



COMPUTABILITY SIMPLE (FOR REAL)



Gabriel Rovesti

1 SUMMARY

1	Summary.....	0
2	A Useful Introduction: Swiss Knife for Everything Needed	6
2.1	How to do the exercises and FAQ	6
2.1.1	How to prove implications for URM machines.....	6
2.1.2	How to prove the primitive recursive exercises	8
2.1.3	What are those $E\bar{x}$ and $W\bar{x}$ I see everywhere?.....	8
2.1.4	What is exactly $\varphi\bar{x}$ I see everywhere?.....	8
2.1.5	Why the subtraction has a point on top?	9
2.1.6	How to prove the smn-theorem exercises	10
2.1.7	How to write non-computable functions	11
2.1.8	How to use diagonalization	11
2.1.9	How to prove decidability/semidecidability.....	13
2.1.10	How to write computable functions	13
2.1.11	What is that set K I always see?	15
2.1.12	How to use the minimalisation	15
2.1.13	What are those $w1, w2 \dots$ I see everywhere?	16
2.1.14	What is that subscript 1 over function composition?	17
2.1.15	What is the universal function and how to use it?.....	17
2.1.16	What are those S and H functions I see everywhere?	18
2.1.17	How to do recursive/r.e. exercises.....	19
2.1.18	How to write the negated sets	23
2.1.19	How to do the Second Recursion Theorem exercises.....	24
2.2	Swiss Knife of Practical Definitions.....	26
2.2.1	Totality and diagonalization	26
2.2.2	Minimalization	26
2.2.3	Why do we need to focus this much on Ackermann Function?.....	27
2.2.4	Recursiveness and types.....	28
2.2.5	Recursively enumerable and enumeration	29
2.2.6	Decidability and Semidecidability.....	30
2.2.7	Functionals and Fixed Points	30
2.3	Symbols and Acronyms	31
2.3.1	URM Machines Symbols	33
2.3.2	General Functions and Notation.....	34
2.3.3	Sets, Predicates and Characteristic Functions	35
2.3.4	All book notations.....	37

2.4	Swiss Knife of Useful Theoretical Definitions	39
2.4.1	Computable function	39
2.4.2	URM-Machine	39
2.4.3	URM-Computable function.....	39
2.4.4	Reduction.....	39
2.4.5	Recursive Set.....	40
2.4.6	Recursively Enumerable Set	40
2.4.7	Decidable Predicate.....	41
2.4.8	Bounded Minimalisation.....	41
2.4.9	Unbounded Minimalisation	41
2.4.10	Semi-decidable predicate.....	41
2.4.11	Partially recursive functions	42
2.4.12	Primitivfunctione Recursive Functions.....	42
2.4.13	Smn-Theorem	43
2.4.14	Structure Theorem	44
2.4.15	Projection Theorem.....	45
2.4.16	Saturated set	45
2.4.17	Rice's Theorem	45
2.4.18	Finite Function and Sub-function	46
2.4.19	Rice-Shapiro's Theorem	46
2.4.20	Myhill-Shepherdson Theorem.....	47
2.4.21	First Recursion Theorem	48
2.4.22	Second Recursion Theorem.....	49
3	Introduction to the course	50
4	Algorithms, effective procedures, non-computable functions	52
4.1	Existence of non-computable functions.....	54
5	URM Computability	58
5.1	URM-computable functions and examples	60
5.2	Exercises	62
6	Decidable predicates	70
6.1	Computability on other domains	71
7	Generation of computable functions	74
7.1	Generalized composition.....	75
7.2	Primitive recursion	77
7.2.1	Functions defined by primitive recursion.....	80
7.3	Bounded sum, bounded product and bounded quantification	82

7.4	Bounded minimalisation	84
7.5	Encoding in pairs	88
7.6	Unbounded minimalisation	89
8	Partial recursive functions	94
9	Primitive recursive functions	99
9.1	The Ackermann Function	99
9.2	Partially Ordered/Well Founded Posets	101
9.3	Complete/Well-founded Induction and Ackermann Proof	103
10	Enumerating URM programs	108
10.1	Exercises	115
11	Cantor Diagonalization Technique	118
11.1	Examples	119
12	Parametrisation/smn-theorem	125
12.1	smn-theorem	127
12.2	Simplified smn-theorem	131
12.3	Examples	132
13	Universal Function	134
13.1	Definition	135
13.2	Related Exercises	140
13.3	Effective operations of computable functions	143
13.4	Other exercises solved in lessons	148
14	Recursive sets	155
14.1	Definition	156
14.2	Reduction and Related Problems	157
15	Saturated Sets and Rice's Theorem	163
15.1	Saturated Sets	164
15.2	Rice's theorem	166
15.3	Examples	167
16	Recursively Enumerable Sets	170
16.1	Definition	170
16.2	Existential quantification	172
16.3	Structure Theorem	173
16.4	Projection Theorem	174
16.5	Other exercises from lessons	176
16.6	Recursively Enumerable Sets and Reducibility	178
17	Rice-Shapiro's Theorem	183

17.1	Definition.....	184
17.2	Proof.....	186
17.3	Examples	190
18	First Recursion Theorem	192
18.1	Recursive Functionals.....	195
18.2	Myhill-Shepherdson's theorem	196
18.3	Definition.....	197
19	Second Recursion Theorem.....	199
19.1	Definition and Proof Idea	199
19.2	Application Examples	201
20	Ending Lessons – Exercises	206
20.1	Exam of 19/01/2022	206
20.2	Various Exercises Solved (1/2)	212
20.3	Various Exercises Solved (2/2)	217
20.4	Solution of the Exercise on Random Numbers	221
21	Tutoring lessons 2023-2024	223
21.1	Tutoring 1: Primitive Recursion Exercises	223
21.2	Tutoring 2: Exercises on Diagonalization and Partial Recursive Functions.....	229
21.3	Tutoring 3: smn-theorem exercises	231
21.4	Tutoring 4: R.E. Sets	232
21.5	Tutoring 5: R.E. Sets and Reduction.....	234
21.6	Tutoring 6: R.e./Rice-Shapiro Exercises	237
21.7	Tutoring 7: R.e. and Reductions.....	239
21.8	Tutoring 8: All Kinds of Exercises	240
22	Many solved exercises with full commentary	242
22.1	URM Machines	242
22.2	Primitive Recursive Functions	255
22.3	smn-theorem	263
22.4	Decidability and Semidecidability	271
22.5	Numerability and Diagonalization.....	274
22.6	Functions and Computability	276
22.7	Reduction, Recursiveness and Recursive Enumerability.....	290
22.8	Characterization of sets	303
22.9	Second Recursion Theorem	350

Disclaimer

This file is the result of months of work, on a subject which is on paper the most interesting, but it's math all over again (oh no, Mario like) and a very formal one at that. The teacher is really good, but problem is: too much forced formal stuff and no real explanation on how to think and reason on the exercises, so you see stuff in front of your eyes which you never know where it comes from (because you were never told).

Here, every single concept is explained in precision, trying to simplify as much as possible concepts without loss of formality and generality. You will hopefully get the grasp and you will also find a lot of exercises trying to be solved or that were solved like that by the teacher or tutors of this course.

This subject is not for everybody; it is followable, but you have to reason in abstract and expect to be confused a lot. It's totally normal, still. Apart from a few geniuses (which I am not - this is all hard work, would love to get things first time, yeah), you are not alone. That's the reason of this file existence.

Hopefully, given this course is unfollowable in class without having absolutely complete notes on it first (you just write, or you just listen), this is a resource for you have just that. Consider also this is a work by a single person; the notes present on Moodle were translated in LaTeX by three different people and the professor gave assistance.

Here I did everything alone. Hope it will be useful; I was rarely thanked for works like this (I did many over the years, with the goal of being *simple for real*), but if at least I gave you some help, well, at least this makes something out of it. But I always do this for passion and never for any other goal rather than spreading knowledge and hopefully improve, making a change directly, in people. I believe in this (this is not slogan, it's what I think) – it can be me or anyone else. The only thing I care is doing something nice.

Learning can be fun, even with subjects like this. Humbly, this tries to display passion and careful precision over everything. Here, nothing is took for granted. Each notation, concept, function, is explained in words and tries to be understood in a simple and concrete way.

On this course organization:

- Consider the lessons are done until half of December, then the professor dedicates two/three lessons only for exercises
- The tutoring lessons are done until the end of January; in the dedicated chapter for exercises, you find the link to find recordings of old tutorings and even the ones of this year's file writing (23-24)
- The course had a partial exam up until it was in Italian, almost up until 19/20; topics touched for those were just before universal function
- In any case, the exam is written, oral is optional (if you are crazy enough to want "cum laude")

Consider also:

- there is an entire chapter with subsections of solved exercises, both from "exercises.pdf" or exams (even Italian exercises absent from "exercises.pdf" of Moodle, which I translated)
- for each exercises I try to give my take following Baldan notes as close as possible – I do not assume those to be correct, just consider them for your own idea and in case just for reference
 - o given the file of this work for a single person, I think you should understand :-D

We are computer scientists, yes. But we are also human beings, at least sometimes.

So, help is needed and do not be afraid to leave feedback over this file, we can discuss it together. Also, to thank me, it doesn't kill me that much.

2 A USEFUL INTRODUCTION: SWISS KNIFE FOR EVERYTHING NEEDED

This chapter includes explanation for every possible thing:

- an entire subsection dedicated first thing first to understand exercises (because that's what we all want to know, given for most of the course everything is taken for granted and never explained everywhere in detail, yay!)
- an entire subsection dedicated to make you get the grasp of the single concepts and understand everything in practice (so, it will be basically "get the idea")
- an entire subsection dedicated to symbols (which were never written comprehensively anywhere, yay again!)
- an entire subsection dedicated to theory definitions (for proofs, just take the notes – these are both for the crazy ones of you who want to take the oral exam but also for us ordinary people to easily get the grasp on where to use stuff and we care about it)

2.1 HOW TO DO THE EXERCISES AND FAQ

This entire subsection represents what an introduction should be: something useful and mandatory in a logical way. We like to understand stuff, right? This is actually meant for that, no more and no less. This is the result of careful observation and putting puzzle pieces together of concepts never actually explained by anybody. Here you go then.

2.1.1 How to prove implications for URM machines

This type of exercise and kind of straightforward (and usually asked in partial exams, so not focusing so much on those, I'd say, but always be prepared).

Usually here there is a language with more/different capabilities from the normal URM machine and there are two ways for this implication:

- set which contains more/different capabilities contained inside normal set
- normal set contained inside which contains more/different capabilities

This can be done in two ways (alternative to each other):

- Less formal proof showing how the new instruction can be encoded using a combination of "normal" instructions and a subroutine
- More rigorous proof showing inductively on the number of steps, that the maximum of the values contained in the registers at any time is bounded by the maximum value in the initial configuration (see exercise 1.5 inside "exercises.pdf" for this one)

This kind of exercises was mainly present only inside partial exams. More precisely:

- The exercise gives us a variant of the normal URM model which these basic instructions:
 - o *zero* $Z(n)$, which sets the content of register R_n to zero: $r_n \leftarrow 0$
 - o *successor* $S(n)$, which increments by 1 the content of register R_n : $r_n \leftarrow r_n + 1$
 - o *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
- We have to prove the inclusion of the computable sets in both ways
 - From modified URM to normal URM
 - From normal URM to modified URM
- Define \mathcal{C} for URM-machine and \mathcal{C}' (for example) the set of the model you have to show
- First step is showing $\mathcal{C}' \subseteq \mathcal{C}$
 - Not necessarily the new machine is more powerful, infact it may be even less powerful
 - Informally, we simply can code the “new” instruction/s in normal URM machine using a routine of some existing instructions (jump/transfer/successor/jump)
 - This is typically done considering say i the index of an unused register by the program and a subroutine
 - Formally, we prove $\mathcal{C}' \subseteq \mathcal{C}$ showing that, for each number of arguments k and for each program P using both sets of instructions we can obtain a URM program P' which computes the same function i.e. such that $f_{P'}^{(k)} = f_P^{(k)}$
 - The proof goes on by induction on the number of instructions h
 - ($h = 0$), usually trivial, it's already a URM program
 - ($h \rightarrow h + 1$), basically I will describe the logic
 - Describe as j for instance the index of instruction you want to replace and $l(P)$ the length of computed program
 - We can build a program P'' using a register not referenced in P , for instance $q = \max\{\rho(P), k\} + 1$ (ρ is the largest unused register)
 - Show that for the whole length of program, the jump to the subroutine can successfully replace the instruction wanted
 - The program P'' is s.t. $f_{P''}^{(k)} = f_P^{(k)}$ and it contains h instructions. By inductive hypothesis, there exists a URM program P' s.t. $f_{P'}^{(k)} = f_P^{(k)}$, which is the desired program
- Second step is showing $\mathcal{C} \subseteq \mathcal{C}'$
 - The usual question is if inclusion holds both ways or if it is strict
 - If this second part does not hold, then it is not strict
- Usually, this is similar to the one before, but this time around, instructions of normal URM have to be encoded using only the new machine
 - This one follows, if formally, exactly the same steps as before

Exercise to follow fairly complete on this: 1.5.

2.1.2 How to prove the primitive recursive exercises

These basically follow an intuitive track:

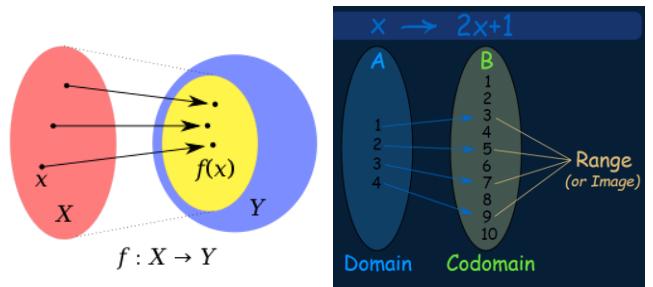
- We give the definition of \mathcal{PR}
- We define a function inside the set of primitive recursive functions by cases, covering base case and inductive case
 - o Assume [these](#) functions are primitive recursive
- In case this function needs other functions to describe it, other can be combined via primitive recursion themselves, hence proving the first one
- I suggest making a lot of practical exercises, for example also considering proofs [here](#)

Exercise to follow complete on this: 2.4 from “exercises.pdf”

2.1.3 What are those E_x and W_x I see everywhere?

Hard to write it explicitly, hey? No worries because here will be specified multiple times, but having a dedicated subsection to this is crucial.

- W_x : *domain* of function
 - o “Where the function is allowed to hit”
 - o All possible input values
- E_x : *codomain* of function
 - o “Where the function *does* hit”
 - o All possible output values



The *image* (or range) is a subset of codomain and are the values reached when you substitute x to get y on a subset (like the image you see); codomain are *all* the possible values reached.

2.1.4 What is exactly φ_x I see everywhere?

This is represented as “phi” as letter (many times here ϕ) – in LaTeX it’s “\varphi” (so, φ).

The first time this term is introduced is inside diagonalization, meaning “the function computed by the program of index x ” where we mean “index 0”, so φ_0 , “index 1”, so φ_1 , etc.

Specifically, φ_x can be defined as the x -th partial computable function, where x is the index of a Turing machine (or an equivalent formalism like the URM model) in some fixed effective enumeration.

Later on, we can see φ_x is simply $f(x)$ given it is defined. Consider the following infact:

Exercise 8.49. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid \varphi_x(y) = y^2 \text{ for infinite } y\}$, i.e., establish if B and \bar{B} are recursive/recursive enumerable.

Solution: We observe that B is saturated, since $B = \{x \mid \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \mid f(y) = y^2 \text{ for infinite } y\}$. Rice-Shapiro's theorem is used to deduce that both sets are not r.e.

Specifically, inside theory, when talking about programs enumeration, we have:

- $\varphi_n^{(k)} : \mathbb{N}^k \rightarrow \mathbb{N}$ (as the function of k arguments (k -ary function) computed by the program $P_n = \gamma^{(-1)}(n)$ can be seen as $\varphi_n^{(k)} = f_{P_n}^{(k)}$)

Consider this is justified by the universal function, which is always computable, and one can say φ_x is computable for program e : $\Psi_U(e, \vec{x}) = \varphi_e^k(\vec{x})$.

So: $\varphi_x(x)$ is $f(x)$, $\varphi_x(y)$ is $f(y)$

Written by Gabriel R.

2.1.5 Why the subtraction has a point on top?

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

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

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:
 - o $3 - 4 = -1 \notin \mathbb{N}$
 - o $3 \cdot\cdot= 0$
- 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”.

2.1.6 How to prove the smn-theorem exercises

Here we have to prove there exists a total computable function which has a particular and defined domain/codomain or both of them.

- In this case, simply define a function of two parameters which is the combination of other computable functions (so, it will probably use minimalization and things like sign functions)
- Use the smn-theorem definition to prove there exist an index capable of computation
- Use domain and codomain accordingly substituting the right indexes

Specifically:

- Give a function of two arguments $g(x, y)$
 - o Define a case for set definition
 - o Define a case for otherwise
- In this case, with smn-theorem exercises, it helps creating a function s.t.
 - o the domain is where the values exist
 - so, the positive case condition is the domain or less than the domain and has to include that case inside condition
 - o the codomain is the output we want to reach
 - after having written the cases, we see if the output/the computable function respects said condition
- It is computable, since it is defined by cases
- By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}$
 - o Write $\phi_{s(x)}(y) = g(x, y)$ and rewrite the function defined initially again
- As observed above
 - o $W_{s(x)} = \{x \mid g(x, y) \downarrow\} =$ prove you stay inside domain, getting the same value
 - o $E_{s(x)} = \{g(x, y) \mid x \in \mathbb{N}\} =$ prove you stay inside codomain, getting the same value

In case you have $E_{k(n)}$ and $W_{k(n)}$ inside the function definition (just notation here, folks, the concept holds the same way, you simply have n in place of x):

- simply use a function $f(n, x)$
- by smn theorem, there is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{k(n)}(x) = f(n, x) \quad \forall n, x \in \mathbb{N}$
- As observed above
 - o $W_{s(x)} = \{x \mid f(n, x) \downarrow\} =$ prove you stay inside domain, getting the same value
 - o $E_{s(x)} = \{f(n, x) \mid x \in \mathbb{N}\} =$ prove you stay inside codomain, getting the same value

2.1.7 How to write non-computable functions

Consider [this](#) one to get the concrete idea. Such functions are always total, since they are always defined (basically, covered by all cases in function definition).

We have different choices to follow:

- diagonalization (subsection ahead)
- use a known non computable function, like χ_K
 - o conditions are dependent on exercise, here reported just as an example

$$f(x) = \begin{cases} 0, & \text{if } x \leq 1 \\ \chi_K(x), & \text{otherwise} \end{cases}$$

- o the general structure would be using χ_K somewhere, it can be both on positive/otherwise case

- sometimes, it happens that we use functions and subfunctions

$$\theta(x) = \begin{cases} f(x), & \text{general condition (e.g. if } x < x_0) \\ \uparrow, & \text{otherwise} \end{cases}$$

$$f(x) = \begin{cases} \theta(x), & \text{general condition (e.g. if } x < x_0) \\ \text{value (e.g. } 0, k), & \text{otherwise} \end{cases}$$

- since the subfunction is finite, the function is too, and one can write it as a computable function

2.1.8 How to use diagonalization

This one is a direct consequence of the previous one.

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:

① $(f_i)_{i \in \mathbb{N}}$ CONSTRUCT f st $\text{dom}(f) \neq \text{dom}(f_i) \quad \forall i \in \mathbb{N}$

$f: \mathbb{N} \rightarrow \mathbb{N}$

$f(i) = \begin{cases} \uparrow & f_i(i) \downarrow \\ 0 & f_i(i) \uparrow \end{cases}$ $\boxed{\exists i}$

if $f_i(i) \downarrow \Rightarrow f(i) \uparrow \Rightarrow \text{dom}(f_i) \neq \text{dom}(f)$
 if $f_i(i) \uparrow \Rightarrow f(i) \downarrow$

- In this case, consider the function are total
 - o 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 $\phi_x(x)$)

Consider (conditions are dependent on exercise, here reported just as an example):

$$g(x) = \begin{cases} \phi_x(x) + 1, & x \in W_x \\ 0, & \text{otherwise} \end{cases}$$

More generally, it might be something like:

$$f(x) = \begin{cases} \text{something involving } \phi_x, & x \in W_x \\ 0, & \text{otherwise (so, } x \notin W_x\text{)} \end{cases}$$

- Consider the following notable examples from the course:

OBSERVATION 10.4. There exists a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

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

f is not computable because it differs from all computable functions. In fact

- if $\varphi_n(n) \downarrow$, then $f(n) = \varphi_n(n) + 1 \neq \varphi_n(n)$
- if $\varphi_n(n) \uparrow$, then $f(n) = 0 \neq \varphi_n(n)$

so

$$\forall n f \neq \varphi_n$$

OBSERVATION 10.5. There are infinitely many total non-computable functions of the following shape

$$f(n) = \begin{cases} \varphi_n(n) + k & n \in W_n \\ k & n \notin W_n \end{cases}$$

The book specifies the following over diagonalization:

We can summarise the diagonal method as we shall be using it, in the following way. Suppose that $\chi_0, \chi_1, \chi_2, \dots$ is an enumeration of objects of a certain kind (functions or sets of natural numbers). Then we can construct an object χ of the same kind that is different from every χ_n , using the following motto:

'Make χ and χ_n differ at n .'

The interpretation of the phrase *differ at n* depends on the kind of object involved. Functions may differ at n over whether they are defined, or in their values at n if defined there; with functions, there is usually freedom to construct χ so as to meet specific extra requirements; for instance, that χ be computable, or that its domain (or range) should differ from that of each χ_n .

In the case of sets, the question at n is whether or not n is a member.

Keep in mind that if you consider [this](#) one (quantification), this is universal and if one component is computable, also the other one is, because it is decidable.

TL; DR

- use K and χ_K
- use $\phi_x(x)$ in some form different by construction

2.1.9 How to prove decidability/semidecidability

In this case, we follow the definition, and we try to build a characteristic/semicharacteristic function, and we know respectively something is decidable/semidecidable. This kind of exercises basically refers to projection/structure theorem and usually involves building predicates in between them and prove those definitions while respecting the meaning of giving characteristic/semicharacteristic functions.

In such cases, projection theorem and structure theorem definitions and proofs come to help. Consider this correspondence (might seem obvious to you, but useful to know):

- recursive = decidable
- recursively enumerable = semidecidable

Consider also:

- structure says that starting from a semidecidable predicate, you build a semidecidable one such that it will terminate in a number of steps t
- projection says that starting from a semidecidable predicate, then you can apply the previous one and encode another semidecidable predicate

2.1.10 How to write computable functions

This is kinda hard to answer because there's no truly generic way to do this, but there are a lot of common situations.

For semicharacteristic functions the general approach is to write them as $1(\mu w \dots)$, and then you need to handle the ..., which you need to write as some sort of search where the condition is decidable/a total function. Writing such search/condition is the hardest part and also what the exercise is all about.

Initially you can write this condition using predicates (for example $=, >$, that is not just functions) and introducing new variables whenever you have a condition in the form "there exists ..." (often this comes up with the number of steps, but not only in those cases).

When you're done you can start bundling all the variables in the w , for example if you used the variables y , t and k you can say that $w = (y, t, k)$, then you can replace all the uses of those variables with $(w)_i$ where i is the index of those variables in w (in this case all the y 's become $(w)_1$, all the t 's become $(w)_2$ and all the k 's become $(w)_3$).

What's important to remember here is to not duplicate variables (that is, each variable should be present in w only once, you should not have something like $w = (y, t, y)$) and that x should not be listed in w because it already exists as the function argument (it may also happen that you have a function with two parameters when using the SMN theorem, in that case you need to exclude both parameters from w).

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

- replace every predicate with its characteristic function (H becomes χ_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 $sg(a - b)$
- $a \geq b$ becomes $sg(a + 1 - b)$
- OR operations become multiplications
- AND operations become additions

- NOT operations become negated signs

In the end you get 0 if true and 1 if false, which is what you need to end the unbounded minimization when the condition is true.

Other things:

- we use cut-off subtraction (or monus, as you can see [here](#)) to define values search and “something greater than something else”.

$$\begin{array}{ll} \overbrace{x}^z - \overbrace{y}^0 & x \leq y \\ x - (\underbrace{x-y}_0) & \\ x - (\underbrace{x-y}_0) = y & x > y \end{array}$$

- we use the sign function (or negated sign) whenever we have something which depends on “greater/lesser” like before, but also can be either 1 or 0 (if first 1, second case 0, use sign, otherwise use negated sign).

Consider the examples of remainder function definition or bounded minimalisation:

$$\begin{aligned} \text{rem}(x, 0) &= 0 \\ \text{rem}(x, y+1) &= \begin{cases} \text{rem}(x, y)+1 & \text{if } \text{rem}(x, y)+1 < x \\ 0 & \text{otherwise} \end{cases} \\ &= (\text{rem}(x, y)+1) * \underbrace{\text{sg}(x - (\text{rem}(x, y)+1))}_{\begin{array}{l} \nearrow \text{something} \\ \swarrow \begin{array}{l} 1 \quad \text{if } \text{rem}(x, y)+1 < x \\ 0 \quad \text{otherwise} \end{array} \end{array}} \end{aligned}$$

$$\begin{aligned} &= h(\vec{x}, y) + \text{sg}(y - h(\vec{x}, y)) + \\ &\quad (y + \text{sg}(f(\vec{x}, y))) \overline{\text{sg}}(y - h(\vec{x}, y)) \end{aligned}$$

↑ if $h(\vec{x}, y) < y$
0 otherwise

- decidable predicates have to be transformed into semicharacteristic functions, for example like:

$$\begin{aligned} sc_A(x) &= \mathbf{1}(\mu w. (S(x, (w)_1, (w)_2, (w)_3) \wedge (w)_2 \in Y)) \\ &= \mathbf{1}(\mu w. (|\chi_S(x, (w)_1, (w)_2, (w)_3) * \chi_Y((w)_2) - 1|)) \end{aligned}$$

We define different cases:

- constants will be multiplied by their expression

$$g(x, y) = \begin{cases} y & \text{if } \neg H(x, x, y) \\ 0 & \text{otherwise} \end{cases} = y \cdot \chi_{\neg H}(x, x, y)$$

$$g(x, y) = \begin{cases} y + 1 & \text{if } x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

The function $g(x, y)$ is computable, since

$$g(x, y) = (y + 1) \cdot sc_K(x)$$

- equalities become absolute values subtractions from right value to left value

$$A = \{x \in \mathbb{N} : x \in W_x \wedge \varphi_x(x) = x^2\} \quad sc_A(x) = \mathbf{1}(\mu w. |x^2 - \varphi_x(x)|) = \mathbf{1}(\mu w. |x^2 - \Psi_U(x, x))|$$

- “less than” becomes subtraction from bigger value to lesser value

$$\mathbb{A} = \{x \in \mathbb{N} : \varphi_x(x) \downarrow \wedge \varphi_x(x) < x + 1\} \quad sc_A(x) = sg(x + 1 - \varphi_x(x))$$

Note that “less than” and not “less or equal then” (so, $<$ instead of \leq) is expressed via $(+1)$, as follows:

$$f(x, y) = \begin{cases} qt(2, y) & \text{if } y < 2x \\ \uparrow & \text{otherwise} \end{cases} \quad g(x) = \begin{cases} 0 & \text{if } x < x_0 \\ M - M' & \text{otherwise} \end{cases} = (M - M') \ sg(x + 1 - x_0)$$

Observe that $f(x, y) = qt(2, y) + \mu z. (y + 1 - 2x)$

- “greater than” becomes subtraction between bigger value and lesser one as cutoff subtraction

$$f(n, x) = \begin{cases} 2 * (x - n) & \text{if } x \geq n \\ \uparrow & \text{otherwise} \end{cases} = 2 * (x - n) + \mu z. (n - x)$$

- \forall values smaller than another one require a minimum bound

$$f(x) = \begin{cases} x & \text{sg } \forall y \leq x. \varphi_y \text{ total} \\ 0 & \text{otherwise} \end{cases}$$

Solution: Let $y_0 = \min\{y \mid \varphi_y \text{ is not total}\}$.

$$f(x) = \begin{cases} x & \text{if } x < y_0 \\ 0 & \text{otherwise} \end{cases} = x \cdot sg(y_0 - x)$$

- we subtract 1 when we consider the effect of unbounded minimalisation, which can give 0 as you will see after these ones, considering predicates, like here, might be true (so give 1), we consider that in the writing

$$\exists w. H^{(k+1)}(e, (\vec{x}, (w)_1), (w)_2)$$

$$sc_A(x) = \mathbf{1}(\mu w. (S(x, (w)_1, (w)_2, (w)_3) \wedge (w)_2 \in Y))$$

$$= \mathbf{1}(\mu w. (|\chi_S(x, (w)_1, (w)_2, (w)_3) * \chi_Y((w)_2) - 1|)) \quad sc_P(\vec{x}) = \mathbf{1}(\mu w. |\chi_{H^{(k+1)}}(e, (\vec{x}, (w)_1)), (w)_2) - 1|)$$

2.1.11 What is that set K I always see?

That set is the halting set, so we are talking about this one specifically:

$$K = \{x \mid x \in W_x\} = \{x \mid \varphi_x(x) \downarrow\} = \{x \mid P_x(x) \text{ terminates}\}$$

The set is recursively enumerable, since it can be written with a semicharacteristic function, but it is not recursive and so not computable.

