CS3102: Theory of Computation

Class 18: Proving Undecidability

Spring 2010
University of Virginia
David Evans

Menu

- Revisiting the Halting Problem
  - Proof by Paradox
  - Universal Programming Languages
- Reduction Proofs
- Barbara Liskov’s Turing Award: CLU and Data Abstraction

Halting Problem

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

If \( \text{HALTS}_{TM} \) is decidable, there is some TM \( H(\langle M, w \rangle) \) that decides \( \text{HALTS}_{TM} \).

\[ X(\langle M, w \rangle) = \text{Simulate } H(\langle M, w \rangle). \text{ If it accepts, loop forever. If it rejects, accept.} \]

What is \( X(\langle X, w \rangle) \)?

Halting Problem is Undecidable

\[ X(\langle M, w \rangle) = \text{Simulate } H(\langle M, w \rangle). \text{ If it accepts, loop forever. If it rejects, accept.} \]

Paradox

If \( X(\langle X, w \rangle) \) halts, then \( H(\langle X, w \rangle) \) accepts, and \( X(\langle X, w \rangle) \) runs forever.
If \( X(\langle X, w \rangle) \) doesn’t halt, then \( H(\langle X, w \rangle) \) rejects, and \( X(\langle X, w \rangle) \) halts.

Suppose \texttt{halts} solves Halting Problem

```python
def halts(code):
    ... ? ...

>>> halts('3 + 3')
True
>>> halts('"
    i = 0
    while i < 100:
        i = i \ast 2""
')
False
```
def is_sum_of_two_primes(n):
    for a in range(2, n/2):
        for b in range(2, n/2):
            if a + b == n and is_prime(a) and is_prime(b):
                return True
    return False

i= 2
while is_sum_of_two_primes(i): i= i+ 1
return False

Goldbach Conjecture: Every even integer can be written as the sum of two primes. (Open problem since 1742.)

def paradox():
    if halts('paradox()'):
        while True:
            pass
    Does paradox() halt?
    Yes?: If paradox halts, the if test is true and it evaluates to an infinite loop: it doesn’t halt!
    No?: If paradox doesn’t halt, the if test is false and it finishes. It halts!

Universal Programming Language

Universal Turing Machine: a Turing machine that can simulate every other Turing machine
– Every algorithm can be implemented by a UTM

Universal Programming Language: a programming language that can simulate a Universal Turing Machine
– All real implementations have limits (can’t really simulate infinite tape), but all common PLs are effectively universal

Proofs of Undecidability

To prove a language is undecidable, need to show there is no Turing Machine that can decide the language.

This is hard: requires reasoning about all possible TMs.

Proof by Reduction

0. We know X does not exist.
   (e.g., X = a TM that can decide A_{TM})

1. Assume Y exists.
   (e.g., Y = a TM that can decide B)

2. Show how to use Y to make X.

3. Contradiction: Since X does not exist, but Y could be used to make X, then Y must not exist.

Reduction Proofs

A reduces to B
means

Y can be used to make X
that can decide B

Hence, A is not a harder problem than B.

The name “reduces” is confusing: it is in the opposite direction of the making.
Converse?

A reduces to B

\[ Y \quad \text{that can solve B} \quad \text{can be used to make} \quad X \quad \text{that can solve A} \]

A is not a harder problem than B.

Does this mean B is as hard as A?

No! Y can be any solver for B. X is one solver for A. There might be easier solvers for A.

Reduction Pitfalls

- Be careful: the direction matters a great deal
  - To show \( L_B \) is at least as hard to decide as \( L_A \), we need to show that a machine \( M_B \) that decides \( L_B \) could be used to build a machine \( M_A \) that decides \( L_A \).
  - To show equivalence, need reductions in both directions.
- You can’t assume anything about \( M_B \) other than it decides \( L_B \).
- The construction of \( M_A \) must involve only things you know you can do: otherwise the contradiction might be because something else doesn’t exist.

What does can do mean here?

