Lessons touched by this meeting according to schedule:

- 8. 04/11/2024
  - Proof of the fact that the class of partial recursive functions coincide with the class of URM-computable functions
  - Primitive recursive functions. [§3.3]
  - Ackermann's functions: total, computable and not primitive recursive [partially in §2.5.5]
- 9. 05/11/2024
  - Enumerating URM programs and computable functions [§4.1, §4.2]

Class R (Partial Recursive Functions) is defined as the smallest class that:

a) Contains basic functions:

- Zero function
- Successor function
- Projections

b) Closed under:

- Composition
- Primitive recursion
- Minimalization

$R = C$ (URM-computable functions)

PR is defined similarly but WITHOUT minimalization:

- Contains same basic functions
- Closed under composition and primitive recursion only
- All functions in PR are total

Key property: PR functions use only bounded iteration

The Ackermann function $\psi : N^2 \to N$ is defined as:

$\psi(0,y) = y + 1$

$\psi(x+1,0) = \psi(x,1)$

$\psi(x+1,y+1) = \psi(x,\psi(x+1,y))$

	y=0	y=1	y=2	y=3	y=4	
x=0	1	2	3	4	5	(y+1)
x=1	2	3	4	5	6	(y+2)
x=2	3	5	7	9	11	(2y+3)
x=3	5	13	29	61	125	(2^(y+3)-3)
x=4	13	65533	...	...	...	(grows extremely fast)

The function directly addressed Hilbert's conjecture that $PR = R \cap Tot$ (that all total computable functions could be primitive recursive). The materials show this was false by proving:

- $PR \subsetneq R \cap Tot$ (strict inclusion)
- $\psi \in R \cap Tot$ but $\psi \notin PR$

The function demonstrates fundamental differences between computational models:

- `C_for (FOR loops only) = PR`
- `C_for,while (FOR and WHILE loops) = C = R`

This shows:

- PR corresponds to bounded iterations (FOR loops)
- Some computable functions require unbounded iterations (WHILE loops)
- The Ackermann function requires the latter

Properties:

1. ψ is total (always terminates and is defined for all inputs)

2. ψ is computable (ψ ∈ R ad it ca be programmed)

3. ψ ∉ PR (not primitive recursive) – FOR loops alone not sufficient for all computable functions, given some total computable functions require unbounded iteration (WHILE)

- The function operates on (N^2, ≤lex)
- Each recursive call reduces arguments in lexicographical order
- Well-foundedness guarantees termination since:
    - No infinite descending chains exist
    - Each call must eventually reach base case
    - Its growth rate shows that its nesting depth cannot be bounded by any PR function, since arguments decrease in lexicographical order
    - This happens since having "j" for loops cannot contain the elementary growth of the function, since PR functions do not have enough power to do that

We also know how to enumerate URM programs:

`γ: P → N (where P is the set of URM programs)`

This gives us a way to:

- Assign a unique number (Gödel number) to each program
- Convert between programs and numbers effectively
- Create a systematic way to talk about all possible programs

This holds for all functions:

ϕ^n_k: the k-ary function computed by program P_n

W_n = domain of ϕ_n

E_n = codomain of ϕ_n

Key points:

- The set of all URM programs is countable
- The set of all computable functions is countable
- We can list all possible programs/functions

- Not all functions are computable (uncountability argument)
- Many programs compute the same function
- There are infinitely many programs for each computable function

We represent the enumeration of functions and programs the following way:

Functions:	ϕ_0	ϕ_1	ϕ_2	ϕ_3	...
Inputs:					
0	a_{00}	a_{10}	a_{20}	a_{30}	...
1	a_{01}	a_{11}	a_{21}	a_{31}	...
2	a_{02}	a_{12}	a_{22}	a_{32}	...
3	a_{03}	a_{13}	a_{23}	a_{33}	...
...	...	...	...	...	...

We can define a function f that differs from every computable function by:

$$f(n) = \begin{cases} \phi_n(n) + 1 & \text{if } \phi_n(n)\downarrow \\ 0 & \text{if } \phi_n(n)\uparrow \end{cases}$$

For any n:
- If $\phi_n(n)\downarrow$, then $f(n) = \phi_n(n) + 1 \neq \phi_n(n)$
- If $\phi_n(n)\uparrow$, then $f(n) = 0 \neq \phi_n(n)$

Therefore: $f \neq \phi_n$ for all n

A more concrete example and point:

$\phi_0(0) = 3$	$f(0) = 4$	(differs from ϕ_0)
$\phi_1(1) = 1$	$f(1) = 2$	(differs from ϕ_1)
$\phi_2(2)\uparrow$	$f(2) = 0$	(differs from ϕ_2)
$\phi_3(3) = 0$	$f(3) = 1$	(differs from ϕ_3)

- If all functions were computable
- Then f would be computable
- Then $f = \phi_k$ for some k
- But $f(k) \neq \phi_k(k)$ by construction
- Contradiction!