We use this set with reduction from the set we have to this one; if proven correct, the set is not recursive.

Its complement, instead, so \overline{K} is not r.e. and a reduction from this one proves your set is not r.e.

2.1.12 How to use the minimalisation

This is intended in words as: “I am looking for something”. This holds in two cases:

- the “normal use”, because you use partially recursive functions, you use *unbounded* minimalisation
- in the case of primitive recursive functions, you have to use the *bounded* minimalisation

Look at the definition:

Given a total function $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$, we define a function $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ as follows:

$$h(\vec{x}, y) = \mu z < y. f(\vec{x}, z) = \begin{cases} \text{minimum } z < y \text{ such that } f(\vec{x}, z) = 0 & \text{if it exists} \\ y & \text{otherwise} \end{cases}$$

As you can see, the computation is always bounded, specifically in the range $0 \leq x \leq f(\vec{x}, z)$. Unlike unbounded minimization, the bounded version is guaranteed to halt due to the restriction on the search range.

The unbounded minimalisation, instead, is similar to the last one but the search is not bounded, given the underlying function is not total, this operation is not guaranteed to halt for all inputs. Look at the following definition; the function might never halt and so might not give the desired output.

DEFINITION 6.31. Let $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ be a function. Then the function $h : \mathbb{N}^k \rightarrow \mathbb{N}$ defined through **unbounded minimalisation** is:

$$h(\vec{x}) = \mu y. f(\vec{x}, y) = \begin{cases} \text{least } z \text{ s.t.} & \begin{cases} f(\vec{x}, z) = 0 \\ f(\vec{x}, z) \downarrow \quad f(\vec{x}, z') \neq 0 \quad \text{for } z < z' \\ \uparrow \qquad \qquad \qquad \text{otherwise, if such a } z \text{ does not exist} \end{cases} \end{cases}$$

We often use these ones to guarantee the desired function will terminate for sure, depending on the nature of said functions and cases to consider.

The broader class of partial recursive functions is defined by introducing an unbounded search operator. The use of this operator may result in a partial function, that is, a relation with at most one value for each argument but does not necessarily have any value for any argument. An equivalent definition states that a partial recursive function is one that can be computed by a Turing machine. A total recursive function is a partial recursive function that is defined for every input.

Consider oftentimes we use a “fake” minimalization, which means we minimize on a variable not present inside our set (so, for instance $\mu z.$ (and z is not present) or μw) or instead do a “real” one when we look for “the smallest y compared to x ” say for instance, so you have $\mu y.$ (*expression with y and x*)

2.1.13 What are those $(w)_1, (w)_2 \dots$ I see everywhere?

$(w)_1, (w)_2$ are meant to be encoding in pairs and represent basically tuples – they are used to correctly replace w, z, y and variables like that *inside* minimalization operator. They exist basically because there is not a pair-minimizing operator. Nested minimalization doesn’t work either, because I would scroll the table first only on the columns and then only on the rows.

Basically, they are used to map x, y as projection elements to transform a predicate into a mathematical expression (coding a couple as an integer). Consider this example which extends what was written before; basically, we use this encoding to replace x, y, t (example taken from exercise 8.26 – one of the very few to make us understand because the process is clearly written – would love it if was always like that):

$$\begin{aligned} sc_A(x) &= \mathbf{1}(\mu(y, z, t).H(x, y, t) \wedge S(x, z, y, t)) \\ &= \mathbf{1}(\mu w.H(x, (w)_1, (w)_3) \wedge S(x, (w)_2, (w)_1, (w)_3)) \end{aligned}$$

Usually, in other cases (but the encoding depends on the specific problem, remember):

- $(w)_1: y$
- $(w)_2: t$ (number of steps)

The introduction of their variables is explained at the end of the Universal Function lesson; basically, when we talk about the inverse function to determine if it terminates over input x in a defined number of steps, we might use the encoding in pairs π , but instead we use the exponent of the first prime number 1 and the exponent of the second prime number 2. So, we have:

$$\begin{aligned}
 f^{-1}(y) &= \mu \omega \\
 \pi^{-1}(\omega) &= \cancel{\langle \pi_1(\omega), \pi_2(\omega) \rangle} \\
 \omega \rightarrow & \quad \underbrace{(\omega)_1}_{x}, \underbrace{(\omega)_2}_{m}
 \end{aligned}$$

Consider we do a minimalization on w on these ones given we are using the encoding with w letters.

Important: 95% of the cases, you are forced to use w encoding because you do not have π (which is the encoding in pairs as just said). So, if the exercises gives you $\pi(x, y)$, use that instead. Check for reference exercise 8.52 (which can be used in 8.20) – both are using π .

From here, the predicate over index e , on input x , produces y in n steps. When we found w , we get x as first component. The encoding is not injective, but we don't care about this: we only care it's done in a defined number of steps, hence finitely described.

$$\begin{aligned}
 f^{-1}(y) &= (\mu \omega . \quad S(e, (\omega)_1, y, (\omega)_2))_1 \\
 \pi^{-1}(\omega) &= \cancel{\langle \pi_1(\omega), \pi_2(\omega) \rangle} \\
 \omega \rightarrow & \quad \underbrace{(\omega)_1}_{x}, \underbrace{(\omega)_2}_{m}
 \end{aligned}$$

2.1.14 What is that subscript 1 over function composition?

I simply mean $(\dots)_1$. It simply means “get the first component of such composition/minimalization, etc.” This is often used in semicharacteristic functions writing, but often it's simply forgot, because it's often implied by the reasoning of said writing.

$$\chi_A(x) = (\mu w. S(e_0, x, (w)_1, (w)_2) \vee S(e_1, x, (w)_1, (w)_2))_1$$

For instance, the following are equivalent

and Baldan is not strict if you don't put that 1.

$$1(\mu w. (S(x, (w)_1, (w)_2, (w)_3) \wedge (w)_2 \in Y))$$

2.1.15 What is the universal function and how to use it?

Consider this is always computable and one can say ϕ_x is computable for program e : $\Psi_U(e, \vec{x}) = \phi_e^k(\vec{x})$

From what I got looking at the exercises solutions, it is a very limited case:

- basically, when you write semicharacteristic functions, so you are doing r.e./not r.e. exercises considering exercises with $\phi_x(x)$ inside the exercise definition, there is:
 - o $sc_A(x) = (\dots - \phi_x(x))$
- you simply replace ϕ_x with $\Psi_U(x, x)$
 - o $sc_A(x) = (\dots - \Psi_U(x, x))$

Consider also (to make you understand how to write this):

- $\phi_x(z) = \Psi_U(x, z)$
- $\phi_y(z) = \Psi_U(y, z)$

2.1.16 What are those S and H functions I see everywhere?

In this kind of exercises, there are also parts of proofs that use S and H function, never actually explained properly, apart from putting the puzzle pieces together, present but not in a straightforward way.

General form of semicharacteristic function writing:

$$H(x, y, t) \wedge S(x, y, z, t)$$

Syntax:

- S - function defined to stop in a defined number of steps
 - $S(x, y, z, t)$ is the usual form – constrain execution to x , depend on y and z , terminate in t steps
- When using the program code $\phi_e = f$ you can write
 - $S(e, x, y, t)$ – constrain execution to e , depend on x and y , terminate in t steps
 - e is the code, x is the input, y is the output, t the number of steps
 - This is also written as χ_S when expressed as semicharacteristic function and often it's better, when writing the semicharacteristic functions, to pass to this form instead (so, from S to χ_S to transform it from predicate to function) – Baldan it's not strict on this one
- H – this is the “halts” function, defining where the function will stop, specifically on which input and which output
 - $H(e, x, t)$ checks if e on input x terminates in t steps
 - e is the code, x is the input, y is the output, t the number of steps
 - The usual form of this one is $H(x, x, y)$ used in the case of the halting set
 - x is the code, x is the input, y the number of steps
 - It's often used in the case of \bar{K} which means $\neg H(x, x, y)$ holds (does not halt)
 - This is also written as χ_H when expressed as semicharacteristic function when expressed as semicharacteristic function and often it's better – see above

Nice explanation by this year tutor:

- The S function is almost the same as the H function, except it has an additional argument (the 3rd one) that constrains the output of the program execution
- That is, $H(e, x, t)$ only checks if the program with index e on input x halts after t steps, while $S(e, x, y, t)$ checks if the program with index e on input x halts with output y after t steps
- You would use H when you only care whether a program halts or not, and S when you also care about its output
 - the usual example is the inverse of a function, where you want to know whether there exists an input x such that $f(x) = y$, in which case you do care that the output is y

Found inside page 79 of notes their notation:

COROLLARY 12.3. *The following predicates are decidable:*

- (a) $H_k(e, \vec{x}, t) \equiv "P_e(\vec{x}) \downarrow \text{in } t \text{ or less steps}"$
- (b) $S_k(e, \vec{x}, y, t) \equiv "P_e(\vec{x}) \downarrow y \text{ in } t \text{ or less steps}"$

Also inside page 96 (a bit late I would say) – S should be uppercase there:

- $H(x, y, t) = "P_x(y) \downarrow \text{in } t \text{ steps or less}"$;
- $s(x, y, z, t) = "P_x(y) \downarrow z \text{ in } t \text{ steps or less}"$;

From what I investigated:

- if you have to express the fact the program halts or not (or you care about the input/domain), use the function H (useful especially when writing semicharacteristic functions)
- if you care about halting and the output, use S (so we are talking about the codomain - useful especially when writing semicharacteristic functions)

2.1.17 How to do recursive/r.e. exercises

For dummies:

- start to see if set is saturated
 - usually if saturated, then not recursive, but can be r.e.
- if it is saturated, see also if it is r.e. (so, you can write a semicharacteristic function)
- if it is r.e., check if recursive
 - use Rice's Theorem/halting set to prove it's not
 - 98% of the times sets are not recursive – apart from exercise 8.58, usually “cool proof but not real case” exercise to consider for this
- if it is not r.e. use Rice-Shapiro or negation of halting set
 - if not r.e. then it is not recursive

Also consider all logical implications, written in detail later.

Longer explanation; two macro-cases:

- We use Rice-Shapiro (this usually happens with functions with definitions like W_x or E_x) and show the set is saturated, then show both the set and its complement are not r.e.
 - Since they are not r.e., they are not recursive
 - Basically we have at the same time a function which is defined but a finite subfunction which is not defined
 - Conversely we have at the same time a function which is not defined but a finite subfunction which is defined
 - On this arises the contradiction given by Rice's theorem
 - If we have a set r.e. (because we can write its semicharacteristic function)
 - Then we have to use Rice's Theorem to understand if it's recursive or not
- We use the reduction (see also what was written [here](#)) from the halting set (this happens for more “practical” cases, which happens when we have definitions that do not include domain/codomain)
 - If a function can be reduced from K it is not recursive
 - In this case, the function is assumed to terminate (so, you have $H(x, x, y)$ terminating)
 - Since the set is not recursive, the complement is not recursive or r.e.
 - If a function can be reduced from \bar{K} it is not r.e.
 - In this case, the function is assumed to not terminate (so, you have $\neg H(x, x, y)$)
 - Since the set is not r.e., it is not recursive and also the same holds from the normal set, so not recursive or r.e.

We know a set is not saturated if there is y depending on x or $f(x) \in W_x/E_x$

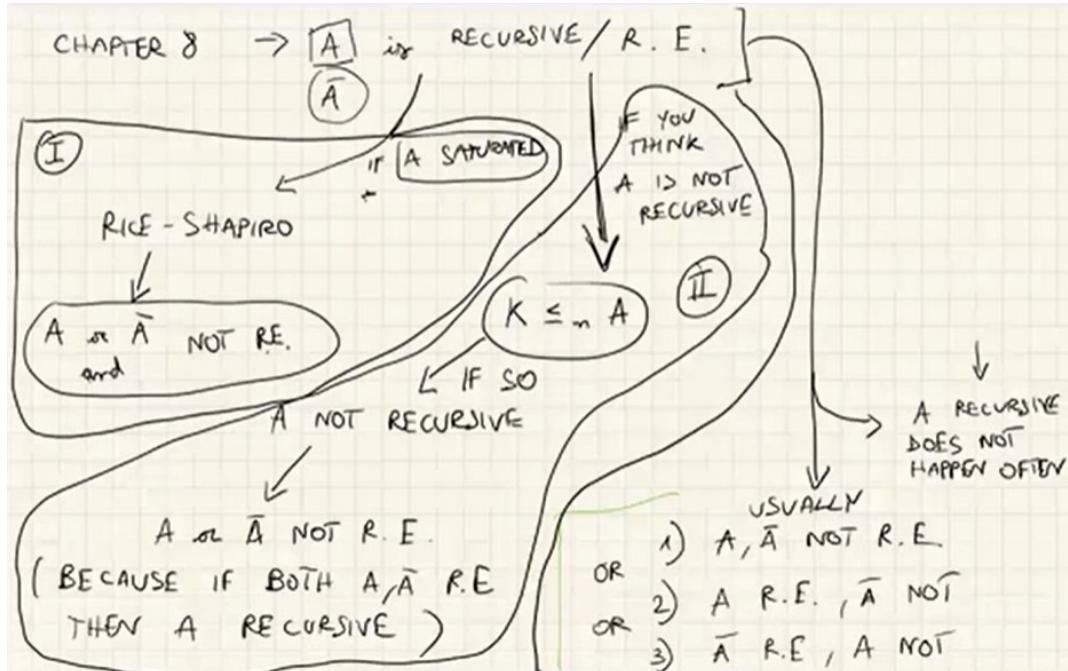
Practical observations:

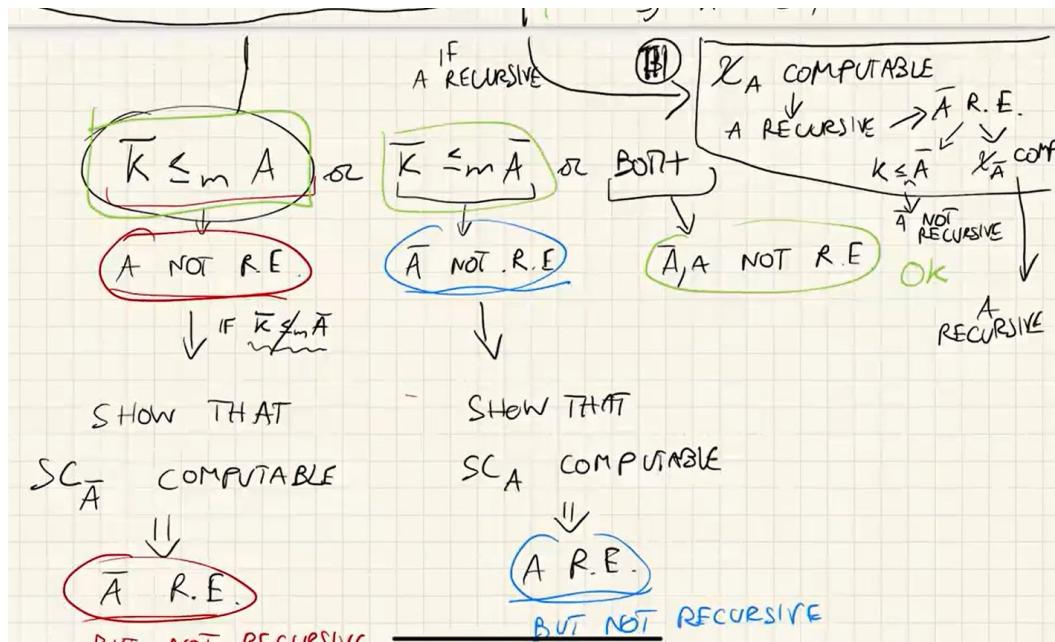
- Exercises with only domain and codomain in the definition (so, W_x and E_x) always require the use of Rice-Shapiro
- Exercises with more normal definitions (like combination of y, x), something using ϕ_x or both of them, then it requires a reduction from K (or its complement)

How to use Rice-Shapiro correctly? Well, considering id as the identity (always defined for every natural number) and \emptyset as the always undefined function (so, always undefined for every natural number).

- A is not r.e. (the cases are alternative between each other – so, if one holds, the other does not)
 - o We can have notable functions like $id/\emptyset \in \mathcal{A}$ and a function $\theta \notin \mathcal{A}$
 - Where the subfunction needs to be defined by cases
 - Other times we have $f \in \mathcal{A}$ (f is the original definition function) and $\theta \notin \mathcal{A}$
 - o We can have $id/\emptyset \notin \mathcal{A}$ and a function $\theta \in \mathcal{A}$
 - Same observation for finite subfunction
 - Other times we have $f \notin \mathcal{A}$ and $\theta \in \mathcal{A}$
- \bar{A} is not r.e. (the cases are alternative between each other)
 - o We can have notable functions like $id/\emptyset \in \mathcal{A}$ and a function $\theta \notin \mathcal{A}$
 - Where the subfunction needs to be defined by cases
 - Other times we have $f \in \mathcal{A}$ (f is the original definition function) and $\theta \notin \mathcal{A}$
 - o We can have $id/\emptyset \notin \mathcal{A}$ and a function $\theta \in \mathcal{A}$
 - Same observation for finite subfunction
 - Other times we have $f \notin \mathcal{A}$ and $\theta \in \mathcal{A}$

A general schema coming from an old tutor:





2.1.17.1 Rice-Shapiro

- We use this one if A is saturated
 - o This usually happens when the exercises gives W_x, E_x or both of them
 - o $A = \{x \in \mathbb{N} \mid \phi_x \in \mathcal{A}\}$ and $\mathcal{A} = \{f \mid \dots\}$
 - o You replace W_x with $\text{dom}(f)$ and E_x with $\text{cod}(f)$
- This way, we show A and \bar{A} are not r.e.
 - o This may not always be the case; sometimes a set is saturated, but the set is r.e. (it means you can write a semicharacteristic function χ_A)
 - In this case, if A is r.e. \bar{A} is not r.e. (hence not recursive)
 - Conversely, if \bar{A} is r.e., A not r.e. (hence not recursive)
- Applying the definition it means either:
 - o we have a function which is in the set but a finite subfunction not in the set
 - o we have a function which is not in the set but a finite subfunction which is in the set
- Usually, we use id and \emptyset
 - o identity = defined for all natural numbers
 - o always undefined function = undefined for all natural numbers
- Sometimes, one can use the constant $\mathbf{1}$ function
- It usually works showing you have (as above, but replace f with a logically correlated function to the exercise definition of specified set)
 - o $f \notin A$, but $\exists \theta \text{ finite}, \theta \in \mathcal{A}$
 - o $f \in A$, but $\forall \theta \text{ finite}, \theta \notin \mathcal{A}$
- This usually holds for both sets
 - o If both sets are not r.e. they are not recursive either

There are the following implications:

- if A is r.e. but not recursive, also \overline{A} is not r.e. (also not recursive, otherwise they would be both recursive)
- if A is recursive, then χ_A is computable. We have \overline{A} is r.e. and:
 - o if $K \leq_m \overline{A}$, then \overline{A} is not recursive
 - o if $\chi_{\overline{A}}$ is computable then \overline{A} is recursive
- If A r.e., then \overline{A} is not – if A is r.e., it means sc_A exists, but is not recursive
- If \overline{A} r.e. then A is not – if \overline{A} is r.e., it means $sc_{\overline{A}}$ exists, but is not recursive

Side note (important):

- One can show a set is not recursive by using Rice's theorem
 - o This occurs when the set is saturated and maybe is r.e. but we ask if it is recursive
 - o Then, you use $e_0 \in id/\mathbf{1}$ and $e_i \in \emptyset$ to prove $e_0 \in A, e_1 \notin A$ hence $A \neq \emptyset, \mathbb{N}$
 - for example $e_0 s.t. \phi_{e_0} = id/\mathbf{1}$ or $e_1 s.t. \phi_{e_1} = \emptyset$

2.1.17.2 Reduction

- We use this one if A is not recursive ($K \leq_m A$)
 - o usually something like $g(x, y) = \begin{cases} y \text{ (or value), } x \in K \\ \uparrow, \text{ otherwise} \end{cases}$
 - o a variant with the same meaning is $g(x, y) = \begin{cases} 1 \text{ (or value), } x \in W_x \\ \uparrow, \text{ otherwise} \end{cases}$
 - o it is computable and thus, by the smn theorem, we deduce that there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ such that, for each $x, y \in \mathbb{N}$, $g(x, y) = \phi_{s(x)}(y)$
- It can be shown to be the correct reduction function
 - o if $x \in K, \phi_{s(x)}(y) = g(x, y) = y \text{ (or value) } \forall y \in \mathbb{N}$. Therefore $s(x) \in W_{s(x)} = \mathbb{N}$ and $\phi_{s(x)}(s(x)) = s(x)$. Therefore, $s(x) \in A$
 - the function here is the value; if we had y^2 it would have been $(s(x))^2$
 - o if $x \notin K, \phi_{s(x)}(y) = g(x, y) \uparrow \forall y \in \mathbb{N}$. Therefore $s(x) \notin W_{s(x)} = \emptyset$ and so $s(x) \notin A$
- We can also use the complement of the same set to show it is not r.e. ($\overline{K} \leq_m A$)
 - o usually something like $g(x, y) = \begin{cases} y \text{ (or value), } \neg H(x, x, y) \\ \uparrow, \text{ otherwise} \end{cases}$
 - o this starts from a computable function, like $g(x, y) = \begin{cases} 1 \text{ (or value), } x \in K \\ \uparrow, \text{ otherwise} \end{cases}$
 - o it is computable since we have $g(x, y) = \text{value} * sc_K(x)$ and thus, by the smn theorem, we deduce that there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ such that, for each $x, y \in \mathbb{N}$, $g(x, y) = \phi_{s(x)}(y)$

The problem is that the reduction may not work for cases $x \notin \overline{K} \Rightarrow f(x)$ not in the set, since $g(x, y)$ may still have some values in the domain/codomain before it starts to always diverge, and those may be enough to satisfy the set's requirements.

- It can be shown to be the correct reduction function
 - o if $x \in \overline{K}, \phi_{s(x)}(y) = g(x, y) = y \text{ (or value) } \forall y \in \mathbb{N}$. Also, we can say $H(x, x, y)$ is false $\forall y \in \mathbb{N}$. Therefore $s(x) \in W_{s(x)} = \mathbb{N}$ and $\phi_{s(x)}(s(x)) = s(x)$. Therefore, $s(x) \in A$
 - o if $x \notin \overline{K}, \phi_{s(x)}(y) = g(x, y) \uparrow \forall y \in \mathbb{N}$. Also, we can say $H(x, x, y)$ is true $\forall y \in \mathbb{N}$. Therefore $s(x) \notin W_{s(x)} = \emptyset$ and so $s(x) \notin A$

- If this reduction from complement holds, A is not r.e.
- It can also happen $\bar{K} \leq_m \bar{A}$ and so \bar{A} is not r.e.
- If both are valid (so $\bar{K} \leq_m A$ and $\bar{K} \leq_m \bar{A}$), both sets (A, \bar{A}) are not r.e.

Some logical implications I had also written in another subsection:

- Any recursive set is also recursively enumerable
- A set is recursive iff the set and its complement are r.e.
- If a set is not r.e. then is not recursive
- A set is not recursive when a reduction from the halting set (K) works
- A set is r.e. if one can write the semicharacteristic function, which is computable
- A set can be shown to be not r.e. using a reduction from halting set complement (\bar{K}) or via Rice-Shapiro
- Usually, if a set is r.e., the complement is not r.e. (this depends on the problem conditions) hence not recursive (otherwise, they would be both recursive hence r.e.)
- If a set is saturated, then it is not recursive (can be shown via Rice's theorem)
- If a function does not terminate, we argue $\neg H(x, x, y)$ otherwise it terminates so $H(x, x, y)$

Some from reading the book:

- An infinite set is recursive iff it is the range of a total increasing computable function i.e. if it can be recursively enumerated in increasing order

2.1.18 How to write the negated sets

In Rice-Shapiro exercises, it's useful to write the negated sets to actually understand the track to follow. Remember the following rules (they come from logic, so consider this and De Morgan laws to be precise):

- \exists becomes \forall
- \forall becomes \exists
- \wedge becomes \rightarrow

All other obvious ones regard: = becomes \neq , $>$ becomes $<$ etc. Consider as examples:

Exercise 8.26. Study the recursiveness of the set $A = \{x : \forall y. \text{if } y + x \in W_x \text{ then } y \leq \varphi_x(y + x)\}$, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Solution: The set $\bar{A} = \{x : \exists y. y + x \in W_x \wedge y > \varphi_x(y + x)\}$ is not recursive, since $K \leq_m \bar{A}$.

Exercise 8.69. Classify the following set from the point of view of recursiveness

$$B = \{x \in \mathbb{N} : \forall y \in W_x. \exists z \in W_x. (y < z) \wedge (\varphi_x(y) > \varphi_x(z))\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is saturated, given that $B = \{x : \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \in \mathcal{C} : \forall y \in \text{dom}(f). \exists z \in \text{dom}(f). (y < z) \wedge (f(y) > f(z))\}$.

