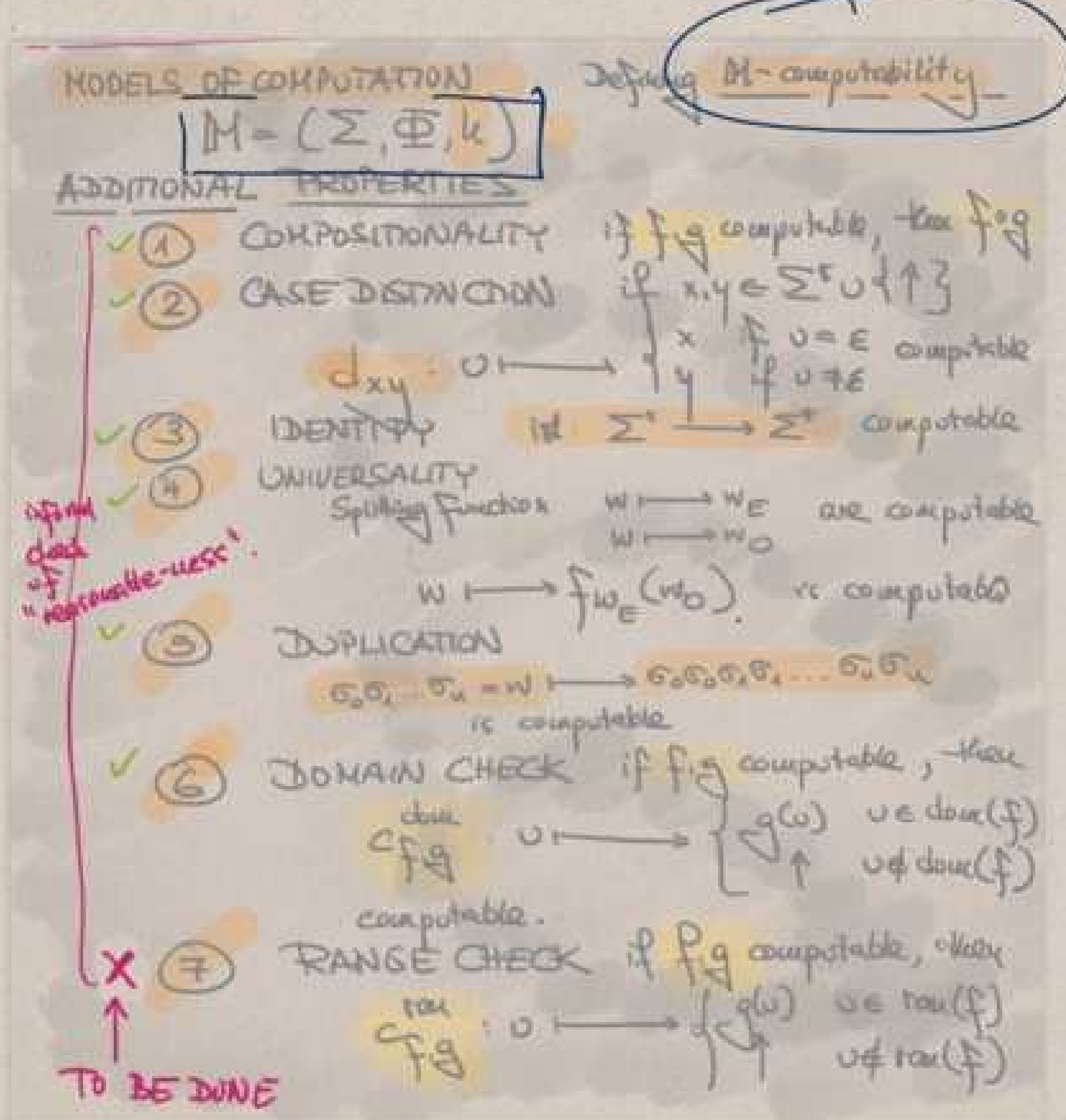
RECURSION THEORY
Fifth Lecture

28 November 2021

FROM LECTURE IV

MODELS OF COMPUTATION
WITH SEVEN "REASONABLE ASSUMPTIONS"



