Lessons covered by this meeting according to schedule:

- 01/10/2024
  - Historical introduction. From logics to computability to modern computer science.
- 07/10/2024
  - Effective procedures and computable functions. Relations, functions, sets and cardinality. Existence of non-computable functions.
- 08/10/2024
  - URM-computability. The class C of URM-computable functions. Examples. [§1.2, §1.3]
- 14/10/2024
  - Exercises on some variants of the URM machine
  - Decidable predicates. [§1.4]
  - Computability on domains different from the natural numbers. [§1.5, §3.6]
- 15/10/2024
  - Generating computable functions: Notation. Closure under generalized composition (Substitution) [§2.1, §2.2, §2.3]
  - Primitive recursion and examples [§2.4]

Actual lesson

- **Existence of computable functions**

Description of effective procedures as sequences of elementary steps transforming input into outputs. Make people think about the sense of functions (mapping/etc.) - § page 52 of my notes.

Notion of total function: domain and codomain/image

- W_x: function domain
  - Letter W *probably* stands for *writable*, so it is possible to write down all the inputs for which the function is defined)

- E_x : image of a function (aka, all the function outputs or the values in the codomain – the first definition is less math-like and more human-like, I'd say, but let's try to be precise)
  - remember the codomain is also written as $cod(f)$, where f is a function
  - remember also the image can be written as $img(f)$
  - Letter E *probably* stands for *enumerable* because it is possible to enumerate all the outputs that the function can produce.

Notion of subtraction (monus) with example

$$\textbf{subtraction } x \dot{-} y = \begin{cases} x - y & x \geqslant y \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{ll} x \dot{-} 0 = x & f(x) = x \\ x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 & g(x,y,z) = z \dot{-} 1 \end{array}$$

This represent a subtraction which *will never give you negative results*, so it's always positive and well-defined with no problems (if the subtraction gives say -1, the calculation will give 0, something like that).

More precisely:

- the subtraction with a point on top indicates something like a normal subtraction, it saturates to zero when we get a negative value (not like a normal subtraction but defined only for natural numbers). Examples:
  - $3 - 4 = -1 \notin \mathbb{N}$
  - $3 \dot{-} 4 = 0$
- this is technically called "truncated subtraction", as shown here

> Regular subtraction is not well-defined on the natural numbers. In natural number contexts one often deals instead with *truncated subtraction*, which is defined:
>
> $$a \dot{-} b = \begin{cases} 0, & \text{if } a \leq b \\ a - b & \text{if } a \geq b \end{cases}$$

You can see here it's called *monus* (or also *cut-off subtraction* as evidenced by Cutland, p. 241). As you can see above, the function is primitive recursive and usually you put a point because you subtract from the highest value the lowest, so this ensures "it's all good".

- **Existence of non-computable function**

Definition of partial function/totality

Total = defined for all natural numbers (*all set*), assigning a unique element to each output from input

- A function is total if it's defined for cases and returns an output for every single possible input (covers all possibilities, in other terms) – so, a function by cases is total by construction
- Technically, a function is total when f is defined for every input on set X
- Diagonalization here is a powerful tool
  - We can use diagonalization to make a partial computable function that differs from every total computable function
  - It states that there are sets where you can't list all of their members sequentially. It assumes to have an infinite list of elements, which can't be because the underlying thing is total
    - If a list of a set of these strings exists…
    - …then there also exists a string that is not in the list
  - Given it does not happen, there is the contradiction: the function is total, but you find a value which is not on the list, while being defined for all values

Partial: defined *only on a subset* of a specified set. Consider a partial function from x to y, this will assign at most one element of y to every element of x

Computable = when referred to a specific model of computation (usually Turing Machines, here also URM machines). Recall the definition given here:

- A computable problem/function is one where the steps of computation are precisely defined and execution of the algorithm will always terminate in a finite number of steps, yielding a well-defined output.

- Examples of computable problems include addition, multiplication, exponentiation for integer inputs, as well as problems like determining if a number is prime or solving linear equations - all of which can be solved by unambiguous, terminating algorithms.