For the complement $\bar{B} = \{f \mid \exists y \in \text{dom}(f). \forall z > y. (z \notin \text{dom}(f)) \vee (f(y) \leq f(z))\}$, we observe

Exercise 8.70. Classify the following set from the point of view of recursiveness

$$B = \{x \in \mathbb{N} : \forall y \in W_x. \exists z \in W_x. (y < z) \wedge (\varphi_x(y) < \varphi_x(z))\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is saturated, given that $B = \{x : \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \in \mathcal{C} : \forall y \in \text{dom}(f). \exists z \in \text{dom}(f). (y < z) \wedge (f(y) < f(z))\}$.

2.1.19 How to do the Second Recursion Theorem exercises

The process here is fairly simple:

- you state the theorem
- you define a function of two arguments which is computable
- by the smn-theorem, this works (simply write its definition)
- by the second recursion theorem, there exists only a single index on which this holds
- use the second recursion theorem definition substituting e inside the function definition

Here you have two cases:

- use the theorem to prove there exist a single index respecting a problem definition
- use the theorem to prove a function is not computable
- use the theorem to prove a function is total
- use the theorem to prove the set is not saturated

In all three cases previously written, just to what was written above and

- if the function was not computable or total to begin with, with the second recursion theorem, any possible index won't be inside the domain or just will not be defined
- if the function is total or there exists a single index, simply substitute e in place of x inside the function definition and call it a day

For the fourth case (set not saturated):

- everything is the same, just change the conclusions
 - o Given any $e' \neq e$ and such $\phi'_e = \phi_e$, we have that $f(e') \neq f(e)$ and so (after proving that on the exercise definition), $e' \notin A/\mathcal{C}$
 - where A is the set, other times there is calligraphic \mathcal{C} to represent the class of computable functions

2.1.19.1 Show there exist an index s.t. function is total/computable

- Give the theorem definition
- Give a function of two arguments $g(x, y)$ for instance defined by cases
 - o case for the normal condition
 - o case for otherwise
- Since it is defined by cases, it's computable (since it is total, holds)
- By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y)$
- By the Second Recursion Theorem, there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{s(e)}$
- You use the function previously defined and replace $g(x, y)$ with $\phi_e(y) = \phi_{s(e)}(y) = g(e, y)$
 - o inside the function, replace x with e
- You conclude since you fixed the point in which all the condition you posed hold

2.1.19.2 Show there exist an index s.t. function is not computable

- Give the theorem definition
- Note the function is computable but it is usually total, so you have say $\phi_x \neq \phi_{h(x)}$
- By the Second Recursion Theorem, there exists $e \in \mathbb{N}$ such that $\phi_e \neq \phi_{s(e)}$
- So, the original function cannot be computable

2.1.19.3 Show that a set A is not saturated

- Give the theorem definition
- Give a function of two arguments $g(x, y)$ for instance defined by cases
 - o case for the normal condition
 - o case for otherwise
- Since it is defined by cases, it's computable
- By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y)$
- By the Second Recursion Theorem, there exists e such that $\phi_e = \phi_{s(e)}$
- You use the function previously defined and replace $g(x, y)$ with $\phi_e(y) = \phi_{s(e)}(y) = g(e, y)$
 - o inside the function, replace x with e
- Now, just take $e' \neq e$ such that $\phi_{e'} = \phi_e$ (which exists since there are infinitely many indices for the same computable function)
- So, we have e in A and $e' \notin A$ So, A is not saturated

2.2 SWISS KNIFE OF PRACTICAL DEFINITIONS

2.2.1 Totality and diagonalization

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
 - o We can use diagonalization to make a partial computable function that differs from every total computable function
 - o 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
 - o 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
 - o Tl;dr – use $\phi_x(x)$ as [this](#) section specifies

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](#):

- o 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.
- o 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.
- o 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.

2.2.2 Minimalization

Minimalization is the process of finding the smallest value that satisfies a given property within a certain range or domain. In the context of computability theory, minimization often involves searching for the smallest input that makes a specific function output. Integrate this section with [this](#) one.

We use the so called μ -operator, denotes as μy , can be either unbounded or bounded. In the unbounded form, it searches for the least natural number y such that a given predicate $R(y, x_1, \dots, x_k)$ holds. The bounded form, as described by Kleene, involves finding the least $y < z$ s.t. $R(y)$ with appropriate conditions.

Let's clarify the differences between bounded and unbounded minimalization.

- unbounded

The unbounded minimization of a partial computable function $\phi(x, y)$ with respect to y searches for the smallest y s.t. $\phi(x, y) = 0$ (0 in the context of this example, keep in mind). If such y exists, the function returns that value, otherwise it doesn't halt.

In formula: $\mu y[\phi(x, y) = 0]$

- bounded

The bounded minimization of a partial computable function $\phi(x, y)$ with respect to y searches for the smallest y less than or equal to k s.t. $\phi(x, y) = 0$. If such y exists, the function returns that value, otherwise it doesn't halt.

In formula: $\mu y \leq k[\phi(x, y)] = 0$

2.2.3 Why do we need to focus this much on Ackermann Function?

It is never required inside the exercises, but it's fundamental for many things:

- we need an order of elements, at least partial
- by induction, this can be proven to be defined on more elements respecting a specific property, becoming well-founded
- for loops (bounded minimalization/PR definitions) cannot be general, given there are also non total functions and there while loops (unbounded minimalization) is needed

At the end of the day:

- To be able to define all total functions you also need minimalization: otherwise, some functions might be too powerful to express traditionally, like happens [here](#).

Consider [this](#) one:

- We can't do unbounded loops with only composition and primitive recursion b/c composition is necessarily finite and primitive recursion "counts down" as it were, to the base case.
- In other words, when we use primitive recursion we must specify a base case (usually when $y = 0$) and a "demoting case", that is, a case that starts with y' and ends with y (when $y' \neq 0$).
- This clearly describes a bounded loop - yet in order to compute all the functions that Turing machines can compute, we must be allowed to execute unbounded loops
- in a sense, we need to be able to ask questions that may or may not have any answer (we need to be able to run our program forever, since we know not where the answer lies – $f(0), f(1), f(2), f(3)$, etc..)
- This is what minimization seems to get us - since it asks the question - "At what point does the function f return 0 for some inputs x_1, \dots, x_n ?" - even though the function may never return 0 for that input.
- The quintessential example of a function which is not definable without minimization is the Ackermann function

2.2.4 Recursiveness and types

- **recursive**: A set is recursive if it can be computed in a finite amount of time/number of steps
 - o Concretely, it means the characteristic function (gives 1 or 0) is computable
 - o In function terms, it simply means we have a function giving 1 or 0 with letter "chi" (χ)
$$\chi_A = \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases}$$
 - o Consider recursive is a subset of recursively enumerable, which means that recursive first depends on the fact the set is r.e. or not (so check that first)
- **primitive recursive**: A function is primitive recursive if it can be obtained from basic initial functions through a finite number of applications of certain predefined recursion schemes.
 - o According to a useful practical definition [here](#):
 - any function you can write where the only loops are those of the form
 - "for $i = 1$ to n do ..."
 - o Here n is fixed in advance (before the loop starts), and you cannot (explicitly) change i nor n inside the loop. So the number of times the loop executes is determined in advance. These conditions make infinite loops impossible.
- In other terms:
 - o A simplified answer is that primitive recursive functions are those which are defined in terms of other primitive recursive functions, and recursion on the structure of natural numbers.
 - o Primitive recursive functions are a (mathematician's) natural response to the halting problem, by stripping away the power to do arbitrary unbounded self-recursion (because you can define a function *recursively going only on function you decide yourself, not out of all the possible ones present out there*) – this reasoning and phrasing comes from [here](#)

A good enough distinction from [here](#):

- *Primitive recursive* functions are those that can be computed (from the trivial initial functions) by using *for* loops as the basic programming structure.

There are *bounded* loops [we know as we enter how many cycles to execute]. We can, though, nest them one inside the other, and chain together such nested loops. 'For loops' correspond to definitions by primitive recursion.

- *Recursive* functions are those that can be computed (from the trivial initial functions) by using *for loops* and/or *do until* loops.

'Do until' loops involve *unbounded* searches until some condition is satisfied [as we enter the loop, we don't know how many times we will need to cycle around], so correspond to definitions by minimization.

- **partial recursive**: A function is partial recursive is if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion and minimization.
 - o In other words, the function might be undefined (or "partial") for certain inputs. The set of partial recursive functions includes both recursive and primitive recursive functions
 - o Again: this one include minimalisation, while primitive recursive functions do not.

2.2.5 Recursively enumerable and enumeration

- enumeration: An enumeration is a listing or indexing of elements of a set in some order. It provides a way to iterate through all elements of the set one by one.
- recursively enumerable: A set is recursively enumerable if we can enumerate recursively the number of steps (which means “sometimes” we find a function terminating/TMs accepting the language)
 - o Concretely, it means its semicharacteristic function is computable (so, it's a superset of recursive) and mean
 - o The semicharacteristic function is a partial function that indicates membership in the set for certain elements but may not provide information for others.
 - o Usually we write:

$$sc_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

Quoting Wikipedia, “for these sets, it is only required that there is an algorithm that correctly decides when a number *is* in the set; the algorithm may give no answer (but not the wrong answer) for numbers not in the set”. So you see that:

- we can list the elements, so we are able to describe recursively a set
- but not always this set ends
 - o it may happen *on a finite set of indices* and not for the other – meaning saturated

Precisely, on the recursive/r.e. nature some logical implications:

- Any recursive set is also recursively enumerable
- A set is recursive iff the set and its complement are r.e.
- If a set is not r.e. then is not recursive
- A set is not recursive when a reduction from the halting set (K) works
- A set is r.e. if one can write the semicharacteristic function, which is computable
- A set can be shown to be not r.e. using a reduction from halting set complement (\bar{K}) or via Rice-Shapiro
- Usually, if a set is r.e., the complement is not r.e. (this depends on the problem conditions) hence not recursive (otherwise, they would be both recursive hence r.e.)
- If a set is saturated, then it is not recursive (can be shown via Rice's theorem)
- If a function does not terminate, we argue $\neg H(x, x, y)$ otherwise it terminates so $H(x, x, y)$

Some from reading the book:

- An infinite set is recursive iff it is the range of a total increasing computable function i.e. if it can be recursively enumerated in increasing order

2.2.6 Decidability and Semidecidability

- decidable: A predicate is decidable if its characteristic function is (URM) computable (and so it is total)
 - o Concretely, it means there is a characteristic function like the following for a predicate:

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

- o If you look better at this, you will see it's something like the recursive set case
- semidecidable: A predicate is semidecidable if and only if the predicate is r.e. – so there is a semicharacteristic function for that
 - o In other words, a set is recursively enumerable if there is an algorithm that can list its elements, though this algorithm might not halt for elements not in the set
 - o Concretely, you will see this is similar to the recursively enumerable case

$$sc_P(\vec{x}) = \begin{cases} 1 & \text{if } P(\vec{x}) \\ \uparrow & \text{otherwise} \end{cases}$$

Of a set, such that there is a deterministic algorithm such that if an element is a member of the set, the algorithm halts with the result "positive", and if an element is not a member of the set, the algorithm does not halt, or if it does, then with the result "negative".

- saturated: A subset $A \subseteq \mathbb{N}$ is considered saturated or extensional if for every pair of natural numbers m and n , if m belongs to A and the functions represented by the programs ϕ_m and ϕ_n are equal, then n must also belong to A .
 - o In simpler terms, this definition asserts that if a program with a specific property is part of the set, then all programs that compute the same function must also be part of the set.
 - o This notion is essential to try to use Rice-Shapiro

2.2.7 Functionals and Fixed Points

- functional: A functional (also called "operator") is a total function $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^h)$ (considering F is the set of all the functions and the others are the arguments and the indices of the k arguments)
 - o A functional has to be effective, given both input and output can be infinite
 - o They calculate in finite time using only a finite part of the input function (according to Cutland book definition)
 - o In general, a *functional type*, in which a function takes in input a function of same type and gives as output another function of same type
- To do this, we introduce the concept of recursive functionals
 - o A functional $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^h)$ is recursive if there is a total computable function $\phi: \mathbb{N}^{h+1} \rightarrow \mathbb{N}$ s.t. $\forall f \in F(\mathbb{N}^k)$
 $\forall \vec{x} \in \mathbb{N}^h$
 $\Phi(f)(\vec{x}) = y \quad \text{iff there exists } \theta \subseteq f \text{ s.t. } \phi(\tilde{\theta}, \vec{x}) = y$
 - o In simpler terms, recursive functionals essentially produce outputs of the same type as a finite part of the input function, acting as both input and output themselves.

- **fixed point:** A function is a fixed point/fixpoint of a functional Φ (a function which is not changed from the transformation and is an element mapped to itself by the function) , i.e. $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\Phi(f) = f$.
 - o Looking [here](#), a fixed point x in a set X s.t. $x \in X$ is a fixed point with a map to itself such that $f(x) = x$.

2.3 SYMBOLS AND ACRONYMS

First thing first:

- Inside “notes.pdf” on Moodle, you will often find the subtraction with a point, something like $- \cdot$

What does this mean (because never explicitly told, not even by the book)? 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:
 - o $3 - 4 = -1 \notin \mathbb{N}$
 - o $3 - \cdot = 0$
- 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)

Just to pinpoint in grammar:

- according to the English language, minimalization, better minimization, it's the right term. Here, both the book and the notes use the “s” writing (minimalisation), which is less correct in English. Both are used here, mainly the first one being more correct (at least from what I found myself).

Continuing with other explanations:

- $(\dots)_1$: when you see an expression like this, with a 1 as a subscript (it doesn't matter what's inside), means "projection on the first component of the computation", which in other words means "take the output from the first register" and will always hold as such for the other ones given it is defined.
- \uparrow (does not terminate/halt)
- \downarrow (terminates/halts)

The \square symbol specifies the end of a proof.

Moving on:

- \in = inside a set
- \notin = not inside a set
- $s.t.$ or the pipe sign $|$ = such that ("tale che" in italiano)
- \equiv = congruent to
- \sum = sum of all things / \prod = product of all things
- \neg = negation
- \circ = composition
- \leq_m = reduction (simplification hence usage of smn-theorem for parametrize) of function of left of the symbol to the one on the right
- \subseteq = For a given set B, the set A is a subset of B if every element that is in A is also in B. This is denoted by $A \subseteq B$
- \vec{x} = vector of inputs for a given function (which means we take them all and apply projection), so we have all of the inputs on a possible computation
- γ : effective bijective enumeration of set of all programs
- π : encoding in pairs of natural numbers (n-tuples defined by recursion)

Just to clarify:

- Set Intersection (\cap):

The intersection of two sets A and B, denoted $A \cap B$, contains only the elements that are common to both sets A and B.

It "filters out" elements that are not in both sets.

Examples:

$$\{1, 2, 3\} \cap \{2, 3, 4\} = \{2, 3\}$$

$$\{a, b\} \cap \{b, c\} = \{b\}$$

- Set Union (\cup):

The union of two sets A and B, denoted $A \cup B$, contains all elements that are in either set A or set B or both.

It combines the elements of both sets without removing any.

Examples:

$$\{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$$

$$\{a, b\} \cup \{b, c\} = \{a, b, c\}$$

So we can describe in symbols:

- \vee = or
- \wedge = and

Continuing with other symbols:

- \mapsto = maps to

2.3.1 URM Machines Symbols

- *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:
 - o if $r_m = r_n$ then jumps to I_t (jumps to t -th instruction)
 - o otherwise, it will continue with the next instruction
- U_i^k : projection

The jump actually does many things:

- $J(1, 1, SUB)$ = it's a jump on a subroutine, which you can define, to encode further instructions of a machine
- $J(1, 1, END)$ = jumps to the end of program

Other ones to note (considering a URM program P):

- $l(P)$ = program length/number of instructions
- $\rho(P)$ = largest register index (the letter is "rho")

2.3.2 General Functions and Notation

- μ : minimalization (there is a “point” (.) between μ and bounded function)
 - o It uses a random w to only prove the minimalization argument for which the expression is less than w thanks to μ
- sg : sign function (used to represent a binary condition, usually something you don’t want to become zero)
- \overline{sg} (negative sign function, same for binary, opposite of the last one)

The two “sign functions” are often used to properly represent all the possible combinations of computation when dealing with a defined function. It means we usually consider “binarily” the cases we have; if we have two cases, for example, we use sign or negated sign “to represent $f(x)$ subtracted from the other case in the expression”, hence representing it is computable otherwise. Often it is used with minimalization.

- qt : quotient function)
- 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 $f(x)$
- rm : remainder, will give 1 if the division does not have a null remainder, 0 otherwise)

Many times you will find these ones, not so often explicitly told:

- W_x : function domain
 - o 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)
 - o remember the codomain is also written as $cod(f)$, where f is a function
 - o remember also the image can be written as $img(f)$
 - o Letter E *probably* stands for *enumerable* because it is possible to enumerate all the outputs that the function can produce.

Specifically, we can precise that:

- image, codomain is always N if we use the classical convention
- codomain = target set and image = set of images of domain elements
- E is the image while the codomain is N

Other notable ones:

- $\pi(x, y)$: a pairing function
- γ : program coding

One thing important to remember is that (may seem confusing, but all cases are listed thoroughly):

- If you have *computable* functions and perform *computable* operations (addition, multiplication, composition of computable functions), the result is always computable.
- If you have *computable* elements but perform operations that involve *non-computable functions*, the result may be non-computable.
- If you have *non-computable* elements but perform *computable* operations, the result can be either computable or non-computable, depending on the specific nature of the non-computable elements.
 - o In the specific case of functions, if you combine a *computable* function with a *non-computable* function through a *computable* operation, the result is *computable*.
 - o However, if you combine a *computable* function with a *non-computable* function through a *non-computable* operation, the result may be *non-computable*.
- If both elements and operations involved are *non-computable*, the result is typically *non-computable*. Non-computability tends to propagate, and complex interactions may lead to undecidable or incomputable outcomes.

2.3.3 Sets, Predicates and Characteristic Functions

- \mathcal{PR} = set of primitive recursive functions (useful in the dedicated exercises and with specific properties specified [here](#))
 - K = halting function set (we can have its complement \overline{K})
 - o if a set reduces to K is r.e. (but not recursive)
 - o if a set reduces to \overline{K} is not r.e. (also not recursive)
- $$K = \{x \mid x \in W_x\} = \{x \mid \varphi_x(x) \downarrow\} = \{x \mid P_x(x) \text{ terminates}\}$$
- A = decidable predicate (many other times, simply a set, which can also be B)
 - o consider many times we need to write the negated set, useful for recursiveness exercises

Other notable sets: \mathbb{P} - set of even numbers, Pr set of prime numbers

- $Q(x_1, \dots, x_k)$: k -ary predicate
- Sometimes you will find \mathcal{A} which is the *set of computable functions* (called “calligraphic A”)
 - o many times in this file, will be written as A and the normal set as A
 - o this was because I discovered late the solution of calligraphic A (bear with me, given the quality of file)
- Similarly, you will find \mathcal{C} (definable normally as “calligraphic C”) which represents the *class of computable functions*
- Similarly, you will find \mathcal{F} (definable normally as “calligraphic F”) which represents the *set of all functions* (possibly not computable)

- χ_K : characteristic function of the halting set, which is 1 if input halts ($n \in K$), 0 otherwise) - Greek letter here is “chi” (the strange X here)
- χ_A : characteristic function of set A which is 1 if input $\in A$, 0 otherwise)
- sc_A : semi-characteristic function (0/1 cases) of a predicate
 - o if the semicharacteristic function is decidable, the function is semidecidable
- Ψ_U : universal function (Greek letter is “Psi”)
 - o it is usually of two arguments: e (program) and \vec{x} (input arguments)
 - o it is generally used when there is a fixed index for the function or a fixed input
 - o in many cases, it simply substitutes $\phi_x(x)$ as $\Psi_U(x, x)$ (as written [here](#))
- **1**: characteristic function, seen also in LaTeX as `\mathbb{1}` (for your reference)
 - o both in Moodle notes and here is used the bold notation, for easiness of use
 - o Over functions, you will see many times combinations of binary function which are trying to express the binary conditions, as you can see by the following example. In this case, it can be read as “all the possible combinations thanks to which in can be either 1 or 0”
- θ : finite subfunction
 - o 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
- e : index of computation for functions (also called many times simply “program”)
 - o ϕ_e = partial recursion over the specific index
 - o usually, it is used inside the Second Recursion Theorem exercises or simply to compose function by compositions (so, $f = \phi_e$ and then you write $f(x)$ as the combination of tuples with e as index)

In Rice-Shapiro exercises, the following functions are often used:

- id (identity function)
 - o it is used usually for the normal version of set to show for every natural number the finite subfunction does not respect the conditions of the specified set and is not inside of it
 - o this holds because it is defined for every natural number
 - o it has finite indices hence finite programs calculating it
 - o a subvariant sometimes used is $id(x)$, so you can define a subfunction, like constant 1 or constant 0
- \emptyset (always undefined function – same symbol as empty set)
 - o it is used usually inside the complement of set to show the finite subfunction does not respect the conditions of the complement and is not inside the set
 - o it has infinite indices hence infinite programs calculating it
 - o this holds because it is not defined for any natural number

Inside last part of notes, there is:

- Φ (uppercase phi), which represents a functional
- f_Φ , which represents a least fixed point

2.3.4 All book notations

Just because I am a good person, I compiled every notation here stole by courtesy of Cutland book.

Chapter 1	
R_n	n th register 9
r_n	contents of R_n 9
$Z(n)$	zero instruction 10
$S(n)$	successor instruction 10
$T(m, n)$	transfer instruction 10
$J(m, n, q)$	jump instruction 11
$r_n := x$	r_n becomes x 10
$P(a_1, a_2, \dots)$	computation under program P 16
$P(a_1, a_2, \dots) \downarrow$	the computation stops 16
$P(a_1, a_2, \dots) \uparrow$	the computation never stops 16
$P(a_1, a_2, \dots, a_n) \downarrow b$	the final value in R_1 is b 17
$\mathcal{C}, \mathcal{C}_n$	computable functions 17
$f_P^{(n)}$	n -ary function computed by P 21
c_M	characteristic function of M 22

Chapter 2	
U_i^n	projection functions 25
P	
PQ or Q	concatenation of programs 27
$\rho(P)$	denotes registers affected by P 2
$P[l_1, \dots, l_n \rightarrow l]$	28
$x - y$	cut-off subtraction 36

Chapter 3	
$\mathcal{R}, \mathcal{R}_0$	(partial) recursive functions 49
\mathcal{PR}	primitive recursive functions 51
$\mathcal{T}C$	Turing-computable functions 56
\xrightarrow{Q}	obtained by productions in Q 59
$\vdash_{\mathcal{G}}$	Post-system \mathcal{G} generates 59
$T_{\mathcal{G}}$	strings generated by \mathcal{G} 59
$\hat{\sigma}$	coding of a word σ 61
\tilde{n}	word representing n 61
$G(f)$	graph of f 62
\mathcal{PC}	Post-computable functions 63

Chapter 4	
\mathcal{I}	URM instructions 74
\mathcal{P}	URM programs 74
γ	program coding function 75
P_n	n th program = $\gamma^{-1}(n)$ 75
$\phi_a^{(n)}, \phi_a$	functions computed by P_a 76-77
$W_a^{(n)}, W_a$	domain of $\phi_a^{(n)}, \phi_a$ 77
$E_a^{(n)}, E_a$	range of $\phi_a^{(n)}, \phi_a$ 77

Chapter 5	
$\psi_U, \psi_U^{(n)}$	universal functions 86
$c_n(e, x, t)$	configuration code 87

Chapter 6	
$j_n(e, x, t)$	next instruction 87
$\sigma_n(e, x, t)$	state function 87
$T_n(e, x, t)$	Kleene T -predicate 89
$\text{Rec}(f, g)$	function obtained by recursion from f, g 91
$\text{Sub}(f, g_1, \dots, g_m)$	function obtained by substitution from f, g_1, \dots, g_m 91

Chapter 7	
$A \oplus B$	$\{2x : x \in A\} \cup \{2x + 1 : x \in B\}$ 122
$A \otimes B$	$\{\pi(x, y) : x \in A \text{ and } y \in B\}$ 122
K	$\{x : x \in W_x\}$ 123

Chapter 8	
\neg, \vee	logical symbols for 'not', 'or' 143
\mathcal{L}	statements of language L 144
\mathcal{T}, \mathcal{F}	true, false statements of L 144
θ_n	$(n+1)$ th statement of \mathcal{S} 144
$n \in K$	formal counterpart of $n \in \mathbb{K}$ 145
\mathcal{P}_i	provable statements 147
Pr^*	$\{n : n \in K \text{ is provable}\}$ 148
Ref^*	$\{n : n \notin K \text{ is provable}\}$ 148

Chapter 9	
$A \leq_m B$	A is many-one reducible to B 158
\equiv_m	many-one equivalent 161
$d_m(A)$	the m -degree of A 161
$a \leq_m b$	partial order on m -degrees 162
0_m	m -degree of recursive sets 163
\mathbf{o}, \mathbf{n}	m -degrees of \emptyset and \mathbb{N} 162
$0_m'$	m -degree of K 163

$a \cup b$	least upper bound of degrees a, b	165
$O(n)$	oracle instruction	167
P^χ	URMO program P with χ in the oracle	168
\mathcal{C}^χ	χ -computable functions	169
\mathcal{R}^χ	χ -partial recursive functions	170
$\phi_m^{\chi,n}, \phi_m^\chi$	functions computed by Q_m^χ	170–171
W_m^χ, E_m^χ	domain and range of ϕ_m^χ	171
$\psi_U^{\chi,n}$	universal function for χ -computability	171
K^χ	$=\{x: x \in W_x^\chi\}$	172
$P^A, \mathcal{C}^A, \phi_m^A, W_m^A, E_m^A, K^A$	relativised notions for A -computability	172
$A \leq_T B$	A is Turing reducible to B	174
\equiv_T	Turing equivalent	174
$d_T(A)$	Turing degree of A	175
$a \leq b$	partial order on T-degrees	176
0	T-degree of recursive sets	176
$0'$	T-degree of K	176
A'	jump of A	177
a'	jump of a	177
$a b$	a, b are incomparable degrees	179

Chapter 10

\mathcal{F}_n	n -ary partial functions	182
θ	a finite function	183
$\tilde{\theta}$	code for a finite function θ	183
f_Φ	least fixed point for Φ	192
f_τ	function defined by program τ	196

Chapter 11

E_k	sequence of computable functions enumerated by ϕ_k	208
D	diagonal enumeration	208

Chapter 12

$t_P^{(n)}(x)$	number of steps taken by P to compute $f_P(x)$	213
$t_e^{(n)}(x)$	$t_{P_e}^{(n)}(x)$	213
$\mathfrak{C}_b, \mathfrak{C}_b^*$	complexity classes of b	223, 233

\mathcal{E} elementary functions 225

$b_k(z)$  230

2.4 SWISS KNIFE OF USEFUL THEORETICAL DEFINITIONS

You don't need all the definitions to know here – e.g. computable function, URM machine/URM-Computable/Myhill theorem/First Recursion theorem/finite function one are here for clarification. The other ones actually came out in exercises or exams. Each one will be clarified concretely and by the point of view of the specific exercise in which it is used.

