CS109B Notes for Lecture 6/7/95

Unsolvable Problems

- Some problems have "efficient" solutions, *i.e.*, they have algorithms that run in time polynomial in the length of input.
  - □ Examples: testing whether a propositional formula is true under a truth assignment, a graph is bipartite
- NP-complete problems are "intractable"; there seem to be no efficient algorithms for them
  - □ Examples: testing whether a propositional formula is satisfiable, a graph is tripartite
- Some problems are "unsolvable"; there *cannot exist any* algorithms for them !
  - □ Examples: stick around

Algorithms are ML functions

We can represent the inputs for a problem as a string in ML. We can then write an algorithm for the problem as an ML function of type `string -> ...`

Example: We can easily represent propositional formulas as strings.

Then, we can write the algorithm for satisfiability as an ML function

```
Sat : string -> bool
```

with `Sat` returning `true` if the formula is satisfiable and `false` otherwise.

Church's Thesis: Every algorithm can be programmed as an ML function.

Thus a problem is solvable only if we can write an ML function to solve the problem. Conversely, to prove that a problem is unsolvable it suffices to show that there is no ML function that would solve the problem.

Halting Problem

Given the definition of an ML function `f` of type `string -> bool` and an input string `s` for `f`, does `f` halt on argument `s`?

Question: Does there exist an algorithm to solve the halting problem? Equivalently, can we write an ML function

$$\texttt{HaltTester} : \texttt{string} * \texttt{string} -> \texttt{bool}$$

such that for strings `p`, `s`

- If `p` is a valid definition of an ML function `f:string->bool` then
  - □ `HaltTester(p,s)` returns `true` if `f(s)` halts
  - □ `HaltTester(p,s)` returns `false` if `f(s)` goes into an infinite loop
- If `p` is not a valid definition of an ML function, `HaltTester(p,s)` returns `false`

Self Reference

We can apply an ML function `f:string -> bool` *to itself* in the following sense.

Suppose the definition of `f` is

```
fun f(s) = ...
```

Then `"fun f(s) = ..."` is just a string and thus we can write

```
f "fun f(s) = ... "
```

Nothing strange about this. For example, if

```
fun len(s) = length(explode(s))
```

then

```
len "fun len(s) = length(explode(s))"
```

returns `31`

Diagonalization

Assume we can write an ML function

```
fun HaltTester(p,s) = ...
```

Then we can write the following ML function

```
fun weird(s) = if HaltTester(s,s)
               then loop(s)
               else true
```

where

```
fun loop(s) = loop(s)
```

What does `weird` do?

- If `f` `"fun f(s) = ..."` halts then
  `weird "fun f(s) = ..."`
  goes into an infinite loop
- If `f` `"fun f(s) = ..."` goes into an infinite loop then
  `weird "fun f(s) = ..."`
  returns `true`, *i.e.*, it halts

Why `weird` is weird
Consider what happens if we apply `weird` to itself, *i.e.*,

```
weird "fun weird (s) = ... "
```

This either halts or goes into an infinite loop

- If it halts then
  `weird "fun weird (s) = ... "`
  is supposed to go into an infinite loop `#%!&`
- If it goes into an infinite loop then
  `weird "fun weird (s) = ... "`
  is supposed to halt `#%!&`

Therefore, our assumption that there is an ML function `HaltTester` must be wrong!

That is, *the halting problem is unsolvable.*

Reductions
Can now show that other problems are unsolvable by reducing the halting problem to them.

Example: Given the definition of an ML function `f:string->bool`, does `f` halt on all inputs?

Suppose this problem was solvable, *i.e.*, we can write a function

```
AllHalt : string -> bool
```

such that `AllHalt "fun f ..."` returns `true` if `f` halts on all strings and `false` otherwise.

Then, we can solve the halting problem. Suppose we are given the definition of a function `"fun f ... "` and a string `s` and we want to know whether `f` halts on `s`. Then, we can construct the function

```
fun g(x) = let fun f ... in f(s) end
```

and run `AllHalt` on `g` and return the same answer.

Other Unsolvable Problems

- Given an ML function `f`, is there any input on which `f` halts?
- Given two grammars G_1, G_2, are their languages the same?
- Given a grammar G, is its language regular?

Class Problem
Are the following problems solvable?

- Given a boolean ML expression `b` (represented as a string), does `b` halt when executed? Given an arbitrary ML expression `e` does `e` halt when executed?
- Given two ML expressions `f` and `g` of type `string -> bool`, do `f` and `g` compute the same function (*i.e.*, are they algorithms for solving the same problem)?

Final Thought
Suppose we wanted to invent a language PerfectML such that all the programs that we could write in PerfectML always halted. At the same time, PerfectML should be powerful enough to be able to program all computable functions in it. Unfortunately, *this is impossible!*

PerfectML is useful only if we have an algorithm to execute its programs on inputs. For example, we should be able to write an ML function

```
execute : string * string -> bool
```

which given the definition of a PerfectML function `f:string->bool` written as a string and an input `s` returns the result of running `f(s)` in PerfectML.

Then, we can write the following function in ML

```
fun diag(s) = not(execute(s,s))
```

and the function computed by `diag` would not be programmable in PerfectML.

Good luck for the final exam. You can rest assured that it will be solvable!