CDI V

FIFTH LECTURE

9 May 2023

Computability, Decidability, Incompleteness

NEXT WEEK: PENTECOSTAL BREAK
NO LECTURE ON 16 MAY
NEXT LECTURE: 23 MAY

Lecture IV
Page 11

Closure properties

(1) COMPOSITION

Suppose $f : \mathbb{W}^n \dashrightarrow \mathbb{W}$ and $g_1, \ldots, g_m : \mathbb{W}^k \dashrightarrow \mathbb{W}$ are partial functions, then the partial function h defined by

$$h(\vec{w}) := f(g_1(\vec{w}), \ldots, g_m(\vec{w}))$$

is called the *composition of f with $(g_1, \ldots, g_m)$*. The notational convention used for operations applies here as well: if any term on the right hand side is undefined, then so is the left hand side.

(2) RECURSION

Suppose $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ and $g : \mathbb{W}^{k+2} \dashrightarrow \mathbb{W}$ are partial functions, then the function h defined by the recursion equations

$$h(\vec{w}, \varepsilon) = f(\vec{w}) \text{ and}$$
$$h(\vec{w}, s(v)) = g(\vec{w}, v, h(\vec{w}, v))$$

is called the *recursion result of f and g*.

(3) MINIMISATION

Suppose $f : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ is a partial function, then the partial function h defined by

$$h(\vec{w}) := \begin{cases} v & \text{if for all } u \leq v \text{, we have that } f(u)\downarrow \text{ and} \\ & v \text{ is } \leq\text{-minimal such that } f(\vec{w}, v) = z \text{ or} \\ \uparrow & \text{if for all } v,\ f(\vec{w}, v) \neq z \end{cases}$$

is called the *minimisation result of f*.

The class of **RECURSIVE** functions is the smallest class containing all basic functions and closed under composition, recursion, & minimisation.

Already done:

1. All basic functions are computable.
2. Computable fns closed under composition.

Goal.

Lecture IV
Page 13

Goal Show the following theorem

THEOREM Every recursive fn is computable.

[We already know that all basic functions are computable; we already know that the computable fns are closed under ①.

If we can show that computable fns are closed under ②, ③, then since recursive fns are smallest such class:

Rec ⊆ Comp.]

Proof. We need to show that the computable functions are closed under ② & ③.

② **Recursion**

f, g computable

$$h(\vec{w}, 0) = f(\vec{w})$$
$$h(\vec{w}, s(v)) = g(\vec{w}, v, h(\vec{w}, v))$$

So: h is the result of recursion on f & g.

We'll show that h can be computed by RM.

1. Find two irrelevant registers k, l and empty them.
2. Calculate $f(\vec{w})$ and write it into reg. l.

   [Write u for the CURRENT register content of k
   u^* for the CURRENT register content of l]

3. Check whether $v = u$. If so, then output u^* and halt.
4. If not, calculate $g(\vec{w}, u, u^*)$ and write this in register l.

5. Apply s to v and write the result in register k.
(s ↑ successor fn)

6. go to 3.

Clearly, this routine computes h.

---*---

(3). Suppose f is computable and

$$h(\vec{w}) := \begin{cases} v & \text{if } \forall u \leq v \; f(u)\downarrow, \; \& \\ & v \text{ is } \leq\text{-minimal s.t. } f(\vec{w}, v) = \varepsilon \\ \uparrow & \text{o/w} \end{cases}$$

Show that h can be computed by a RM.

1. Find irrelevant register k and empty it. As before, write v^* for the CURRENT content of register k.
2. Calculate $f(\vec{w}, v^*)$. If this halts, check whether $f(\vec{w}, v^*) = \varepsilon$.
3. If so, output v^*.
4. If not, apply s to v^* and go to 2.
(s ↑ successor fn)

Clearly, this routine computes h. q.e.d.

