CS61A Lecture #28: The Halting Problem and Incompleteness

- An interpreter (or compiler) is a program that operates on programs.
- In fact, there are numerous other ways to operate on programs. For example,
  - Given a one-parameter function in some language, produce the function that computes its derivative.
  - Given a C program, add statements that check for memory index bounds errors.
- The development of program-analysis tools of this sort is an active research area.

The Halting Problem

- For example, would be very useful to know "Is there some input to Scheme function P that will cause it to go into an infinite loop?" Is there a program that operates on programs that will answer this question correctly in finite time?
- This question was answered negatively in the 1930s by Alan Turing. In fact, there isn't even a program that fully meets the following specification:

```
;; True iff DEFN is a Scheme definition that defines a one-argument
;; function that eventually halts given the input X.
(define (halts? defn x) ...)
```

Biting Your Tail: Proof of Impossibility

```
(define (halts? defn x) alleged definition of halts?)
(define halts?-bogus-program
  (quote (define (halts?-bogus x)
           (define (halts? defn x) alleged definition of halts?)
           (define (loop) (loop))
           (if (halts? x x) (loop) #t))))
(halts? halts?-bogus-program halts?-bogus-program) ; (*)
```

- Assume that `halts?` works as specified: `(halts? defn y)` returns true if `defn` is a Scheme definition of some one-argument function that halts (does not loop) when given input `y`.
- Then if the line marked (∗) returns true, it is supposed to mean that `(halts?-bogus halts?-bogus-program)` halts.
- But `halts?-bogus` computes `(halts? x x)` during its execution, with the value of `x` being `halts?-bogus-program`.
- That would presumably return `true`, which would make `halts?-bogus` loop infinitely.
- So clearly, if `halts?` works, line (∗) cannot return true after all; it must return false.

Biting Your Tail (II)

```
(define (halts? defn x) alleged definition of halts?)
(define halts?-bogus-program
  (quote (define (halts?-bogus x)
           (define (halts? defn x) alleged definition of halts?)
           (define (loop) (loop))
           (if (halts? x x) (loop) #t))))
(halts? halts?-bogus-program halts?-bogus-program) ; (*)
```

- But if the line marked (∗) returns false, then the execution of `halts?-bogus` would terminate, which would mean that `halts?` had gotten the wrong answer.
- The only way out is to conclude that `halts?` never returns in this case—it does *not* answer the question for all possible inputs.
- Putting it all together, we must conclude that

  No possible definition of `halts?` works all the time.

Not Just a Trick

- Nothing in this argument is specific to Scheme.
- Furthermore, Scheme is capable of representing any "effectively computable" function on symbolic data (i.e, computable via some finitely describable algorithm that terminates).
- Therefore, the impossibility of the halting problem is fundamental: the `halts?` function is uncomputable.
- If `halts?` always returns a correct result (when it returns), then there must be an *infinite number* of inputs for which it fails to give any answer at all (i.e., loops infinitely). Why infinite?

Consequences

- There's a lot of fallout from the impossibility of writing `halts?`.
- For example, I cannot tell in general whether two programs compute the same thing. [Why not?]
- Therefore,

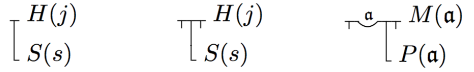
  Perfect anti-virus software is theoretically impossible.

  Anti-virus software must either miss some viruses, or prevent some innocent programs from running (or freeze your computer.)
- Many analyses that might be useful cannot be done in general. For example, even if I know that a given program will terminate, I cannot necessarily predict in general how long it will take to do so.

The Mathematics of Mathematics

Gottlob Frege (1879) is usually credited with introducing the first modern *formal system* for expressing mathematical and logical statements and arguments. He was attempting to put mathematics on a firm foundation—to make it clear when a proof was a proof, for example.

