Computability

Algorithms 2024

Hilbert's Program

- *Grundlagenkrise* in Mathematics (~ 1900):
  - How to be sure that Mathematics is true
    - Attempts suffer from paradoxes
      - Example Naïve Set Theory: Russel's set of all sets that do not contain themselves as an element
- Answers to the Grundlagenkrise
  - Intuitionism:
    - Mathematics is a human activity, it does not discover universal truth
  - Logicism:
    - All mathematics derives from logic
  - Formalism:
    - Mathematics is a game with certain rules that conform to our thinking processes

Hilbert’s Program

- A formulation of all mathematics
- Completeness:
  - Proof that all true mathematical statements can be proved in the formalism.
- Consistency:
  - Proof that no contradiction can be obtained in the formalism of mathematics.
- Conservation:
  - Proof that any result about "real objects" obtained using reasoning about "ideal objects" (such as uncountable sets) can be proved without using ideal objects.
- Decidability
  - There is an algorithm for deciding the truth or falsity of any mathematical statement.

Hilbert’s Program

- Hilbert’s program:
  - Find an algorithm that can decide the truth or falsity of an arbitrary statement in first-order predicate calculus applied to integers

- Gödel’s incompleteness result (1931)
  - No such effective procedure can exist

Hilbert's Program

- Formalization of "effective procedure"
  - Each procedure should be described finitely
  - Each procedure should consist of discrete steps, each of which can be carried out mechanically

  - Number of proposals
    - λ-calculus
    - Turing machines (in different versions)
    - RAM machines (computers with infinite memory)

Hilbert's Program

- Lambda calculus:
  - Write functions with a lambda expression
    - `lambda x: x**2` (Python)
    - $\lambda x \,.\, x^2$ (Lambda Calculus)
  - Apply a function to a value:
    - `(lambda x: x**2)(5)` (Python)
    - $(\lambda x \,.\, x^2)(5)$ (Lambda Calculus)

Hilbert's Program

- Lambda Calculus:
  - Let Λ be a set of expressions
    1. If x is a variable, then $x \in \Lambda$
    2. (Abstraction) If x is a variable and $\mathscr{M} \in \Lambda$ then $(\lambda x \,.\, \mathscr{M}) \in \Lambda$
    3. (Application) If $\mathscr{M}, \mathscr{N} \in \Lambda$ then $\left(\mathscr{M}\mathscr{N}\right) \in \Lambda$

Hilbert's Program

- Three rules:
  - **α-equivalence:**
    - $\lambda x \,.\, x$ and $\lambda y \,.\, y$ are equivalent, extend to more complicated formulae
      - (Names of free variables do not matter)
  - **β-reductions:**
    - $(\lambda x \,.\, x^3)(y)$ is the same as y^3, extend to all occasions to plug in
  - **η-reductions:**
    - $\lambda x \,.\, \mathscr{M} x$ is the same as $\mathscr{M}$
    - `lambda x: math.sin(x)` is the same as `math.sin`

Hilbert's Program

- This apparatus is enough to define integers, Booleans, etc.
  - Example:
    - $\lambda xy\,.\,x$ and $\lambda xy\,.\,y$ *function* the same as True and False
    - If-then-else: ifthenelse$(c, a, b) = \lambda c\,.\,\lambda x\,.\,\lambda x\,.\,cxy$
    - This works:
      - ifthenelse $T\ a\ b\ = (\lambda xy\,.\,x)\ a\ b = a$
      - ifthenelse $F\ a\ b\ = (\lambda xy\,.\,y)\ a\ b = b$

Hilbert's Program

- Number 0 is represented by not applying a function
    - $\lambda s\,.\,\lambda x\,.\,x$
- Number 1 is represented by applying a function exactly once
    - $\lambda s\,.\,\lambda x\,.\,sx$
- Number 2 is represented by applying a function exactly twice
    - $\lambda s\,.\,\lambda x\,.\,s(s(x))$
- …
- We then can introduce successor function, addition, multiplication, …

Hilbert's Program

- Church Turing Result:
  - λ-calculus and Turing machines have the same computational power
- Church Hypothesis
  - Turing machines are equivalent to our intuitive notion of a computer
  - What is computable by a human is what is computable by a computer which is what is computable by a Turing machine

Turing

- Early career is as a Mathematical Logician
  - Idea: What is computable
  - Proposes the Turing machine as a simple example of what a Mathematician can calculate (without the brilliance)
    - I.e.: A very simple formal way to compute
    - Idea: If something is possible in that simple system then a human Mathematician can do it as well

Turing

- *Entscheidungsproblem:* Can every true statement in first order logic (with quantifiers) be derived in first order logic
- Answers a dream of *Gottfried Leibniz*: Build a machine that could manipulate symbols in order to determine the truth values of mathematical statements.