2.4.1 Computable function

DEFINITION 1.2 (Computable function). A function f is *computable* if there exists an algorithm that computes f .

2.4.2 URM-Machine

A URM-machine formalizes the notion of computable function by using an abstract machine called URM-machine (Unlimited Register Machine) which computes instructions effectively and finitarily thanks to the Church-Turing thesis. It has:

- unbounded memory that consists of an infinite sequence of registers, each of which can store a natural number
- a computing agent capable of executing an URM program
- a URM program, i.e. a finite sequence of instructions that can “locally” alter the configuration of the URM

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:
 - o if $r_m = r_n$ then jumps to I_t (jumps to t -th instruction)
 - o otherwise, it will continue with the next instruction

2.4.3 URM-Computable function

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

In this case we say that P computes f .

2.4.4 Reduction

DEFINITION 13.5. Let $A, B \subseteq \mathbb{N}$. We say that the problem $x \in A$ reduces to the problem $x \in B$ (or simply that A reduces to B), written $A \leq_m B$ if there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ computable and total such that, for every $x \in \mathbb{N}$

$$x \in A \Leftrightarrow f(x) \in B$$

In this case, we say that f is the *reduction function*.

More concisely:

Given sets $A, B \subseteq \mathbb{N}$, we say that $A \leq_m B$ if there exists a total computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for all $x \in \mathbb{N}$, it holds $x \in A$ iff $f(x) \in B$.

Usually, we use these reductions:

- $K \leq_m A$: to prove a set is not recursive
- $\overline{K} \leq_m A$: to prove a set is not r.e.

Why do we care?

Usually, we use K (also, respectively \overline{K}) which is not a recursive set (respectively not r.e.), and we show A is not recursive (respectively not r.e)

How to use it

We write a function of two arguments which basically goes on like this:

- if $x \in K$, it means there is a function of two arguments defined by cases
 - o its positive case will be $x \in K$ or $H(x, x, y)$ (meaning it halts)
 - o its negative case will be otherwise

The function is computable, given it can be written as the composition of computable functions or just the product/composition of sc_K (semicharacteristic function which gives 1 or 0) and f .

Use the smn-theorem proving there exists $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $f(x, y) = \phi_{s(x)}(y) \forall x, y \in \mathbb{N}$ and so $K \leq_m A$.

- if $x \in K$, we will have the positive case, so the domain will usually be the natural set and conditions/indices will be respected, hence $s(x) \in A$
- if $x \notin K$ we will have \uparrow and the domain will be the empty set, so indices/conditions will not be define, hence $s(x) \notin A$

2.4.5 Recursive Set

DEFINITION 13.1. A set $A \subseteq \mathbb{N}$ is *recursive* if its characteristic function

$$\chi_A : \mathbb{N} \rightarrow \mathbb{N}$$

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

is computable.

Also, remember, in words:

- A set is recursive if it can be expressed finitely and totally by cases and holds a trivial property
 - o if the property is not trivial, it's not recursive
- Specifically, a set is recursive because it is finite

2.4.6 Recursively Enumerable Set

DEFINITION 15.1 (Recursively enumerable set). We say that $A \subseteq \mathbb{N}$ is *recursively enumerable* if the semi-characteristic function

$$sc_A(x) = \begin{cases} 1 & x \in A \\ \uparrow & \text{otherwise} \end{cases}$$

is computable.

Specifically:

- A set is r.e. if I can check a property on a finite number of points
- A set is not r.e. if I have to check the property on an infinite number of points

If I am able to determine the property finitely because there are values “which I can search and find”, then I use and write a semicharacteristic function.

2.4.7 Decidable Predicate

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

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

is computable.

The correspondence for functions is the recursive case.

2.4.8 Bounded Minimalisation

Given a total function $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$, we define a function $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ as follows:

$$h(\vec{x}, y) = \mu z < y. f(\vec{x}, z) = \begin{cases} \text{minimum } z < y \text{ such that } f(\vec{x}, z) = 0 & \text{if it exists} \\ y & \text{otherwise} \end{cases}$$

2.4.9 Unbounded Minimalisation

DEFINITION 6.31. Let $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ be a function. Then the function $h : \mathbb{N}^k \rightarrow \mathbb{N}$ defined through **unbounded minimalisation** is:

$$h(\vec{x}) = \mu y. f(\vec{x}, y) = \begin{cases} \text{least } z \text{ s.t.} & \begin{cases} f(\vec{x}, z) = 0 \\ f(\vec{x}, z) \downarrow & f(\vec{x}, z') \neq 0 \quad \text{for } z < z' \\ \uparrow & \text{otherwise, if such a } z \text{ does not exist} \end{cases} \end{cases}$$

2.4.10 Semi-decidable predicate

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

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

is computable.

Again:

- If I am able to determine the property finitely because there are values “which I can search and find”, then I use and write a semicharacteristic function

The correspondence for functions is the recursively enumerable (r.e.) case.

2.4.11 Partially recursive functions

DEFINITION 7.1 (Partially recursive functions). The class \mathcal{R} of **partially recursive functions** is the least class of partial functions on the natural numbers which contains

- (a) zero function;
- (b) successor;
- (c) projections

and **closed** under

- (1) composition;
- (2) primitive recursion;
- (3) minimalisation.

2.4.12 Primitive Recursive Functions

Definition

The general one is:

DEFINITION 8.1 (Primitive recursive functions). The class of *primitive recursive functions* is the smallest class of functions \mathcal{PR} containing

- (a) zero function
- (b) successor
- (c) projections

and closed under

- (1) composition
- (2) primitive recursion

The one to use in the exam is:

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

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

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

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

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

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

A shorter one from an exam (I don't exactly know how precise you should be, this here is just for logical reference, given it was as said inside of one):

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

2.4.13 Smn-Theorem

Definition

Given $m, n \geq 1$ there is a total computable function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ such that $\forall \vec{x} \in \mathbb{N}^m, \forall \vec{y} \in \mathbb{N}^n, \forall e \in \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

What does it mean

It means that, whatever the exercise and the argument, we are parametrizing the function “to get where we want”. Concretely, this uses a function of two parameters thanks to which we are proving some conditions. Consider this as an intermediate to get to other things and it’s used everywhere.

The smn-theorem states that given a function $g(x, y)$ which is computable, there exists a total and computable function s such that $\phi_{s(x)}(y) = g(x, y)$, basically “fixing” the first argument of g – usually, we fix x in favor of y . It’s like partially applying an argument to a function. This is generalized over m, n tuples for x, y .

Usually, you use it to create a reduction function by first finding an appropriate $g(x, y)$ and then using the smn-theorem to say that there exists the previously cited function s , which is also the reduction function. The difficult part is finding the appropriate function $g(x, y)$, then the application of the smn-theorem is always the same (so, finding a “mapping function” computable, then parametrize and showing it is).

Note there is the simplified version of said theorem [here](#) (we refer to the definition present above anyway)

The Cutland book says: “The smn theorem is often useful in reducing $x \in W_x$ to other problems”.

2.4.14 Structure Theorem

Definition

Let $P(\vec{x}) \subseteq \mathbb{N}^k$ a predicate. Then $P(\vec{x})$ is semidecidable iff there is a decidable predicate $Q(t, \vec{x}) \subseteq \mathbb{N}^{k+1}$ s.t. $P(\vec{x}) = \exists t. Q(t, \vec{x})$

$$\text{def } P(\vec{x}) \subseteq \mathbb{N}^k \text{ a predicate} \\ P(\vec{x}) \text{ semi-decidable} \Leftrightarrow \begin{array}{l} \text{there is } Q(t, \vec{x}) \subseteq \mathbb{N}^{k+1} \text{ decidable} \\ \text{s.t. } P(\vec{x}) = \exists t. Q(t, \vec{x}) \end{array}$$

Note: in the “notes.pdf” the predicate is written as “decidable”, but prof. says it’s semidecidable like written here. Keep this in mind.

Why do we care

It's often asked in the oral exam; you need this theorem to show the second one (projection).

This reasoning is useful inside theoretical exercises about decidability/semidecidability because it's literally the same reasoning, reported here for the sake of completeness.

PROOF. (\Rightarrow) Let $P(\vec{x})$ be semi-decidable. It has a computable semi characteristic function sc_P so

$$P(\vec{x}) \equiv \exists t. H(e, \vec{x}, t)$$

therefore if we can rewrite H as $Q(t, \vec{x}) = H(e, \vec{x}, t)$, in this way Q is decidable as we wanted and

$$P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$$

(\Leftarrow) Let $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$ with $Q(t, \vec{x})$ decidable. Observe that

$$sc_P(\vec{x}) = \mathbf{1}(\mu t. |\chi_Q(t, \vec{x}) - 1|)$$

which is computable by definition, and therefore $P(\vec{x})$ is semi-decidable.

The converse does not hold, for example $P(\vec{x}, y) \equiv (x \in W_x) \exists x. P(x, y)$

Alternatively:

- $P(x, y) \equiv (y = 1) \wedge (x \notin W_x) Q(y) \equiv \exists x. P(x, y) \equiv (y = 1)$
- Suppose $P(x, y)$ holds if $\phi_x(x) \uparrow$, $P(x, y)$ non-semi-decidable, otherwise \overline{K} would be r.e.. We know there are programs inside \overline{K} , e.g. the ones calculating the always undefined function, but then $\exists x. P(x, y)$ always holds and so it would always be inevitably undecidable

2.4.15 Projection Theorem

Definition

THEOREM 15.6 (Projection theorem). Let $P(x, \vec{y})$ be semi-decidable; then

$$\exists x.P(x, \vec{y}) = P'(\vec{y})$$

is semi-decidable.

Why do we care

It's often asked in the oral exam.

This reasoning is useful inside theoretical exercises about decidability/semidecidability because it's literally the same reasoning, reported here for the sake of completeness.

Proof

Let $P(x, \vec{y}) \subseteq \mathbb{N}^{k+1}$ semi-decidable. Hence, by the structure theorem, there is $Q(t, x, \vec{y}) \subseteq \mathbb{N}^{k+2}$ decidable s.t. $P(x, \vec{y}) \equiv \exists t.Q(t, x, \vec{y})$.

Now $R(\vec{y}) \equiv \exists x.P(x, \vec{y}) \equiv \exists x.\exists t.Q(t, x, \vec{y}) \equiv \exists w.Q((w)_1, (w)_2, \vec{y})$ is decidable.

Hence, R is the existential quantification of a decidable predicate and by the structure theorem is semi-decidable.

2.4.16 Saturated set

A set $A \subseteq \mathbb{N}$ is saturated whenever, if it includes the index (program) for a computable function, it includes also all the other indexes (programs) for the same function. Formally, for all $x, y \in \mathbb{N}$ if $x \in A$ and $\varphi_x = \varphi_y$ then $y \in A$.

- A set is saturated if it describes non trivial properties or if it contains/regards properties of functions – it means that, taking two indices of a computation, not always they would have to compute the same property, so it is non-trivial (I think in words this gives you the idea, thank me later)
- Conversely, is not saturated if it does not regard a property of a function

2.4.17 Rice's Theorem

Definition

THEOREM 14.6 (Rice's theorem). Let $A \in \mathbb{N}, A \neq \emptyset, A \neq \mathbb{N}$ be saturated. Then it is not recursive.

What does it mean in practice?

It's used to show that a set is not recursive – usually, it's more of an help when we know the set is saturated, but we don't want to use the reduction from K (much longer).

Consider you usually need to know the idea of the proof of this one, because it allows you to say $\phi_e = f$ (so you use e as the index of computation to prove some properties over a defined number of steps).

How to use it

You need to prove the conditions above hold, particularly using the fact the set is saturated.

Remember a set $A \subseteq \mathbb{N}$ is considered saturated or extensional if for every pair of natural numbers m and n , if m belongs to A and the functions represented by the programs ϕ_m and ϕ_n are equal, then n must also belong to A .

There are a few examples of usage inside the exercises on Moodle or in [this](#) section.

Basically, if you used Rice-Shapiro for proving set is not r.e., just use Rice Theorem conditions to prove is not recursive, which is basically an extension and it's way shorter than doing a reduction.

2.4.18 Finite Function and Sub-function

DEFINITION 16.1 (Finite function). A finite function is a function $\theta : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{dom}(\theta)$ is finite.

Also, we define the *subfunction* this way:

Given $f : \mathbb{N} \rightarrow \mathbb{N}$, θ is a sub-function of f if $\theta \subseteq f$

This concept might appear obvious, but it's not. It's used in the following one, so Rice-Shapiro.

2.4.19 Rice-Shapiro's Theorem

Definition

Let $\mathcal{A} \subseteq \mathcal{C}$ be a set of computable functions. Then if set $A = \{x \mid \phi_x \in \mathcal{A}\}$ is r.e. then

$$\forall f (f \in \mathcal{A} \Leftrightarrow \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A})$$

What does it mean in practice?

It's used to show that sets are not r.e. – alternatively (in a few cases), it can be shown a set is not r.e. using a reduction from the complement of the halting set \overline{K} . Usually, it's the easiest way. If a set is not r.e., then it is also not recursive (given r.e. it's a superset of recursive).

How to use it

Generally, it can be used in two ways:

- $\exists f \in \mathcal{C}. f \notin \mathcal{A} \wedge \exists \theta \subseteq f \text{ finite}, \theta \in \mathcal{A} \Rightarrow A \text{ not r.e.}$
- $\exists f \in \mathcal{C}. f \in \mathcal{A} \wedge \forall \theta \subseteq f \text{ finite}, \theta \notin \mathcal{A} \Rightarrow A \text{ not r.e.}$

First, the set has to be saturated (which means there is a set of computable functions holding the exercise property, just writing *dom/cod* in place of *W/E*). We can use here functions like:

- *id*, which is the identity function, always defined for every natural number
 - o This one is *usually* used to show *id* $\in \mathcal{A}$ and $\exists \theta \notin \mathcal{A}$ (or viceversa) is not r.e.
 - o So, this is a function and also requires the use of subfunctions
- \emptyset , which is not the empty set, but a function with empty domain, called “always undefined function”
 - o This one is usually used as a subfunction (so first you need to have a function, such as the identity or something), so like $f \in \mathcal{A}$ but $\emptyset \notin \mathcal{A}$ (or viceversa) so A is not r.e.
- Other times, constant functions fit the bill (like **0, 1**) or just create custom functions/subfunctions

Not always a set and its complement are not r.e. – infact, sometimes a semicharacteristic function can be written and the set is r.e. Just consider the problem conditions and see if you can find such a function – often, it is just not recursive and not r.e. but don't go autopilot.

You can start from either the function or the subfunction - just prove that f is outside. Technically it is enough to find a pair of functions (f, θ) where theta is finite subfunction of f , theta is in the set and f is not in the set, this is because both are under an existential quantifier (f exists... and θ exists...). Thus there is no order between the two, it is just that introducing f first is generally preferable.

Why it's used

Because it makes recursiveness proofs much shorter than using a reduction. If you don't believe me, you will understand overtime and prove me right.

2.4.20 Myhill-Shepherdson Theorem

Definition

(1) Let $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^i)$ be a recursive function. Then, there exists a total computable function $h_\Phi: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \Phi(\phi_e^{(k)}) = \phi_{h_\Phi(e)}^{(i)}$ and h_Φ is extensional.

(2) Let $h: \mathbb{N} \rightarrow \mathbb{N}$ be a total computable function and h extensional.

Then, there is a unique recursive functional $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^i)$ s.t. for all $e \in \mathbb{N}$ (possible programs)

$$\Phi(\phi_e^{(k)}) = \phi_{h(e)}^{(i)}$$

What does it mean

Myhill-Shepherdson's theorem establishes a significant relationship between recursive functions and total computable functions. In the first part, it states that for any recursive function Φ mapping functions from $\mathbb{N}^k \rightarrow \mathbb{N}^i$, there exists a total computable function $h_\Phi: \mathbb{N} \rightarrow \mathbb{N}$ that captures the behavior of Φ on computable functions. This means that the actions of the recursive functional on computable functions can be entirely represented by a total extensional function operating on the indices of those functions.

In the second part, the theorem addresses the reverse scenario. Given a total computable and extensional function $h: \mathbb{N} \rightarrow \mathbb{N}$, there exists a unique recursive functional Φ such that, for all possible programs $e, \Phi(\phi_e^{(k)}) = \phi_{h(e)}^{(i)}$. This implies that computable extensional functions uniquely identify computable functions through program transformations.

Why do we care

This is basically a bridge for the two recursion theorems, and it is essential to understand the true meaning of the second one, which is quite powerful given the general result it holds.

In particular, if you look at the definition of it, you see that we're actually using this theorem every time we plug an index inside of Second Recursion Theorem, describing a property holds extensionally because there is at least one fixed point to consider.

2.4.21 First Recursion Theorem

Definition

THEOREM 17.9 (First recursion theorem (Kleene)). *Let $\Phi : \mathcal{F}(\mathbb{N}^k) \rightarrow \mathcal{F}(\mathbb{N}^h)$ be a recursive functional. Then Φ has a least fixed point f_Φ which is computable, i.e.*

- (1) $\Phi(f_\Phi) = f_\Phi$
- (2) $\forall g \in \mathcal{F}(\mathbb{N}^k) \quad \Phi(g) = g \Rightarrow f_\Phi \subseteq g$
- (3) f_Φ is computable

and we can see that $f_\Phi = \bigcup_n \Phi^n(\emptyset)$.

What does it mean

Prerequisites: the definitions of functionals and fixed points given [here](#).

This is also called “Kleene’s First Recursion Theorem” or Fixed-point theorem (of recursion theory). The Cutland Computability book specifies it is used to give “meaning” to programs, computing a recursive program, ensuring implementing the program will be defined rigorously over its inputs in a correct way.

The theorem above implies the closure of the set of computable functions with respect to extremely general forms of recursion.

Why do we care

This theorem allows us to characterize a program, considering a fixed point, with a single index. This index succinctly represents the entire set through a computable function. In practical terms, this means having a single program that encapsulates the entire problem definition or a specific exercise.

The function recursively defined are always computable as a consequence of this one.

How to use it

While the practical application might not be directly evident, the theoretical understanding gained from this theorem is foundational. It serves as a crucial tool in proving the existence of a single program that precisely defines an entire problem or exercise. This program is not just a static solution but is defined recursively on itself, making it computable.

In essence, understanding and applying Kleene's First Recursion Theorem is fundamental for asserting that "there exists a single program that clearly defines the whole problem definition" and that "it is defined recursively on itself," thus ensuring its computability.

This is used only as a theoretical result to understand the following theorem. No more, no less.

2.4.22 Second Recursion Theorem

Definition

The Second Recursion Theorem says that: for all functions $f : \mathbb{N} \rightarrow \mathbb{N}$, if f is total and computable then there is $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{f(e)}$.

What does it mean

Imagine you have a total computable function, let's call it f , which takes natural numbers as inputs and produces natural numbers as outputs. The theorem asserts that there exists a program, denoted as e_0 , such that when this program is applied to its own code, it computes the same function as f .

In other words, there's a program that, when executed, transforms itself into another program (denoted as ϕ_{e_0}) that computes the exact same function as the original function f . The key here is that the program is changed during this transformation, but the function it computes remains the same.

The theorem emphasizes that this transformation is not just about making an identical copy of the original program. Even if two programs compute the same thing (they have the same input-output behavior), the Second Recursion Theorem tells us that they can be transformed into each other while still computing the same function. This property holds true even when the function f is not extensional, meaning it doesn't depend on the specific representations of its programs.

How to use it

There is a dedicated category of exercises on this explained fairly well what do precisely [here](#).

Basically, we use a single index to prove there is a fixed point and the whole set will respect on that function the overall definition. I think intuitively is just this.

3 INTRODUCTION TO THE COURSE

Course reference page: <http://www.math.unipd.it/~baldan/Computability>

(This lesson is based on the only set of slides of the course, available as "Intro-en.pdf")

We start by a simple reflection; can we give the enumeration of all numbers and store them efficiently? A suggestion might be, “rather than the phone number itself, you might store a program that generates the number”. So, instead of 0123456789 We can write *for i = 0 to 39 do print (i mod 10)*

It isn't convenient; there are *numbers n* such that, for all *program P* generating *n*, $\text{size}(n) \leq \text{size}(P)$. These are defined as *random numbers*; we observe there are an infinite number of them. There is no program capable of determining whether a number is random or not, because such a program does not exist.

Exercise (coming from the 8th slide – solution of this exercise made at the end of the course and present [here](#); in any case, see it after having at least a grasp of everything, like 100% completion of the course):

- 1) Prove that there are infinitely many random numbers
- 2) Prove there is no program able whether a number is random or not

Notes on the previous:

What we do know is that not all problems are not solvable by a computer, because of power constraints and limitations of machines, e.g. the halting problem and the program correctness (it's impossible for even simple specifications). A natural question we naturally ask: “Which problems can we solve by a computer / by an effective procedure?”. Some problems are intrinsically theoretical, so they are completely independent from the underlying computation model.

Other specific questions:

- What is an *effective procedure*?
 - o Maybe the simple program can do the job, but we must prove it formally
- What does it mean that a *problem is solved by an effective procedure*?
- Characterize the problems that can and those that cannot be solved
 - o Problems that are not always binary
- Relating *unsolvable* problem (degree of unsolvability)

We tend to classify *solvable/unsolvable problems without limitations on the use of resources* (memory and time). For example, the complexity theory, considering the resources and classifying solvable problems in an hierarchy according to their “difficulty”.

Computability theory is a branch of computer science and mathematics that explores the theoretical limits of computation, this well before its proper birth. It revolves around the concept of decidability and undecidability, focusing on what can and cannot be computed algorithmically.

So, *computer science* may be described as “the ability of building and using tools, according to some (codified) procedure, is a distinctive feature of human beings”. It depends on “how we use the tools and what we find out when we do”, according to *Dijkstra*.

We don't tend to think meaningfully always, but to think *according to patterns*, because there is a general combinatoric procedure to find all truths, reasoning and deriving consequences from a set of premises.

Thing is, it doesn't depend on the language, but we can try to represent things abstractly as a set of customized symbols (creating laws or languages), compute them logically (arithmetically) without contradiction and evaluating problems with procedures, to avoid controversies of decidability and solvability as criteria (*Leibniz, Boole, Lullus and others*) using *logic* as the main foundation.

Others posed the need of an artificial language, formally with syntactic and manipulation rules that can be programmed via *variables* and *statements*.

Using cases like Russell's paradox, we can use the same tools we already have to contradict ourselves and pushing further, even finding new meanings, possibly having a *consistent* system, where it proves itself as correct solidly (*Hilbert*). Many times, this observation led to creation of special-purpose machines, able to compute a specific class of problems.

We might try to take problems considering a small set of rules, which may not be always complete or prove the consistency of the theory (*Gödel*). There may be a machine which computes a problem given a computable function and the same language, given a specific input and an output (*Turing*).

We may express a universal machine to make *any kind of calculation*, storing the result of operations (memory) and solving problem discretely (*Von Neumann*).

Other things:

- On Moodle there are unofficial notes
- There are the exercises with solutions
 - o note for the reader: the exercises, sadly, are not in order of difficulty
- There will be tutoring activities for this course
 - o thank God, you will see more later why :D
 - o in [this](#) section, you can also find some recordings of tutorings

4 ALGORITHMS, EFFECTIVE PROCEDURES, NON-COMPUTABLE FUNCTIONS

An effective procedure it's just a sequence of *elementary steps* which are describing a procedure intended to solve a problem (reaching some objective mechanically), transforming some *input* into some *output*.

We can see an algorithm as a black box of sort:



If this is deterministic, we can mathematically describe a function $f: \{\text{inputs}\} \rightarrow \{\text{outputs}\}$, where each possible input will uniquely determine the corresponding output (we will see later this happens on *partial functions*, so maps between two sets X and Y that may not be defined on the entire set X ; an example might be the square root, where not all real numbers have real square roots so *we can compute it but not always solve it*).

A function f is computable if *there exists* an algorithm such that the induced function is f (so f is the function computed if f is *effectively computable*). It's important to note the algorithm that computes f must exist.

We informally expect some functions to be computable, given the definition above, such as:

- $\text{GCD}(x, y) = \text{greatest common divisor}$ (*Euclid's algorithm*)
- $f(n) = \begin{cases} 1, & n \text{ is prime} \\ 0, & \text{otherwise} \end{cases}$
- $g(n) = p_n$, where p_n is the n^{th} prime number (eventually an n -th prime will be found)
- $h(n) = n^{\text{th}} \text{ digit in } \pi$
 - o this is a series that converges to π and we work with techniques to allow rounding the error, such as truncating the series or rounding the computation

Let's give an interesting example:

- $f(n) = \begin{cases} 1, & \text{if in } \pi \text{ there are exactly } n \text{ consecutive 5's} \\ 0, & \text{otherwise} \end{cases}$
 - o example: if $\pi = 3.14 \dots 755552 \dots$
 - o $f(4) = 1$
 - o More generally, it can be written, for example as $g(3) = 1$ iff $\pi = 3.14 \dots i555j \dots i, j \neq 5$
 - (where iff means "if and only if")

The naïve idea of this last one is:

- compute all the digits of π
- check if there are n digits of 5 in a row

This, however, is not an algorithm, because we can't exclude entirely the generation of n 5's at some point.

Since π 's decimal expansion is non-repeating and doesn't follow a simple pattern, we cannot guarantee that the algorithm won't eventually find the desired sequence of n 5's (given π is an irrational number), so we may run it indefinitely and will eventually become infeasible, because we have no way of returning 0.

Is this function computable? In the case of this one, we don't have an effective procedure known to us to determine whether it's computable or not (hence, it's *not an effective procedure*). The fact that we can't exclude the existence of an effective procedure *doesn't mean* the function is computable, but it also *doesn't definitively prove* that it is computable.

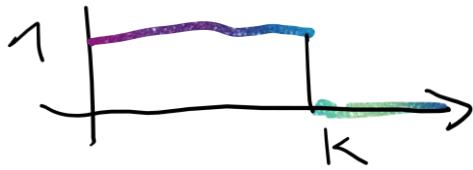
Let's consider now a slightly different example, for a function $g: N \rightarrow N$:

- $g(n) = \begin{cases} 1, & \text{if } \pi \text{ includes at least } n \text{ digits 5 in a row} \\ 0, & \text{otherwise} \end{cases}$
 - o if $\pi = 3.14 \dots 755552 \dots$
 - o We deduce that, somehow, we will reach 1 as constant substituting the values
 - $g(4) = 1, g(3) = 1, g(2) = 1, g(1) = 1, g(0) = 0$
 - o More generally:
 - if $g(n) = 1$ then $g(m) = 1, \forall m \leq n$,

Consider $K = \sup\{n \mid \text{there are } n \text{ digits 5 in a row in } \pi\}$

We then have two possibilities (with plot of the functions reported here, given its quite simple shape):

- K finite, so $g(n) = \begin{cases} 1, & \text{if } n \leq K \\ 0, & \text{otherwise} \end{cases}$



function $g(n)$;
if $n \leq K$ return 1
else return \emptyset

- K infinite, so $g(n) = 1, \forall n \in N$



function $g(n)$;
return 1

This implies the function is computable, because it behaves regularly (step function, so either 1 or 0, or just a constant function, so they can be computed by simple programs). Even though we won't know the exact shape of the function, this way we proved it's computable (the function shape is irrelevant in knowing which program will compute the function, but if finite, they can be a simple tool to see it).

