61A Extra Lecture 12

~~Announcements~~

Quines

Quine: A Program That Evaluates to Itself

Self-evaluating expressions evaluate to themselves

Some expressions evaluate to themselves by constructing expressions

(Demo)

Church-Turing Thesis

An abbreviated history:

- Hilbert asks whether some process can separate mathematical truths from falsehoods
  - Originally a question about whether certain polynomials had rational solutions
  - Expanded to include whether logical conclusions could be derived from axioms
- Various scholars attempt to formalize the intuitive notion of a computational process
- Church and Kleene propose lambda calculus and derive many results about integers
- Turing describes a hypothetical machine (now called a Turing machine) &
  - Asserted that it could compute anything given the right configuration (program)
  - Proved that it could compute exactly the same set of functions as lambda calculus
  - Defined a universal Turing machine that could simulate any other Turing machine

Church Turing thesis (1936). Turing machines can do anything that can be described by any physically harnessable process of this universe. (Equivalently for lambda calculus)

The Church-Turing thesis is a not a theorem, but instead a claim that has withstood time

Self-Replicating Programs are a Feature of Programming Languages

Hartley Rogers (1967): If **F** is a total computable function, it has a fixed point

F has a fixed point if for some input **e**, **F**(**e**) is equivalent to **e**

Interpretation: **e** is a program and **F** is some transformation on programs

E.g., **F(e)** prints out the source code for **e**

Since **F** has a fixed point, there must be some **e** that prints out the source code of **e**

This result was proved about functions on integers, but every function was given a number and the proof involves calling a function on itself

Self-Reference

The name "Quine" was given by Douglas Hofstadter in honor of mathematician W. V. O. Quine

Self-reference turned out to be important in the history of mathematics

Hilbert proposed three questions (as summarized by Stephen Hawking):

1. To prove that all true mathematical statements could be proven, that is, the **completeness** of mathematics.
2. To prove that only true mathematical statements could be proven, that is, the **consistency** of mathematics,
3. To prove the **decidability** of mathematics, that is, the existence of a decision procedure to decide the truth or falsity of any given mathematical proposition."

The outcome hinged on self-referential mathematical statements, akin to Quine's paradox:

"Yields falsehood when preceded by its quotation" yields falsehood when preceded by its quotation.

Halting Problem

Computability

An interpreter can simulate any other program, but can it determine its properties?

We can (define (eval exp env) ...) and (define (apply procedure args) ...), so can we

(define (length exp) ...) that computes the length of an expression?

(define (errors? procedure arg) ...) that returns whether (procedure arg) would error?

(define (halts? procedure arg) ...) that returns whether (procedure arg) would terminate?

The Halting Problem

Exercise (from SICP). Given a one-argument procedure **p** and an object **a**, **p** is said to *halt* on **a** if evaluating the expression (p a) returns a value (as opposed to terminating with an error message or running forever). Show that it is impossible to write a procedure **halts?** that correctly determines whether **p** halts on **a** for any procedure **p** and object **a.** Use the following reasoning: If you had such a procedure **halts?,** you could implement the following program:

```
(define (run-forever) (run-forever))

(define (try p)
  (if (halts? p p)
      (run-forever)
      'halted))
```

Now consider evaluating the expression (try try).

The Halting Problem

Show that it is impossible to write a procedure **halts?** that correctly determines whether **p** halts on **a** for any procedure **p** and object **a.**

```
(define (run-forever) (run-forever))

(define (try p)
  (if (halts? p p)
      (run-forever)
      'halted))
```

If (try try) halts, then (halts? p p) must return false, so (halts? try try) is false!

If (try try) runs forever, then either:

- (halts? p p) returns true, so (halts? try try) is true!
- (halts? p p) runs forever, which means halts? doesn't return the answer we desire

If (try try) errors, then it must be because (halts? p p) errors, which is also wrong

Decidability

The Halting Problem is Undecidable

There is no Scheme procedure that can compute whether an arbitrary Scheme procedure halts

This property holds for all programming languages that can compute anything

If you can build a Scheme interpreter in a language, then it can compute anything

Problems (such as halts?) that cannot be computed are called *undecidable*

Error Detection

Assume we (define (errors? procedure arg) ...), which takes a procedure and its argument and returns whether or not evaluating (procedure arg) would cause an error.

```
(define (halts? p a)
  (and (not (errors? p a))
       (errors? (lambda (x) (p x) (/ 1 0)) a)))
```

Because defining **errors?** would allow us to define **halts?,** we know it is impossible to define **errors?.**

Undecidable problems:

- Does a procedure terminate on all inputs?
- Is a sub-expression within a procedure ever executed?
- What is the shortest program that is equivalent to some procedure?
- Are two procedures equivalent?