MODELS OF COMPUTATION   Defining $\mathbb{M}$-computability

$\boxed{\mathbb{M} = (\Sigma, \Phi, k)}$

ADDITIONAL PROPERTIES

✓ (1) COMPOSITIONALITY if f, g computable, then $f \circ g$

✓ (2) CASE DISTINCTION if $x, y \in \Sigma^* \cup \{\uparrow\}$

$d_{xy} : v \longmapsto \begin{cases} x & \text{if } v = \varepsilon \\ y & \text{if } v \neq \varepsilon \end{cases}$ computable

✓ (3) IDENTITY $\mathrm{id} : \Sigma^* \longrightarrow \Sigma^*$ computable

✓ (4) UNIVERSALITY
Splitting function $w \longmapsto w_E$, $w \longmapsto w_O$ are computable

$w \longmapsto f_{w_E}(w_O)$. is computable

✓ (5) DUPLICATION
$\sigma_0\sigma_1 \ldots \sigma_n = w \longmapsto \sigma_0\sigma_0\sigma_1\sigma_1 \ldots \sigma_n\sigma_n$
is computable

✓ (6) DOMAIN CHECK if f, g computable, then

$c^{\mathrm{dom}}_{f,g} : v \longmapsto \begin{cases} g(v) & v \in \mathrm{dom}(f) \\ \uparrow & v \notin \mathrm{dom}(f) \end{cases}$

computable.

X (7) RANGE CHECK if f, g computable, then

$c^{\mathrm{ran}}_{f,g} : v \longmapsto \begin{cases} g(v) & v \in \mathrm{ran}(f) \\ \uparrow & v \notin \mathrm{ran}(f) \end{cases}$

defining class of "reasonable-ness"!

TO BE DONE

In Lecture IV, we saw two concrete models of computation:

(1) TURING MACHINES

(2) REGISTER MACHINES

(1) Turing machines

$\rightarrow S$ finite set of states

Σ finite alphabet

TURING MACHINE PROGRAM

$$P : S \times \Sigma \cup \{\emptyset\} \longrightarrow S \times \Sigma \cup \{\emptyset\} \times \{\leftarrow, \rightarrow, \emptyset\}$$

TURING MACHINES as Models of Computation

$$\overline{\Sigma} = \Sigma \cup \{\emptyset\} \cup \{\sigma_s ; s \in S\}$$

Configurations: $\overline{\Sigma}^*$ $\mathbb{M}_{TM}$

(2) Register machines

S finite set of states

Σ finite alphabet

I set of instructions $\left[\begin{array}{l} +(k, \sigma, s) \\ -(k, s, s') \end{array}\right.$

REGISTER MACHINE PROGRAM

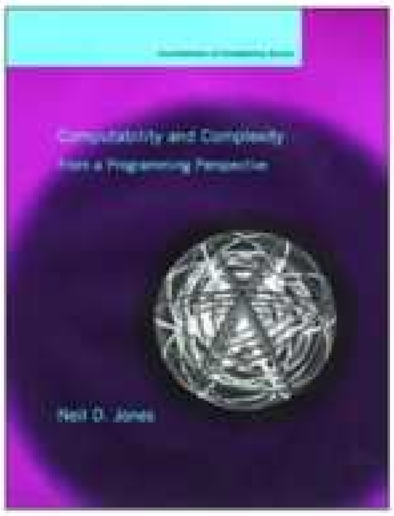
$$\rightarrow P : S \longrightarrow I$$

REGISTER MACHINES as Models of Computation

$$\overline{\Sigma} = \Sigma \cup \{R\} \cup \{\sigma_s ; s \in S\}$$

Configurations: $\overline{\Sigma}^*$. $\mathbb{M}_{RM}$

(3) WHILE PROGRAMMING LANGUAGES



From Foundations of Computing

Computability and Complexity

From a Programming Perspective

By Neil Deaton Jones

MIT Press 1997

Definition 2.1.3 Let Vars $= \{\mathtt{V}_0, \mathtt{V}_1, \ldots\}$ be distinct variables. We use the conventions d, e, f, ... $\in \mathbb{D}$ and X, Y, Z, ... $\in$ Vars. Then the syntax of WHILE is given by the following grammar:

```
Expressions  ∋  E, F  ::=  X                      (for X ∈ Vars)
                        |  d                      (for atom d)
                        |  cons E F
                        |  hd E
                        |  tl E
                        |  =? E F
Commands     ∋  C, D  ::=  X := E
                        |  C; D
                        |  while E do C
Programs     ∋  P     ::=  read X; C; write Y
```