Can we use the same argument for f ?

$$f(n) = \begin{cases} 1, & \text{if in } \pi \text{ there are exactly } n \text{ consecutive 5's} \\ 0, & \text{otherwise} \end{cases}$$

Let $A = \{n \mid \text{there are exactly } n \text{ digits 5 in a row in } \pi\}$

and take:

function $f(n)$
if $n \in A$:
 return n
else
 return \emptyset

Problem is, $f(n)$ is not computable in the slightest, because the set A is possibly infinite and there is no such a thing as a finite representation for it (in the notes, it's also present an example of a function $G: N \rightarrow N$ which is 1 if $P = NP$, 0 otherwise; since the condition does not depend on the variable, it can have either way 1 or 0 as value, so the function G remains computable, but if posed inside the set A would be equally incomputable).

4.1 EXISTENCE OF NON-COMPUTABLE FUNCTIONS

This poses the question for the existence of non-computable functions, because it suggests $f(n)$ is computable, because the set is possibly infinite, so we can't provide a finite representation.

A good algorithm should satisfy the following characteristics which can be ideally implemented in a theoretical machine we call *computational model*, this way being considered *effective*:

- it has a *finite length*
- there exists a *computing agent* able to execute the algorithm instructions
 - o this agent has a *memory* to store the input, results and steps and it is *unbounded*
 - even if the algorithm will be finite, we assume it is unbounded for the sake of analyzing if it's computable or not (large, but never using the full space)
 - this way, we will be able to define algorithms working on any possible input and there is no limit on the memory that can be used
 - o the computation consists in *discrete steps*, not probabilistic or not-deterministic
 - o finite limit to number of instructions and the power of their complexity
 - this way representing a finite machine
- the computation can
 - o terminate in a finite yet unbounded number of steps → output
 - o diverge (never terminate) → no output

Let's recall the *math notation* needed to understand the subsequent inference of non-computable functions for every "effective" computational model.

- $\mathbb{N} = \{0, 1, 2, \dots\}$ set of *natural numbers* (so finite and always with a successor)
- $A \times B = \{(a, b) \mid a \in A, b \in B\}$ as *Cartesian product* (combine two sets to create an ordered one)
 - o We will write, having A set, $A^n = A \times A \times A \dots \times A$ (n times)

- *binary relation* or *predicate* as $r \subseteq A \times B$

$$\text{m divides m} \xrightarrow{\quad} \text{divides} = \left\{ (m, m * k) \mid m, k \in \mathbb{N} \right\}$$

$$\mathbb{N} \times \mathbb{N}$$

- $f : A \rightarrow B$, the *partial function*, special relation $f \subseteq A \times B$ such that

$$\forall a \in A, \forall b, b' \in B, (a, b), (a, b') \in f \rightarrow b = b'$$

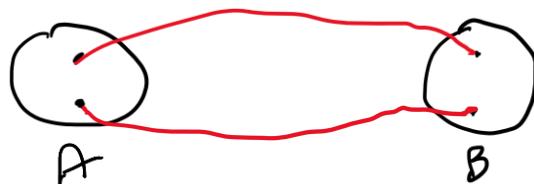


- o $\text{dom}(f) = \{a \mid \exists b \in B, (a, b) \in f\}$
- o We write $f(a) \downarrow$ for $a \in \text{dom}(f)$ and $f(a) \uparrow$ for $a \notin \text{dom}(f)$

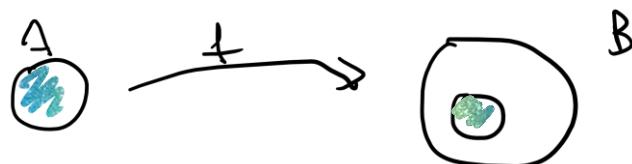
- In words, we essentially say it's a mathematical relationship that associates elements from a set A to elements in a set B , but it may not be defined for all elements in A (for example, not all pairs)
- When you apply the partial function to an element a in its domain, you write $f(a) \downarrow$ to indicate that the function is defined and yields a result. Conversely, if you try to apply the function to an element a outside its domain, you write $f(a) \uparrow$ to signify that the function is undefined for that input.

Given a set A , we indicate with $|A|$ the cardinality (number of elements), then we define, for sets A and B :

- $|A| = |B|$ if there is $f: A \rightarrow B$ bijective (unique and complete mapping)



- $|A| \leq |B|$ if there is $f: A \rightarrow B$ injective (no two different inputs map to the same output)

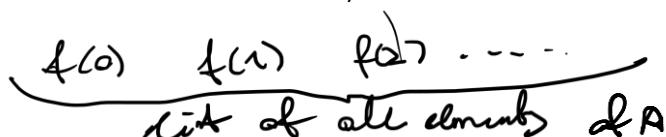


- equivalently, if there is a surjection $g: B \rightarrow A$ (covering the entire codomain – all possible outputs)



Observe also that if $A \subseteq B$, $|A| \leq |B|$, having injectivity in between.

- A countable (denumerable), $|A| \leq \mathbb{N}$, having a surjective function $f: \mathbb{N} \rightarrow A$ (listing all the elements one after the other)



- If A, B countable $\rightarrow A \times B$ countable
- A countable union of countable sets is countable: A_0, A_1, A_2 countable sets $\rightarrow \bigcup_{i \in \mathbb{N}} A_i$ countable

Idea (just to visualize the whole thing, place the elements in a matrix and enumerate them in diagonals):

A	a_0	a_1	$a_2 \dots$
B	b_0	b_1	$b_2 \dots$
	a_0	a_1	a_2
	b_0	(a_0, b_0)	(a_1, b_0)
	b_1	(b_1, a_0)	(b_1, a_1)
	b_2	(b_2, a_0)	(b_2, a_1)

dove tail enumeration

This so called “dove tail enumeration” means systematically listing all functions from A to B :

- Begin by listing the element at position $(0, 0)$ in the matrix, which is the function that maps a_0 to b_0 .
- Then, move along the diagonals of the matrix, listing the elements in order

Let's come back to the existence of non-computable functions: we focus on unary function over the natural numbers (function that takes a single argument or input variable and produces a single output):

$$\mathcal{F} = \{f \mid f: \mathbb{N} \rightarrow \mathbb{N}\} \text{ set of all partial unary functions on } \mathbb{N}$$

We then fix a model of computation, which then induces a set of algorithms, for example a set \mathcal{A} of all algorithms inside of it. Given an algorithm $A \in \mathcal{A}$, we compute a function $f_A : \mathbb{N} \rightarrow \mathbb{N}$, which is said to be *computable* in our model if there exists an algorithm that computes it.

Hence, we define *functions computable in A* like:

$$\mathcal{F}_\mathcal{A} = \{f \mid \text{there exists } A \in \mathcal{A} \text{ s.t. } f_A = f\} = \{f_A \mid A \in \mathcal{A}\}$$

Clearly we have $\mathcal{F}_\mathcal{A} \subseteq \mathcal{F}$. Is this inclusion strict? (so $\mathcal{F}_\mathcal{A} \subsetneq \mathcal{F}$, which means (is there a non-computable function?) ! yes)

The answer is yes, because the algorithms are too few to compute all the functions, so they must be countable in some way, hence by logical closure computable.

By assumption, an algorithm is a finite sequence of instructions from an instruction set I , which we assume finite. We can interpret all of this as a big union of finite algorithms.

$$\mathcal{A} = I \cup I^I \cup I^{I^I} \cup \dots = \bigcup_{i \geq 1, i \in \mathbb{N}} I^i \text{ (countable union of countable sets} \rightarrow \text{countable)}$$

Given $|\mathcal{A}| \leq \mathbb{N}$ and since we have $\mathcal{A} \rightarrow \mathcal{F}_\mathcal{A}$, $\mathcal{A} \rightarrow F_A$ (which means it is surjective by definition), we have:

$$|F_A| \leq |A| \leq |\mathbb{N}|$$

What we say in words is this: the set of all algorithms \mathcal{A} in our fixed computational model and F_A , the set of computable functions are as many as the natural numbers.

On the other hand, the set of all functions F is not countable. Why? Assume for the sake of contradiction that it is so:

$$|F| \leq |N|$$

We can list the elements of F like we did before (taking, with diagonalization, the main diagonal, then systematically changing diagonal values):

	f_0	f_1	f_2	f_3	\dots
0	$f_0(0)$	$f_1(0)$	$f_2(0)$	$f_3(0)$	\dots
1	$f_0(1)$	$f_1(1)$	$f_2(1)$	$f_3(1)$	\dots
2	$f_0(2)$	$f_1(2)$	$f_2(2)$	$f_3(2)$	\dots
3	\vdots	\vdots	\vdots	\vdots	

change it systematically
if $\downarrow \rightsquigarrow \uparrow$
 $\uparrow \rightarrow 0$

then build a function on that, like this one:

$$d : N \rightarrow N$$

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

d is a function which is *total* (so, defined for every natural number) in F so there is $n \in N$ s.t. $d = f_n$

- if $f_n(n) \downarrow$ then $d(n) \uparrow \neq f_n(n)$ (meaning d is not defined at n , since $f(n) = f_n(n) + 1 \neq f_n(n)$ and it means we are not enumerating the current function inside the natural numbers, which we assume we can always do since is countable; so there is the contradiction)
- if $f_n(n) \uparrow$ then $d(n) = 0 \neq f_n(n)$ (again, d not defined in n and we do not enumerate the function assuming we can, hence another contradiction)

Since d is distinct from all the functions in the enumeration, it demonstrates that the set of all functions F is uncountable, because it cannot be put in one-to-one correspondence with the set of natural numbers N .

Summing up in math notation (there are more function than natural numbers, even though finite algorithms are as many as natural numbers):

$$\mathcal{F} \text{ not countable}, |\mathcal{F}| > |N|$$

$$\mathcal{F}_A \subseteq \mathcal{F}$$

$$|\mathcal{F}_A| \leq |N| \leq |\mathcal{F}| \rightarrow \mathcal{F}_A \not\subseteq \mathcal{F}$$

Note that we can't count non-computable functions, so:

$$|\mathcal{F} \setminus \mathcal{F}_A| > |N|$$

We conclude that:

- no computational model can compute all functions
- there are more non-computable than computable functions

5 URM COMPUTABILITY

To give a good notion of computability, we must choose a good model of computation, inducing a class of algorithms and computable functions. There can be:

- Turing Machines (finite-state control, reading, writing, initial/final configuration)
- λ -calculus (a design of programming where one designs/applies functions based on primitives)
- Partial recursive functions (functions calculated with specific function that build partially)
- Canonical deduction systems (system used to create proofs logically via connectives and trees)
- URM (Unlimited Register Machines)

Whatever the model, we may concern if a specific theory may be valid for the specific model.

According to the Church-Turing thesis, a function is computable by an *effective procedure* if and only if it's *computable by a Turing machine* (we resort to this to shorten the proof that a certain function is computable and it's used informally as notion of effectiveness, then must be supported by evidence). This says that a function is computationally robust and we can choose whatever model one likes.

The notion of computable function will be formalized by using the URM-machine, abstraction based on the Von Neumann's model. It has many characteristics:

- *memory is unbounded*, using an infinite number of *registers* storing each a natural number (where a sequence of registers is called *configuration*);



- it executes a program, based on a finite list of instructions (and a *computing agent* able to execute it);

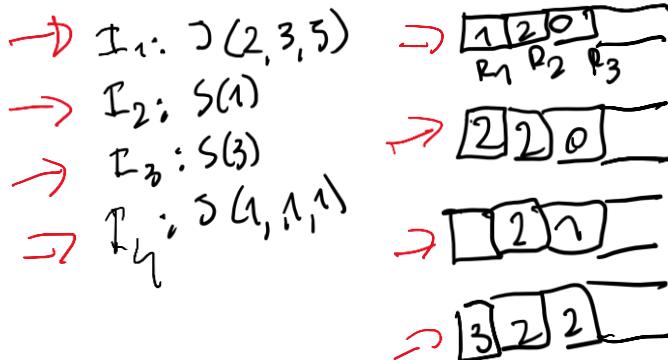
I_1, I_2, \dots, I_s (instructions set)
 $\nwarrow (s = 5)$

- it has *arithmetic instructions*, characterized by the fact that the instruction to be executed in the next step is the one following the current instruction in the program. They are:
 - o *zero Z(n)*, which sets the content of register R_n to zero: $r_n \leftarrow 0$
 - o *successor S(n)*, which increments by 1 the content of register R_n : $r_n \leftarrow r_n + 1$
 - o *transfer T(m, n)*, which transfers the content of register R_m into R_n , which R_m staying untouched: $r_n \leftarrow r_m$
 - o *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

The computation:

- starts from an initial configuration of registers and executes I_1
- terminates if
 - o the instruction to be executed does not exist
 - o it's the last instruction
 - o you jump out of the program yourself

An example might be the following one:



(continuing now
increments register 1,
register 3 and
jump if the
registers have
the same value,
then again jumping)

In LaTeX form, to not kill your eyes that much, coming from the notes:

R_1	R_2	R_3	\dots	$\xrightarrow{I_1, I_2}$	R_1	R_2	R_3	\dots	$\xrightarrow{I_3}$	R_1	R_2	R_3	\dots	$\xrightarrow{I_4, I_1, I_2}$
1	2	0	\dots		2	2	0	\dots		2	2	1	\dots	
R_1	R_2	R_3	\dots	$\xrightarrow{I_3}$	R_1	R_2	R_3	\dots	$\xrightarrow{I_4, I_1, I_5}$	R_1	R_2	R_3	\dots	
3	2	1	\dots		3	2	2	\dots					\dots	

As we're using the Church-Turing thesis, we're defining a machine, so we must describe which *states* it has: there is a *register configuration* c , taking the register content and index the next instruction via a program counter t . Also, *operational semantics* can be defined via $\langle c, t \rangle$.

A computation can possibly diverge (not terminate); consider for instance this program:

$$I_1: S(1)$$

$$I_2: J(1, 1, 1)$$

Then the computation will not terminate. For instance

R_1	R_2	R_3	\dots	$\xrightarrow{I_1, I_2}$	R_1	R_2	R_3	\dots	$\xrightarrow{I_1, I_2}$	R_1	R_2	R_3	\dots	$\rightarrow \dots$
0	0	0	\dots		1	0	0	\dots		2	0	0	\dots	

Let P be an *URM* program. Given a sequence of natural numbers $a_1, a_2, \dots \in \mathbb{N}$, $P(a_1, a_2, \dots)$ indicates the computation of P from (a_1, a_2, \dots) :

R_1	R_2	R_3	\dots
a_1	a_2	a_3	\dots

- $P(a_1, a_2, \dots) \downarrow$ if the computation eventually terminates (*halts*)
- $P(a_1, a_2, \dots) \uparrow$ if the computation diverges (*never halts*)

We work on computations that start from an initial configuration where only a *finite number of registers contain a non-zero value*. So, given $a_1, \dots, a_k \in \mathbb{N}$, $P(a_1, \dots, a_k)$ denotes $P(a_1, \dots, a_k)$ for $P(a_1, \dots, a_k, 0, \dots, 0)$.

The notation then extends to the previous ones, stating that at the end of a program we will have a valid value $\rightarrow P(a_1, \dots, a_k) \downarrow a$ for $P(a_1, \dots, a_k) \downarrow a$ and in final configuration $r_1 = a$.

5.1 URM-COMPUTABLE FUNCTIONS AND EXAMPLES

For URM-computable functions, given a function $f: \mathbb{N}^k \rightarrow \mathbb{N}$ (possibly partial), we say is URM-computable if there is a *URM* program P such that, $\forall (a_1, \dots, a_k) \in \mathbb{N}^k, \forall a \in \mathbb{N}, P(a_1, \dots, a_k) \downarrow a$ if and only if $(a_1, \dots, a_k) \in \text{dom}(f)$ and $f(a_1, \dots, a_k) = a$.

In words, for any input tuple of natural numbers, if you run the URM program on this input it will eventually have a result equal to the output of the function for input. This way, P computes f .

We then define, $\mathcal{C}^k = \{f \mid f: \mathbb{N}^k \rightarrow \mathbb{N} \text{ computable (URM)}\}$ as the *classes of computable functions*.

Therefore $\mathcal{C} = \bigcup_{k \geq 1} \mathcal{C}^k$ is the union of all of them.

We next list some examples of URM-computable functions, providing the corresponding programs:

$$(1) \quad f: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$f(x, y) = x + y$$

$$I_1: J(2, 3, 5)$$

$$I_2: S(1)$$

$$I_3: S(3)$$

$$I_4: J(1, 1, 1) \quad // \text{ unconditional jump}$$



Idea: Incrementing R_1 and R_3 until R_2 and R_3 contain the same value, resulting in adding to R_1 the content of R_2 . Specifically:

loop: $J(2, 3, \text{stop})$
 $S(1)$
 $S(3)$
 $J(1, 1, \text{loop})$

$$(2) \quad f: \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = x - 1 = \begin{cases} 0 & x = 0 \\ x - 1 & x > 0 \end{cases}$$

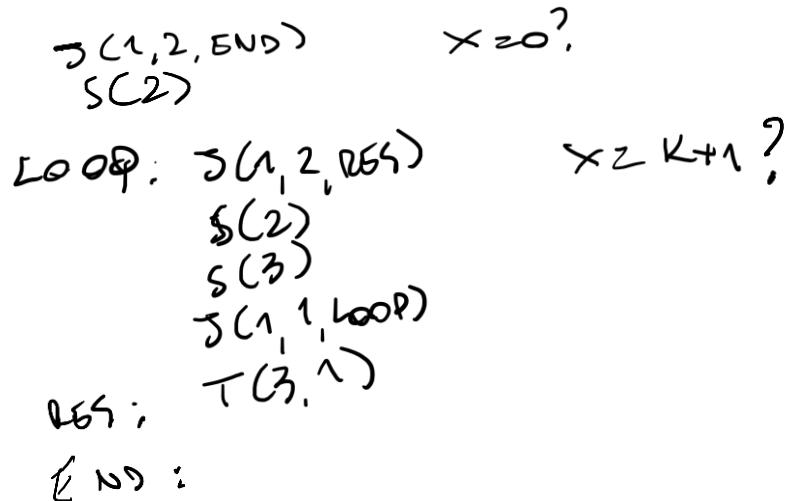
otherwise



Now, let's analyze how this program works:

- If $x = 0$, it will jump to instruction 8, which presumably indicates the end of the program.
- If $x = 1$, it will go through instructions 2, 3, and 7, effectively setting $R_1(x)$ to 0.
- If $x > 1$, it will go through instructions 2, 3, 4, 5, 6, and 7, effectively setting $R_1(x)$ to $x - 1$.

Idea: if $x = 0$ it trivially terminates; if $x > 0$, it keeps a value $k - 1$ in R_2 and k in R_5 , with $k > 1$ ascending until $R_3 = x$, at that point $R_2 = x - 1$.



The core concept behind this program is to continually subtract from the input value, considering the partial nature of the function. This means that the program might not always terminate, even if the function is computable, or it might terminate when the function is not computable.

In this specific example, the program checks if two values are equal; if they are, it jumps to a different instruction. If they are not equal, it subtracts one from the value. This subtraction continues until there is memory available for further operations.

Courtesy of notes (slightly different, but same example):

Idea: if $x = 0$ it trivially terminates; if $x > 0$, it keeps a value $k - 1$ in R_2 and k in R_5 , with $k > 1$ ascending until $R_3 = x$, at that point $R_2 = x - 1$.

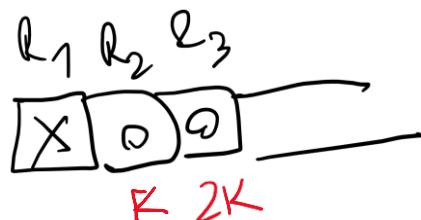
Here's the program

- $I_1: J(1,3,8)$
- $I_2: S(3)$
- $I_3: J(1,3,7)$
- $I_4: S(2)$
- $I_5: S(3)$
- $I_6: J(1,1,3)$
- $I_7: T(2,1)$

Let's consider a different function:

$$(3) \quad f : \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ even} \\ \uparrow & \text{otherwise} \end{cases}$$



The function behaves as follows:

- If the input number x is even, the function returns half of x (store an increasing even number in R_2 then storing its' half in R_3)
- If the input number x is odd, the function does not terminate (indicated by the symbol \uparrow).

The program continues executing these instructions in a loop. If the initial input x is even, it will eventually reach a point where $R_1(x)$ equals the even number in R_2 , and it will jump to instruction 6, halving the input x . If x is initially odd, the program keeps increasing the even number in R_2 , and it never reaches the halting condition, resulting in a non-termination, as indicated by \uparrow .

*Loop: $S(1, 3, \text{res})$
 $S(2)$
 $S(3)$
 $J(1, 1, \text{loop})$
 Dots, $T(2, 1)$*

Given a program P , for some fixed number of parameters $k \geq 1$, there exists a unique function computed by P that we denote as follows as $f_P^k: \mathbb{N}^k \rightarrow \mathbb{N}$. More precisely:

$$f_P^{(k)}(a_1, \dots, a_k) = \begin{cases} a & \text{if } P(a_1, \dots, a_k) \downarrow a \\ \uparrow & \text{if } P(a_1, \dots, a_k) \uparrow \end{cases}$$

In words: given a fixed number of parameters, the program halts if there is a final character of computation, otherwise the function will terminate when the program terminates. Remember:

- a program terminates or not, a function is defined or not. A function is not computing, only the program does (they are correlated, of course).
- the same function can be computed via different algorithms, which means different problems

Question: given $f: \mathbb{N}^k \rightarrow \mathbb{N}$ how many computing are there computing F ?

Answer: We can have infinitely many if and the only if the function is computable; so, 0 or infinitely many

5.2 EXERCISES

Exercise

Consider URM^- , class of URM machines without transfer instructions (so, no $T(m, n)$). We indicate C^- the class of URM computable functions. How does C^- compare to C ? (in math notation, $C = C^-$)

(Thoughts)

We can use $T(m, n)$ as we can zero in and increment the register m until it reaches n .

The idea is:

$Z(n)$

LOOP: $J(m, n, DONE)$ (if the registers are equal, it exits the loop)

$S(n)$

Written by Gabriel R.

$J(1,1,LOOP)$ (back the program to the loop beginning)

In plain terms, this program aims to achieve a similar effect as the transfer instruction $T(m, n)$ by repeatedly incrementing the value in register R_n until it matches the value in register R_m . Once they are equal, it exits the loop.

Proof

We show that $C^- = C$. Let $f: \mathbb{N}^k \rightarrow \mathbb{N}$ computable in URM^- $f \in C^-$ i.e. there is P in URM^- . Just observe that P is also a URM program $\rightarrow f \in C$. As said in thoughts, ideally we replace the transfer instruction $T(m, n)$ as the $t - th$ step with the previous subroutine:

```

    Z(n)
    LOOP : J(m, n, END)
    S(n)
    J(1, 1, LOOP)
    END:
  
```

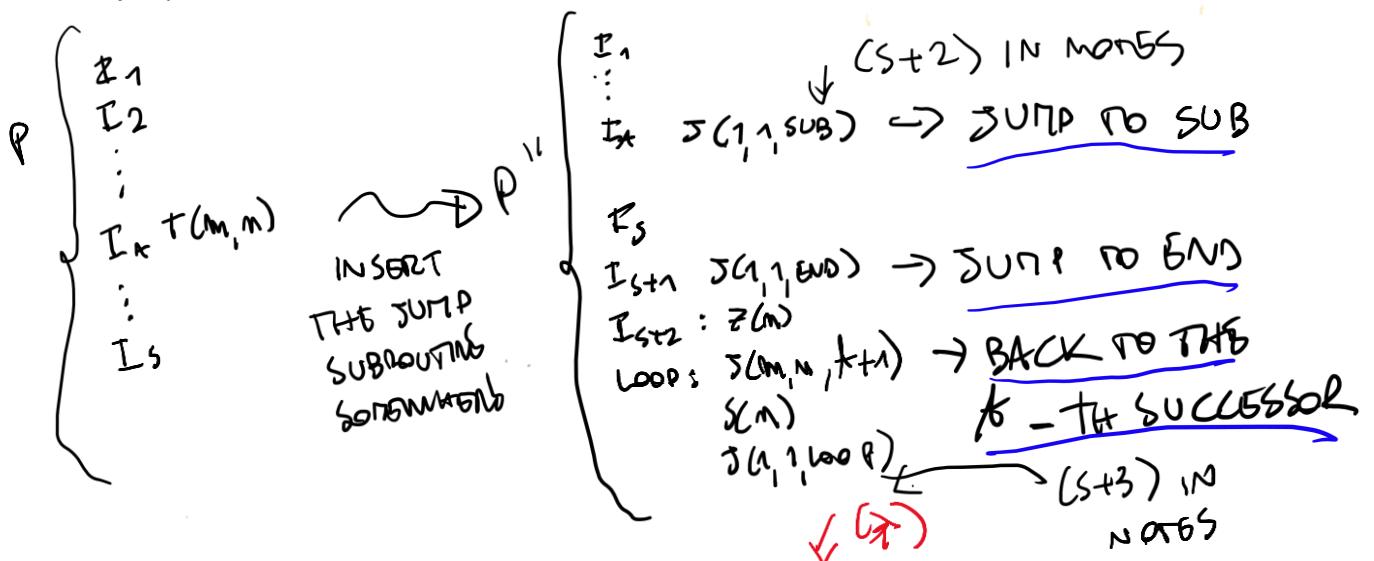
Let then $f \in C, f: \mathbb{N}^k \rightarrow \mathbb{N}$. Hence there is a URM program P such that $f = f_p^k$.

We show there exists $P^- URM^-$ machine such that $f_{p'}^k = f_p^k$.

Remember: We're assuming P is well formed: if it terminates, it will at instruction $n + 1$.

We proceed by induction on h , which is the number of transfer instructions $T(m, n)$ in P . We can assume, without loss of generality, that when a program halts it does so at the index of the last instruction plus one (induction logic at its core, in words).

- $h = 0$: trivial, as P with no transfer instructions is already a URM^- program, hence $P^- = P$
- $h \rightarrow h + 1$: let P be the URM program with $h + 1$ transfer instructions. Hence, we replace T with a jump to the subroutine:



We call this program P'' which has h transfer instructions and $f_p^k = f_{p''}^k$. By inductive hypothesis there is a $P' URM^-$ program such that $f_{p'}^k = f_{p''}^k$. Putting things together:

$$f_p^k = f_{p''}^k = f_{p'}^k \quad \text{with } P' \text{ URM}^- \text{ program}$$

Note: for any URM program P there is a well-formed program P' computing the same function. In fact:

$I_1, \dots, I_s, J(m, n, t), \text{if } t > s, \text{ replace it with } J(m, n, S + 1)$

In words:

For any URM program P , you can create an equivalent well-formed program P' that computes the same function. To do this, you replace any conditional jump instruction $J(m, n, t)$ where t is greater than the total number of instructions (S) with $J(m, n, S + 1)$. This ensures that the program always jumps to a valid instruction and remains well-formed while achieving the same computational result as the original program.