Halting Problem is Undecidable

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

Proof by Contradiction.
Assume \( \text{HALTS}_{TM} \) is Turing-decidable.
There is some TM \( H \) that decides \( \text{HALTS}_{TM} \).
We could use \( H \) to construct a TM \( A \) that decides \( A_{TM} \):

\[ A(\langle M, w \rangle) = \text{simulate } H(\langle M, w \rangle). \]

If \( A \) accepts, simulate \( M \) on input \( w \) and output the result. Otherwise, reject.

Since we know \( A_{TM} \) is undecidable, we know the machine \( A \) that decides \( A_{TM} \) must not exist. But, if \( M \) that decides \( \text{HALTS}_{TM} \) exists, we could construct \( A \). Thus, \( M \) must not exist and \( \text{HALTS}_{TM} \) must be undecidable.

What are \( L_B, L_A, M_B, M_A \)?

Reduction =

Proof by Contradiction and Construction

Assume \( M_B \) is a TM that decides \( L_B \).
Do a construction using \( M_B \) to build \( M_A \), a TM that decides \( L_A \).
Since \( L_A \) is undecidable, \( M_A \) cannot exist.
We have reached a contradiction, so (as long as nothing else is questionable) our assumption must be wrong.

\[ L_B = \text{HALTS}_{TM}, M_B = H \quad \text{(TM that decides halt)} \]
\[ L_A = A_{TM}, M_A = A \]
**Reduction Proof**

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

\( HALT_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on } w \} \)

**Proof by Reduction.** We prove \( HALT_{TM} \) is undecidable by showing \( A_{TM} \) (which is known to be undecidable) reduces to \( HALT_{TM} \). Suppose \( H \) is a TM that decides \( HALT_{TM} \). Then, we can construct a TM that decides \( A_{TM} \):

\[
A(H, w) = \text{simulate } H(\langle M, w \rangle). \text{ If it accepts, simulate } M \text{ on input } w \text{ and output the result. Otherwise, reject.}
\]

**Equivalence of Machines**

\( EQ_{DT} = \{ \langle D, M \rangle \mid D \text{ is a description of a DFA, } M \text{ is a description of a TM, and } L(D) = L(M) \} \)

Is \( EQ_{DT} \) decidable?

**Relation Proof Assumption**

Suppose \( M_{EQ} \) decides \( EQ_{DT} \). Can we use \( M_{EQ} \) to decide \( HALT_{TM} \)?

\( EQ_{DT} = \{ \langle D, M \rangle \mid D \text{ is a description of a DFA, } M \text{ is a description of a TM, and } L(D) = L(M) \} \)

\( HALT_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on } w \} \)

**Reduction Proof Construction**

\( D \) is a DFA only accepts \( w \)

\( M' \) is a TM that decides \( HALT_{TM} \)

\( M_{EQ} \) is a TM that decides \( EQ_{DT} \)

\( M_{H} \) is a TM that decides \( HALT_{TM} \)

\( M_{H}(\langle M, w \rangle) = \text{simulate } M_{EQ}(\langle D, M' \rangle) \)

where \( D \) describes a DFA that accepts all strings and \( M' \) is a Turing Machine that writes \( w \) as the input and then simulates \( M \). If it accepts or rejects, accept.
**EQ\text{DT} Is Undecidable**

If we had a TM that decides \( EQ\text{DT} \), we could use it to do something we know is impossible: build a TM that decides \( \text{HALTS}_{\text{TM}} \).

---

**Empty Language**

\[
\mathcal{E}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a description of a TM, and } \mathcal{L}(M) = \emptyset \}
\]

---

**Proving Undecidability**

![Diagram showing the process of reducing \( \text{HALTS}_{\text{TM}} \) to \( \mathcal{E}_{\text{TM}} \)]

---

**Reducing \( \text{HALTS}_{\text{TM}} \) to \( \mathcal{E}_{\text{TM}} \)**

\( M_H(\langle M, w \rangle) = \) A TM that constructs \( M' \), a Turing Machine that:

1. Checks if the input is \( w \). If not, reject.
2. Simulates \( M \) except with all \( q_{\text{reject}}'s \) in the output of \( M \)'s \( \delta \) function replaced by \( q_{\text{accept}} \).

Then, it simulates \( M_E(\langle M' \rangle) \). If it accepts, reject; if it rejects, simulate accept.

