual” heads.
- rescans to write new symbols and move heads
- $S$ tries to move virtual head onto $\#$ when $M$ is trying to move head onto unused blank square. $S$ writes blank $\underline{\#}$ on tape, and shifts rest of the tape one square to the right.
RAM

- CPU
- 3 Registers (Instruction Register (IR), Program Counter (PC), Accumulator (ACC))
- Memory
- Operation:
  - Set IR $\rightarrow$ MEM[PC]
  - Increment PC
  - Execute *instruction* in IR
  - Repeat
- Instructions are typically compare, add/subtract, multiply/divide, shift left/right.
- All are doable on a TM.
RAM

- CPU
  - Registers
    - Instruction Register
    - Program Counter
    - Accumulator
  - Memory
    - Addresses: 0001, 0002, 0003, ..., 0015, ...

## RAM

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>HALT</td>
</tr>
<tr>
<td>01</td>
<td>LOAD x</td>
</tr>
<tr>
<td>02</td>
<td>LOADI x</td>
</tr>
<tr>
<td>03</td>
<td>STORE x</td>
</tr>
<tr>
<td>04</td>
<td>ADD x</td>
</tr>
<tr>
<td>05</td>
<td>ADDI x</td>
</tr>
<tr>
<td>06</td>
<td>SUB x</td>
</tr>
<tr>
<td>07</td>
<td>SUBI x</td>
</tr>
<tr>
<td>08</td>
<td>JUMP x</td>
</tr>
<tr>
<td>09</td>
<td>JZERO x</td>
</tr>
<tr>
<td>10</td>
<td>JGT x</td>
</tr>
</tbody>
</table>
**RAM**

Here is a program that multiplies two numbers (in locations 1000 & 1001), and stores the result in 1002

<table>
<thead>
<tr>
<th>Memory</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>011000</td>
<td>LOAD 1000</td>
</tr>
<tr>
<td>0002</td>
<td>031003</td>
<td>STORE 1003</td>
</tr>
<tr>
<td>0003</td>
<td>020000</td>
<td>LOADI 0</td>
</tr>
<tr>
<td>0004</td>
<td>031002</td>
<td>STORE 1002</td>
</tr>
<tr>
<td>0005</td>
<td>021003</td>
<td>LOAD 1003</td>
</tr>
<tr>
<td>0006</td>
<td>090012</td>
<td>JZERO 0012</td>
</tr>
<tr>
<td>0007</td>
<td>070001</td>
<td>SUBI 1</td>
</tr>
<tr>
<td>0008</td>
<td>031003</td>
<td>STORE 1003</td>
</tr>
<tr>
<td>0009</td>
<td>011002</td>
<td>LOAD 1002</td>
</tr>
<tr>
<td>0010</td>
<td>041001</td>
<td>ADD 1001</td>
</tr>
<tr>
<td>0011</td>
<td>080004</td>
<td>STORE 1002</td>
</tr>
<tr>
<td>0012</td>
<td>000000</td>
<td>HALT</td>
</tr>
</tbody>
</table>
Theorem: A multi-tape Turing machine can simulate this RAM model.

Idea:

- One tape for each register (IR, IP, ACC)
- One tape for the Memory
- Memory tape will be entries of the form <address> <contents>

A sketch of such simulation (not a formal proof) will hopefully be given in the recitation.
RAMs, Computers & TMs

- A RAM can be modeled (simulated) by a Turing Machine.
- Any current machine (architecture, manufacturer, operating system, power supply, etc.) can be modeled by a Turing Machine.
- If there is an algorithm for it, a Turning Machine can do it.
- Note that at this point, we don’t care how long it might take, just that it can be done.
Turing Completeness

- A computation model is called “Turing Complete” if it can simulate a (general) Turing Machine.

- Turing Complete $\Rightarrow$ can compute anything

- Of course it might not be convenient...
Non-Deterministic Turing Machines

Transition function:

\[ \delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \]

- Computation of a deterministic TM can be viewed as a path in configuration space.
- Computation of a non-deterministic TM (NTM) can be viewed as a tree in configuration space.

