The Halting Problem

Mark Greenstreet, CpSc 421, Term 1, 2006/07

- Halting for Java Programs
- Implications and Turing Machines

The Halting Problem

- Let J be a Java program, does J halt when we run it or does it go on forever (in an infinte loop, inifinite recursion, etc.)?
- Maybe J halts for some inputs and not for others.
  Does J halt when run with input I?
  - Does the input include mouse events? communication with other processes? communication with other computers? ...
  - Let's keep it simple.
    The program reads from `stdin`. The input is finite (i.e. eventually there is an EOF).
  - Thus, the input is a string.
- How can we describe J|
  - Does J consist of multiple modules?
  - Again, we'll go for simplicity. J will be one module with one class. The only other classes that it uses are the standard ones from `java.lang`.
  - Thus, the program is a string.

A halt method

- We want:

```
boolean halt(String J, String I) {
   // return true if program J halts when run with input I;
   // return false otherwise.
   ...
}
```

- Note things that we can check with a compiler:
  - Syntax errors.
  - Undefined variables, classes, methods.
  - Type mismatches.
  - Some cases of uninitialized variables.
  - Wouldn't it be nice to detect infinite loops?

Let's say we could write halt()

```
// Given an array of programs, JA, find the first one that halts
// if run on input I and return its index.
// If there is no such program, return -1.
int firstGoodOne(String[] JA, String I) {
   for(int k = 0; k ¡ JA.length; k++) {
      if(halt(JA[k], I))
         return(k);
   }
   return(-1);
}
```

Another program that uses halt()

```
boolean contrary(String J, String I) {
   if(halt(J, I))
      while(true); // go into an infinite loop
   return(true);
}
```

- This program does the opposite of what its arguments would do:
- If J would halt on input I, then contrary loops forever.
- On the other hand, if J runs forever, then contary halts.

From contrary to turing

```
boolean turing(String X) {
   if(halt(X, X))
      while(true); // go into an infinite loop
   return(true);
}
```

- If X halts when run with its own source code as input, then turing loops forever.
- On the other hand, if X runs forever, then turing halts.

Self Reference

Why would a program have itself as input?

- Compilers:
  - Often, the first compiler for a language is written in some other language. E.g. the first Java compilers were written in C.
  - Once the early compilers are working, subsequent compilers are typically written in their own language:
    - javac is written in Java.
    - gcc is written in C.
  - This makes upgrades easier – if you're interested in working on a better C compiler, you're probably interested in C, and probably already have a C compiler handy.
- Theory of computation:
  - This is an example of self-reference.
  - Self-reference plays a central role in computer science, and is the key to several of the most profound intellectual discoveries of the 20th century.

Back to Turing

```
boolean turing(String X) {
   if(halt(X, X))
      while(true); // go into an infinite loop
   return(true);
}
```

- Let T be the string for the program shown above.
- What happens if we invoke the turing method passing it T as its parameter?
  - This is running T with T as its input.
  - If (halt(T, T)), then .
  - Otherwise, ¬(halt(T, T)), then .
- We've shown that halt cannot be written!

Halting Recap

- For the sake of contradiction, assume that halt(String J, String I) is a function that returns true if program J halts when run with input I.
- Write

```
boolean turing(String X) {
  if(halt(X, X))
    while(true); // go into an infinite loop
  return(true);
}
```

  Let T be the string for the source code of this program (including the source code for halt, etc.).
- Consider what happens if we run T with T as its input string.
  Whether halt(T, T) is true or false, we get a contradiction.
- Thus, our assumption that we could write halt must be wrong (we can definitely write turing).
- $\therefore$ It is not possible to write halt().

Implications

- We'll show that the result for halting can be used to show that it is impossible to decide any dynamic property of program behavior:
  - Does the program throw an exception?
  - Does the program ever execute a particular line of code?
  - Does the program compute the “right” answer?
  - Is the program a virus?
- This doesn't mean the are no programs for which we can decide these things. For example:

```
public static void main(String[] args) {
  System.out.println("hello world"); }
```

  definitely halts.
- What this does mean is that we can't implement tests for these things that work for all programs.

Formalizing these ideas

- Is the undecidability of halting a quirk of Java programs, or do other languages have the same limitation?
- Did it matter that we restricted the program and our notion of input?
- Wait a second! Any real computer has some limited amount of memory. Therefore, any real program is a finite automaton. Can't we figure out these things about a finite automaton?
- We'll take on each of these objections in the next few weeks.

Turing Machines

- A Turing Machine (TM) is a very simple computer.
- A TM has a tape and a finite automaton.
- The tape is infinitely long.
  - The tape initially holds the input.
  - There is a special tape symbol □ (blank).
  - Initially, all the tape after the input is an infinite string of □'s.
- At each step, the finite automaton
  - reads the symbol on the tape;
  - based on the symbol and the state of the finite automaton, the TM
    - writes a symbol on the current square;
    - moves to a new state;
    - moves the tape head one square (the head can move either left or right).

What’s next?

- We’ll study TMs.
  - Any reasonable model for computation can be simulated by a TM. This includes Java programs, C programs, etc.
  - It’s a convenient approximation to assume that typical programming languages allow an infinite amount of memory. With this assumption, most programming languages can simulate a TM.
- The halting problem is undecidable for TMs.
  - We’ll look at many implications of this.
  - TMs give us an simple framework in which to develop these results.
  - The results apply to more common formulations of computation.