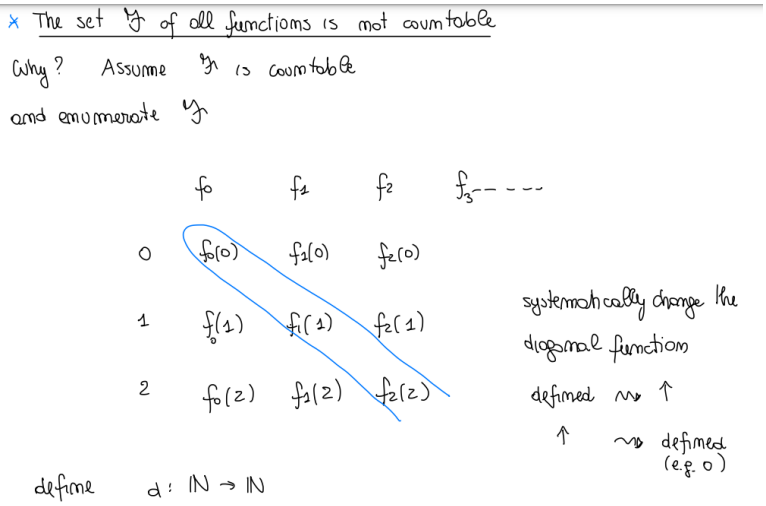
- In contrast, problems like the halting problem are not computable as there is no single algorithm that can correctly determine the behavior of all programs in a finite number of steps.

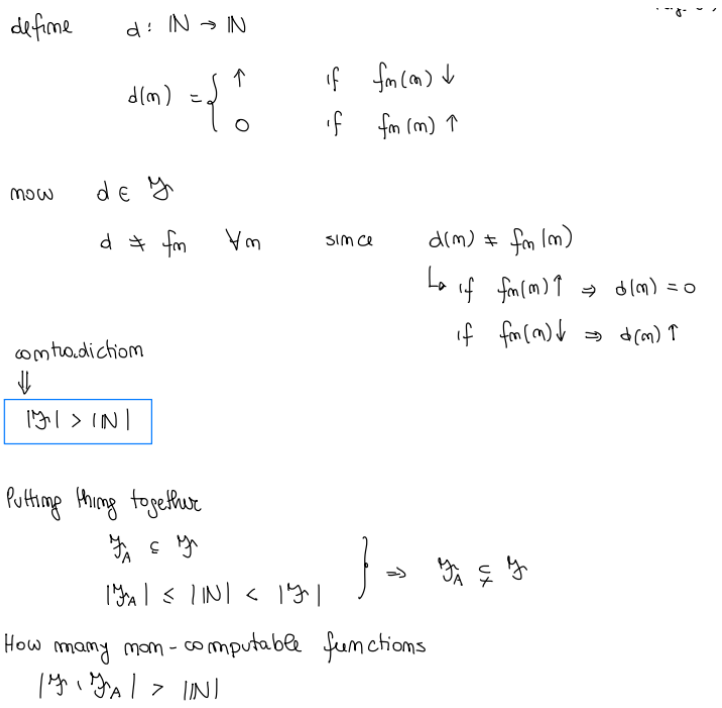
In computability theory, a general recursive function is a partial function from the integers to the integers; no algorithm can exist for deciding whether an arbitrary such function is in fact total.