- NTM accepts an input if there is (\(\exists\)) an accepting branch in its computation tree.
- NTM rejects an input if all (\(\forall\)) branches in its computation tree are either rejecting or infinite (looping).
Equivalence

**Theorem:** A language is enumerable \((\mathcal{RE})\) if and only if there is some non-deterministic Turing machine that accepts it.

One direction is trivial.

To prove the other direction, we will show how to convert a non-deterministic TM, \(N\), into an equivalent deterministic TM, \(D\).
Simulating Non-Determinism

Basic idea:

- $D$ tries all possible branches
- If $D$ finds any accepting branch, it accepts.
- If all branches reject, $D$ rejects.
- If all branches reject or loop, $D$ loops.
Simulating Non-Determinism (2)

$N$’s computation is a tree in the configurations’ space.

- each tree branch is branch of $N$’s non-deterministic computation
- each tree node is a configuration of $N$
- root is starting configuration
- the number of children of each node, denoted by $b$, is at most the number of $N$’s states, times the size of $\Gamma$, times 2 (left/right).
- depth-first search doesn’t work (why?)
- breadth-first search does work, as we’ll show.
Simulating Non-Determinism (3)

The simulating machine, $D$, has three tapes

- the input tape is never altered (only read from)
- the simulation tape is a copy of $N$’s tape
- the address tape keeps track of $D$’s location in $N$’s computation tree.
Simulating Non-Determinism (4)

The address tape:
- every node in the tree has at most $b$ children
- every node in the tree is assigned an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$
- to get to node with address 231:
  - start at root
  - take second child of root
  - take third child of current node
  - take first child of current node
- ignore meaningless addresses (choices not available for configuration along branch)
Simulating Non-Determinism (5)

- Initially, the input tape contains $w$, and the other two tapes are empty.
- Copy input tape to simulation tape.
- Use simulation tape to simulate $N$ on input $w$ on a finite portion of one non-deterministic branch. On each choice, consult the next symbol on address tape. Accept if accepting configuration reached. Skip to next step if
  - symbols on address tape are exhausted
  - non-deterministic choice is invalid
  - rejecting configuration was reached
- Replace string on address tape with the lexicographically next string. Go back to Step 2, to simulate this branch of $N$’s computation.
Decidability vs. Enumerability

- **Decidability** is a stronger notion than enumerability.

- If a language $L$ is decidable then clearly it is enumerable (the other direction does not hold, as we’ll show in a couple of lectures).

- It is also clear that if $L$ is decidable then so is $\overline{L}$, and thus $\overline{L}$ is also enumerable.

- Let $\mathcal{RE}$ denote the class of enumerable languages, and let $\text{co} \mathcal{RE}$ denote the class of languages whose complement is enumerable.

- Let $\mathcal{R}$ denote the class of decidable languages. Then what we just argued implies $\mathcal{R} \subseteq \mathcal{RE} \cap \text{co} \mathcal{RE}$.
Decidability vs. Enumerability (2)

**Theorem:** $\mathcal{R} = \mathcal{RE} \cap \text{coRE}$.  

**Proof:** We should prove the $\supseteq$ direction. Namely if $L \in \mathcal{RE} \cap \text{coRE}$, then $L \in \mathcal{R}$.

In other words, if both $L$ and its complement are enumerable, then $L$ is decidable.

Let $M_1$ be a TM that accepts $L$.
Let $M_2$ be a TM that accepts $\overline{L}$.
We describe a TM, $M$, that decides $L$.
On input $x$, $M$ runs $M_1$ and $M_2$ in parallel.
If $M_1$ accepts, $M$ accepts.
If $M_2$ accepts, $M$ rejects.
Should now show that indeed $M$ decides $L$.  

Reformulation

**Theorem:** A language is decidable if and only if it is both enumerable and co-enumerable.

**Proof:** We must prove two directions:

- If $L$ is decidable, then both $L$ and $\overline{L}$ are enumerable.
- If $L$ and $\overline{L}$ are both enumerable, then $L$ is decidable,
One Direction

Claim: If $L$ is decidable, then both $L$ and $\overline{L}$ are enumerable.

