CS420
Reducibility
Wednesday, April 12, 2023

```c
DEFINE DOESITHALT(PROGRAM):
{
    RETURN TRUE;
}
```

THE BIG PICTURE SOLUTION TO THE HALTING PROBLEM
Announcements

• HW 8 out
  • Due Tue 4/11 11:59pm
  • Due Thu 4/13 11:59pm

• HW 9 out
  • Due Sun 4/23 11:59pm (extended)

• No Class next Monday 4/17
  • Patriot’s Day

• No quiz today!
Last Time: Undecidability Proofs

• We proved \( A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \} \) undecidable ...

• ... by contradiction:
  • Use hypothetical \( A_{TM} \) decider to create an impossible decider “\( D \)”!
    
    **reduce “\( D \) problem” to \( A_{TM} \)**

• Step # 1: coming up with “\( D \)” --- **hard**!
  • Need to invent **diagonalization**

• Step # 2: **reduce “\( D \)” problem to \( A_{TM} \) --- easier**!

• From now on: undecidability proofs **only need step # 2**!
  • And we now have two “impossible” problems to choose from
Last Time: The Halting Problem

\[ \text{HALT}_\text{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \} \]

**Thm:** \( \text{HALT}_\text{TM} \) is undecidable

**Proof**, by contradiction:

- **Assume:** \( \text{HALT}_\text{TM} \) has decider \( R \); use it to create decider for \( A_{\text{TM}} \):
  \[ A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \]

- ...  

- But \( A_{\text{TM}} \) is undecidable and has no decider!

**Reduce** (from known undecidable) \( A_{\text{TM}} \) to \( \text{HALT}_\text{TM} \)
The Halting Problem

**Thm:** \( \text{HALT}_{\text{TM}} \) is undecidable

**Proof, by contradiction:** Using our hypothetical \( \text{HALT}_{\text{TM}} \) decider \( R \)

- **Assume:** \( \text{HALT}_{\text{TM}} \) has decider \( R \); use it to create decider for \( A_{\text{TM}} \):

  \[
  A_{\text{TM}} = \{ \langle M, w \rangle | \text{M is a TM and } M \text{ accepts } w \} 
  \]

  \( S = \) “On input \( \langle M, w \rangle \), an encoding of a TM \( M \) and a string \( w \):

  1. Run TM \( R \) on input \( \langle M, w \rangle \).
  2. If \( R \) rejects, reject.
  3. If \( R \) accepts, simulate \( M \) on \( w \) until it halts.
  4. If \( M \) has accepted, accept; if \( M \) has rejected, reject.”

**Termination argument:**

- **Step 1:** \( R \) is a decider so always halts
- **Step 3:** \( M \) always halts because \( R \) said so
**Last Time:** The Halting Problem

\[ \text{HALT}_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \} \]

**Thm:** \( \text{HALT}_{\text{TM}} \) is undecidable

**Proof, by contradiction:**

- **Assume:** \( \text{HALT}_{\text{TM}} \) has *decider* \( R \); use it to create decider for \( A_{\text{TM}} \):

  \[ A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \]

  \[ S = \text{“On input \( \langle M, w \rangle \), an encoding of a TM \( M \) and a string } w:} \]
  
  1. Run \( T \text{M} \) \( R \text{ on input } \langle M, w \rangle \).
  2. If \( R \text{ rejects, reject.} \)
  3. If \( R \text{ accepts, simulate } M \text{ on } w \text{ until it halts.} \)
  4. If \( M \text{ has accepted, accept; if } M \text{ has rejected, reject.”} \)

- **But** \( A_{\text{TM}} \) *is undecidable!* i.e., this decider does not exist!
  - So \( \text{HALT}_{\text{TM}} \) is also undecidable!

**Undecidability Proof Technique #1:**

- **Reduce** (directly) from \( A_{\text{TM}} \)
- (by creating \( A_{\text{TM}} \) decider)
Interlude: Reducing from $\text{HALT}^\text{TM}$

A practical thought experiment ...  
... about compiler optimizations

Your compiler changes your program!

If TRUE then A else B $\rightarrow$ A

$1 + 2 + 3 \rightarrow 6$
Compiler Optimizations

Optimization - docs

- **-00**
  - No optimization, faster compilation time, better for debugging builds.

- **-02**
- **-03**
  - Higher level of optimization. Slower compile-time, better for production builds.

- **-OFast**
  - Enables higher level of optimization than **-03**. It enables lots of flags as can be seen in the source code (`-ffloat-store, -ffast-math, -ffinite-math-only, -O3 ...`)

- **-finline-functions**
- **-m64**
- **-funroll-loops**
- **-fvectorize**
- **-fprofile-generate**

Types of optimization

Techniques used in optimization can be broken up among various scopes which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

**Peephole optimizations**

These are usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like “looking through a peephole” at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself (this example is also an instance of strength reduction).