Applications

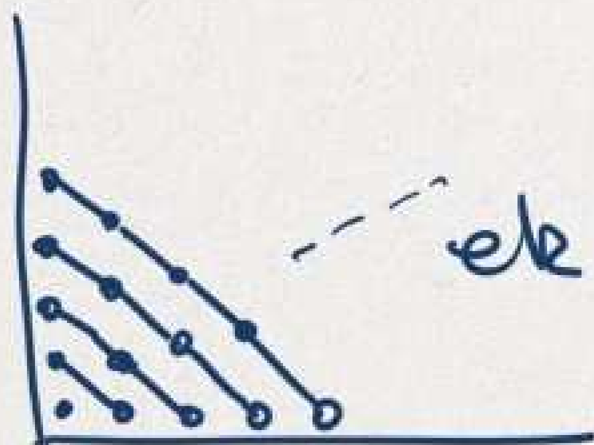
1. This allows us to use recursion in our proofs.
2. This allows us to mimic all arithmetical fns that we proved in Lecture N to be recursive.

[$v + w = u$ iff $\#v + \#w = \#u$ is recursive, ~~thus~~ computable; so is $v \cdot w = u$ iff $\#v \cdot \#w = \#u$ etc.]

<u>Example</u> Cantor's Zigzag function

$$z : (i,j) \longmapsto \frac{(i+j)(i+j+1)}{2} + j.$$

This is the famous bij. between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$



Via our theorem, this yields a <u>computable</u>

$$\bar{z} : W^2 \longrightarrow W$$

For this, we write

$v * w$

MERGING FUNCTION

Similarly, we can take a word w and split it into two:

$$w_{(0)} \quad \& \quad w_{(1)}$$

s.t. $w_{(0)} * w_{(1)} = w$.

Both $w \longmapsto w_{(0)}$ and

$w \longmapsto w_{(1)}$

are computable.

SPLITTING FUNCTIONS

4.7 Software and universality

Fix an alphabet Σ and enlarge it by new symbols to a larger alphabet Σ':

$$\mathbf{0} \quad \mathbf{1} \quad \boldsymbol{\varepsilon} \quad + \quad ? \quad - \quad (\quad) \quad , \quad \mapsto \quad \square.$$

delimiter

INSTRUCTION

We'll use these symbols to encode all of the elements of our descriptions of Σ-register machines. First of all, a natural number k will be represented in binary notation using $\mathbf{0}$ and $\mathbf{1}$, e.g., 13 will be represented by $\mathbf{1101}$; we write $\mathrm{code}(k)$ for this word. The ability to refer directly to natural numbers with words (rather than via the #-function that we get from the shortlex ordering) allows us to write arithmetical functions in a more direct way. Note that the function $w \mapsto \mathrm{code}(k)$ where $\#(w) = k$ is computable (use recursion) and has a computable inverse. This means that the function $h(\mathrm{code}(k), \mathrm{code}(\ell)) := \mathrm{code}(k+\ell)$ is computable: find w and v such that $\#(w) = k$ and $\#(v) = \ell$, use the arithmetical functions defined in § 4.5 to obtain u such that $\#(u) = k + \ell$, and transform u into $\mathrm{code}(k+\ell)$.

We'll encode the numbers occurring in instruction in binary using $\mathbf{0}$ & $\mathbf{1}$.

Define $\mathrm{code}(k) :=$ the binary representation of k.

Similarly, we shall represent states by binary sequences: as we have seen before, the actual set of states does not matter for a register machine, only its size. As a consequence, we can assume w.l.o.g. that the states of a register machine are binary number representations; again, we write code(q) for the word in $\{\mathbf{0},\mathbf{1}\}^*$ that represents the state q.

Instructions are represented by the obvious string of letters in $\mathbb{W}$ using $\mathbf{+}$, $\mathbf{?}$, and $\mathbf{-}$ to represent the types of instructions. E.g., $+(k,q,q')$ will be represented by the word $\mathbf{+}(\text{code}(k)\mathbf{,}\text{code}(q)\mathbf{,}\text{code}(q'))$. If I is an instruction, we once more write code(I) for the word representing it.

By the fact that only the number of states matters, we can assume a fixed reservoir of countably many states $\{q_i \,;\, i \in \mathbb{N}\}$ used by all machines.
Then we encode the state q_i by

code(i)

[using our code for natural numbers].

Instructions are, e.g., $+(k,q,q')$, already strings of symbols, numbers, letters, and states. So we can encode it as the corresponding Σ-word.