Meaning of diagonalization and meaning with examples

- ϕ_x : primitive recursive $k - ary$ function given by the $x - th$ step of enumeration, so it can be different from the function (so, subinputs over following computations of a function). Usually, this is useful for diagonalization arguments, which is different from $f(x)$

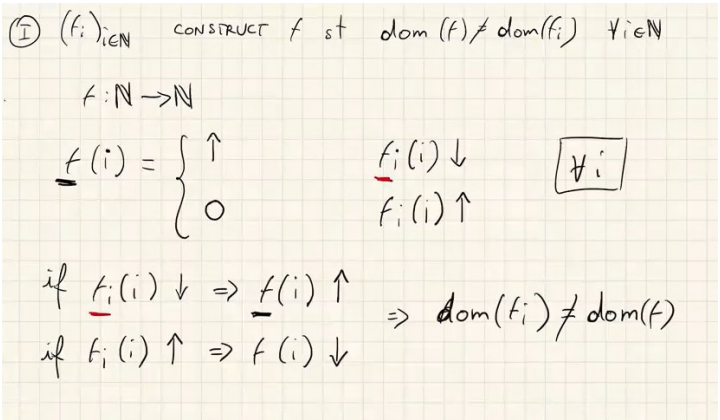
Let's think on the actual proof...





The idea of these exercises is to build a function which itself is built to be different from every single other function of the same family, otherwise it is undefined.

Then, we have that the recursion of the function, say $\phi_x(x) \neq f(x)$, which happens because the domain is built to be different from every value. An example that puts this explicitly:



- In this case, consider the function are total
  - So, they have to define and handle all cases by definition
- In this case, there are notable total non-computable functions; the function is built to differ from its own values by recursion
- We then say $f(x) \neq \phi_x(x)$ since this holds by construction (just use the problem conditions replacing $f(x)$ with $\phi_x(x)$)

Description of functions and subfunctions, which may be finite or not

- θ: finite subfunction
  - used in Rice-Shapiro context to show there is at least one part which has properties the rest of the considered set does not have

> Is there a non-computable total function $f : \mathbb{N} \to \mathbb{N}$ such that $img(f) = \mathbb{N}\backslash\{0\}$? (recall that $img(f) = \{f(x) \mid x \in \mathbb{N}\}$) Provide an example or show that such a function cannot exist.
>
> **Solution:** Yes, such a function exists. For instance one can define $f : \mathbb{N} \to \mathbb{N}$
>
> $$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in W_x \\ 1 & \text{otherwise} \end{cases}$$
>
> Observe that
>
> - The function f is total by construction.
> - The function is not computable, since for all $x \in \mathbb{N}$, $f \neq \varphi_x$. In fact if $\varphi_x(x) \downarrow$ then $f(x) = \varphi_x(x) + 1 \neq \varphi_x(x)$. If instead, $\varphi_x(x) \uparrow$ then $f(x) = 1 \neq \varphi_x(x)$.
> - $img(f) = \mathbb{N}\backslash\{0\}$.
>
> We prove separately the two inclusions. Clearly $img(f) \subseteq \mathbb{N}\backslash\{0\}$. In fact, for all $x \in \mathbb{N}$, we have $f(x) = 1$ if $\varphi_x(x) \uparrow$ and $\varphi_x(x) + 1$ when $\varphi_x(x) \downarrow$. Thus $f(x) > 0$ i.e., $f(x) \in \mathbb{N}\backslash\{0\}$. Conversely, also $\mathbb{N}\backslash\{0\} \subseteq img(f)$. In fact, for all $n \in \mathbb{N}\backslash\{0\}$, take any index e of the constant $n - 1$. Then $f(e) = \varphi_e(e) + 1 = n - 1 + 1 = n$. Thus $n \in img(f)$.

- **URM machine**

It has different instructions:

- *zero* $Z(n)$, which sets the content of register R_n to zero: $r_n \leftarrow 0$
- *successor* $S(n)$, which increments by 1 the content of register R_n: $r_n \leftarrow r_n + 1$
- *transfer* $T(m, n)$, which transfers the content of register R_m into R_n, which R_m staying untouched: $r_n \leftarrow r_m$
- *conditional jump*: $J(m, n, t)$, which compares the content of register R_m and R_n, so:
  - if $r_m = r_n$ then jumps to I_t (jumps to t-th instruction)
  - otherwise, it will continue with the next instruction

Existence of URM-computable functions

> DEFINITION 3.6 (URM-computable function). A function $f : \mathbb{N}^k \to \mathbb{N}$ is said to be **URM-computable** if there exists a URM program P such that for all $(a_1, \ldots, a_k) \in \mathbb{N}^k$ and $a \in \mathbb{N}$, $P(a_1, \ldots, a_k) \downarrow$ if and only if $(a_1, \ldots, a_k) \in dom(f)$ and $f(a_1, \ldots, a_k) = a$.
>
> In this case we say that P computes f.