Turing

- Made it plausible that a Mathematician is not more powerful than the Turing calculus
- Proved limitations on what a Turing calculus can achieve
- Post thought that Turing’s machine was too complicated and proposed a cleaner definition of the machine

Post-Turing Machine

- A Turing machine consists of
  - An infinitely-long tape divided into squares that are initially blank (denoted by a symbol ‘b’)
  - A read-write head that can read and write symbols
  - A control unit that consists of a state machine
    - In a given state and when reading a given symbol:
      - The machine goes to a new state
      - The machine writes a new symbol
      - The machine moves to the left or the right by one step.

Post-Turing Machines

- Turing machine input
  - A string on the tape, with all other symbols being blanks.

- Turing machine output
  - Turing machines can make decisions:
    - By writing them on the tape
    - By entering an “accepting” or a “rejecting” state
    - These possibilities are actually equivalent

http://morphett.info/turing/turing.html

Post-Turing Machines

- Turing machine programs:
  - A program consists of a set of transition rules:
    - Current state, Current Symbol —> New State, New Symbol, Move

- Note: All Turing machine programs are finite

Post-Turing Machine

- Despite its simplicity, a Turing machine can imitate any computer (known today)

Turing Machine Simulator

- https://morphett.info/turing/turing.html

Post Turing Machine

- Turing machine programs
  - consists of lines

<curr. state> <curr. symb> <new symb> <dir> <new state>

Turing Machine Example

- Palindrome detector
  - Accepts if the input — binary string surrounded by blanks — is a palindrome
  - Algorithm:
    - Find the left-most symbol, erase it, and remember it
    - Go to the right until we are over a blank
    - Move one to the left and check the symbol, erasing it
    - Continue until
      - A discrepancy is discovered
      - Until no more symbols are left over

Turing Machine Example

- go to the left until we find a blank

```
state0, 0, 0, left, state0
state0, 1, 1, left, state0
state0, b, b, right, state1
```

- now we are at the beginning of the word
    - we erase the symbol, but remember the symbol (through the state) and go right

```
state1, 0, b, right, state_seen_zero
state1, 1, b, right, state_seen_one
```

Turing Machine Example

- we go right until we hit a blank, then we go back one step to compare

```
state_seen_zero, 0, 0, right, state_seen_zero
state_seen_zero, 1, 1, right, state_seen_zero
state_seen_zero, b, b, left, state0end

state_seen_one, 0, 0, right, state_seen_one
state_seen_one, 1, 1, right, state_seen_one
state_seen_one, b, b, left, state1end
```

Turing Machine Example

- We are now over the last symbol
  - If the symbol does not match, we go to the non-acceptance state
  - If the symbol matches, we start moving left until we hit the blank that we created

```
state0end, 1, b, stop, not_accepted
state0end, 0, b, left, state_go_left
state1end, 0, b, stop, not_accepted
state1end, 1, b, left, state_go_left
```

Turing Machine Example

- We just go left until we hit the blank, at which point we go right and start over

```
state_go_left, 0, 0, left, state_go_left
state_go_left, 1, 1, left, state_go_left
state_go_left, b, b, right, state1
```

Turing Machine Example

- When do we stop:
  - If there are only blanks on the tape
    - We are then in state1 and we encounter another blank

```
state1, b, b, stop, accept
```

Turing Machine Example

- You can run this example at
  - http://morphett.info/turing/

The Universal Turing Machine

- We can extend the model of the Turing machine
  - E.g. we can have Turing machines with two tapes
  - But we do not get anything more,
    - Because we can <u>*emulate*</u> a Turing machine with two tapes with a Turing machine with one tape
    - How?
      - Even cells are for tape 0, odd cells are for tape 1, and a more complicated state machine