**Local optimizations**

These only consider information local to a basic block. Since basic blocks have no control flow, these optimizations need very little analysis, saving time and reducing storage requirements. But this also means that no information is preserved across jumps.

**Global optimizations**

These are also called ‘interprocedural methods’ and act on whole functions. This gives them more information to work with, but often makes expensive computations necessary. Worst case assumptions have to be made when function calls occur or global variables are accessed because little information about them is available.

**Loop optimizations**

These act on the statements which make up a loop, such as a for loop, for example loop-invariant code motion. Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

**Prescient store optimizations**

These allow store operations to occur earlier than would otherwise be permitted in the context of threads and locks. The process needs some way of knowing ahead of time what value will be stored by the assignment that it should have followed. The purpose of this relaxation is to allow compiler optimization to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs.

**Interprocedural, whole-program or link-time optimization**

These analyze all of a program’s source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information, i.e., within a single function. This kind of optimization can also allow new techniques to be performed. For instance, function inlining, where a call to a function is replaced by a copy of the function body.

**Machine code optimization and object code optimizer**

These analyze the executable task image of the program after all of an executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression which saves space by collapsing common sequences of instructions, are more effective when the entire executable task image is available for analysis.
The Optimal Optimizing Compiler

**Thm:** The Optimal (C++) Optimizing Compiler does not exist

**Proof,** by contradiction:

**Assume:** *OPT* is the Perfect Optimizing Compiler

Use it to create *HALT* \(_{TM}\) decider (accepts \(<M, w>\) if \(M\) halts with \(w\), else rejects):

\(S = \) On input \(<M, w>\), where \(M\) is C++ program and \(w\) is string:

- If \(OPT(M) = \text{for(;;)}\)
  
  a) Then **Reject**
  
  b) Else **Accept**

---

In computer science and mathematics, a **full employment theorem** is a term used, often humorously, to refer to a theorem which states that no algorithm can optimally perform a particular task done by some class of professionals. The name arises because such a theorem ensures that there is endless scope to keep discovering new techniques to improve the way at least some specific task is done.

For example, the full employment theorem for compiler writers states that there is no such thing as a provably perfect size-optimizing compiler, as such a proof for the compiler would have to detect non-terminating computations and reduce them to a one-instruction infinite loop. Thus, the existence of a provably perfect size-optimizing compiler would imply a solution to the halting problem, which cannot exist. This also implies that there may always be a better compiler since the proof that one has the best compiler cannot exist. Therefore, compiler writers will always be able to speculate that they have something to improve.
Summary: The Limits of Algorithms

- \( A_{\text{DFA}} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts input string } w \} \) Decidable
- \( A_{\text{CFG}} = \{ \langle G, w \rangle | G \text{ is a CFG that generates string } w \} \) Decidable
- \( A_{\text{TM}} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \} \) Undecidable
- \( \text{HALT}_{\text{TM}} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \} \) Undecidable

It's straightforward to use hypothetical \( \text{HALT}_{\text{TM}} \) decider to create \( A_{\text{TM}} \) decider
Summary: The Limits of Algorithms

- \( A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \} \)
  - Decidable
- \( A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates string } w \} \)
  - Decidable
- \( A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \)
  - Undecidable
- \( \text{HALT}_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \} \)
  - Undecidable
- \( E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \} \)
  - Decidable
- \( E_{\text{CFG}} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \} \)
  - Decidable
- \( E_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \} \)
  - Undecidable

How can we use a hypothetical \( E_{\text{TM}} \) decider to create \( A_{\text{TM}} \) or \( \text{HALT}_{\text{TM}} \) decider?
Reducibility: Modifying the TM

Thm: \( E_{TM} \) is undecidable

Proof, by contradiction:

• Assume \( E_{TM} \) has decider \( R \); use it to create decider for \( A_{TM} \):

\[
S = \text{“On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w:\n\]

First, construct \( M_1 \)

• Run \( R \) on input \( \langle M \rangle_1 \)

Note: \( M_1 \) is only used as arg to \( R \); we never run it!

• If \( R \) accepts, reject (because it means \( \langle M \rangle \) doesn’t accept \( w \))

• if \( R \) rejects, then accept (\( \langle M \rangle \) accepts \( w \))

• Idea: Wrap \( \langle M \rangle \) in a new TM that can only accept \( w \):

\[
M_1 = \text{“On input } x:\n\]

1. If \( x \neq w \), reject.
2. If \( x = w \), run \( M \) on input \( w \) and accept if \( M \) does.”

Input is \( w \), maybe accept

Input not \( w \), always reject

\( M_1 \) accepts \( w \) if \( M \) does
Reducibility: Modifying the TM

**Thm:** $E_{\text{TM}}$ is undecidable

**Proof,** by contradiction:

- Assume $E_{\text{TM}}$ has decider $R$; use it to create *decider* for $A_{\text{TM}}$:

  \[
  S = \text{"On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w:\n  \]

