COMPSCI 501: Formal Language Theory

Lecture 13: Church-Turing Thesis

Marius Minea
marius@cs.umass.edu

University of Massachusetts Amherst

20 February 2019

From Mathematics to Algorithms

1900: David Hilbert's 23 open problems

Tenth problem:
"Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers."

This asks to *devise* an algorithm.

To answer (esp. negatively), requires defining what is **computable**

Polynomial Roots with Turing Machines

How to encode this problem as a language ?

Can define set $D = \{p \mid$
p is an integer-coefficient polynomial in x with an integral root$\}$

Polynomial = string of input symbols $\Rightarrow$ D is a *language*

Is D Turing-recognizable? Yes

TM evaluates p with values 0, -1, 1, -2, 2, ... and checks if zero.
Accepts or runs forever
Multivariable case: can enumerate tuples

Can we argue that at some point we've checked enough?
easy for single variable (highest term dominates)
proved impossible for multivariable (1970)

$\Rightarrow$ language is not Turing-*decidable*

Hilbert and the Entscheidungsproblem (decision problem)

1889: Peano formalization of arithmetic

1900: Hilbert's 2nd problem:
Prove that the axioms of arithmetic are consistent

1928: Entscheidungsproblem
Determine whether an arbitrary statement in first-order logic is *valid* (equivalent to it being *provable*)

Gödel: Incompleteness Theorems (1931)

First incompleteness theorem

Any *consistent* formal system F rich enough to formalize elementary arithmetic is *incomplete*, i.e., there are statements in F that can neither be proved nor disproved in F.

Gödel encodings of statements as numbers
string $s_1 s_2 \ldots s_n$ encoded as $2^{s_1} 3^{s_2} 5^{s_3} \ldots p_n^{x_n}$ + encodings of proofs (is P a proof of statement S ?)
since F reasons about numbers, it can encode statements about F construct number/statement that says it is not provable in F

Second incompleteness theorem:

For any consistent system F that formalizes elementary arithmetic, the consistency of F cannot be proved in F itself.

Gödel & Herbrand (1933): General Recursive Functions

- Zero function
- Successor function $S(x) \stackrel{def}{=} x + 1$
- Projection function $P_i(x_1, x_2, ..., x_n) = x_i$
- Function composition
- Primitive recursion: given k-ary $g(x_1, ..., x_k)$ and $k+2$-ary $h(y, z, x_1, ..., x_k)$, define f:
  $f(0, x_1, ..., x_k) = g(x_1, ..., x_k)$ (base case)
  $f(y+1, x_1, ..., x_k) = h(y, f(y, x_1, ..., x_k), x_1, ..., x_k)$
- Minimization operator μ: $\mu(f)(x_1, ..., x_k) = z$ iff $f(z, x_1, ..., x_k) = 0$, and $f(i, x_1, ..., x_k) > 0$ for $i < z$

Church: Lambda Calculus (1936)

$e ::= x$ variable
$\lambda x.e$ function abstraction (definition)
$e_1 e_2$ function application

Evaluation is defined by *substitution*

Church encodings (for everything else): booleans, numbers, pairs, if-then-else, recursion

true: $\lambda x.\lambda y.x$
false: $\lambda x.\lambda y.y$
if-then-else: $\lambda p.\lambda x.\lambda y.p\ x\ y$
0: $\lambda f.\lambda x.x$
1: $\lambda f.\lambda x.fx$
succ: $\lambda i.\lambda f.\lambda x.f(i\ f\ x)$

Turing: On Computable Numbers (1936)

numbers that can be generated digit by digit

"*"it is almost equally easy to define and investigate computable functions*"

"*In a recent paper Alonzo Church has introduced an idea of "effective calculability", which is equivalent to my "computability", but is very differently defined. Church also reaches similar conclusions about the Entscheidungsproblem. The proof of equivalence between "computability" and "effective calculability" is outlined in an appendix to the present paper.*"

- Showed that a solution to the Entscheidungsproblem would imply it could be determined if a Turing machine prints 0

Church-Turing Thesis

- Church's lambda calculus and Turing machines are equivalent models of computation (also with general recursive functions) – this has been proved
- Every effective computation can be carried out by a Turing machine

(generally accepted as definition of what is computable)

Encoding Algorithms as Turing Machines

Will usually not do either

formal description (states, transitions)
implementation description ("move until finding symbol X", "cross out all a's", etc.)

High-level description

start with description of tape input no low-level details of encoding, but any object can be encoded
(graphs, grammars, automata, polynomials, etc.) serialize all objects $O_1, O_2, ...O_k$ on tape

Operations expressed differently (no array index, etc.), but "mark object X", "copy contents of Y", etc.

Example: Decide Graph Connectivity

$A = \{\langle G\rangle | G \text{ is an undirected connected graph}\}$

$\langle G\rangle = (1,2,3,4)((1,2),(2,3)(3,1),(1,4))$

Implement some generic form of graph traversal:

- mark first node
- repeat until no new nodes marked:
- mark any node attached to a marked node
- scan nodes; *accept* if all marked, *reject* othjerwise