Model of computation: also like Turing Machines and whatnot.

Concepts of: register configuration, program counter

Introduces the concept of memory, convergence and divergence (possibly partial).

How to structure such program proofs:

- Like a TM → variant contained inside of normal (subseteq) and vice versa
- It has to be well-formed (without loss of generality – wlog)
- Has to be shown by induction on the number of steps of execution by a program
- Usually, it happens on some kind of jump to subroutines then loop

- Has to happen that "every program executes such that $f_p^k = f_{p'}^k$" on the same number of instructions

Exercise 1.4(p). Consider the subclass of URM programs where, if the i-th instruction is a jump instruction $J(m, n, t)$, then $t > i$. Prove that the functions computable by programs in such subclass are all total.

Solution: Given a program P prove, by induction on t, that the instruction to execute at the $t + 1$-th step has an index greater than t. This implies that the program will end in at most $l(P)$ steps. □

Let C^f be the class of functions computable by URM programs in this subclass. We will prove that C^f ⊆ C and that all functions in C^f are total.

Step 1: Show C^f ⊆ C

This is trivial as any program in our subclass is also a valid URM program.

Step 2: Prove totality

Let P be a program in this subclass. We will prove that P always terminates, thus computing a total function.

Without loss of generality (wlog), assume P is in standard form, i.e., if it terminates, it does so at instruction l(P) + 1.

We prove by induction on the number of execution steps t that after t steps, P is executing an instruction with index j > t.

Base case (t = 0):

At step 0, P is at its first instruction I_1. This holds trivially as 1 > 0.

Inductive hypothesis:

Assume that for some k, 0 ≤ k < l(P), after k steps, P is executing instruction I_j with j > k.

Inductive step:

We need to prove that after k+1 steps, P will be executing an instruction I_l with l > k+1.

Case 1: If I_j is not a jump instruction, then P moves to instruction I_(j+1). Since j > k, we have j+1 > k+1.

Case 2: If I_j is a jump instruction J(m,n,t), then by the condition of the subclass, t > j. Since j > k, we have t > k+1.

In both cases, for step k+1, P moves to an instruction with index greater than k+1.

By the principle of mathematical induction, for all steps t, 0 ≤ t < l(P), P is always executing an instruction with index greater than t.

Therefore, P will inevitably reach a step l(P) where it attempts to execute a non-existent instruction I_(l(P)+1), causing termination.

Formally, we can state:

∀P ∈ subclass, ∀x ∈ N^k, ∃t ≤ l(P) such that P(x) terminates in t steps.

This proves that for any f ∈ C^f, ∀x ∈ N^k, ∃y ∈ N such that f(x) = y.

Therefore, all functions in C^f are total.

Conclusion:

We have shown that C^f ⊆ C and that every f ∈ C^f is total. Thus, the functions computable by programs in this subclass are all total.

- **Decidable predicates**

A predicate $Q(\vec{x}) \subseteq \mathbb{N}^k$ is decidable if the characteristic function $\chi_Q : \mathbb{N}^k \to \mathbb{N}$ defined by

$$\chi_Q(\vec{x}) = \begin{cases} 1 & \text{if } Q(\vec{x}) \\ 0 & \text{otherwise} \end{cases}$$

is computable.

Description of meaning of characteristic function.