Here X and Y are the not necessarily distinct *input* and *output variables*. □

```
read X;                       (* X is (d.e) *)
  A := hd X;                  (* A is d *)
  Y := tl X;                  (* Y is e *)
  B := nil;                   (* B becomes d reversed *)
  while A do
    B := cons (hd A) B;
    A := tl A;
  while B do
    Y := cons (hd B) Y;
    B := tl B;
write Y                       (* Y is list d with e appended *)
```

```
read X;
 GO := true; Y := false;
 while GO do
  if D then
    D1 := hd D; D2 := tl D;
    if D1 then
      if E then
        E1 := hd E; E2 := tl E;
        if E1 then
          D := cons (hd D1) (cons (tl D1) D2));
          E := cons (hd E1) (cons (tl E1) E2))
        else GO := false
      else GO := false
    else
      if E then
        if (hd E) then GO := false
        else
          D := tl D;  E := tl E
      else GO := false
  else
    if E then GO := false
    else
      Y := true; GO := false;
write Y
```

Σ is the alphabet of symbol on the keyboard, so Σ^* is just strings of text symbols.

$* \in \Sigma^*$.

MODEL OF COMPUTATION

M_{WHILE}

(4) Everything we can describe with any reasonable specification for programming languages is a model of computation.

Pseudocode

From Wikipedia, the free encyclopedia

> This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
> *Find sources:* "Pseudocode" – news · newspapers · books · scholar · JSTOR *(August 2016) (Learn how and when to remove this template message)*

In computer science, **pseudocode** is a plain language description of the steps in an algorithm or another system. Pseudocode often uses structural conventions of a normal programming language, but is intended for human reading rather than machine reading. It typically omits details that are essential for machine understanding of the algorithm, such as variable declarations and language-specific code. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications to document algorithms and in planning of software and other algorithms.

No broad standard for pseudocode syntax exists, as a program in pseudocode is not an executable program; however, certain limited standards exist (such as for academic assessment). Pseudocode resembles skeleton programs, which can be compiled without errors. Flowcharts, drakon-charts and Unified Modelling Language (UML) charts can be thought of as a graphical alternative to pseudocode, but need more space on paper. Languages such as HAGGIS bridge the gap between pseudocode and code written in programming languages.

Contents [hide]

Often in CS: if you need to show that something can be done by a machine, you don't produce evidence in the form of a real (running) computer program. Instead, you write it in pseudocode which is an informal specification that allows every reader to implement this in their favourite programming language.

Remark This is precisely what we did in our previous "informal arguments" that properties (1) – (7) are "reasonable":

we provided a high-level pseudocode description of an algorithm that performs the appropriate transformation.

Consider the models of computation M_{TM}, M_{RM} and check whether they satisfy (1) – (7).

[If we were more interested in the computational processes in detail, we'd spend several lectures on checking (1) – (7) precisely for one of the models.]

We look at two examples:

Example 1 ③ for M_{TM}.

IDENTITY

$id : \Sigma^* \longrightarrow \Sigma^*$ is M-computable.

TM program

$$P : S \times \Sigma \cup \{\emptyset\} \longrightarrow S \times \Sigma \cup \{\emptyset\} \times \{\leftarrow, \rightarrow, \emptyset\}$$

S states: s_0 START

s_1 HALT

$$P(s_0, \sigma) = (s_1, \sigma, \emptyset)$$

where $\sigma \in \Sigma \cup \{\emptyset\}$

It halts

It doesn't change anything.

It doesn't move.

The program P will immediately halt and not modify what's on the tape.

Remark In general, doing things with TM is hard due the fact that they have a single storage unit and that even moving along the storage unit requires many individual steps.

Example 2 (1) for M_{RM}

COMPOSITION:
if f and g are M-computable, then so is $f \circ g$.

Register machine programs:

$I \quad +(k,s,s)$
$\quad\; -(k,s,s')$

$P : S \longrightarrow I$

If f and g are M_{RM}-computable, then they have RM-programs P and Q s.t.