Using diagonalization, we can always construct a function different from all computable functions.

Consider an example (5.2):

> **Exercise 5.2**(p). A function $f : \mathbb{N} \to \mathbb{N}$ is called *total increasing* when it is total and for each $x, y \in \mathbb{N}$, if $x < y$ then $f(x) < f(y)$. Prove that the set of total increasing functions is not countable.

Assume the set is countable. Then there exists an enumeration: $\{f_n\}_n \in \mathbb{N}$ of all total increasing functions. We are constructing a new function:

Define g: $\mathbb{N} \to \mathbb{N}$ as:

$$g(x) = 1 + \Sigma_{n=0}^{x} f_n(n)$$

We prove g is total increasing:

a) g is total (clearly defined for all inputs)

b) g is increasing:

$$g(x+1) = g(x) + f_{x+1}(x+1)$$

$$> g(x) \quad \text{(since } f_{x+1}(x+1) > 0\text{)}$$

For any $n \in \mathbb{N}$:

$$g(n) = 1 + \Sigma_{n\,i=0} f_i(i) \geq 1 + f_n(n)$$

Therefore $g(n) > f_n(n)$

Concluding:

- g is a total increasing function

- $g \neq f_n$ for all n

- This contradicts our assumption that $\{f_n\}$ enumerated ALL total increasing functions

- Therefore, the set cannot be countable

Providing another example - <u>exercise</u>:

> 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 $n \in img(f)$.

Exercise

Given a function $f: \mathbb{N} \to \mathbb{N}$:

`Z(f) = {g: ℕ→ℕ | ∀x∈ℕ. g(x)=f(x) ∨ g(x)=0}`

Show that Z(id) is not countable. Is it true that for every function f, Z(f) is not countable? In other words, Z(f) contains all functions that either match f or return 0 at each point.

Solution

- Part 1: Prove Z(id) is not countable

For id (identity function), Z(id) contains functions that at each point can either be x or 0

For any subset S ⊆ ℕ, we can define function gs:

```
gs(x) = {
    x  if x ∈ S
    0  if x ∉ S
}
```

- Every such gs is in Z(id)
- Different subsets S give different functions gs
- Since P(ℕ) (power set of ℕ) is uncountable, Z(id) must be uncountable

Is Z(f) uncountable for every function f?

- No! Counter-example: Consider constant function f(x) = 0
- Then Z(f) contains only one function (the constant 0 function)
- Because for each x:
    - either g(x) = f(x) = 0
    - or g(x) = 0
    - in both cases, g(x) = 0
- Therefore Z(f) is countable (finite in this case)

Therefore:

1. Z(id) is uncountable
2. Z(f) is not always uncountable (the constant 0 function provides a counterexample)

Exercise 6.6(p). Say if there can be a non-computable function $f : \mathbb{N} \to \mathbb{N}$ such that for any other non-computable function $g : \mathbb{N} \to \mathbb{N}$ the function $f + g$ defined by $(f + g)(x) = f(x) + g(x)$ is computable. Justify your answer (providing an example of such f, if it exists, or proving that cannot exist).

No such function f can exist. We will prove this by contradiction.

Assume there exists a non-computable function f : N → N such that for every non-computable function g : N → N, the function (f + g) is computable, where (f + g)(x) = f(x) + g(x).

Since the quantification over g is universal, this property must also hold when g = f. Therefore, taking g = f:

(f + f)(x) = f(x) + f(x) = 2f(x) must be computable.

However, if 2f(x) is computable, then:

f(x) = (2f(x))/2

would also be computable, since division by 2 is a computable operation.

This contradicts our initial assumption that f is non-computable.

Therefore, we can conclude that no such function f can exist.

This solution uses proof by contradiction and leverages the fact that if f + f is computable, then f itself would be computable through basic arithmetic operations, which contradicts the requirement that f be non-computable.

The key insight is that by considering g = f, we force f to be computable through the closure properties of computable functions, contradicting our assumption. This demonstrates the impossibility of finding such a function f.

Infact:

Solution: It cannot exist otherwise, since the quantification over g is universal, the property should also hold for $g = f$. Thus $f + f = 2f$ would be computable, which implies f computable. □

Let's pass to other kinds of exercises:

Exercise 2.5(p). Give the definition of the set $\mathcal{PR}$ of primitive recursive functions and, using only the definition, prove that $p_2 : \mathbb{N} \to \mathbb{N}$ defined by $p_2(y) = |y - 2|$ is primitive recursive.

Now, let's prove that p2 ∈ PR:

First, define the function p1 : N → N by p1(y) = |y - 1|. We can define p1 using primitive recursion:

p1(0) = 1

p1(y+1) = y

This can be written as:

p1(0) = S(Z(y))

p1(y+1) = P_1^1(y)

Both S(Z(y)) and P_1^1(y) are in PR, so p1 ∈ PR by the primitive recursion clause.