Exercise

Variant URM^S machine where there are no traditional transfer instructions such that $T(m, n)$, but $T^S(m, n)$ (swap instructions) like $r_m \Leftrightarrow r_n$ to exchange context of registers.

How does C^S compare to C ? (is $C^S = C$?)

My take on the proof

Let us prove $C^S \subseteq C$. We can replace the swap instructions with a few transfer instructions, formalizing how $T_S(m, n)$ can be encoded in means of the routine. We can explain this in terms of how a swap instruction works in programming: we allocate a new register/new variable, we assign the variable to save to the new variable, then the new variable will get assigned to the second variable.

So, we create something like:

$T(n, new)$

$T(m, n)$

$J(new, m)$

Formally, having a function $f \in C^S, f: \mathbb{N} \rightarrow \mathbb{N}$, we have a program P as a URM^S program such that $f_{p'}^k = f_p^k$. Proceeding by induction:

- $h = 0$, the program is already a URM program and $P' = P$ (such as before)
- $h \rightarrow h + 1$, where the program, by induction, must have at least some transfer instructions to realize how a swap works. So, if $f_{p'}^k = f_p^k$ only if this program uses both URM and URM^S instructions (the swap can't be explained otherwise, and we need this statement to make this work correctly).
 - o This way we can prove that with the $(h + 1)^{th}$ swap, will have at least h swap instructions, given the swap will be given via a jump instruction reaching the transfer instructions, hence creating the swap.
 - o Inductively, there exists a URM program P^1 s.t. $f_{p_1}^k = f_p^k$ for P_1 and P_2 , concluding $f_{p_1}^k = f_p^k$ for P_1 having $h + 1$ swaps recursively

Exercise

Consider URM^{--} without jump (where the apex indicates “minus minus”). How does C^{--} compare with C ? (is $C^{--} = C$?). [To note: it's difficult, but one can start characterizing the shape of the functions in C^{--}]

My take on the proof

Let us prove $C^{--} \subseteq C$. As said from the hint, we can characterize the shape of functions inside of it. We first observe that C^{--} is strictly contained in C , since there are total computable functions in C that cannot be computed by a URM^{--} machine due to the lack of jumps.

If we try to think logically, we have zero, transfer, successor as the available functions. This means this function is strictly linear and can only perform execution as a fixed sequence of inputs, potentially up until a constant number of operations. This says they always terminate, so we can have:

$$f(x) = c$$

Or also (having c as the constant we were discussing above, which will be inside N):

$$f(x) = x + c$$

This can be proven by induction, but operating with something that makes the computation possible. In this case, it should be something with a number, just to prove x can come out of it. So, recursively it must recreate the shape of $x + c$. We will use a register r_1 describing the execution of a given number of steps, say s , so (s, x) is equal to $x + c$.

- $h = 0$, we have $r_1(0, x) = x$, fine because with $c = 0$ the base case is trivial, having already $f(x) = c$ or $f(x) + c$ which will turn it 0 as $f(x)$ alone
- $h \rightarrow h + 1$, so in this case the only thing this can do is the other three functions:
 - $Z(n)$, concluding trivially because the next step, $r_1(s + 1, x) = r_1(h, x)$ having $x = 0$ fr $n = 1$ and we conclude we're inside and this is hence respected. When $n = 1$, infact, the operation resets to 0 and the function will keep its form
 - $S(n)$, so $r_1(s + 1, x)$ will allow us to get the sum of the instruction, given $r_1(s + 1, x) = r_1(s + x)$, again concluding by inductive hypothesis. This way the function will have its expanded form $f(x) = x + c$, because we continuously sum
 - $T(m, n)$; this is uncertain because the function depends on two values this time around, m, n ; when they are different from each other, the result way be unknown (one can be 1, the other we can't know for sure, making the underlying function assume shapes unknown.)
 - When $n > 1$ (or n, m equal) we will know $r_1(s + 1, x) = r_1(s, x)$ will do exactly the transfer of x steps; otherwise, if $m > 1$, we won't know what happen for sure, it can jump many instructions
 - The proof goes well if we assume we have exactly s steps, so the function for c or even $c + x$ can go exactly linearly assuming we will execute exactly *only that* number of steps. This happens because we will keep inside the function

$$f(x) = x + c$$

Let's give the official solution to the previous exercises:

- URM^S , where we replace the transfer instruction ($T(m, n)$) with the swap one ($T^S(m, n)$).

Proof

We want to prove the two sets are equal.

- (Case $C \subseteq C^S$)

Given $f \in C, f: N^k \rightarrow N$, so $f \in C^S$.

If $f \in C$ then there is a program P URM program s.t. $f_p^k = f$. We know that there is P' URM program without T instructions s.t. $f_{p'}^k = f_p^k$. But P' is also a URM^S -machine program.

In this case, so $f_{p'}^k = f_p^k = f \in C^S$.

- (Case $C^S \subseteq C$)

Take $f: N^k \rightarrow N, f \in C^S$ and let P a URM^S program s.t. $f = f_p^k$. We want to "transform" P into a URM program P' s.t. $f_{p'}^k = f_p^k$.

So, the instruction $T^S(m, n)$ can be encoded in a new subroutine, using i which is something new and not used by the program. So:

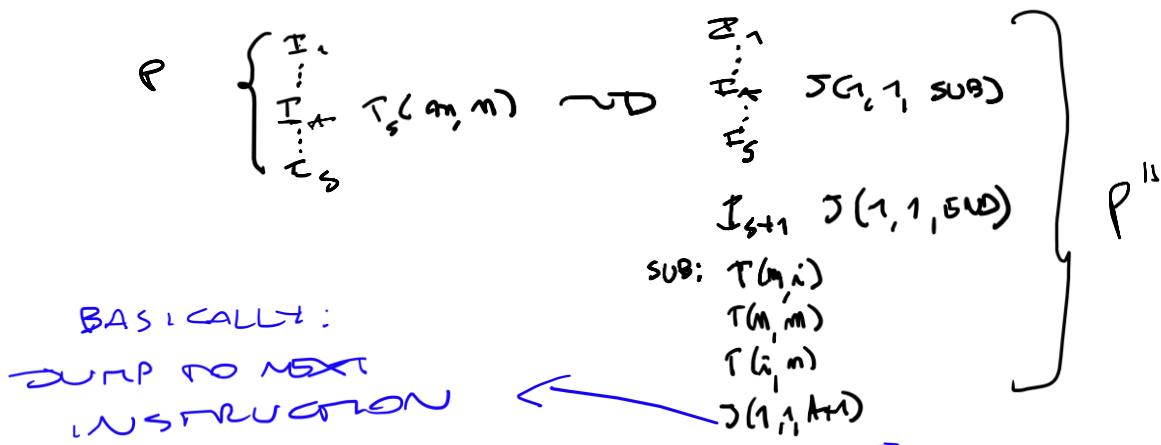
$$T_s(m, n) \rightsquigarrow \begin{array}{l} T(m, i) \\ T(n, m) \\ T(i, m) \end{array} \quad R_i \text{ not used in } P$$

$T^S(m, n)$ is replaced with:

1. $T(m, i)$: This moves the value at location m to a new, unused location i .
2. $T(n, m)$: This swaps the value at location n with the value at location m , performing the swap.
3. $T(i, m)$: Finally, it restores the original value at location m to the new location i .

A URM^S program P can be transformed into a URM-program P' s.t. $f_p^k = f_{p'}^k$. We proceed by induction on h , which is the number of T_s instructions in P .

- ($h = 0$) $\rightarrow P$ is already a URM program, take $P' = P$
- ($h \rightarrow h + 1$) \rightarrow Let P has $(h + 1)$ T_s instructions. The program can be seen as:



To complete the proof, we need:

- P always terminates (if it does) at time $s + 1$
- $i = \max(\{n \mid R_n \text{ is used in } P\} \cup \{k\}) + 1$
 - o This equation calculates the maximum of two sets: the set of registers used in program P and the set k . The purpose of this is to ensure that the value of i is chosen to be greater than any register used in the program P .

Then, $f_p'^k = f_p^k$ and P'' has $h T_s$ instructions (hence, they compute the same instruction). Hence, by inductive hypothesis, there is a URM program $P' s.t. f_p'^k = f_p''^k$. Thus, $f = f_p^k = f_p''^k = f_p'^k$, i.e. $f \in C$

The proof is wrong: we're using the inductive hypothesis on P'' which is not a URM^s -program (it contains both T and T_s). You can make it work by proving a stronger assertion, specifically:

"Every program P which uses all instructions, including T and T_s can be transformed in a URM -program $P' s.t. f_p^k = f_p'^k$." This way, using all we already know up until now, we can conclude the proof solidly (so, if it works for all values in induction, we can safely conclude).

- Consider URM^{--} without jump instructions. $C^{--} \not\subseteq C$

Proof

A URM^{--} program has this structure, and we know it terminates after $l(p) = \# \text{instructions in } P$ steps:

$$P \left\{ \begin{array}{c} r_1 \\ \vdots \\ r_s \end{array} \right. \quad l(p) = \text{length of program } P \\ p \text{ terminates after } l(p) \text{ steps}$$

All functions in C^{--} are total (defined for all possible input values from its domain), so $C^{--} \not\subseteq C$, e.g. $f: N \rightarrow N, f(x) \uparrow \forall x \in N$ (meaning it diverges for all values, because "program without jump always terminate")

$$f \in C \quad J(1,1,1)$$

$$f \notin C^{--} \text{ because it is not total}$$

(not sufficient to say "it uses jump" $J(1,1,2)$ computes $f(x) = x \in C^{--}$; we're basically saying this does not hold inside C^{--} because not in all cases can terminate if there's a jump it doesn't terminate and diverges).

Let's restrict the program executing to unary functions (which take one argument or input); since there is no jump and it was the only way to alter the control independently from the input, we will always do the same thing

$$\boxed{x} \quad \boxed{\oplus} \quad \dots \quad \text{restrict to unary functions}$$

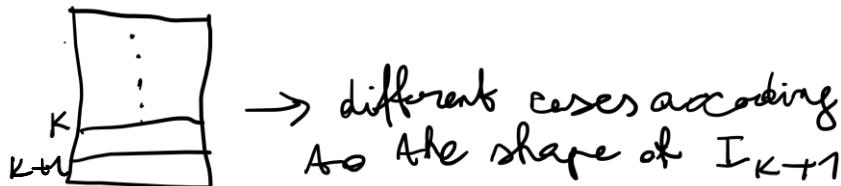
The shape of the functions will be either $f(x) = c$ or $f(x) = x + c$, for a c suitable constant.

Denote $r_1(x, k) = \text{content of } R_1 \text{ after } k - \text{step of computation starting from } \boxed{x}$

We prove by induction on k that $r_1(x, k) = c$ or $r_1(x, k) = x + c$

- $(k = 0) \rightarrow r_1(x, 0) = x = x + 0$, so $c = 0 \rightarrow \text{OK}$
- $(k \rightarrow k + 1) \rightarrow$ By inductive hypothesis $r_1(x + k) = c$ or $r_1(x, k) = x + c$ for $c \in \mathbb{N}$

In this case we will have only three possibilities (given the fact that we can't jump):



As said, three cases, so:

1) $I_{k+1} = Z(n)$, then we have two subcases (ind. hypoth.):

- $n = 1, r_1(x, k + 1) = 0$ (base case)
- $n > 1, r_1(x, k + 1) = r_1(x, k) \rightarrow \text{OK}$, by inductive hypothesis, we have zeroed correctly

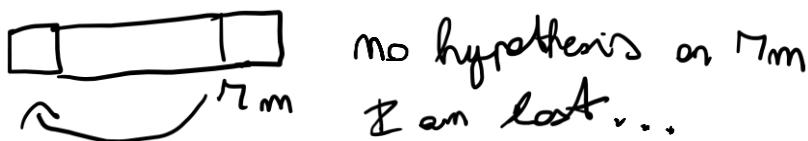
2) $I_{k+1} = S(n)$, again two subcases:

- $n = 1, r_1(x, k + 1) = r_1(x, k) + 1 \rightarrow \text{OK}$, by inductive hypothesis (successor zero and all good)
- $n > 1, r_1(x, k + 1) = r_1(x, k) \rightarrow \text{OK}$, by inductive hypothesis (proceeding inductively works)

3) $I_{k+1} = T(m, n)$, again two subcases:

- $n > 1 \text{ or } m = 1, r_1(x, k + 1) = r_1(x, k) \rightarrow \text{OK}$ by inductive hypothesis
- $n = 1 \text{ and } m > 1$

In this subcase we're lost, because transfer instructions can cause issues when trying to maintain a specific structure for *unary functions*, particularly when the transfer instructions lead to values that cannot be effectively controlled within the defined structure of unary functions (in other case, as seen inductively, we know which instruction comes next, here we don't know it for sure).



So, how do we proceed?

Idea 1: $T(m, n)$ is "useless". Ok, but this observation requires the jump to make it work.

The key observation is that the same property holds for all registers:

$r_1(x, k) = \text{content of } R_n \text{ after } k \text{ steps of computation, starting from } \boxed{x \quad 0 \quad 0} \dots$

Show by induction on k that for all k

$$r_1(x, k) = \begin{cases} c \\ x + c \end{cases} \text{ for "c" suitable constants}$$

The proof goes smoothly in this case ([exercise here](#), so we get one input plus the constant for the specific function). For h -ary functions:

$$f(x_1, \dots, x_n) = \begin{cases} c & 1 \leq j \leq h \\ x_j + c & \text{otherwise} \end{cases}, c \text{ constant}$$

Solution (made by me, to take with a grain of salt):

Given all these values, let's try to solve this inductively in cases as seen until now. We have the function here, which will be used to compute all values and r_j will store the intermediate computation value. We will express such function, for $f(x_1, \dots, x_n)$ with a new function g that computes $h + 1$ steps as:

$$f(x_1, \dots, x_n) = g(r_1(x_k), r_2(x_k), \dots, r_n(x_k))$$

where this function operates on its arguments. Let's show this inductively:

- $k = 0 \rightarrow r_j(x_j, 0) = 0 + c = c$ or $r_j(x_j, 0) = x_j + c$
- $k \rightarrow k + 1$, assuming that for step k , $r_j(x, k) = c$ or $r_j(x, k) = x + c$, we will now show how this assumption extends to step $k+1$. $r_j(x, k) = c$ or $r_j(x, k) = x + c$, we will assume $r_j(x, k + 1) = c$ or $r_j(x, k + 1) = x + c$
 - o We introduce the function g as $f(x_1, \dots, x_n) = g(r_1(x_k), r_2(x_k), \dots, r_n(x_k))$ to represent the inductive computation. We will consider all the subcases as before, given the instruction I_{k+1} for g and for a suitable constant c :
 - $Z(n)$
 - $n = 1, r_j(x, k + 1) = 0, g(r_1(x)) \rightarrow OK$
 - $n > 1, r_j(x, k + 1) = r_j(x, k), g(r_1(x, k)) \rightarrow OK$
 - $S(n)$
 - $n = 1, r_j(x, k + 1) = r_j(x, k) + 1, g(r_1(x, k + 1)) \rightarrow OK$
 - $n > 1, r_j(x, k + 1) = r_j(x, k), g(r_1(x, k)) \rightarrow OK$
 - $T(m, n)$
 - $n > 1$ or $m = 1, r_j(x, k + 1) = r_j(x + k)$
 - $n = 1$ and $m > 1$, here it will hold for each instruction before given the property can be seen as transfer of data so $r_j(h + 1, x)$ is given by $g(r_1(x_k), r_2(x_k), \dots, r_{j-1}(x_k))$ given it's defined linearly for all the function before

After examining the inductive step, we have shown that the properties we assumed for $r_j(x, k)$, namely $r_j(x, k) = c$ or $r_j(x, k) = x + c$, extend to step $k + 1$. This extension has been demonstrated through the function g , which operates on the intermediate values $r_1(x_k), r_2(x_k), \dots, r_n(x_k)$ to represent the inductive computation for $f(x_1, \dots, x_n)$.

In summary, we have successfully established that for all steps k , the properties for $r_j(x, k)$ hold, and by extension, the computed function $f(x_1, \dots, x_n)$ is of the desired form:

$$f(x_1, \dots, x_n) = c \text{ or } f(x_1, \dots, x_n) = x_j + c$$

This completes the inductive proof, confirming that the functions adhere to the specified structure.

6 DECIDABLE PREDICATES

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

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

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

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

$$\text{div}: \mathbb{N} \times \mathbb{N} \rightarrow \{\text{true}, \text{false}\}$$

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

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.

The k -ary predicate on \mathbb{N} indicates the property $Q(x_1, \dots, x_k)$ can be true or false for a set of values, formally describing:

- a function $Q : \mathbb{N}^k \rightarrow \{\text{true}, \text{false}\}$ (note that we represent $0 = \text{true}$, $1 = \text{false}$ as values)
- a set $Q \subseteq \mathbb{N}^k$

We write $Q(x_1, \dots, x_k)$ to denote $(x_1, \dots, x_k) \in Q$ or $Q(x_1, \dots, x_k) = \text{true}$. This means Q will be computable if there exists a k -tuple (x_1, \dots, x_k) returning *true* if $Q(x_1, \dots, x_k)$, *false* otherwise.

Then, given $Q(x_1, \dots, x_k) \subseteq \mathbb{N}^k$. We say it's decidable if the *characteristic function* (also called *indicator function*, used to represent a specific property or set membership in a binary way) is like:

$$x_Q: \mathbb{N}^k \rightarrow \mathbb{N}$$

$$x_Q(x_1, \dots, x_k) = \begin{cases} 1 & \text{if } Q(x_1, \dots, x_k) \text{ is URM-computable} \\ 0 & \text{otherwise} \end{cases}$$

Remember also x_Q is a total function (again, defined for all possible input values from its domain and decidability of predicates involves only total functions in the process).

Let's give some examples of decidable predicates:

1) Equality

$$Q(x_1, x_2) \subseteq \mathbb{N}^2$$

$$Q(x_1, x_2) = x_1 = x_2 \text{ decidable}$$

$$x_Q: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$x_Q(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 = x_2 \\ 0 & \text{otherwise} \end{cases}$$

$\Rightarrow (1, 2, \text{true})$
 FALSE: $\Rightarrow (1, 1, \text{false})$
 TRUE: $\Rightarrow (3, 3)$
 RCS: $\Rightarrow (2, 1)$



This function essentially encodes the result of applying the predicate $Q(x_1, x_2)$ to a pair of natural numbers. It returns 1 if the numbers are equal (satisfying the predicate Q) and 0 if they are not.

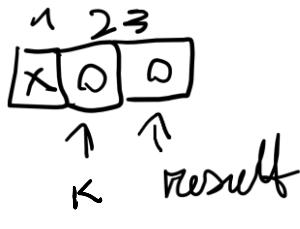
Now, let's see how this program works to compute $x_Q(x_1, x_2)$:

- If $x_1 = x_2$, the program executes the jump in instruction 1, which sends it to instruction 3. It then increments register 3, making it 1, and transfers this value to register 1 (x_1).
- If $x_1 \neq x_2$, the program keeps looping at instruction 2 (the self-jump) without changing the value of register 3. Therefore, register 1 (x_1) remains 0.

So, after the execution of this program, register 1 will contain either 1 (if $x_1 = x_2$) or 0 (if $x_1 \neq x_2$), which corresponds to the value of $x_Q(x_1, x_2)$.

2) Parity of a number

$$Q(x) = "x \text{ is even}" \text{ decidable}$$



EVEN: $J(1, 2, YES)$
 $S(2)$
ODD: $J(1, 2, NO)$
 $S(2)$
 $J(1, 1, EVEN)$
YES: $S(3)$
NO: $T(3, 1)$

The program essentially starts with k at 0 and checks whether x is equal to k . If x is equal to k , it means that x is even, so it increments the result r . If x is not equal to k , the program increments k and repeats the process. This continues until x is equal to k , at which point r is set to 1, indicating that x is even.

In memory, there is the following situation:

x	k	r
---	---	---

 in memory where k is a growing index and r is the result.

The program employs a simple iterative approach to determine if a given number x is even, and it does so by incrementing k until it matches x . If the program exits with r equal to 1, it means that x is even. This program effectively computes the characteristic function for the predicate " x is even," making it a decidable problem.

Let's make a digression, using computability not only confined to a specific model, but resorting to the notion of effective encoding (used to map elements from one set [the domain] to elements in another set, typically inside natural numbers) in a way that is algorithmically or effectively computable.

6.1 COMPUTABILITY ON OTHER DOMAINS

This allows us to extend the concept to other domains, defining then the computability on other domains. Consider we're interested in computability of a domain D of objects, which is countable (so one-to-one correspondence with natural numbers), and:

$$\alpha: D \rightarrow \mathbb{N}, \text{bijective "effective"}$$

Let's specify each notation verbally:

- *bijective* means "establishing a one-to-one correspondence (bijective mapping) between elements in the domain and the set of natural numbers".
- *effective* means "the process of encoding an element from the domain to a natural number should be algorithmically computable"
- there exists an *inverse function*, which should map natural numbers back to elements in the domain *effectively* (so α and α^{-1} are effective)
- once an effective encoding is established ($\alpha: D \rightarrow \mathbb{N}$), it can be employed to define computability on the domain D . This means that functions and predicates over D can be represented using natural numbers through the encoding.

Consider for example the strings domain Σ , where its size is smaller than real numbers set and it's countable or other sets like A^ω (infinite sequences of elements from a given set, also called *streams*), Q, Z .

Let's define a *computable function on a generic domain*; given $f: D \rightarrow D$ function we say is computable if:

$$f^* = \alpha \circ f \circ \alpha^{-1}$$

$$\begin{array}{ccc} D & \xrightarrow{f} & D \\ \alpha^{-1} \uparrow & & \downarrow \alpha \\ \mathbb{N} & \xrightarrow{f^*} & \mathbb{N} \end{array}$$

is URM-computable (the symbol \circ means the composition of functions)

In words: if f is defined, if α and its inverse are effective, f^* is computable (you can see the mapping).

Practically, if we act on domains unrelated to that of natural numbers and want to check whether such a function is computable, it will be sufficient if its respective encoded function is.

Let's see this more concretely, shall we? Suppose we want to pose *computability on the integer numbers* (over Z). We need an encoding $\alpha: Z \rightarrow \mathbb{N}$, given the following encoding:

$$\alpha(z) = \begin{cases} 2z & z \geq 0 \\ -2z - 1 & z < 0 \end{cases}$$

which is an effective function with inverse:

$$\alpha^{-1}(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ \frac{-(n+1)}{2}, & n \text{ is odd} \end{cases}$$

Consider then the absolute value function:

$$f(z) = |z|$$

Is this one computable? In this encoding, it is.

$$\begin{aligned}
 f^*(n) &= (\alpha \circ f \circ \alpha^{-1})(n) \\
 &= \begin{cases} (\alpha \circ f)\left(\frac{n}{2}\right) & n \text{ even} \\ (\alpha \circ f)\left(-\frac{n+1}{2}\right) & \text{otherwise} \end{cases} \\
 &= \begin{cases} \alpha\left(\frac{n}{2}\right) & n \text{ even} \\ \alpha\left(\frac{n+1}{2}\right) & \text{otherwise} \end{cases} \\
 &= \begin{cases} n & n \text{ even} \\ n+1 & \text{otherwise} \end{cases}
 \end{aligned}$$

In the final part where $f(n)$ is expressed for even and odd cases, it shows how the composition of functions and the encoding function α results in a computable function (for all cases). Specifically:

1. If n is even: In this case, $f(n)$ is computed as $\alpha(|n|)$, which simplifies to $\alpha(n)$. This means that when n is even, the absolute value of n is the same as n itself, and the composition function f is equal to $\alpha(n)$.
2. If n is odd: In this case, $f(n)$ is computed as $\alpha(|n|)$, which simplifies to ' $\alpha(n+1)$ '. When n is odd, the absolute value of n is $n+1$ because the negative of an odd integer is one more than its absolute value. Therefore, the composition function f is equal to $\alpha(n+1)$ when n is odd.

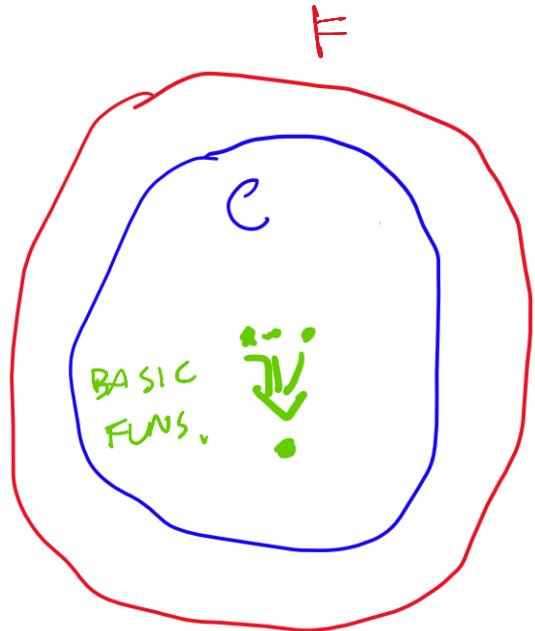
The expressions show how $f(n)$ behaves for even and odd values of n in terms of the encoding function α . The goal of these expressions is to demonstrate that f is URM-computable for all cases, making the function $f(z) = |z|$ computable on the integers Z by encoding and decoding integers using α and its inverse α^{-1} .

7 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} .

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 or use the closure theorems of C choosing the operations carefully (the ones listed above).

The basic functions following are URM-computable:

1) Constant zero

$$Z: \mathbb{N}^k \rightarrow \mathbb{N}, Z(\vec{x}) = 0, (x_1, \dots, x_k) \mapsto 0, \forall \vec{x} \in \mathbb{N}^k$$

2) Successor

$$S: \mathbb{N} \rightarrow \mathbb{N}, S(x) = x + 1, \forall x \in \mathbb{N}$$

3) Projection

$$U_j^k, U_j^k(\vec{x}) = x_j, (x_1, \dots, x_k) \mapsto x_i, \forall x \in \mathbb{N}^k$$

They are in C as they are computed respectively by:

1) z computed by $Z(1)$

2) s computed by $S(1)$

3) U_i^k computed by $T(i, 1)$

Consider also, as a side note, identity is a special projection, basically over all natural numbers.