$f(w)\downarrow \iff P$ halts on input w

$g(w)\downarrow \iff Q$ halts on input w.

W.l.o.g., let's assume that the states S used in P and the states T used in Q are disjoint.

In S, s_0 is start state
$\quad s_1$ is the haltstate

In T, t_0 is start state
$\quad t_1$ is halt state

P:

s_0	I_0
s_1	I_1
s_2	I_2
$\vdots$	$\vdots$
s_k	I_k

Q:

t_0	J_0
t_1	J_1
$\vdots$	$\vdots$
t_ℓ	J_ℓ

P:

$$\begin{array}{ll} s_0 & I_0 \\ s_1 & I_1 \\ \vdots & \vdots \\ s_k & I_k \end{array}$$

Q:

$$\begin{array}{ll} t_0 & J_0 \\ t_1 & J_1 \\ \vdots & \vdots \\ t_l & J_l \end{array}$$

Find all instructions J_i that contain the Q-halt state t_1 and replace them by s_0 (the start state of P). Call this replaced instruction J_i^*.

R:

$$\begin{array}{ll} s_0 & I_0 \\ s_1 & I_1 \\ \vdots & \vdots \\ s_k & I_k \\ t_0 & J_0^* \\ t_1 & J_1^* \\ \vdots & \vdots \\ t_l & J_l^* \end{array}$$

with
start state t_0
halt state s_1.

Then R calculates $f \circ g$.

Example 3

All of our previous informal arguments can be seen as a proof that the properties ① – ⑦ hold in $M_{\text{pseudocode}}$.

And much more is true: with the power of reasonable programming languages, we can argue that many more functions are computable.

Ex.

$\sigma \in \Sigma$

$w \in \Sigma^*$

$$w \longmapsto \sigma w$$

is clearly computable in the pseudocode sense; also possible with TM or RM but very unpleasant.

One could speculate that the ease of showing that something is computable indicates that $M_{\text{pseudocode}}$ is more powerful than M_{TM}.

Def. We say that a computational framework is an assignment

$$\Sigma \longmapsto \mathbb{F}(\Sigma)$$

that assigns to each finite alphabet Σ a model of computation

$$\mathbb{F}(\Sigma) = (\Sigma, \Phi, h).$$

Observation The previously discussed models of computation are actually computational frameworks:

$$\mathbb{F}_{TM}, \mathbb{F}_{RM}, \mathbb{F}_{WHILE}$$

Definition If $\mathbb{F}_0, \mathbb{F}_1$ are computational frameworks, we say that they are equivalent if for every Σ, there are $\Sigma_0, \Sigma_1 \supseteq \Sigma$ s.t.

if f is $\mathbb{F}_0(\Sigma)$-computable, then it is $\mathbb{F}_1(\Sigma_1)$-computable

and

if f is $\mathbb{F}_1(\Sigma)$-computable, then it is $\mathbb{F}_0(\Sigma_0)$-computable.

Theorem

$\mathbb{F}_{TM}$, $\mathbb{F}_{RM}$ and $\mathbb{F}_{WHILE}$

are all equivalent.

Proof sketch.

The key ingredient is that each individual operation of the model of computation $\mathbb{F}_0(\Sigma)$ has to be mimicked by $\mathbb{F}_1(\Sigma_1)$.

E.g., $\mathbb{F}_{RM}$ can do the operation

$$\boxed{\sigma_s R w_0 R w_1 \dots R w_n} \qquad s \longmapsto +(k, \sigma, s')$$

$$\downarrow$$

$$\sigma_{s'} R w_0 R w_1 \dots R w_k \sigma \dots R w_n$$

Goal for the proof: describe this transformation with a Turing machine.

If you have compositionality, this is enough if you do it for every single operation of $\mathbb{F}_0(\Sigma)$.