Proof: Argued this three slides ago!
Other Direction

**Claim:** If $L$ and $\overline{L}$ are both enumerable, then $L$ is decidable,

- Let $M_1$ be the acceptor for $L$, and
- $M_2$ the acceptor for $\overline{L}$.

$M = \text{On input } w$
1. Run both $M_1$ and $M_2$ in parallel.
2. If $M_1$ accepts, accept; if $M_2$ accepts, reject.
In Parallel?

**Question:** What does it mean to run $M_1$ and $M_2$ in parallel?

- $M$ has two tapes
- $M$ alternates taking steps between $M_1$ and $M_2$. 
Claim

We claim that $M$ decides $L$.

- Every string is in $L$ or in $\overline{L}$ (of course not in both).
- Thus either $M_1$ or $M_2$ accepts the input $w$.
- Consequently, since $M$ halts whenever $M_1$ or $M_2$ accepts, $M$ always halts, and hence is a decider.
- Moreover, $M$ accepts strings in $L$ and rejects strings in $\overline{L}$.

Therefore, $M$ decides $L$, so $L$ is decidable. ♣
Revised View of the World of Languages

- Regular
- Context Free
- Decidable
- Enumerable
Enumerators

We said a language is *enumerable* if it is accepted by some Turing machine. But why *enumerable*?

**Definition:** An *enumerator* is a TM with a printer.

- TM sends strings to printer
- may create infinite list of strings
- TM *enumerates* a language – all the strings produced.
Theorem

**Theorem:** A language is accepted by some Turing machine if and only if some enumerator enumerates it.

Will show

- If $E$ enumerates language $A$, then some TM $M$ accepts $A$.
- If $M$ accepts $A$, then some enumerator $E$ enumerates it.
Theorem

Claim: If $E$ enumerates language $A$, then some TM $M$ accepts $A$.

On input $w$, TM $M$

- Runs $E$. Every time $E$ outputs a string $v$, $M$ compares it to $w$.
- If $v = w$, $M$ accept.
- If $v \neq w$, $M$ continues running $E$. 
Theorem

Claim: If $M$ accepts $A$, then some enumerator $E$ enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in $\Sigma^*$ (e.g. strings in lexicographic order).
The enumerator, $E$

- repeat the following for $i = 1, 2, 3, \ldots$
- run $M$ for $i$ steps on each input $s_1, s_2, \ldots, s_i$.
- if any computation accepts, print out the corresponding $s$.

Note that with this procedure, each output is duplicated infinitely often.
How can this duplication be avoided?
Theorem

Claim: A language $L$ is decidable if and only if there is some enumerator $E$ that enumerates $L$ in lexicographic order.

Proof: Left as an exercise.
What is an Algorithm???????
Indeed, What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it :-(

- Historically,
  - notion has long history in Mathematics (starting with Euclid’s gcd algorithm), but
  - not precisely defined until 20th century
  - informal notions rarely questioned,
  - still, they were insufficient
Remarks

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models were shown to be equivalent to Turing machines.
- All “reasonable” programming languages (e.g. Java, Pascal, C, Python, Scheme, Mathematica, Maple, Cobol, . . . ) are equivalent.
- The notion of an algorithm is model-independent!
- We don’t really care about Turing machines per se.
- We do care about understanding computation, and because of their simplicity, Turing machines are a good model to use.
Church-Turing Thesis

Formal notions appeared, starting in 1936:

- $\lambda$-calculus of Alonzo Church
- Turing machines of Alan Turing
- Recursive functions of Gödel and Kleene
- Counter machines
- Unrestricted grammars
- Two stack automata
- Random access machines (RAMs)

These definitions look very different, but are provably equivalent.
Church-Turing Thesis

These definitions look very different, but are provably equivalent.

The Church-Turing Thesis:

“The intuitive notion of reasonable models of computation equals Turing machine algorithms”.
Wild Models

What about “wild” models of computation?
Consider MUnTel’s $\aleph$-AXP© processor (to be released Labor Day 2009).