<u>Example</u> code$(+(k,q,q')) :=$

$\mathbf{+}\mathbf{(}\text{code}(k)\mathbf{,}\,\text{code}(q)\mathbf{,}\,\text{code}(q')\mathbf{)}$

A program line of the form $q \mapsto P(q)$ will be represented by $\mathrm{code}(q) \mapsto \mathrm{code}(P(q))$, and finally, a register machine $M = (\Sigma, Q, P)$ will be represented by the word

$$\mathrm{code}(q_0) \mapsto \mathrm{code}(P(q_0)), \mathrm{code}(q_1) \mapsto \mathrm{code}(P(q_1)), \ldots, \mathrm{code}(q_n) \mapsto \mathrm{code}(P(q_n))$$

if Q has $n+1$ elements; we write $\mathrm{code}(M)$ for that word.

We encode a sequence $\vec{w} \in \mathbb{W}^{n+1}$ by a single word in the enlarged alphabet Σ', viz. $w_0\Box\ldots\Box w_n\Box$; we write $\mathrm{code}(\vec{w})$ for this word. If $q \in Q$ and $\vec{w} \in \mathbb{W}^{n+1}$, we encode the configuration $C = (q, \vec{w})$ by the word $\mathrm{code}(q)\Box\mathrm{code}(\vec{w})$; once more, we write $\mathrm{code}(C)$ for this word.

A program was a map from Q to the set of instructions.

If $q \longmapsto P(q)$ is a program line, we write

$$\mathrm{code}(q \longmapsto P(q)) := \mathrm{code}(q) \mapsto \mathrm{code}(P(q))$$

Similarly, a whole program now becomes

code(M) :=

$$\mathrm{code}(q_0) \mapsto \mathrm{code}(P(q_0)), \ldots, \mathrm{code}(q_n) \mapsto \mathrm{code}(P(q_n))$$

[Note that the order in which program lines occur does not matter for the machine, but needs to be fixed in our coding.]

Sequences of words

$$\mathrm{code}(\vec{w}) := w_0 \Box w_1 \Box \ldots \Box w_n \Box.$$

$\vec{w} = (w_0, \ldots, w_n)$

C configuration $C = (q, \vec{w})$

$$\mathrm{code}(C) := \mathrm{code}(q) \Box \mathrm{code}(\vec{w}).$$

IMPORTANT OBSERVATION

Suppose I am given

$c \mapsto c' \triangleright \cdots \triangleright c'' \mapsto c''' =$ code(M) and

code$(q)\,\square\,$code$(\vec{w}) =$ code(C)

where M is a RM and C is a configuration.

There is a unique C' s.t. M "transforms C to C'".

Then the operation

$$\text{code}(M), \text{code}(C) \longmapsto \text{code}(C')$$

can be performed by a RM.

<u>Proposition</u> The total operation that takes code(M), code$(\vec{w})$ and v and outputs the code of $C(M, \vec{w}, \#v)$ can be performed by a RM.

<u>Proof.</u> Remember that C was defined by recursion:

$$h(\text{code}(M), \text{code}(\vec{w}), \varepsilon) := \text{code}(q_S)\,\square\,\text{code}(\vec{w})$$

(*) $$h(\text{code}(M), \text{code}(\vec{w}), s(v)) := \text{code}(C')$$

where $C(M, \vec{w}, \#v)$ is transformed by M to C'. This is an application of recursion, so h is computable.
q.e.d.

Corollary 4.24. The total operation "check whether M has halted with input $\vec{w}$ after at most $\#v$ steps" can be performed by a register machine. We call the corresponding total (characteristic) function

$$t_{M,k}(\vec{w}, v) := \begin{cases} a & M \text{ has halted with input } \vec{w} \text{ after at most } \#v \text{ steps and} \\ \varepsilon & \text{otherwise} \end{cases}$$

the *truncated computation of M*.

$$t_{M,k}(\vec{w}, v) = \begin{cases} a & M \text{ has halted with input } \vec{w} \text{ after at most } \#v \text{ steps} \\ \varepsilon & \text{o/w} \end{cases}$$

Proof.