Now, we can define p2 using p1:

p2(y) = |y - 2| = |(y - 1) - 1| = p1(p1(y))

This is a composition of p1 with itself, and since p1 ∈ PR, we have p2 ∈ PR by the composition clause.

Therefore, p2(y) = |y - 2| is primitive recursive.

Exercise 2.4(p). Give the definition of the set $\mathcal{PR}$ of primitive recursive functions and, using only the definition, prove the function $half : \mathbb{N} \to \mathbb{N}$, defined by $half(x) = x/2$, is primitive recursive.

Let's start by defining two helper functions that are known to be primitive recursive:

1. The function sg : N → N, defined as: sg(0) = 1 sg(x+1) = 0
2. The function rm2 : N → N, which returns the remainder of the division of x by 2: rm2(0) = 0 rm2(x+1) = sg(rm2(x))

Now we can define the function half using primitive recursion:

half(0) = 0 half(x+1) = half(x) + rm2(x)

Let's verify that this definition satisfies the conditions for primitive recursive functions:

1. The base case, half(0) = 0, is the constant zero function, which is primitive recursive.
2. For the recursive case, half(x+1) = half(x) + rm2(x):
    - half(x) is the recursive call on a smaller argument, which is allowed in primitive recursion.
    - rm2(x) is a primitive recursive function, as shown above.
    - The addition of two primitive recursive functions (half(x) and rm2(x)) is also primitive recursive, due to the closure of primitive recursive functions under composition.

Therefore, half : N → N, defined by half(x) = x/2, is a primitive recursive function.

Exercise

Consider the URM variant of the URM machine obtained by removing the successor instruction S(n) and jump instruction J(m,n,t), and adding the instruction JS(m,n,t), which compares the contents of registers m and n, and if they coincide, it jumps to instruction t, otherwise it increments the m-th register and executes the next instruction.*

Determine the relation between the set C of functions computable by a URM* machine and the set C of functions computable by a standard URM machine. Is one included in the other? Is the inclusion strict? Justify your answers.*

Solution:

Clearly the instruction JS(m,n,t) can be simulated in the URM machine as follows: J(m,n,t) S(m)

Conversely, the instruction S(n) cannot be simulated. In fact, starting from a configuration with all registers at 0, there is no way to modify the content of any register: this would require the presence of two registers with different content and there are none.

More formally, given a program P and a number of arguments $k \in N$, denote by $q = \max\{\rho(P), k\} + 1$ the index of the first register not used and therefore initially at 0. By replacing in P each instruction S(n) with the instruction T(q,n), we obtain a URM* program that computes exactly the same function.

Therefore, $C^* \subsetneq C$, i.e., the inclusion is strict.

How to write minimalization – Exercises

You will see this more overtime, but – ways to write mu-operator:

For the sign functions part, I think the most "mechanical" way to do it is to:

- replace every predicate with its characteristic function (H becomes X_H for example). Remember that those are 1 when the predicate is true, 0 otherwise. Use the negated sign to make them 0 if true, 1 if false

- a=b becomes sg(|a-b|), which is 0 if true and 1 if false

- a>b becomes sg(a - b)

- a>=b becomes sg(a + 1 - b)

- OR operations become multiplications

- AND operations become additions

- NOT operations become negated signs

Exercise 1

Define the function absDiff : $N^2 \to N$ that computes the absolute difference between two natural numbers: absDiff(x, y) = |x - y|

Solution

We can express this using the sg and rm functions:

```
absDiff(x, y) = (x - y) * sg(x - y) + (y - x) * (1 - sg(x - y)) =
(x - y) * sg(x - y) + (y - x) * sg(y - x)
```

Since sg(x - y) = 1 if x ≥ y and 0 otherwise, this function will return x - y if x ≥ y, and y - x otherwise, effectively computing the absolute difference.

We can also write this using the μ operator and the rm function:

```
absDiff(x, y) = μz. (rm(z, x - y) = 0)
```

Here, the μ operator will find the least z such that z is divisible by |x - y|, which is exactly |x - y| itself.

Exercise 2

Define the functions quot : $N^2 \to N$ and rem : $N^2 \to N$ that compute the quotient and remainder of the division of x by y, respectively:

```
quot(x, y) = the largest integer q such that q * y ≤ x
```

```
rem(x, y) = x - quot(x, y) * y
```

Solution

We can express these using the μ operator and the rm function:

```
quot(x, y) = μq. (rm(x, q * y) < y)
```

```
rem(x, y) = rm(x, y * quot(x, y))
```

The μ operator in quot will find the largest q such that the remainder of x divided by q * y is less than y, effectively computing the quotient.