  - First, construct $M_1$
  - Run $R$ on input $\langle M_1 \rangle$
  - If $R$ accepts, reject (because it means $\langle M \rangle$ doesn’t accept $w$)
  - If $R$ rejects, then accept ($\langle M \rangle$ accepts $w$)

- **Idea:** Wrap $\langle M \rangle$ in a new TM that *can only* accept $w$:

  \[
  M_1 = \text{"On input } x: \]
  
  1. If $x \neq w$, reject.
  2. If $x = w$, run $M$ on input $w$ and accept if $M$ does."
Summary: The Limits of Algorithms

- $A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$
  Decidable
- $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates string } w \}$
  Decidable
- $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$
  Undecidable
- $E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \}$
  Decidable
- $E_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \}$
  Decidable
- $E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \}$
  Undecidable
- $EQ_{DFA} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$
  Decidable
- $EQ_{CFG} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}$
  Undecidable
- $EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$
  Undecidable
Reduce to something else: $EQ_{TM}$ is undecidable

$EQ_{TM} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Proof, by contradiction:

- **Assume**: $EQ_{TM}$ has decider $R$; use it to create decider for $A_{TM}$:

  \[
  E_{TM} = \{\langle M \rangle | M \text{ is a TM and } L(M) = \emptyset\}
  \]

  $S = \text{"On input } \langle M \rangle, \text{ where } M \text{ is a TM:}\$

  1. Run $R$ on input $\langle M, M_1 \rangle$, where $M_1$ is a TM that rejects all inputs.

  2. If $R$ accepts, accept; if $R$ rejects, reject."
Reduce to something else: $EQ_{TM}$ is undecidable

$EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$

Proof, by contradiction:

• **Assume**: $EQ_{TM}$ has decider $R$; use it to create **decider** for $E_{TM}$:

  $S = "\text{On input } \langle M \rangle, \text{ where } M \text{ is a TM:}\$
  
  1. Run $R$ on input $\langle M, M_1 \rangle$, where $M_1$ is a TM that rejects all inputs.
  2. If $R$ accepts, accept; if $R$ rejects, reject."

• But $E_{TM}$ is undecidable!
Summary: Undecidability Proof Techniques

- **Proof Technique #1:**
  - Use hypothetical decider to implement impossible \( A_{\text{TM}} \) decider
  - Example Proof: \( A_{\text{TM}} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \} \)

- **Proof Technique #2:**
  - Use hypothetical decider to implement impossible \( A_{\text{TM}} \) decider
  - But first modify the input \( M \)
  - Example Proof: \( E_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \} \)

- **Proof Technique #3:**
  - Use hypothetical decider to implement non-\( A_{\text{TM}} \) impossible decider
  - Example Proof: \( E_{Q_{\text{TM}}} = \{ \langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \} \)
Summary: Decidability and Undecidability

- $A_{DFA} = \{\langle B, w \rangle | B \text{ is a DFA that accepts input string } w \}$ Decidable
- $A_{CFG} = \{\langle G, w \rangle | G \text{ is a CFG that generates string } w \}$ Decidable
- $A_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$ Undecidable
- $E_{DFA} = \{\langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$ Decidable
- $E_{CFG} = \{\langle G \rangle | G \text{ is a CFG and } L(G) = \emptyset \}$ Decidable
- $E_{TM} = \{\langle M \rangle | M \text{ is a TM and } L(M) = \emptyset \}$ Undecidable
- $E_{DFA} = \{\langle A, B \rangle | A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$ Decidable
- $E_{CFG} = \{\langle G, H \rangle | G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}$ Undecidable
- $E_{TM} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$ Undecidable
Also Undecidable ...

- $\text{REGULAR}_{TM} = \{ <M> \mid M \text{ is a TM and } L(M) \text{ is a regular language} \}$
Theorem: $\text{REGULAR}_{\text{TM}}$ is undecidable

$\text{REGULAR}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is a regular language} \}$

Proof, by contradiction:

- **Assume:** $\text{REGULAR}_{\text{TM}}$ has decider $R$; use it to create decider for $A_{\text{TM}}$:

  $S = \text{“On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w:$
  
  - First, construct $M_2(\cdot)$
  - Run $R$ on input $\langle M_2 \rangle$
  - If $R$ accepts, accept; if $R$ rejects, reject

  **Want:** $L(M_2) =$
  - regular, if $M$ accepts $w$
  - nonregular, if $M$ does not accept $w$
Thm: \( \text{REGULAR}_{\text{TM}} \) is undecidable (continued)

\[ \text{REGULAR}_{\text{TM}} = \{ \langle M \rangle | M \text{ is a TM and } L(M) \text{ is a regular language} \} \]

\[ M_2 = \text{“On input } x: \]

1. If \( x \) has the form \( 0^n1^n \), accept.
2. If \( x \) does not have this form, run \( M \) on input \( w \) and accept if \( M \) accepts \( w \).