Frege invented a universal *syntax* for expressing mathematical statements. Examples (with modern notation underneath):



$S(s) \to H(j)$ $\quad$ $S(s) \& H(j)$ $\quad$ $\neg\forall a(P(x) \to \neg M(a))$ or $\exists a(P(a) \& M(a))$

Formal Systems

- A formal system then consists of a set of symbols that are supposed to have meanings (constants, functions, predicates), plus a finite set of *axioms* (like $\forall x, y. x + y = y + x$), *axiom schemas* (templates for axioms, like $A \wedge B \Rightarrow A$), and mechanical *inference rules.*
- Creation of formal systems turned out to be tricky:
  - **Russell's Paradox:** Frege's original system allowed the definition (in effect) of $S = \{x | x \notin x\}$, the set of everything that is not a member of itself.
  - This is a highly problematic set! Can prove both that $S \in S$ and $S \notin S$.
  - Therefore, Frege's system was *inconsistent*, which is bad.
- Fortunately, a syntax such as Frege's is very well defined; sentences and proofs are themselves mathematical objects. So, perhaps we can build a mathematics of mathematics ("metamathematics") and within it prove our that formal systems are consistent: *Hilbert's Program*.

From Syntax to Semantics

- Notations like these provide notation (*syntax*) without meaning (*semantics*), ...
- ...except for a few key symbols with fixed meanings:
  - Logical connectives, such as '&', '¬', '→'.
  - Quantifiers, such as '∀' (for all), '∃' (there exists), and the variables they apply to (but we don't say what set ("*domain*") they quantify over.)
  - (Sometimes) the predicate '='.
- But otherwise, the functions and predicates (true/false functions) are *uninterpreted.*
- When we restrict ourselves so that we cannot apply ∀ or ∃ to function symbols or predicates, we have what is called the *first-order predicate calculus.* Basically all of standard mathematics uses it.
- But what good is it? How can we get meaningful information by just manipulating meaningless symbols?

Meaning from Assertions

- Even if we can't say exactly what a symbols means, we *can* assert various sentences about it that *constrain its possible meanings.*
- For example, suppose that, besides the standard logical connectives, quantifiers, and =, we allow *only* the relation predicate $\leq$.
- If we say nothing else, $\leq$ could mean anything.
- But suppose we assert a few things:

$$\forall x, y(x \leq y \vee y \leq x)$$
$$\forall x, y(x \leq y \ \& \ y \leq x \to x = y)$$
$$\forall x, y, z(x \leq y \ \& \ y \leq z \to x \leq z)$$

- This restricts the possible meanings of $\leq$ to total orderings.
- Certain other things must now be true. E.g., $\forall x(x \leq x)$.
- But there are additional statements involving only $\leq$ whose truth is not so constrained. Example? $\exists y \forall x(y \leq x)$
- For our "theory of $\leq$", it is possible to add additional axioms to eliminate all such *independent* statements. Is this always possible?

Proofs

- **Big Idea:** If we can add enough constraints to get the properties we want for our symbols, we can dispense with messy meanings (semantics) and do everything by manipulations of syntax (e.g., which we could represent as operations on Scheme expressions).
- We call these constraining assertions
  - *Axioms:* (e.g, $\forall x, y(x \leq y \vee y \leq x)$)
  - *Axiom schemas:* templates standing for an infinite number of axioms, such as $\mathcal{A} \ \& \ \mathcal{B} \to \mathcal{A}$.
- A *proof* of a statement, A, is defined as a finite sequence of finite statements ending with A such that each statement is either
  - An axiom (like $\forall x, y. x + y = y + x$), or an instance of an axiom schema (like $x < y \wedge y < z \Rightarrow x < y$, which is the result of plugging $x < y$ and $y < z$ into $A \wedge B \Rightarrow A$);
  - The result of applying one of a few *inference rules* to preceding statements in the proof. Most well-known inference rule is *modus ponens:* can add D to a proof if there are preceding statements C and $C \Rightarrow D$. Usually don't have too many other rules.

Proofs (II)

- The set of axioms and schemas is finite, and a program can tell if it is looking at an axiom.
- Likewise, the inference rules must be finite and algorithmically checkable.
- Given an alleged formal proof, it is a *purely clerical task* to determine that it actually *is* a proof.
- A mathematician's secretary or a program can make this determination.
- Furthermore, if a proof of A exists, can find it in finite (albeit enormous) time by generating and checking all possible proofs.