To prove the closure properties we will need to “combine” programs so we need some notation that we will give now. Given a *URM* program P , we define:

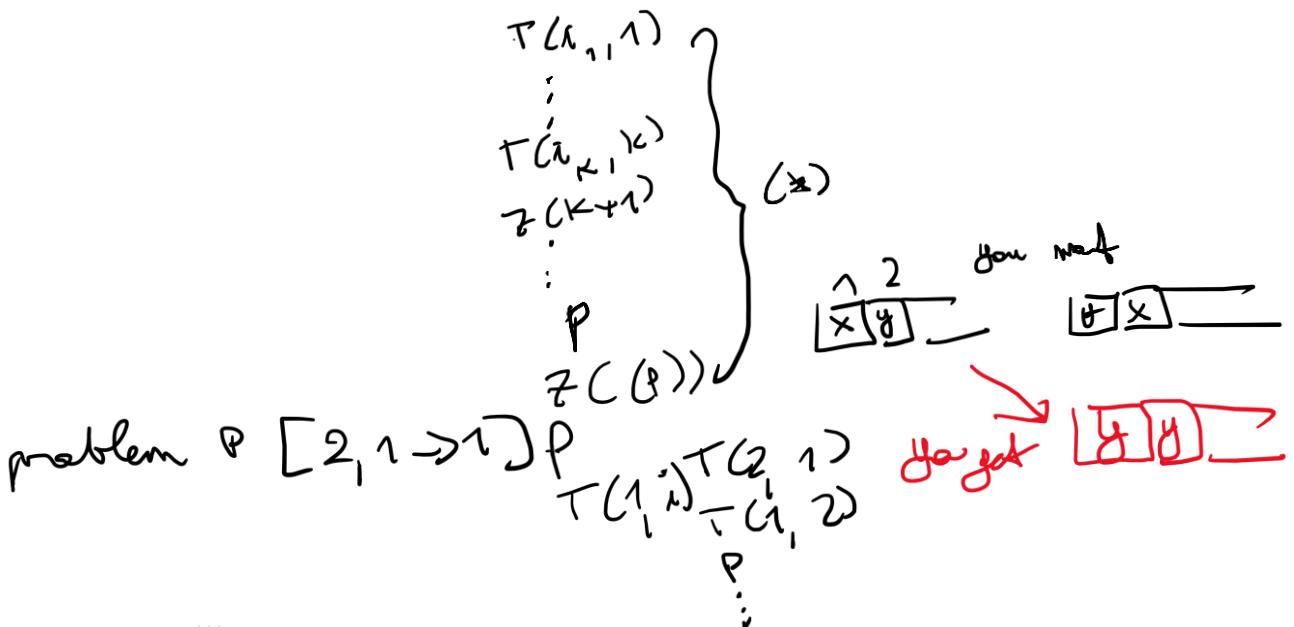
- $\rho(P) = \max\{n \mid \text{register } R_n \text{ is referred in } P\}$ [aka *largest register index*]
- $l(P) = \text{length of } P$ [aka *number of instructions* in P]
- P if in *standard form* if, whenever it terminates, it does so at an instruction $l(P) + 1$
 - o for each $J(m, n, t)$ instruction, $t \leq l(P) + 1$ (stopping at instruction $l(P) + 1$ as just said)

We can define concatenation of programs: given P, Q programs (from now on we assume they are standard), we combine programs in such a way that is computable.

So, starting from P, Q , their concatenation is obtained by considering P followed by the instructions of Q and updating instructions properly:

$$\begin{array}{ccc} P & \xrightarrow{\quad} & P \\ Q & & Q' \leftarrow \text{update } J(m, m^A) \\ & & \text{with } J(n, m, t + l(p)) \\ & & (\Rightarrow) \end{array}$$

- Given P a program we write $P[i_1, \dots, i_k \rightarrow i]$ program taking the input from R_{i_1}, \dots, R_{i_k} and outputs in R_i without assuming registers different from the input are set to 0. We do this by using transfer and reset operations, executing up until ρ , so last instruction given the whole register space.
 - o More precisely, it we express $P[i_1, \dots, i_k \rightarrow i]$ as follows:



Exercise: Write (*) properly in this case

Solution: We write the program as $T(2,1), Z(2), P, T(1,1)$, so we just transfer a value from the output register, do a reset operation and transfer the value back again inside the original register, considering the problem structure.

7.1 GENERALIZED COMPOSITION

We define the composition, given $f: \mathbb{N}^k \rightarrow \mathbb{N}, g_1, \dots, g_k: \mathbb{N}^n \rightarrow \mathbb{N}$ you define $h: \mathbb{N}^n \rightarrow \mathbb{N}$ 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)

For example, consider:

$$z(x) = 0, \forall x$$

$$\emptyset(x) \uparrow \forall x \quad z(\emptyset(x)) \uparrow \forall x$$

(In words: we use the empty set function, so the emptying on all values is defined for all x and subdomains)

Another example:

$$U_1^2(x, y) = x \quad U_1^2(x, \emptyset(y)) \uparrow \forall x, y$$

(In words: composition holds for both values inside functions and if value is not zero, it will output the first, otherwise it just diverges)

Now, we argue \mathcal{C} is closed under *generalized composition*.

Proof

Given $f: \mathbb{N}^k \rightarrow \mathbb{N}, g_1, \dots, g_k: \mathbb{N}^n \rightarrow \mathbb{N}$ in \mathcal{C} and consider $h: \mathbb{N} \rightarrow \mathbb{N}, h(\vec{x}) = f(g_1(\vec{x}), \dots, g_k(\vec{x}))$ is in \mathcal{C} .

Let F, G_1, \dots, G_k be programs (in standard form) for f, g_1, \dots, g_k . The program for h can be:

1	...	m	$m + 1$...	$m + n$	$m + n + 1$...	$m + n + k$
...		x_1	...	x_n	$g_1(\vec{x})$...		$g_k(\vec{x})$

It is important to note that the registers from $m + 1$ onwards can be used freely without the risk of interferences. Let us consider here the largest possible register, so $m = \{\max(\rho(F), \rho(G_1), \dots, \rho(G_k), k, n)\}$.

The program h for composition then is:

$$\begin{aligned} & T(1, m + 1) \\ & \dots \\ & T(n, m + n) \\ & G_1[m + 1, \dots, m + n \rightarrow m + n + 1] \\ & \dots \\ & G_k[m + 1, \dots, m + n \rightarrow m + n + k] \\ & F[m + n + 1, \dots, m + n + k \rightarrow 1] \end{aligned}$$

In words, the composition program provides inputs using additional registers and auxiliary input, then applying transfer operation on each following register and finally computing the result.

Because the following registers can compute the result, if it was defined before, the property continues to hold.

We then conclude that $h \in \mathcal{C}$

Let's give another example: $f(x_1, x_n) = x_1 + x_2$ known to be in \mathcal{C} . We define $g: \mathbb{N}^2 \rightarrow \mathbb{N}$ with $g(x_1, x_2, x_3) = x_1 + x_2 + x_3 = f(f(x_1, x_2), x_3)$. Are we really doing generalized composition? Yes, but we can use projection.

We define such projection on $f(f(x_1, x_2), x_3)$ as $U_1^3(\vec{x}), \mathbb{N}^3 \rightarrow \mathbb{N}$ (we use projection on x_1) and then x_2 as $U_2^3(\vec{x}), \mathbb{N}^3 \rightarrow \mathbb{N}$ (again, projection on x_2). This way, we will have a function of three arguments correctly using generalized composition, having finally:

$$g(x_1, x_2, x_3) = f(f(x_1, x_2), x_3) = f(f(U_1^3(\vec{x}), U_2^3(\vec{x})), U_3^3(\vec{x})), \text{ that is computable.}$$

Basically, in drawing form:

$$\begin{aligned}
 & f(f(x_1, x_2), x_3) \xrightarrow{x} (x_1, x_2, x_3) \\
 & f(f(U_1^3(x)), U_2^3(x), U_3^3(x)) \\
 & N^3 \rightarrow N \quad N^3 \rightarrow N \quad N^3 \rightarrow N \\
 & f : N^2 \rightarrow N \\
 & \underbrace{\qquad\qquad}_{N^3 \rightarrow N} \qquad \underbrace{\qquad\qquad}_{N^3 \rightarrow N} \\
 & \qquad \qquad \qquad \underbrace{\qquad\qquad\qquad}_{N^3 \rightarrow N}
 \end{aligned}$$

Another example: let $F: \mathbb{N} \rightarrow \mathbb{N}$ computable, $Q_f(x, y) = "f(x) = y"$ decidable?

$\chi_{Q_f}(x, y) = \{1 \text{ if } f(x) = y, 0 \text{ otherwise}\}$ computable?

$$\chi_{Q_f}(x, y) = \begin{cases} 1, & \text{if } f(x) = y \\ 0, & \text{otherwise} \end{cases}$$

We know that $\chi_{Eq}: \mathbb{N}^2 \rightarrow \mathbb{N}$, $\chi_{Eq}(x, y) = \begin{cases} 1, & \text{if } f(x) = y \\ 0, & \text{otherwise} \end{cases}$ is computable. We then obtain the computable result via composition (χ_{Eq} means equality and we know it's computable).

Therefore $\chi_Q(x, y) = \chi_{Eq}(f(x), y) = \chi_{Eq}(f(U_1^2(x, y)), U_2^2(x, y))$, and thus χ_Q is computable.

There is a big problem: we're not considering the case of F undefined (we assumed F was total, but that seems not to be the case, because the function is partial, so we map *some* values). To have it correct, change the definition of the predicate putting: "let $F: \mathbb{N} \rightarrow \mathbb{N}$ computable and total", thus it will work.

7.2 PRIMITIVE RECURSION

Recursion is a familiar concept to us computer scientists: it allows to define a function specifying its values in terms of other values of the same function (while other functions are possibly already defined).

Two classic examples of those:

(1) the *factorial* (the product of all positive integers less than or equal to a given positive integer)

$$\begin{cases} 0! = 1 \\ (n+1)! = n! \cdot (n+1) \end{cases}$$

(2) *Fibonacci* (a sequence in which each number is the sum of the two preceding ones)

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n+2) = f(n) + f(n+1) \end{cases}$$

In our case we define a very basic and “controlled” version of recursion (also from domains, you can see they are recursively defined). Let’s give a proper definition then.

Given $f: \mathbb{N}^k \rightarrow \mathbb{N}$ and $g: \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ functions, define $h: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ by primitive recursion as follows:

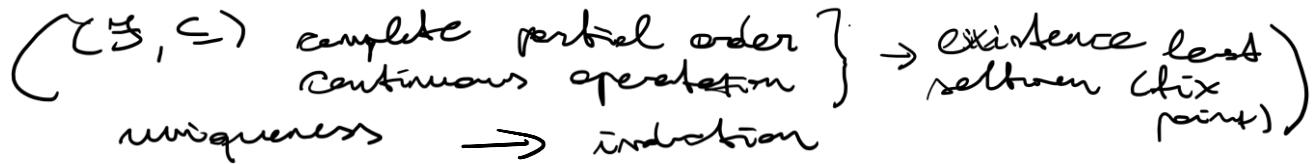
$$\begin{cases} h(\vec{x}, 0) = f(\vec{x}) \\ h(\vec{x}, y + 1) = g(\vec{x}, y, h(\vec{x}, y)) \end{cases}$$

So, the definition of h combines f and g to compute its value for different inputs. It starts with a base case where h equals f for input 0 and then uses the recursive case to compute h for other values by using both g and the previously computed values of h .

The function h exists or that it is unique, and circularity of the definition is avoided by thinking of the values for $h(x, y)$. Unless f, g are total, then h may not be total.

We define a set of functions over the natural numbers, an operator for computing the recursive formula, a function of fixed points (definition of this given [here](#) – will become useful later on) where there is always an upper bound which allows us to do operation in the continuous (so, always inductively defined).

This can be drawn as:



We just say here: there might be problems given the recursive nature of computation if it doesn’t match our requirements. Even more concisely: it’s possible to show this is defined for every natural number “on the same function”, given there are fixed points involved and it’s continuous (always holds).

Let’s give other examples.

Consider the sum function:

$$h: \mathbb{N}^2 \rightarrow \mathbb{N}, h(x, y) = x + y$$

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

$$\begin{aligned} f(x) &= x \\ g(x, y, z) &= z + 1 \end{aligned}$$

Then we define the product function:

$$h': \mathbb{N}^2 \rightarrow \mathbb{N}, h'(x, y) = x * y$$

$$\begin{aligned} x \cdot 0 &= 0 \\ x \cdot (y + 1) &= (x \cdot y) + x \end{aligned}$$

$$\begin{aligned} f(x) &= 0 \\ g(x, y, z) &= z + y \end{aligned}$$

Proposition

C is closed by primitive recursion, so:

- functions obtained from total functions by generalized composition are total
- functions obtained from total functions by primitive recursion are total

Proof

Let: $f: \mathbb{N}^k \rightarrow \mathbb{N}, g: \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ be in C and let F, G programs in standard form for f, g . We want to prove that $h: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ defined through primitive recursion:

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

is computable.

We start from $\boxed{x_1 \ \dots \ x_k \ y \ 0 \ \dots}$

we save the parameters and we start to compute $h(\vec{x}, 0)$ using F .

Basically, we compute and instruction, then save the next and use registers not used to save the next instruction which needs to be computed.

$$\begin{aligned}
 h(\vec{x}, 0) &= f(\vec{x}) \quad (\text{use } F) \\
 h(\vec{x}, 1) &= g(\vec{x}, e, h(\vec{x}, 0)) \quad (\text{use } G) \\
 &\vdots \\
 h(\vec{x}, i) &= g(\vec{x}, i-1, h(\vec{x}, i-1)) \quad (\text{use } G) \\
 i = y ? \quad &\text{if do output} \\
 &\text{no continue with } i+1
 \end{aligned}$$

Essentially, we compute the following instruction until we get to $i = y$.

As usual we need registers not used by F and G , $m = \max\{\rho(F), \rho(G), k + 2\}$ and we build the program for h as follows:

1	...	$m + 1$...	$m + k$	$m + k + 1$...	$m + k + 3$	0
...	\vec{x}	...	i	$h(\vec{x}, 2)$	y	0

$T(1, m + 1)$
 ...
 $T(k, m + k)$
 $T(k + 1, m + k + 3)$
 $F[m + 1, \dots, m + k \rightarrow m + k + 2]$ // compute $h(\vec{x}, 0) \leftarrow h(\vec{x}, 0) = f(\vec{x})$
 $LOOP : J(m + k + 1, m + k + 3, END)$ // $i = y?$
 $G[m + 1, \dots, m + k + 2 \rightarrow m + k + 2]$
 $S(m + k + 1)$ // $i = i + 1$
 $J(1, 1, LOOP)$
 $END : T(m + k + 2, 1)$

$h(\vec{x}, i+1) = g(\vec{x}, i, h(\vec{x}, i))$

In words: we're just making a for loop using closure under primitive recursion (a function that can be computed by a computer program whose loops are all "for" loops) and under composition (so, each previous function can be used to compute the following one).

We simply go ahead until we reach $i = y$; to avoid conflicts, we determine the maximum register m ensuring there's enough space to do so, simply compute continuously given the problem conditions. It's basically like implementing recursion through iteration, given each instruction gets taken and handled iteratively to a possible next register.

7.2.1 Functions defined by primitive recursion

We define a list of computable functions, implementing recursion through composition.

- *sum* $x + y$

$$h: N^2 \rightarrow N, h(x, y) = x + y$$

$$\begin{cases} h(x, 0) = x = f(x) \\ h(x, y + 1) = h(x, y) + 1 = g(h(x, y)) \end{cases} \quad \begin{array}{l} f(x) = x \\ g(x, y, z) = z + 1 \end{array}$$

- *product* $x * y$

$$h': N^2 \rightarrow N, h'(x, y) = x * y$$

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

- *exponential* x^y

$$\begin{array}{lll} x^0 = 1 & h(x, 0) = 1 & f(x) = 1 \\ x^{y+1} = x^y \cdot x & h(x, y + 1) = h(x, y) \cdot x & g(x, y, z) = z \cdot x \end{array}$$

- *predecessor* $y - 1$

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

- *difference* $x \dot{-} y = \begin{cases} x - y & x \geq y \\ 0 & \text{otherwise} \end{cases}$

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

- *sign* $sg(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0 \end{cases}$

$$\begin{cases} sg(0) = 0 \\ sg(1) = 1 \end{cases}$$

$$\begin{array}{lll} sg(0) = 0 & f \equiv 0 \\ sg(1) = 1 & g(y, z) = 1 \end{array}$$

- *negative sign (or complement sign)*

$$\bar{sg}(x) = \begin{cases} 1 & x = 0 \\ 0 & x > 0 \end{cases}$$

exercise $\left(\begin{array}{l} \text{SOLUTION} \\ \bar{sg}(x) = 1 - sg(x) \end{array} \right)$

$$\begin{aligned} sg(0) &= 0 & f \equiv 0 \\ sg(x+1) &= 1 & g(y, z) = 1 \end{aligned}$$

- *minimum*

$$\min(x, y) = x \dot{-} (x \dot{-} y); \quad \begin{array}{c} \overbrace{x}^{\text{---}} \\ x \dot{-} (\overbrace{x-y}^{\text{---}}) \\ x - (\overbrace{x-y}^{\text{---}}) = y \end{array} \quad \begin{array}{l} x \leq y \\ x > y \end{array}$$

- *maximum*

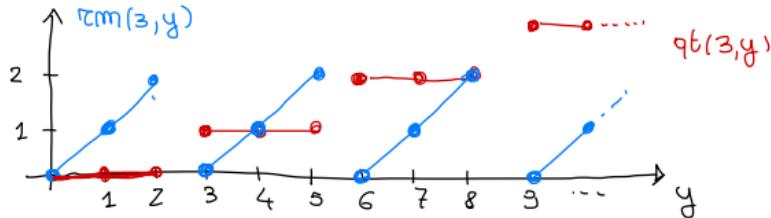
exercise $\left(\begin{array}{l} \text{SOLUTION} \\ \max(x, y) = x + y \dot{-} x \end{array} \right)$

$$\max(x, y) = (x \dot{-} y) + y;$$

- *remainder*, specifically the remainder of the integer division of y by x

$$rm(x, y) = \begin{cases} y \mod x & x \neq 0 \\ y & x = 0 \end{cases}$$

We might see it visually like the following: increment the number and the results changes, incrementing the remainder and the y value accordingly, only if next multiple is not hit.



By far, we use the definition by primitive recursion:

$$\begin{aligned} rm(x, 0) &= 0 \\ rm(x, y+1) &= \begin{cases} rm(x, y) + 1 & rm(x, y) + 1 \neq x \\ 0 & \text{otherwise} \end{cases} \\ &= (rm(x, y) + 1) * \underbrace{sg(x \dot{-} (rm(x, y) + 1))}_{\begin{array}{l} 1 \quad \text{if } rm(x, y) + 1 < x \\ 0 \quad \text{otherwise} \end{array}} \end{aligned}$$

- *quotient*, $qt(x, y) = y \text{ div } x$ (convention $qt(0, y) = y$), we define:

$$\begin{aligned} qt(x, y+1) &= \begin{cases} qt(x, y) + 1 & rm(x, y) + 1 = x \\ qt(x, y) & \text{otherwise} \end{cases} \\ &= qt(x, y) + sg((x \dot{-} 1) \dot{-} rm(x, y)) \quad = \quad qt(x, y) + \overline{sg}(rm(x, y+1)) \end{aligned}$$

Let's give a *definition by cases*; let $f_1, \dots, f_n: \mathbb{N}^k \rightarrow \mathbb{N}$ total and computable and $Q_1(\vec{x}), \dots, Q_n(\vec{x}) \subseteq \mathbb{N}^k$ decidable (mutually exclusive between each other) predicates $\forall x \exists! j Q_j(\vec{x})$ (so, for each, exactly one of Q_1, \dots, Q_n holds) and let $f: \mathbb{N}^k \rightarrow \mathbb{N}$ total computable where:

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & Q_1(\vec{x}) \\ f_2(\vec{x}) & Q_2(\vec{x}) \\ \dots \\ f_n(\vec{x}) & Q_n(\vec{x}) \end{cases}$$

then f is computable and total.

Proofcomputable

$$f(\vec{x}) = f_1(\vec{x}) * x_{Q_1}(\vec{x}) + f_2(\vec{x}) * x_{Q_2}(\vec{x}) + \dots + f_n(\vec{x}) * x_{Q_n}(\vec{x})$$

comp.

So essentially, the right function will be selected and effectively computed, given all the marked functions are computable (sum and product) and composition is itself computable. We then conclude f is computable.

Still, there is a mistake one can do: not assuming the functions are total; this way, the proof will never be correct. Let's show a counterexample:

$$n = 2 \quad f_1(x) = x \quad \forall x \text{ computable} \quad Q_1(x) = \text{true } \forall x$$

$$f_2(x) \uparrow \quad \forall x \quad Q_2(x) = \text{false } \forall x$$

$$f(x) = \{f_1(x) \text{ if } \underline{\underline{Q_1(x)}}, f_2(x) \text{ if } \underline{\underline{Q_2(x)}}\} = f_1(x) = x \quad \forall x$$

$$\begin{array}{c} \text{true} \quad \text{v} \times \quad \text{false} \\ \neq f_1(x) * x_{Q_1}(x) + f_2(x) * x_{Q_2}(x) \end{array}$$

v
X

The mistake in the given statement is the assumption that $f_2(x)$ is computable for all x , which is not the case in this counterexample (because it's explicitly told it diverges). Therefore, the statement is not valid.

Important: We have a proof only if the component functions are total, otherwise it will be always undefined. *The proof is wrong if we don't assume totality of functions*; keep in mind that *for now*.

Let's define the algebra of decidability. Let $Q_1(\vec{x}), Q_2(\vec{x}) \subseteq \mathbb{N}^k$ be decidable predicates. Then:

- 1) $\neg Q_1(\vec{x})$
- 2) $Q_1(\vec{x}) \wedge Q_2(\vec{x})$
- 3) $Q_1(\vec{x}) \vee Q_2(\vec{x})$

can be considered decidable.

Proof

- 1) $x_{\neg Q_1}(\vec{x}) = \{1 \text{ if } \neg Q_1(\vec{x}), 0 \text{ if } Q_1(\vec{x})\} = sg(x_{Q_1}(\vec{x}))$ (evaluates to 1 if true, 0 if false)
- 2) $x_{Q_1 \wedge Q_2}(\vec{x}) = x_{Q_1}(\vec{x}) * x_{Q_2}(\vec{x})$ (evaluates to 1 if both are true, to 0 if one of them is false)
- 3) $x_{Q_1 \vee Q_2}(\vec{x}) = sg(x_{Q_1}(\vec{x}) + x_{Q_2}(\vec{x}))$ (evaluates to 1 if either of them is true, to 0 if both are false)

7.3 BOUNDED SUM, BOUNDED PRODUCT AND BOUNDED QUANTIFICATION

Let $f(\vec{x}, z), f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ be a total computable function and let's define $h: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$.

Then, we define the bounded sum as follows (composition of primitive recursive functions – total since f is:

$$h(\vec{x}, y) = f(\vec{x}, 0) + f(\vec{x}, 1) + f(\vec{x}, y - 1) = \sum_{z < y} f(\vec{x}, z)$$

$$\begin{cases} \sum_{z < 0} f(\vec{x}, z) = 0, \\ \sum_{z < y+1} f(\vec{x}, z) = \sum_{z < y} f(\vec{x}, z) + f(\vec{x}, z) \end{cases}$$

In simpler terms, it's like adding up the values of the function for all z starting from 0 up to y . It starts with 0 for $h(\vec{x}, 0)$ and then, for each increment of y , adds the value of $f(\vec{x}, y)$ to the previous total.

The bounded product $\prod_{z < y} f(\vec{x}, z)$ is defined as follows:

$$\begin{cases} \prod_{z < 0} f(\vec{x}, z) = 1, \\ \prod_{z < y+1} f(\vec{x}, z) = \prod_{z < y} f(\vec{x}, z) \cdot f(\vec{x}, z) \end{cases}$$

It's like taking the product of the results for all z starting from 0 up to y .

By closure under composition, the bound can be a total computable function. Another consequence concerns the decidability of the bounded quantification of predicates.

Let $Q(\vec{x}, z)$ decidable:

$$\begin{array}{ll} 1) \quad \forall z < y, Q(\vec{x}, z) & \text{decidable} \\ 2) \quad \exists z < y, Q(\vec{x}, z) & \rightarrow \left[\begin{array}{l} \text{SHOW IT} \\ \text{AS EXISTENCE} \end{array} \right] \end{array}$$

Solution

Essentially, use this lemma from notes:

LEMMA 6.27. If $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ is total computable then

$$(1) \quad g(\vec{x}, y) = \sum_{z < y} f(\vec{x}, z)$$

$$(2) \quad h(\vec{x}, y) = \prod_{z < y} f(\vec{x}, z)$$

are total computable.

and consider everything is defined by primitive recursion, so you have:

1)

$$g(\vec{x}, 0) = 0$$

$$g(\vec{x}, y + 1) = g(\vec{x}, y) * f(\vec{x}, y)$$

2)

$$g(\vec{x}, 0) = 0$$

$$g(\vec{x}, y + 1) = g(\vec{x}, y) * f(\vec{x}, y)$$

We can also determine, since bound is computable, bounded quantification is decidable.

LEMMA 6.28. Let $Q \subseteq \mathbb{N}^{k+1}$ be a decidable predicate, then:

$$(1) \quad Q_1(\vec{x}, y) \equiv \forall z < y. Q(\vec{x}, z)$$

$$(2) \quad Q_2(\vec{x}, y) \equiv \exists z < y. Q(\vec{x}, z)$$

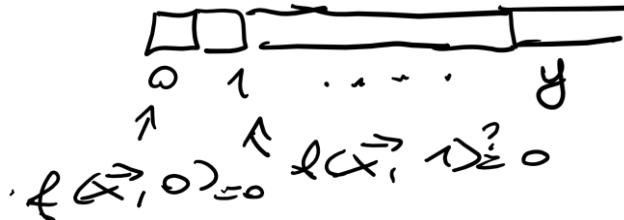
are decidable.

PROOF. (1) observe that $\mathcal{X}_{Q_1}(\vec{x}, y) = \prod_{z < y} \mathcal{X}_Q(\vec{x}, z)$ (2) observe that $\mathcal{X}_{Q_2}(\vec{x}, y) = sg(\sum_{z < y} \mathcal{X}_Q(\vec{x}, z))$

7.4 BOUNDED MINIMALISATION

Let's also define the bounded minimalisation. Given a total function $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$, define a function $h: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ as follows:

$$h(\vec{x}, y) = \begin{cases} z, & \text{minimum } z < y \text{ s.t. } f(\vec{x}, z) = 0 \text{ if it exists} \\ y, & \text{if there is no such } z \end{cases}$$



This is very useful to prove computability of functions; here, we would write $h(\vec{x}, y) = \mu z < y. f(\vec{x}, z) = 0$. As such, this is called *bounded minimalisation operator/bounded μ-operator*.

In simpler terms, h searches for the smallest integer z (less than y) at which the function $f(\vec{x}, z)$ becomes equal to 0. If it finds such a z , it returns that value. If there is no such z , it returns y .