Want: \( L(M_2) = \)
- regular, if \( M \) accepts \( w \)
- nonregular, if \( M \) does not accept \( w \)

Always accept strings \( 0^n1^n \)
\( L(M_2) = \) nonregular, so far

If \( M \) accepts \( w \), accept everything else, so \( L(M_2) = \Sigma^* = \text{regular} \)
Also Undecidable ... 

- $\text{REGULAR}_{\text{TM}} = \{<M> | M \text{ is a TM and } L(M) \text{ is a regular language}\}$

- $\text{CONTEXTFREE}_{\text{TM}} = \{<M> | M \text{ is a TM and } L(M) \text{ is a CFL}\}$

- $\text{DECIDABLE}_{\text{TM}} = \{<M> | M \text{ is a TM and } L(M) \text{ is a decidable language}\}$

- $\text{FINITE}_{\text{TM}} = \{<M> | M \text{ is a TM and } L(M) \text{ is a finite language}\}$

Seems like no algorithm can compute anything about the language of a Turing Machine, i.e., about the runtime behavior of programs ...
An Algorithm About Program Behavior?

main()
{
    printf("hello, world\n");
}

Write a program that, given another program as its argument, returns TRUE if that argument prints “Hello, World!”

TRUE
Write a program that, given another program as its argument, returns `TRUE` if that argument prints "Hello, World!"
Also Undecidable ...

- $\text{REGULAR}_{\text{TM}} = \{ <M> \mid M \text{ is a TM and } L(M) \text{ is a regular language} \}$

- $\text{CONTEXTFREE}_{\text{TM}} = \{ <M> \mid M \text{ is a TM and } L(M) \text{ is a CFL} \}$

- $\text{DECIDABLE}_{\text{TM}} = \{ <M> \mid M \text{ is a TM and } L(M) \text{ is a decidable language} \}$

- $\text{FINITE}_{\text{TM}} = \{ <M> \mid M \text{ is a TM and } L(M) \text{ is a finite language} \}$

- ...

- $\text{ANYTHING}_{\text{TM}} = \{ <M> \mid M \text{ is a TM and “… anything …” about } L(M) \}$
Rice’s Theorem: $ANYTHING_{TM}$ is Undecidable

$ANYTHING_{TM} = \{<M> | M \text{ is a TM and ... } \text{anything} \ldots \text{about } L(M)\}$

• “... Anything ...”, more precisely:
  For any $M_1, M_2$,
  • if $L(M_1) = L(M_2)$
  • then $M_1 \in ANYTHING_{TM} \iff M_2 \in ANYTHING_{TM}$

• Also, “... Anything ...” must be “non-trivial”:
  • $ANYTHING_{TM} \neq \{\}$
  • $ANYTHING_{TM} \neq \text{set of all TMs}$
Rice’s Theorem: \( \text{ANYTHING}_{\text{TM}} \) is Undecidable

\( \text{ANYTHING}_{\text{TM}} = \{<M> \mid M \text{ is a TM and … anything … about } L(M)\} \)

Proof by contradiction

- **Assume** some language satisfying \( \text{ANYTHING}_{\text{TM}} \) has a decider \( R \).
  - Since \( \text{ANYTHING}_{\text{TM}} \) is non-trivial, then there exists \( M_{\text{ANY}} \in \text{ANYTHING}_{\text{TM}} \).
  - Where \( R \) accepts \( M_{\text{ANY}} \).
- **Use** \( R \) to create decider for \( A_{\text{TM}} \):

  **On input** \(<M, w>\):
  - **Create** \( M_w \):
    - \( M_w = \) on input \( x \):
      - Run \( M \) on \( w \)
      - If \( M \) rejects \( w \): reject \( x \)
      - If \( M \) accepts \( w \): Run \( M_{\text{ANY}} \) on \( x \) and accept if it accepts, else reject
  - **Run** \( R \) on \( M_w \)
    - If it accepts, then \( M_w = M_{\text{ANY}} \), so \( M \) accepts \( w \), so accept.
    - Else reject.

**Wait!** What if the TM that accepts nothing is in \( \text{ANYTHING}_{\text{TM}} \)?

These two cases must be different, (so \( R \) can distinguish when \( M \) accepts \( w \)).

Proof still works! Just use the complement of \( \text{ANYTHING}_{\text{TM}} \) instead!
Rice’s Theorem Implication

\{ \langle M \rangle \mid M \text{ is a TM that installs malware} \} \\
Undecidable! (by Rice’s Theorem)
No quiz 4/12!

On gradescope