Topic 5: Random-access machines and $\mu$-recursion

1. Random-access machines and the $\mathcal{L}$ language
2. $\mu$-recursive functions
3. The Church-Turing thesis

Models of algorithmic computation

$\text{TM}-$computable functions $= \text{RAM}-$computable functions $= \text{Recursively-definable functions}$

The Chomsky hierarchy

- Regular languages (accepted by DFAs)
- Context-free languages (accepted by PDAs)
- Recursive (Turing-decidable) sets
- All languages
1. Random-access machines and the $S$ language

- Unlike transition-system-based automata, RAMs have addressable storage
- *Example:* Any microprocessor based system
- $S$, an assembler-like language, implements the RAM model
- Variables are labeled memory cells
- $S$ computes any computable function on natural numbers

**$S$ language definition**

- Identifiers (for any $i \in N$):
  - $X_i$ input
  - $Y_i$ output (initialized to 0)
  - $Z_i$ local variables (initialized to 0)
  - $L_i$ labels for instructions
- $S$ has 3 instructions, for variables $V$
  with values $\in N$:
  - $V \leftarrow V + 1$
  - $V \leftarrow V \div 1$ (monus: $0 \div 1 = 0$)
  - if $V \neq 0$ goto $L$
Example program in $\mathcal{S}$

\[
[A] \quad Y \leftarrow Y + 1 \\
X \leftarrow \max(0, X - 1) \\
\text{if } X \neq 0 \text{ goto } A
\]

- The above program yields $Y = 1$ as output if input $X = 0$; otherwise $Y = X$
- Loop labeled $A$ increments output $Y$ while decrementing input $X$
- This program is close to implementing assignment $Y \leftarrow X$

Macros in $\mathcal{S}$

- Any function that can be computed in $\mathcal{S}$ may be considered added to the language via macros
- Example: unconditional goto, goto $L$, is supported using
  \[
  Z \leftarrow Z + 1 \\
  \text{if } Z \neq 0 \text{ goto } L
  \]
- Jump-on-0 and variable assignment may be implemented by simple macros
Addition in $\mathcal{S}$ using macro

\[\begin{align*}
[A] & \quad \text{if } X_2 = 0 \text{ goto } E \\
Y & \leftarrow X_1 \\
Y & \leftarrow Y + 1 \\
X_2 & \leftarrow X_2 - 1 \\
\text{goto } A
\end{align*}\]

- Implements $+$ via macro
- Some macros are used here
- Label $E$ denotes exit
- **Powerful idea:** In similar ways, can implement $\times$, $\div$, $-$, etc.

Programs with programs as input

- **Problem:** $\mathcal{S}$’s RAM target machine computes only with natural numbers
- **Solution:** By Gödelization, any string $x$ may be encoded as a unique Gödel number, $\#(x)$
- Any number $n$ in $\mathbb{N}$ may be converted to the corresponding string, including an $\mathcal{S}$ program $P = \mathcal{S}(n)$ with $n = \#(P)$
Gödelization

- Let \( x_i \) be the \( i \)th symbol in a program (IF, X, +, etc.)
- Let \( k_i \) be \( x_i \)'s lexical position in the vocabulary of \( \mathcal{S} \)
- For program \( P \) with \( n \) symbols, the Gödel number, \( \#(P) \), is the product of \( n \) prime numbers \( p_i, \ 0 < i \leq n \ (2, 3, 5, \text{etc.}) \), each \( p_i \) raised to the \( k_i \) power
- Gödel used the same concept to encode assertions and proofs

Universal programs

- Consider a program \( U \) in \( \mathcal{S} \) that accepts inputs \( \#(P), x \), decoding \( \#(P) \) to \( P \) and simulates program \( P \) with input \( x \)
- \( U \) can be constructed. Just write a simulator for the RAM that executes \( \mathcal{S} \) code
- \( U \) is called a universal program
- Examples: compilers, interpreters, virtual machines
Halting problem w.r.t. \$S\$

- Define function \$HALT(x,y)\$ as:
  - \(\text{True} \iff \text{program } S(x) \text{ eventually halts on input } y\)
- **Theorem**: \(HALT(x,y)\) is undecidable
- **Proof**: Assume \(HALT(x,y)\) is decidable, so that the subroutine \(Halt\) decides it
  - Construct \(S\) program \(S\) as follows using a program that computes \(HALT\):
    
    \[\text{[A] If } \text{Halt}(X, X) \text{ goto A}\]
    - So for any \(x\), \(Halt(x, #(S)) \iff \neg HALT(x, x)\)
    - Now let \(x = #(S)\). Then \(Halt(#(S), #(S)) \iff \neg HALT(#(S), #(S))\), a contradiction
    - Therefore \(HALT(x,y)\) is undecidable

Cantor’s, Turing’s, and slide 11’s proofs

- Set up a table with \(y\) axis for programs \(0, 1, 2, \ldots\), and \(x\) axis for inputs \(0, 1, 2, \ldots\)
- Entries express values of \(\neg HALTS(P_y, x)\)
- Note that the diagonal, \(P_y(y)\), is the bitwise negation of the behavior of program \(S\) above
- Hence whatever program \(S\) is, its behavior on input \#\(S\) is the opposite of its behavior according to the table
- Hence there is no program \(S\) described
2. $\mu$-recursive functions