Alan Turing
OBE FRS

Turing c. 1928 at age 16

Born	Alan Mathison Turing 23 June 1912 Maida Vale, London, England
Died	7 June 1954 (aged 41) Wilmslow, Cheshire, England
Cause of death	Suicide (disputed) by cyanide poisoning
Resting place	Ashes scattered in gardens of Woking Crematorium
Education	Sherborne School
Alma mater	University of Cambridge (BA, MA) Princeton University (PhD)
Known for	Cryptanalysis of the Enigma Turing's proof Turing machine Turing test Unorganised machine Turing pattern Turing reduction "The Chemical Basis of Morphogenesis"

Alonzo Church

Alonzo Church (1903–1995)

Born	June 14, 1903 Washington, D.C., US
Died	August 11, 1995 (aged 92) Hudson, Ohio, US
Citizenship	United States
Alma mater	Princeton University
Known for	Lambda calculus Simply typed lambda calculus Church encoding Church's theorem Church–Kleene ordinal Church–Turing thesis Frege–Church ontology Church–Rosser theorem Intensional logic

1936

Turing and Church independently solved Hilbert's DECISION PROBLEM with entirely different models of computation: these models, though superficially very different, are equivalent in this sense.

Why do different approaches end up with equivalent computational frameworks?

Answer (Church-Turing)

This is because the class of M-computable partial functions for M_{TM} and M_{Church} is not arbitrary but the natural and "correct" answer to the question

WHAT IS COMPUTATION ?

THE CHURCH-TURING THESIS

Any $\boxed{\text{reasonable}}$ computational framework is equivalent to $\mathbb{F}_{TM}$.

Because of this, the CT thesis is not a provable mathematical statement.

In practice, this means that when we're studying COMPUTABILITY, formally defined as $\mathbb{F}_{TM}$, we can avoid in our proofs to write formal TM programs and instead use verbal descriptions of algorithms, leaving it to the reader to implement them in his or her favourite programming language.

[This is what we already did in arguing for 1 - 7.]

From now on, if we need to prove that something is computable, we provide a step-by-step algorithm with precise criteria for when it halts that can be implemented in a programming language.

DIAGONALISATION / THE ZIG-ZAG METHOD

[This is essentially the argument that (7) RANGE CHECK is reasonable.]

Theorem The intersection of two c.e. sets is c.e.
[computably enumerable]

Proof. If A, B are c.e., then ψ_A, ψ_B are computable :

$$\psi_{A}(w) := \begin{cases} 1 & \text{if } w \in A \\ \uparrow & \text{if } w \notin A \end{cases}$$

B B B

Suppose we have a flag set to 00.

Describe the algorithm in steps:

Step $2n$ Run the computation of φ_A for n steps.
If it has halted switch the first bit of the flag to 1.
If the flag is 11, then halt and output 1.

Step $2n+1$ Run the computation of φ_B for n steps.
If it has halted switch the second bit of the flag to 1.
If the flag is 11, then halt and output 1.

Case 1. This halts at some pt. Then the flag is 11, so $w \in A \cap B$.

Case 2. It never halts, but then at no step, the flag was 11, so $w \notin A \cap B$.

This algorithm computes $\varphi_{A \cap B}$

q.e.d.

Corollary (to the proof)

The intersection of k many c.e. sets is c.e.

Theorem The set

$$\{P;\ P \text{ is a program s.t. } dom(f_P) \neq \emptyset\}$$

is c.e.

Proof. Remember from the argument for RANGE CHECK:

$i \longmapsto w_i$ computable assignment of the i-th word in Σ^*

$\langle i,j \rangle : \mathbb{N}^2 \longrightarrow \mathbb{N}$ computable bijection

I use the same trick as in the argument for RANGE CHECK:

In STEP n of the algorithm

find $n = \langle i,j \rangle$

Compute $f_P(w_i)$ for j steps (*)

If this has halted, then halt and output 1.

If not, move to $n+1$.

Case 1 This algorithm halts at some $n = \langle i, j \rangle$.
Then output is 1 and $w_i \in \operatorname{dom}(f_P)$.

Case 2 It never halts, so $\operatorname{dom}(f_P) = \emptyset$.

Therefore the algorithm describes

$$\psi_A$$

where $A = \{P ; \operatorname{dom}(f_P) \neq \emptyset\}$.

Remark. The same idea also gives

$\{P ; |\operatorname{dom}(f_P)| \geq k\}$ is c.e.

Proof. Set a flag to 0.
Run the algorithm of the last proof. Instead of halting and outputting 1 in (*), we raise the flag by 1 if the word that is accepted in step n is diff. from the others that were accepted before.
Check whether flag $\geq k$. If so, halt and output 1.