In mathematics, we often want to express *properties*. Consider as mathematical property the *divisor*:

$$div(x, y) = \text{x divides y}, div \subseteq \mathbb{N}\ x\ \mathbb{N}$$

$$div = \{(n, m * k) \mid n, k \in \mathbb{N}\}$$

As computer scientists, we can also see the divisor as a function:

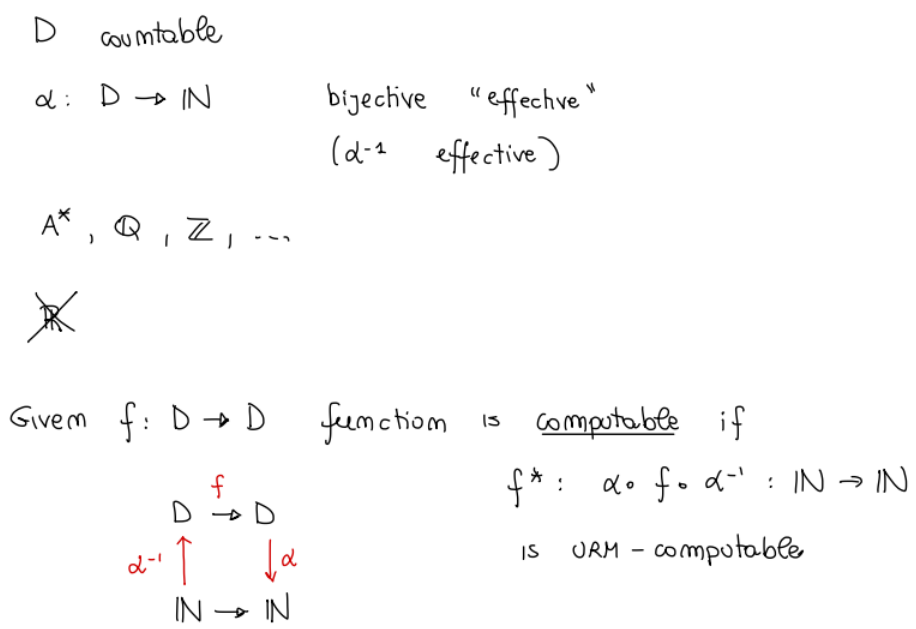
$$div: \mathbb{N}\ x\ \mathbb{N} \to \{true, false\}$$

$$div = \begin{cases} true & \text{if } m \text{ is a divisor of } n \\ false & \text{otherwise} \end{cases}$$

In the context of computability and formal logic, we introduce the concept of a predicate, which is a statement or function that takes one or more inputs and evaluates to either *true* or *false*, typically based on some condition or relationship.

They have exercises: quote Structure Theorem and Projection Theorem (see examples)

Computability on other domains



It's important since we can easily define the functions by cases and "be covered" under all circumstances.

- **Generation of computable functions**

A function will be *computable* if it can be obtained from a set of basic operations that are known to be computable. Essentially, we show that having two functions f_1, f_2 we produce an operation inside $op(f_1, f_2)$ in a way that composing them (for example, via $op(f_1, f_2)$) is still in $\mathcal{C}$.

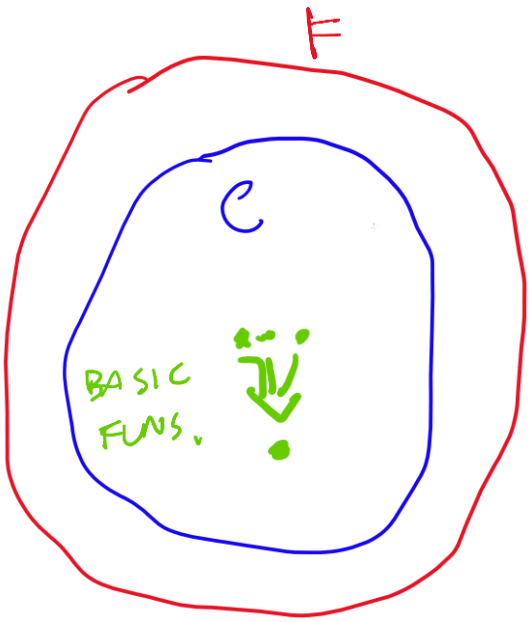
We define the composition, given $f: \mathbb{N}^k \to \mathbb{N}, g_1, \dots g_k: \mathbb{N}^n \to \mathbb{N}$ you define $h: \mathbb{N}^n \to \mathbb{N}$ $for\ \vec{x} \in \mathbb{N}^n$

$$h(\vec{x}) = \begin{cases} f(g_1(\vec{x}), \dots, g_k(\vec{x})) & \text{if } g_1(\vec{x}) \downarrow, \dots, g_k(\vec{x}) \downarrow \text{ and } f(g_1(\vec{x}), \dots, g_k(\vec{x})) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