- Like a Turing machine, except
- Takes first step in 1 second.
- Takes second step in $1/2$ second.
- Takes $i$-th step in $2^{-i}$ seconds . . .

After 2 seconds, the $\aleph$-AXP© decides any enumerable language!

**Question:** Does the $\aleph$-AXP© invalidate the Church-Turing Thesis?
Hilbert’s 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented **23 central mathematical problems**
- challenge for the next (20th) century
- the **10th problem** directly concerned algorithms

**November 2003**: significant progress on the **6th problem**.

But for us, start with some background on the **10th** . . .
Hilbert’s 10th Problems

Too much beer last night? We are supposed to talk about D. Hilbert’s problems, not ’bout Dilbert’s problems, ...
Multivariate Polynomials

- A **term** is a product of **variables** and a constant **coefficient**, e.g. $6x^3yz^2$.

- A **multivariate polynomial** is a sum of terms, e.g. $6x^3yz^2 + 3xy^2 - x^3 - 10$.

- A **root** of a polynomial is an assignment of values to variables so that the polynomial equals **zero**.

- For example, $x = 5$, $y = 3$, and $z = 0$ is a root of the polynomial above.

- Here, we are interested in **integral** roots, namely an assignment of **integers** to all variables.

- Some polynomials have integral roots, some don’t (e.g. $x^2 - 2$).
Hilbert’s Tenth Problem

The Problem: Devise an algorithm that tests whether a polynomial has an integral root.

Actually, what he said (translated from German) was “to devise a process according to which it can be determined by a finite number of operations”.

Note that

- Hilbert explicitly asks that algorithm be “devised”
- apparently Hilbert assumes that such an algorithm must exist, and someone “only” need find it.
Hilbert’s Tenth Problem

- We now know no algorithm exists for this task.
- Even great mathematicians of 1900 (like Hilbert) could not have proved this, because they didn’t have a formal notion of an algorithm.
- Intuitive notions work fine for constructing algorithms (we know one when we see it).
- Formal notions are required to show that no algorithm exists.
Hilbert’s Tenth Problem

In 1970, 23 years old Yuri Matijasevič, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, proved that no algorithm exists for testing whether a polynomial has integral roots

(a survey of the proof)
Reformulating Hilbert’s Tenth Problem

Consider the language:

\[ D = \{ p \mid p \text{ is a polynomial with an integral root} \} \]

Hilbert’s tenth problem asks whether this language is **decidable**.

We now know it is **not decidable**, but it is **enumerable**!
Univariate Polynomials

Consider the simpler language:

\[ D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Here is a Turing machine that accepts \( D_1 \).
On input \( p \),

- evaluate \( p \) with \( x \) set successively to \( x = 0, x = 1, x = -1, x = 2, x = -2, \ldots \).
- if \( p \) evaluates to zero, accept.
Univariate Polynomials (2)

\[ D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Note that

- If \( p \) has an integral root, the machine **accepts**.
- If not, \( M_1 \) **loops**.
- \( M_1 \) is an **acceptor**, but not a **decider**.
Univariate Polynomials (3)

> f := x -> x^3 - 300*x^2 + 10000*x + 1000000;

\[ f := x \rightarrow x^3 - 300x^2 + 10000x + 1000000 \]

> g := x -> 200*x^2 - 2000*x - 1000000;

\[ g := x \rightarrow 200x^2 - 2000x - 1000000 \]

> plot([f(x), g(x)], x=-100..300, color=[red, blue], thickness=3);
Univariate Polynomials (4)

In fact, $D_1$ is decidable.

Can show that all real roots of $p[x]$ lie inside interval

$\left( -\left| kc_{\text{max}} / c_1 \right|, \left| kc_{\text{max}} / c_1 \right| \right)$,

where $k$ is number of terms, $c_{\text{max}}$ is max coefficient, and $c_1$ is high-order coefficient.

Thus it suffices that the TM will check only finitely many values of $x$, and thus it always halts.

By Matijasevič theorem, such effective bounds on range of roots cannot be computed for multivariable polynomials.