The Universal Turing Machine

- We can emulate a Turing machine with n tapes with a standard one
  - This becomes a model for a RAM machine with n memory cells
  - RAM machine stores program in some dedicated memory locations

The Universal Turing Machine

- We can also build a universal Turing machine
  - Initially: a Turing machine program plus input, separated by blanks
  - Machine then simulates the execution of a Turing machine
  - Machine halts when the simulated Turing machine halts

The Universal Turing Machine

- A single machine that can emulate *<u>all</u>* possible Turing machines!!

Diagonalization Proofs

- Mathematical technique developed by Cantor
    - Trick is applying something to itself
    - Example: We can count all rational numbers
        - Use the following scheme

Diagonalization Proofs

1/1	1/2	1/3	1/4	1/5	1/6
2/1	2/2	2/3	2/4	2/5	2/6
3/1	3/2	3/3	3/4	3/5	3/6
4/1	4/2	4/3	4/4	4/5	4/6
5/1	5/2	5/3	5/4	5/5	5/6
6/1	6/2	6/3	6/4	6/5	6/6

Diagonalization Proofs

- Cantor:
  - The real numbers in [0,1] are ***not*** countable
  - Assume that they are:
    - Let $s_1, s_2, s_3, s_4, s_5, \ldots$ be an enumeration of real numbers
    - Write the numbers as binary numbers, leave out the leading dot

Diagonalization Proofs

$$\begin{array}{ccccccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & \cdots \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & \cdots \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & \cdots \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & \cdots \end{array}$$

Diagonalization Proofs

- Now define a new number defined by the enumeration itself

$$t_i = 1 - s_{i,i}$$

  - The i^{th} binary digit of t is the opposite of the i^{th} digit of the i^{th} number

Diagonalization Proofs

- If this would be an enumeration of all real numbers in [0,1], then t would appear in the enumeration
  - Suppose it is the j^{th} element
  - Look at the j^{th} digit of t

$$s_{j,j} = t_j = 1 - s_{j,j}$$

  - So, this is not possible
- Ergo: we cannot enumerate the numbers in [0,1]

Diagonalization Proofs

- This is a similar argument to Russell's paradox:
  - X = The set of all set that do not have themselves as an element.
  - Is $X \in X$

Diagonalization Proofs

- The universal Turing machine allows us to do the same type of self-application to show impossibilities

Impossibility

- Can everything (whatever that means) be computed
- Halting Problem: Will a program stop executing
- Answer: There is no algorithm that can decide whether a given program will stop executing
  - Though most of the time, we can decide so easily

Proof: The Halting Problem is not-computable

- Assume that we have a program that can decide the halting problem
  - Input:
    - A program — basically a long string
    - An input
  - Output: A decision — the program will halt on that input or the program will not halt on that input

Proof: The Halting Problem is not-computable

- Assume that there is such a program

- ```
  def halting(program, input):
      #something really complicated
      if b:
          return True
      else:
          return False
  ```
  ```

Proof: The Halting Problem is not-computable

- Now, we create a new program

```
def z(program):
   if halting(program, program):
        while True:
              x = 0
   else:
         print("I am done")
```

Proof: The Halting Problem is not-computable

- What happens if we calculate z(z)
  - Perfectly legit, since z is a program
  - Will z halt or not?
    - If z halts on z,
      - Then halting(z,z) is True.
      - Then we execute “while True”
      - Therefore z does not halt

```
def z(program):
   if halting(program, program):
         while True:
                x = 0
   else:
          print(“I am done”)
```

Proof: The Halting Problem is not-computable

- What happens if we calculate z(z)
  - Perfectly legit, since z is a program
  - Will z halt or not?
    - If z does not halts on z,
      - Then halting(z,z) is False.
      - Therefore we print “I am done”
      - Therefore z does halt

```
def z(program):
  if halting(program, program):
       while True:
              x = 0
  else:
        print(“I am done”)
```

Proof: The Halting Problem is not-computable

- This is a contradiction
  - Therefore, the function `halting` cannot exist.
  - Therefore, the halting problem cannot be solved by computation