---

**Reducing \( A_{\text{TM}} \) to \( \mathcal{E}_{\text{TM}} \)**

\( M_A(\langle M, w \rangle) = \) A TM that constructs \( M' \), a Turing Machine that:

1. Checks if the input is \( w \). If not, reject.
2. Simulates \( M \).

Then, it simulates \( M_E(\langle M' \rangle) \). If it accepts, reject; if it rejects, simulate accept.

---

**SQUARE**

\[
\text{SQUARE}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a description of a TM, and } \mathcal{L}(M) \subseteq \text{SQUAREFREE} \}
\]

---

If a problem is undecidable, any undecidable problem can be reduced to it. (But not all choices are as simple and elegant.)
SQUARE: Valid Proof?

Not a valid proof. The reduction is in the wrong direction!

Rice’s Theorem

Henry Gordon Rice, 1951

Any nontrivial property about the language of a Turing machine is undecidable.

“Nontrivial” means the property is true for some TMs, but not for all TMs.

Generalizing Rice’s Theorem

Any nontrivial property about the language of a Turing machine is undecidable.

Any nontrivial property about the execution of any universal computing system is undecidable.

Rice’s Theorem: Proof Sketch

$P(\langle M \rangle)$ where $\langle M \rangle$ is a description of a Turing machine, and $P$ is a non-trivial property of a language.

Assume $P$ is decidable. Then, there exists a machine $M_p$ that decides $P$.

Since $P$ is non-trivial, there exists a machine $M_1$ such that $P(\langle M_1 \rangle)$ is true.

$R(\langle M, w \rangle) = \text{Simulate } M_p(\langle M' \rangle) \text{ and output the result where } M' \text{ is a TM that:}$

1. Simulates $M$ on $w$.
2. Cleans the tape and resets.
3. Simulates $M_1$.

What are we assuming about $P$?

Rice Hall

Which of these are Undecidable?

- Does TM $M$ accept any strings?  Undecidable
- Does TM $M$ accept all strings?  Undecidable
- Does TM $M$ accept “Hello”?  Undecidable
- Does TM $M_1$ accept more strings than TM $M_2$?  Undecidable
- Does TM $M$ take more than 1000 steps to process input $w$? Decidable

Note: for PS5 problems 2 and 4, you may use Rice’s theorem to get an intuition about the right answer, but cannot use it for your proof.
Type Safety

\[
\text{WELLTYPED}_{\text{Python}}(\{P\}) = \{ P \text{ is a Python program that cannot produce a type error} \}
\]

```python
>>> s = "hello"
>>> s + 3
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    s + 3
TypeError: Can't convert 'int' object to str implicitly
```

Not decidable: very sketchy proof:
\[\text{halts}(P) = \text{not wellTyped}('\text{removeTypeErrors}(P); s = \text{"hello"}; s + 3')\]

Type Safety

\[
\text{WELLTYPED}_{\text{Java}}(<P>) = \{ P \text{ is a Java program that does not use type casts or array assignments and } P \text{ never produces a run-time type error} \}
\]

This is decidable: your Java compiler should do this (and should always terminate)!

Well-Typed Java?

```java
public class Test {
    static public void main(String args[]) {
        String s;
        s = "Hello";
        s = s + 3;
        System.out.println("s = " + s);
    }
}
```

```java
s = "Hello";
System.out.println("s = " + s);
```

WELLTYPED_{CLU}(<P>) = \{ P \text{ is a CLU program and } P \text{ never produces a run-time type error} \}

CLU Type Safety

```java
public class Test {
    static public void main(String args[]) {
        String s;
        s = "Hello";
        s = s + 3;
        System.out.println("s = " + s);
    }
}
```

```java
javac Test.java
java Test
s = Hello3
```

Well-Typed Java?

```java
javac Test.java
java Test
Test.java:5: operator - cannot be applied to java.lang.String, int
```

Thursday's Class

2008 Turing Award Winner
(Computer Science's Nobel Prize)

Barbara Liskov
Security of Internet Storage

4.1.2010
Chem Auditorium
2pm

LiskovAtUVa.com