Gödel Numbers

- Formulas and proofs in a formal system are just finite sequences of symbols from some finite alphabet. So are programs.
- We can encode any sequence of symbols as an integer in many ways. For example, produce a mapping like

```
'a' => 01, 'b' => 02, ..., '0' => 53, ...,  '+' => 63, '*' => 64, ...
```

  and then, e.g., encode `"a*c"` as 016403.
- Such an encoding is called a *Gödel numbering* of the formulas, proofs, programs, or other symbol string.
- Why is this interesting? It allows us to do symbol manipulation with arithmetic. In fact, it allows us to write and prove theorems about symbols, logical statements, proofs, and programs using the theory of integers.

Incompleteness

- Using nothing but the standard arithmetical operators, logical symbols, and free integer variables p, x, and k, can write a sentence, call it $\mathcal{H}_{p,x,k}$, that means "the program represented by Gödel number p, when given the input x, finishes running in k steps." (It's not difficult, but really tedious; take my word for it).
- So the formula $\exists k.\mathcal{H}_{p,x,k}$ means "program p halts given input x."
- If we can prove this formula, we have shown that program p halts, and if we can prove $\neg\exists k.\mathcal{H}_{p,x,k}$, we have shown that p does not halt.
- But I said in a previous slide that if there is a proof of a statement, a program can find it. So by writing a program that, given x and p, tries to prove both $\exists k.\mathcal{H}_{p,x,k}$ and $\neg\exists k.\mathcal{H}_{p,x,k}$, we could solve the halting problem (the program would generate all possible proofs and check each one to see if it proved one of the two sentences.)
- But the halting problem is unsolvable. Therefore:

  There must be values of p and x such that neither $\exists k.\mathcal{H}_{p,x,k}$ nor $\neg\exists k.\mathcal{H}_{p,x,k}$ can be proven.

The Incompleteness Theorem

- This result is a weak form of Gödel's (First) Incompleteness Theorem (1931). Any consistent mathematical system that includes the theory of the integers must contain an infinite number of *undecidable* propositions where neither the proposition nor its negation have a proof.
- Two big questions surround these formal systems we've been talking about:
  - Are they *consistent:* Is what they purport to prove true?
  - Are they *complete:* Can all the true things be proven?
- Consistency allows us to have faith in our proofs. Completeness allows us to rely on proof exclusively.
- The incompleteness theorem might seem to say that the latter is impossible.

Completeness

- But now things get really strange.
- The year before Gödel proved the first of his incompleteness theorems, he proved the Completeness Theorem:

  Any valid logical sentence is provable.
- But one of $\exists k.\mathcal{H}_{p,x,k}$ and $\neg\exists k.\mathcal{H}_{p,x,k}$ has to be true, so how can they both be unprovable?
- There is but one way out: "valid" doesn't mean what we think.
- A sentence is valid if it is true for all *models:* all choices of what set of values ("domain") $\forall x$ covers and all interpretations of its "non-built-in" symbols (e.g., $\leq$, +, -, *, 0, etc.) that satisfy the axioms.
- So a statement can be true in one model and yet not be valid if it is false under a different model.
- So perhaps it is *not* that we can't know whether some statements are true so much as that we can *choose* whether we want them to be true, by selecting the right model.
- Or, perhaps, that first-order predicate calculus cannot express all the axioms we have in mind when we think of the integers.

Nonstandard Models

- To choose a model (or rather to "unchoose" some other models), we add axioms to our system, narrowing down the possible models.
- Sometimes (as with our "theory of $\leq$"), we can narrow things down to the point where all statements are either provable or disprovable. These systems are *complete.*
- Gödel's result, however, tells us that when a system becomes powerful enough (specifically, when it encompasses enough of the theory of the integers), it is no longer possible to complete it in this fashion, except by adding contradictory axioms that make our system *inconsistent.* (At which point, all statements are provable, which is useless.)
- One implication:

  There must be non-standard models of arithmetic—interpretations in which there are integers other than the familiar 0, 1, 2, ...