1. Find an irrelevant register k and empty it. From now write v^* for the current content of k.
2. Calculate $h(\text{code}(M), \text{code}(\vec{w}), v^*)$.
3. Check whether this starts with $\text{code}(q_H)$. If so, write a in register 0 and halt.
4. If not, check whether $v^* = v$. If so, then write ε in register and halt;
5. if not, apply successor fn to v^* and go to 2.

q.e.d.

Theorem 4.25 (The Software Principle)**.** There is a register machine U, called a *universal register machine* such that for every register machine M and sequence of words $\vec{w}$, we have that

$$f_{U,2}(v,u) = \begin{cases} f_{M,k}(\vec{w}) & \text{if } v = \text{code}(M) \text{ for a register machine } M \\ & \quad \text{and } u = \text{code}(\vec{w}) \text{ for a sequence of words of length } k, \\ \uparrow & \text{otherwise.} \end{cases}$$

REMARK Note that for every $n \in \mathbb{N}$, there are computable functions that cannot be computed by a RM with $< n$ states.
And yet, there is a fixed U with some fixed # of states, that mimics all of them.
It can only do this by using the code of the machine it mimics [note that $|\text{code}(M)| > n$] as input.
This is what we call SOFTWARE.

Proof of the software principle.
We describe U by describing the operation it performs.
1. Check whether v is a code for a RM and whether u is a code for $\vec{w} \in W^k$.
If not, don't halt.
If so, go to 2.

2. Find an irrelevant register, say k and empty it.
From now on, write v^* for the CURRENT content of reg. k.

3. Calculate
$h(v, v, v^*)$ where h is the function producing the computation seq. (*) on page 9

4. Check whether it starts with code(q_H).
If not, then apply succ. fn to v^* and go to 3.

5. If it starts w/ code(q_H), then
$h(v, v, v^*) = \text{code}(q_H) \square \text{code}(\vec{v})$.
Write v_0 into register 0 and halt.

q.e.d.

Theorem 4.26 (The *s-m-n* Theorem)**.** Let $g : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ be any partial computable function. Then there is a total computable function $h : \mathbb{W} \to \mathbb{W}$ such that for all $v \in \mathbb{W}$ and all $\vec{w} \in \mathbb{W}^k$, we have $f_{h(v),k}(\vec{w}) = g(\vec{w}, v)$.

The curious name of this theorem derives from the notation S^m_n used for the function h in the original publication.[16] The *s-m-n* Theorem pulls one of the parameters of the function g into the index. This process is also called *Currying*, after the logician Haskell Curry (1900–1982).[17]

$g : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ partial computable

then there is <u>total computable</u> $h : \mathbb{W} \to \mathbb{W}$

$$f_{h(v),k}(\vec{w}) = g(\vec{w}, v)$$

[It's clear that for each fixed v, there is v' s.t.

$$f_{v',k}(\vec{w}) = g(\vec{w}, v).$$

It's not obvious that there is a computable function calculating such a v'.]

<u>Proof.</u> For any fixed v, the operation

$$\vec{w} \longmapsto (\vec{w}, v)$$

can be performed by a RM, in fact a concretely given RM, let's call it M_v.

The fact that it is concretely given, means that

$$v \longmapsto \mathrm{code}(M_v)$$

is computable.

Since $g : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ was computable, there is an M s.t.

$$f_{M,k+1} = g.$$

$$\text{code}(M_v) \longmapsto (\text{code}(M_v), \text{code}(M))$$

can be performed by a RM.

Put together

$$v \longmapsto \text{code}(M_v) \longmapsto (\text{code}(M_v), \text{code}(M))$$

can be performed by a RM.

$$\vec{w} \longmapsto (\vec{w}, v) \longmapsto g(\vec{w}, v)$$

can be performed by the RM that corresponds to the concatenation of $f_{M_v,k}$ and $f_{M,k+1}$.

Remember that the proof of the subroutine lemma was constructive, so if $M \circ N$ is the concatenation RM of M and N, then

$$(\text{code}(M), \text{code}(N)) \longmapsto \text{code}(M \circ N)$$

is computable.

Together $v \longmapsto \text{code}(M_v) \longmapsto (\text{code}(M_v), \text{code}(M))$
$\longmapsto \text{code}(M_v \circ M)$

is computable. It is total; let h be the function computed by it.

q.e.d.