(In words: the composition function will be made on all the subfunctions if they all halt)

The class $\mathcal{C}$ will be closed under:

- (generalized) composition
- primitive recursion
- unbounded minimalisation



To prove a function f is computable, we can write a URM program of use the closure theorems of C choosing the operations carefully (the ones listed above).

- **Primitive recursion**

Solution: The set $\mathcal{PR}$ of primitive recursive functions is the smallest set of functions that contains the basic functions:

1. $\mathbf{0} : \mathbb{N} \to \mathbb{N}$defined by $\mathbf{0}(x) = 0$ for each $x \in \mathbb{N}$;
2. $\mathbf{s} : \mathbb{N} \to \mathbb{N}$defined by $\mathbf{s}(x) = x + 1$ for each $x \in \mathbb{N}$;
3. $\mathbf{U}_j^k : \mathbb{N}^k \to \mathbb{N}$defined by $\mathbf{U}_j^k(x_1, \ldots, x_k) = x_j$ for each $(x_1, \ldots, x_k) \in \mathbb{N}^k$.

and which is closed with respect to generalized composition and primitive recursion, defined as follows. Given the functions $f_1, \ldots, f_n : \mathbb{N}^k \to \mathbb{N}$and $g : \mathbb{N}^n \to \mathbb{N}$ their generalized composition is the function $h : \mathbb{N}^k \to \mathbb{N}$ defined by:

$$h(\vec{x}) = g(f_1(\vec{x}), \ldots, f_n(\vec{x})).$$

Given the functions $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$ the function defined by primitive recursion is $h : \mathbb{N}^{k+1} \to \mathbb{N}$:

$$\begin{cases} h(\vec{x}, 0) = f(\vec{x}) \\ h(\vec{x}, y+1) = g(\vec{x}, y, h(\vec{x}, y)) \end{cases}$$

A whole set of functions do this: sum, product, exponential, predecessor, difference, sign, negative sign.

Define the class of primitive recursive functions. Using only the definition show that the function $f : \mathbb{N} \to \mathbb{N}$ defined below is primitive recursive

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{if } x \text{ is odd} \end{cases}$$

Solution: The class of primitive recursive functions is the least class of functions $\mathcal{PR} \subseteq \bigcup_k (\mathbb{N}^k \to \mathbb{N})$ containing the base functions (zero, successor, projections) and and closed under composition and primitive recursion.

In order to show that f is primitive recursive observe that it can be defined as

$$\begin{cases} f(0) & = 1 = succ(0) \\ f(y+1) & = \overline{sg}(y) \end{cases}$$

where $\overline{sg}$ is the complemented sign, which can be defined as

$$\begin{cases} \overline{sg}(0) & = 1 = succ(0) \\ \overline{sg}(y+1) & = 0 \end{cases}$$

Exercise

Define PR. Let $f: \mathbb{N}^2 \to \mathbb{N}$ $f(x,y) = 2^y x$

Show $f \in PR$ by using only the definition of PR

$$\begin{cases} f(x,0) = 2^0 \cdot x = x \\ f(x,y+1) = 2^{y+1} \cdot x = 2\, \widetilde{2^y x} \end{cases}$$

$$= 2\, f(x,y) = \text{twice}(f(x,y))$$

$$\begin{cases} \text{twice}(0) = 0 \\ \text{twice}(y+1) = \text{twice}(y) + 2 \end{cases}$$

$$= \text{succ}(\text{succ}(\text{twice}(y)))$$