- Related to lambda calculus (Church)
- Define:
  - Primitive recursive functions
  - Minimalization and $\mu$ recursion
- Equivalence of $\mu$ recursion with TMs, RAMs
- Results are due to K. Gödel, S. Kleene

A recurrence may define a function algorithmically

\[ \text{sum} \ (a, b) = \begin{cases} a & \text{if } b = 0 \\ 1 + \text{sum} \ (a, b - 1) & \text{otherwise} \end{cases} \]

\[ \text{product} \ (a, b) = \begin{cases} 0 & \text{if } b = 0 \\ a + \text{product} \ (a, b - 1) & \text{otherwise} \end{cases} \]

\[ \text{factorial} \ (n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n \times \text{factorial} \ (n - 1) & \text{otherwise} \end{cases} \]
Basic primitive recursive functions

- \( \text{zero}(x) = 0 \)
- \( \text{succ}(x) = x + 1 \)
- \( \text{pred}(x) = x - 1 \) for \( x > 0 \)
- \( \text{proj}_k(x_1, \ldots, x_n) = x_k \)

The above functions are computable by simple \( S \)-language programs and Turing machines.

Composition of primitive recursive functions

Definition: If \( h(x) = f(g(x)) \) then \( h \) is said to be obtained from \( f \) and \( g \) by composition.

Theorem: If \( f \) and \( g \) are \( S \)-computable, and \( h \) is obtained from \( f \) and \( g \) by composition, then \( h \) is \( S \)-computable.

Proof: The following program in \( S \) computes \( h(x) \):

\[
\begin{align*}
Z & \leftarrow g(X) \\
Y & \leftarrow f(Z)
\end{align*}
\]
5. Random access machines and \( \mu \)-recursion

**Primitive recursion**

- **Definition:** With \( k \) fixed, define
  
  \[
  h(0) = k \\
  h(x+1) = g(x, h(x)) \quad \text{for all } x \geq 0
  \]

  Then \( h \) is said to be obtained from \( g \) by *primitive recursion*

- **Example:**
  
  \[
  \text{factorial}(0) = 1 \\
  \text{factorial}(x+1) = g(x, \text{factorial}(x))
  \]

  where \( g(x, y) = (x+1)y \)

  Thus \( \text{factorial} \) is obtained from multiplication by primitive recursion

**Computability of primitive recursive functions**

- **Theorem:** If \( g \) is \( \mathcal{S} \)-computable and \( h \) is obtained from \( g \) by primitive recursion, then \( h \) is \( \mathcal{S} \)-computable

- **Proof:** The program in \( \mathcal{S} \) on the next slide computes \( h(x) \) where
  
  \[
  h(0) = k \\
  h(x+1) = g(x, h(x)) \quad \text{for all } x > 0
  \]
Computability of primitive recursive functions, cont’d

\[
Y \leftarrow k \\
\text{[A]} \quad \text{if } X = 0 \text{ goto } E \\
\quad Y \leftarrow g(Z, Y) \\
\quad Z \leftarrow Z + 1 \\
\quad X \leftarrow X - 1 \\
\text{goto } A
\]

Discussion: \( Y \) gets value of \( h(0) \), then \( h(1) \), \( h(2) \), etc., up to \( h(x) \)

Minimalization

- Let \( \text{min}_y (P(x_1, \ldots , x_n, y)) \) be the smallest value of \( y \) such that \( P(x_1, \ldots , x_n, y) \) is true
- Proper minimalization is applied when the function \( \text{min}_y (P(x_1, \ldots , x_n, y)) \) is total
- Examples:
  - Find shortest Hamiltonian path in a graph
  - Find smallest prime number that has “00000” in its binary expression
  - Find the shortest C program that outputs its own code in fewer than 1 million clock ticks
μ-recursive functions

Define μ (mu), or minimalization, as follows:

\[ \mu y (f(x_1, x_2)) = \min \{ y \mid f(x_1, x_2) = 0 \} \]

Definition: The μ-recursive functions are the primitive recursive functions and functions constructible from PR functions by application of minimalization.

Theorem: Function \[ \mu y (f(x_1, x_2)) \] is \( S \)-computable iff \( f \) is μ-recursive

Proof: If \( f(x,y) \) is computable, then an \( S \) program also exists that increments \( y \) until condition \( f(x,y) = 0 \) is satisfied.

3. The Church-Turing thesis

- Three models of algorithmic computation are equivalent: TMs, RAM with \( S \) language, and μ-recursive functions

- Shown: Any μ-recursive function is computable by some \( S \)-language program

- To show by construction:
  - Any \( S \)-language program computes a μ-recursive function
  - TM ↔ \( S \)-language program (Turing, 1937)

- Church-Turing Thesis: These models capture the intuitive notion of algorithmic computation
5. Random access machines and \( \mu \)-recursion

### TMs, RAMs compute same fns

**TM \( \rightarrow \) RAM**

From TM \( M \), construct a program in \( S \) that simulates \( M \), implementing states, tape, and transition function.

**RAM \( \rightarrow \) TM**

*Given any program in \( S \), construct 4-tape TM:*

- Tape 1 represents memory
- Tape 2 is program counter
- Tape 3 stores memory address or contents
- Tape 4 stores input

### \( \mu \)-recursion and pseudocode compute the same functions

*Proof sketch:*

- A recurrence and an equivalent *while* loop may be easily constructed for any function computable by a single loop.
- Nested *while* loops and nested recurrences may be written as needed.
- To test for minimal value of second parameter, test for all values starting at 0.
References