Observation: If f is computable then $h(\vec{x}, y) = \mu z < y. f(\vec{x}, z) = 0$ computable

The observation correctly emphasizes that when the original function f is computable, the process of finding this minimum value z , as described, is also computable (via composition of computable sum/product/sign)

Proof

definition by primitive recursion

$$\left\{ \begin{array}{l} h(\vec{x}, 0) = 0 \\ h(\vec{x}, y+1) = \begin{cases} \text{if } h(\vec{x}, y) < y & \rightsquigarrow h(\vec{x}, y) \\ \text{if } h(\vec{x}, y) = y & \rightsquigarrow \begin{cases} \text{if } f(\vec{x}, y) = 0 & \rightsquigarrow y \\ \text{if } f(\vec{x}, y) \neq 0 & \rightsquigarrow y+1 \end{cases} \end{cases} \end{array} \right.$$

$\downarrow_z \quad y$

$\uparrow \quad h(\vec{x}, y) \quad \swarrow$

$\uparrow \quad f(\vec{x}, y) = 0 \quad \rightsquigarrow y$

$\uparrow \quad \text{if } h(\vec{x}, y) < y \\ 0 \quad \text{otherwise} \quad \swarrow$

$= h(\vec{x}, y) + sg(y - h(\vec{x}, y))$

$(y + sg(f(\vec{x}, y))) \cdot \bar{sg}(y - h(\vec{x}, y))$

computable by primitive recursion

We then prove the above, defined this way, is then defined by primitive recursion. Let's observe the following functions are computable:

1)

$$\text{div}: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$\text{div}(x, y) = \begin{cases} 1, & \text{if } x \text{ divides } y \\ 0, & \text{otherwise} \end{cases} = \overline{\text{sg}}(\text{rm}(x, y))$$

(the division can be also written as the negation of remainder, so it will be 1 if there is no rest, 0 otherwise, as you see here for dividing)

2)

$$y \leq x + 1$$

$$D(x) = \text{number of divisors of } x$$

$$\sum_{y \leq x} \text{div}(y, x) = \sum_{y \leq x} (\overline{\text{sg}}(\text{rm}(y, x)))$$

We can put the non-strict bound there, like $y \leq x$. Formally, we can say that is defined via composition over the previous function, posing $y \leq x + 1$ (given its recursive nature, we compute the current one if the last one was computed already, so if $y \leq x$ holds, it will hold $y \leq x + 1$).

3)

$$\text{Pr}(x) = \{1 \text{ } x \text{ is prime, } 0 \text{ otherwise}\}$$

x is prime iff the only divisors of x are x and 1 with $x \neq 1$

\Updownarrow

x has exactly z divisors

$$\text{Pr}(x) = \overline{\text{sg}}(|D(x) - z|)$$

(So, it calculates the absolute difference between the number of divisors of x and the given value z . This difference measures how far the number of divisors of x is from z , in other terms, we can compute this absolute difference as $|x - y| = (x - y) + (y - x)$).

4)

$$P_x = x^{\text{th}} \text{ prime number}$$

$$p_0 = 0, p_1 = 2, p_3 = 5, p_4 = 7 \dots$$

We use primitive recursion to do this.

$$\begin{cases} p_0 = 0 \\ p_{x+1} = " \mu z \text{ } z \text{ prime and } z > p_x " \end{cases}$$

The goal is to find a number z that is prime and greater than the x^{th} prime (p_x); there, we use quotes there because this is not a formal definition. So, let's define the search for the prime number properly, bounded to the recursive product of all the other factors.

$$\begin{aligned} P_z(z) \wedge z > p_x \\ = \mu z \leq \left(\prod_{i=1}^n P_i \right) + 1 \quad \overline{\text{sg}}(P_z(z) * \text{sg}(z - p_x)) \end{aligned}$$

infact $p_{x+1} \leq \left(\prod_{i=1}^n p_i \right) + 1$

Let p be a prime divisor of $\prod_{i=1}^n p_i$ then $p \neq p_i \forall i = 1 \dots x$,

otherwise if $p = p_j$ for $j \leq x$ THIS "1" (e.g.) means "divides"

then $p \mid \prod_{i=1}^x p_i$

but $p \mid (\prod_{i=1}^x p_i) + 1$

$\rightarrow p \mid 1 \rightarrow p = 1$ not prime

$$\Rightarrow p \geq p_{x+1} \Rightarrow \left(\prod_{i=1}^x p_i \right) + 1 \geq p \geq p_{x+1}$$

In summary, this argument demonstrates that if p is a prime divisor of the product of the first n prime numbers plus 1, it cannot be equal to any of the first x primes. Instead, it must be greater than or equal to the $(x + 1)^{th}$ prime, which is the next one in the sequence. This establishes the relationship between prime divisors and the order of primes in the sequence, recursively because it's bounded.

5)

$(x)_y = \text{exponent of } p_y \text{ in the prime factorisation of } x$

$$\text{E.g. } 20 = 2^2 3^0 5^1$$

$$(20)_1 = \text{exponent of } p_1 = 2 \rightarrow (20)_1 = 2$$

$$(20)_2 = \dots p_2 = 3 \rightarrow (20)_2 = 0$$

$$(20)_3 = 1$$

$$(20)_4 = 0$$

...

$$(x)_y = \max z \text{ s.t. } p_y^z \text{ divides } x$$

$$= \max z \text{ s.t. } \text{div}(p_y^z, x) = 1$$

$$= \min z \text{ s.t. } \text{div}(p_y^{z+1}, x) = 0$$

$$= \mu z \text{ s.t. } \text{div}(p_y^{z+1}, x)$$

When we calculate $(x)_y$, we're essentially finding the exponent to which the prime number p_y is raised in the prime factorization of x . This implies that we're looking for the minimum z such that if we consider p_y raised to $(z + 1)$, it's no longer a divisor of x . Via bounded minimalisation, $(x)_y$ represents the highest exponent to which the prime number p_y can be raised in the prime factorization of x .

Hence, we conclude this is computable by bounded minimalisation.

Also, the following functions are computable:

- $\lfloor \sqrt{x} \rfloor$

$$\begin{aligned}\lfloor \sqrt{x} \rfloor &= \max y \leq x \cdot y^2 \leq x \\ &= \min y \leq x \cdot (y+1)^2 > x \\ &\mu y \leq x \cdot ((x+1) - (y+1)^2)\end{aligned}$$

- $\text{lcm}(x, y)$

$$\begin{aligned}\text{lcm}(x, y) &= \mu z \leq x \cdot y \cdot (x \mid z \wedge y \mid z) \\ &= \mu z \leq x \cdot y \cdot \bar{s}g(\text{div}(x, z) \cdot \text{div}(y, z))\end{aligned}$$

- $\text{GCD}(x, y)$

$\text{GCD}(x, y) \leq \min\{x, y\}$ and it can be characterized using the minimum number that can be subtracted to $\min\{x, y\}$ to obtain the divisor of x, y

$$\begin{aligned}\text{GCD}(x, y) &\leq \min(x, y) - \mu z \\ &\leq \min(x, y) \cdot (1 - \text{div}(\min(x, y) - z, x) \cdot \text{div}(\min(x, y) - z, y))\end{aligned}$$

- number of prime divisors of x

$$\sum_{z \leq x} \text{pr}(z) \cdot \text{div}(z, x)$$

This seems strictly rigid, but let's reason on another example, the *Fibonacci function*.

The Fibonacci function, as conventionally defined with two base cases and a recursive relationship involving the previous two values, is not a strictly primitive recursive function in the traditional sense of primitive recursion within computability theory.

The reason for this lies in the binary nature of the recursive relationship (adding the previous two values), which goes beyond the simple predecessor relationship found in typical primitive recursive functions. Given that $f(y+2)$ is defined in terms of $f(y)$ and $f(y+1)$, it does not completely adhere to the classical primitive recursion schema (because the inductive step requires a prior pair of values).

$$\left\{ \begin{array}{l} f(0) = 1 \\ f(1) = 1 \\ f(n+2) = f(n) + f(n+1) \end{array} \right. \quad \begin{array}{l} \text{not exactly} \\ \text{a primitive recursion} \end{array}$$

We can show that f is computable by resorting to a new function g :

$$g: \mathbb{N} \rightarrow \mathbb{N}^2$$

$$g(n) = (f(n), f(n+1))$$

7.5 ENCODING IN PAIRS

Let's see an encoding in \mathbb{N} of pairs (and n -tuples) of natural numbers that will be useful for some considerations on recursion. Define a pair encoding as, given $D = \mathbb{N}^2$

$$\begin{aligned}\pi : \mathbb{N}^2 &\rightarrow \mathbb{N} \\ \pi(x, y) &= 2^x(2y + 1) - 1\end{aligned}$$

A pair encoding refers to a method of representing ordered pairs (and n -tuples) of natural numbers using a single natural number. The goal is to encode the information in a way that preserves the relationship between the elements of the pair or tuple while allowing for effective (computable) operations on these encodings.

Note π is bijective (uniquely decode for original pair to encoding) and effective (encoding/decoding are computable), so computable. For example, if you have a pair (x, y) , you can use π to encode it as a single natural number $\pi(x, y)$. Later, you can use the inverse operation to decode the original pair from the encoded value.

We also have π^{-1} effective and can be characterized in terms of two computable functions π_1 and π_2 that give the first and second component of a natural number n seen as pair:

$$\begin{aligned}\pi^{-1} : \mathbb{N} &\rightarrow \mathbb{N}^2 \\ \pi^{-1}(n) &= (\pi_1(n), \pi_2(n))\end{aligned}$$

$$\text{where } \pi_1(n) = (n + 1)_1 \text{ and } \pi_2(n) = (\frac{n+1}{2\pi_1(n)} - 1)/2.$$

It can be easily generalized in case for encoding n -tuples, defining the recursive nature over functions previously computed and with projection always obtain a natural number given π as factor.

On this encoding, let's consider the Fibonacci function, which is defined as:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n + 2) = f(n) + f(n + 1) \end{cases}$$

Given the function is not totally defined as the primitive recursion definition, we can show this is computable using the encoding in pairs. We then define:

$$\begin{aligned}g : \mathbb{N} &\rightarrow \mathbb{N} \\ g(n) &= \pi(f(n), f(n + 1))\end{aligned}$$

therefore, by primitive recursion, we can write fib by primitive recursion:

$$\begin{cases} g(0) = \pi(f(0), f(0 + 1)) = \pi(1, 1) = 2^1(2 * 1 + 1) - 1 = 5 \\ g(n + 1) = \pi(f(n + 1), f(n + 2)) = \pi(\pi_2(g(n)), \pi_1(g(n)) + \pi_2(g(n))) \end{cases}$$

and so g is computable by primitive recursion and $f(n) = \pi_1(g(n))$ is defined computable by composition.

More in detail:

- using the primitive recursion principle, $g(0)$ and $g(n + 1)$ are defined based on the pair encoding and the previously computed values
- since $g(n)$ encodes the pair $(f(n), f(n + 1))$, $f(n)$ can be derived as $\pi_1(g(n))$, where π_1 is a projection function that extracts the first component of an encoded pair. This means that $f(n)$ is defined and computable by composition

7.6 UNBOUNDED MINIMALISATION

Generalized composition and primitive recursion produce total functions when starting from total functions. We then introduce unbounded minimalisation (same as the bounded, but here the search is *not bounded*, hence *not necessarily total*). The key point of this one is this: it allows us to obtain non total functions starting from total functions and we use it basically to "search for something".

Given $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ (not necessarily total), specifically as $f(\vec{x}, y)$, define $h: \mathbb{N}^k \rightarrow \mathbb{N}$, $h(\vec{x}) = \mu y. f(\vec{x}, y) =$

$$\begin{aligned}
 &= \text{least } y \text{ s.t. } f(\vec{x}, y) = 0 \\
 &\rightarrow \text{such } y \text{ could not exist} \\
 &\rightarrow f(\vec{x}, z) \text{ could be undefined before finding } y \\
 &\uparrow \quad (\text{undefined})
 \end{aligned}$$

(so, we set the minimum μ as before, but if $f(\vec{x}, z)$ is always $\neq 0$, then $h \uparrow$)

$$= \begin{cases} y, & \text{if there is } y \text{ s.t. } f(\vec{x}, y) \text{ and } \forall z < y, f(\vec{x}, z) \downarrow \neq 0 \\ \uparrow, & \text{if such a } y \text{ does not exist} \end{cases}$$

You can compute $\mu y. f(\vec{x}, y)$, but the result of such minimalisation is undefined.

$$\begin{aligned}
 f(\vec{x}, 0) &= 0 ? \text{ yes} \rightarrow \text{stop out } 0 \\
 &\quad \text{no} \\
 \rightarrow f(\vec{x}, 1) &= 0 ? \text{ yes} \rightarrow \text{stop out } 1 \\
 &\quad \text{no} \\
 \rightarrow f(\vec{x}, 2) &= 0 ? \text{ yes} \rightarrow \text{stop out } 0
 \end{aligned}$$

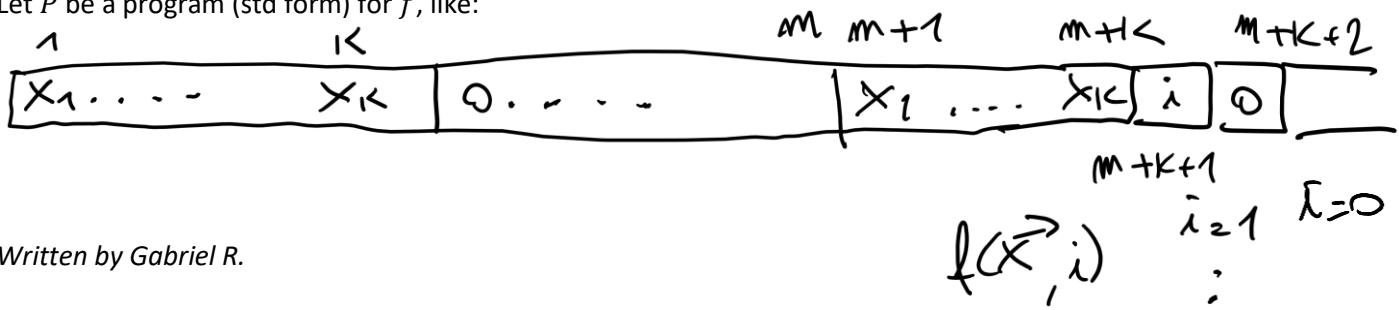
There also can be problems if the function is undefined on a value z' less than z which zeroes the function and h will be undefined also in this case.

Proposition: Class \mathcal{C} is closed under (unbounded) minimalisation

Proof

Let $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ in \mathcal{C} . We want to prove $h \in \mathcal{C}$, $h: \mathbb{N}^k \rightarrow \mathbb{N}$, $h(\vec{x}) = \mu y. f(\vec{x}, y)$

Let P be a program (std form) for f , like:



Pose $m = \max \{\rho(P), k + 1\}$ to get the largest possible register

(not used by the program F). The program for h can be:

```

 $T(1, m + 1) \quad //\text{save input } \vec{x}$ 
...
 $T(k, m + k)$ 
 $\text{LOOP: } P[m + 1, \dots, m + k, m + k + 1 \rightarrow 1] \quad //f(\vec{x}, i) \text{ in } R_1$ 
 $J(1, m + k + 2, \text{END}) \Leftarrow //f(\vec{x}, i) = 0?$ 
 $J(m + k + 1) \quad //i++$ 
 $J(1, 1, \text{LOOP})$ 
 $\text{END: } T(m + k + 1, 1) \quad //\text{output } i$ 

```

In essence, the program for h extends the program for f to perform a minimization operation by iteratively searching for the minimal value of i that makes $f(\vec{x}, i)$ equal to 0.

Observe F may not terminate and this program does not terminate; hence:

- the μ allows us to obtain non-total functions starting from total ones
- unbounded minimalisation is nothing more than a `while` implemented with `goto`.

Example 1 (Perfect square)

$$f: \mathbb{N} \rightarrow \mathbb{N}, f(x, y) = |x - y^2|$$

$$f(x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is a square} \\ \uparrow & \text{otherwise} \end{cases}$$

computable

$$\begin{aligned} f(x) &= \mu y. "y^2 = x" \\ &= \mu y. |y * y - x| \rightarrow \text{computable by minimisation} \end{aligned}$$

Example 2 (Inverse function)

$$g: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$g(x, y) = \begin{cases} \frac{x}{y} & \text{if } y \neq 0 \text{ and } y \text{ divides } x \\ \uparrow & \text{otherwise} \end{cases}$$

We can use only computable functions, like:

$$g(x, y) = \mu z. [z * y - x]$$

$y = 0 \rightarrow 0$
 $x = 0 \rightarrow 0$
 yeah, but you want \uparrow

$$g(x, y) = \mu z . ([z * y - x] + \overline{sg}(y))$$

\uparrow
 \downarrow
↑ if $g = 0$ / ↓ if $y \neq 0$

So, we get:

$$f(x, y) = \mu z . (|yz - x| + \chi_{x=0 \wedge y=0}(x, y))$$

Observation

Every finite (domain) function is computable.

Proof

Let $\theta: \mathbb{N} \rightarrow \mathbb{N}$ a finite domain function (for example, let's define the domain as $\text{dom } (\theta) = \{x_1, \dots, x_n\}$):

$$\theta(x) = \begin{cases} y_1 & x = x_1 \\ \dots & \\ y_n & x = x_n \\ \uparrow & \text{otherwise} \end{cases}$$

then:

$$\theta = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

is computable (note, we use the weighted sum and product, and we map values in pairs).

$$\theta(x) = \sum_{i=1}^n y_i * \overline{sg}(|x - x_i|) + \mu z . \prod_{i=1}^n |x - x_i|$$

— ↗ $(x, ?)$ constant
in z

$\underbrace{\quad}_{\begin{array}{l} \text{if } x = x_i \\ \text{otherwise} \end{array}}$ $\underbrace{\quad}_{\begin{array}{l} \textcircled{1} \text{ if } x \in \text{dom}(\theta) \\ \neq 0 \text{ otherwise} \end{array}}$
 $\underbrace{\quad}_{\begin{array}{l} y_i \text{ if } x = x_i \\ \text{otherwise} \end{array}}$ $\underbrace{\quad}_{\begin{array}{l} \textcircled{0} \text{ if } x \in \text{dom}(\theta) \\ \uparrow \text{ otherwise} \end{array}}$

Example (where both are computable and respect the 0/1/undefined definition for bounded minimalisation given just before):

$$f: N \rightarrow N$$

$$g: N \rightarrow N \text{ fixed a program } P$$

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \text{ and } P \neq NP \\ 1 & \text{if } x = 0 \text{ and } P = NP \\ \uparrow & \text{otherwise} \end{cases}$$

$$g(x) = \begin{cases} 0 & \text{if } x = 0 \text{ and } P(x) \downarrow \\ 1 & \text{if } x = 0 \text{ and } P(x) \uparrow \\ \uparrow & \text{otherwise} \end{cases}$$

Observation

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ computable and injective. Then:

$$f^{-1} = \begin{cases} x & \text{if } x \text{ is s.t. } f(x) = y \\ \uparrow & \text{if there is no } x \text{ d.t. } f(x) = y \end{cases}$$

is computable.

Proof

$$f^{-1}(y) = \mu x. |f(x) - y|$$

The challenge arises because, in some cases, certain input values may not be defined by the function. This problem is consistently related to the concept of a "non-total" function, where not all possible inputs have corresponding outputs. This, in turn, can lead to difficulties in finding inverse values, as there may exist output values for which no valid input exists.

Hence, not working for non-total functions.

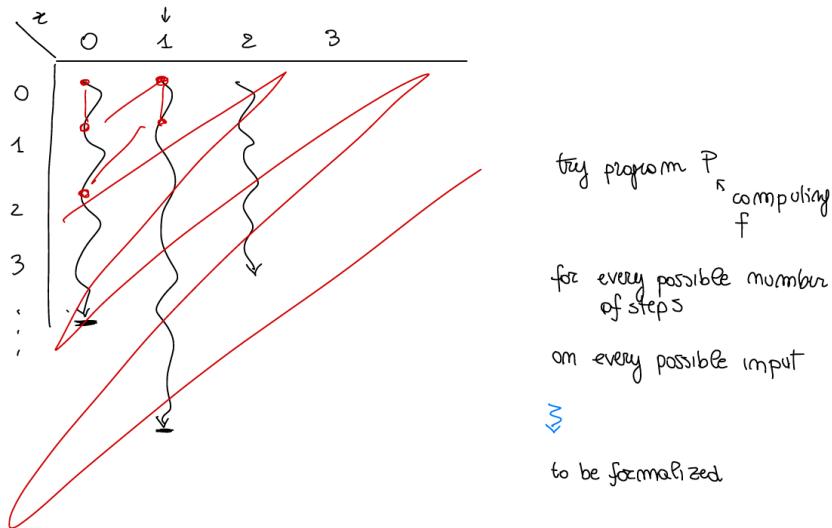
$$f: \mathbb{N} \rightarrow \mathbb{N}$$

$$\begin{aligned} f(x) &= \{x - 1 \quad \text{if } x > 0, \quad \uparrow \quad \text{if } x = 0\} \quad \text{computable} \\ &= (x - 1) + \mu z. \overline{sg}(x) \end{aligned}$$

$$f^{-1}(y) = y + 1 \neq \underbrace{\mu x. |f(x) - y|}_{\text{always undefined}}$$

The key point in this context is that for computable, injective functions, you can define their inverse in a computable way using the minimization operator. Then, it depends when is defined or not.

If f is non-total:



Let's consider the function f and examine its behavior when applied to various inputs.

In words, this is doing in a dove-tail execution pattern (different computations simultaneously):

- 0 steps of the program on argument 0
- 1 step on 0
- 0 steps on 1
- 2 steps on 0

In this graphical representation, each point on the input axis corresponds to a specific input value. We can see that for some inputs, f yields valid output values (red values).

However, there are regions on the graph where there are gaps or undefined points. These gaps represent cases where f is not defined for certain inputs, which makes it a non-total function. This non-totality can pose challenges when attempting to find the inverse function f^{-1} for every possible output (so the red computation continues and goes on and on).

To formalize this process, we can use program P to compute f for every possible number of steps on every possible input, but it's important to be aware of the regions where f may not yield valid results.

Every time the program *terminates* in a certain number of steps k given the argument y , we check the output $f(y)$; if $f(y) = x$ we stop, otherwise we continue.

8 PARTIAL RECURSIVE FUNCTIONS

The *URM* is just one of the various possible computation models to formalize computability. We can alternatively use what we briefly described in the beginning of the course, like:

- Turing Machine
- λ -calculus
- Post system (canonical deduction)

All of these refer to the same class of computable functions, leading to the following thesis (yeah, always this one, for obvious reasons):

Church-Turing Thesis: A function is computable if and only if it is *URM*-computable (so, a *URM*-model)

For our program:

- class \mathcal{R} of partial recursive functions
- prove $\mathcal{R} = \mathcal{C}$ (remember the \mathcal{C} we're talking about was defined [here](#) and we want to prove is equivalent to the *URM*)

We define then the class of partial recursive functions \mathcal{R} , which is the least class w.r.t (with respect to) \mathcal{C} of functions which *contains*:

- (a) zero function
- (b) successor function
- (c) projections

and *closed* under:

1. composition
2. primitive recursion
3. *minimalisation*

We argue this is a well given definition and we will give some remarks in detail.

We can define a rich class of functions if:

1. it contains (a), (b), (c) [so, all basic operations]
2. it is closed w.r.t. (1), (2), (3)

\mathcal{R} is a rich class s.t. for all rich classes \mathcal{A} , we have $\mathcal{R} \subseteq \mathcal{A}$.

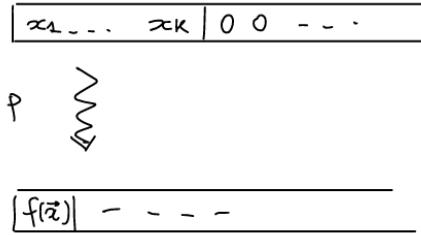
- Note: given $A_i, i \in I$ a family of rich classes then $\bigcap_{i \in I} \mathcal{A}_i$ is rich (so, a rich class is closed under intersection).
- Another note: the class of all functions is rich $\Rightarrow \mathcal{R} = \bigcap_{\mathcal{A} \text{ rich class}} \mathcal{A}$ (it's a way of showing that there is a fundamental set of functions that are rich and that encompasses the richness of other classes)

Equivalently, we note \mathcal{R} is the class of functions (a), (b), (c) which you can obtain from the basic functions using a finite number of times (1), (2), (3).

Theorem: $\mathcal{C} = \mathcal{A}$ (we're now showing that the class of URM-computable functions coincides with the class of partial recursive functions)

Proof

- ($\mathcal{R} \subseteq \mathcal{C}$): just keep in mind \mathcal{R} is a rich class, while \mathcal{C} is the smallest rich class, so this inclusion is trivial and simply $\rightarrow \mathcal{R} \subseteq \mathcal{C}$
- ($\mathcal{C} \subseteq \mathcal{R}$): this is quite more difficult; let $f: \mathbb{N}^k \rightarrow \mathbb{N}, f \in \mathcal{C}$ be a computable function, so there is a URM-program for f , call it P for instance, such that $f_p^k = f$. We want to show $f \in \mathcal{R}$, so we reach with program after all input a function computing all its things (the vector one):



$$\left\{ \begin{array}{l} c_p^1: \mathbb{N}^{k+1} \rightarrow \mathbb{N} \\ c_p^1(\vec{x}, t) = \text{content of } R_1 \text{ after } t \text{ steps of computation of } P(\vec{x}) \end{array} \right.$$

(Above we consider c_p^t which is the content of R_1 after t steps of $P(\vec{x})$ and will give the output of function if the program terminates in less than t steps. This function is clearly total).

Now, we're defining the function for the instruction to be executed after t steps of $P(\vec{x})$ (also a total function), which is

$$j_p^1: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$$

$$\left\{ \begin{array}{l} J_p: \mathbb{N}^{k+1} \rightarrow \mathbb{N} \\ J_p(\vec{x}, t) = \begin{cases} \text{instruction to be executed after } t \text{ steps of } P(\vec{x}) \\ 0 \quad \text{if } P(\vec{x}) \text{ terminates in } t \text{ steps or fewer} \end{cases} \end{array} \right.$$

Let $\vec{x} \in \mathbb{N}^k$

\rightarrow if $f(\vec{x})$, then $P(\vec{x}) \downarrow$ in a number of steps

$t_0 = \mu t. j_p(\vec{x}, t)$ (the least number of steps to reach a state where jump is undefined)

hence $f(\vec{x}) = c_p^1(\vec{x}, t_0) = c_p^1(\vec{x}, \mu t. j_p(\vec{x}, t))$ (the value of function is determined by number of steps to have it undefined)

\rightarrow if $f(\vec{x})$, then $P(\vec{x}) \uparrow$ (must be undefined, given the undefined behavior of jump)

hence $\mu t. j_p(\vec{x}, t) \uparrow$

therefore:

$$f(\vec{x}) = c_p^1(\vec{x}, \mu t. j_p(\vec{x}, t)) \uparrow$$

In all cases (in words: the combination of these functions is able to describe a URM program):

$$f(\vec{x}) = c_p^1(\vec{x}, \mu t. j_p(\vec{x}, t))$$

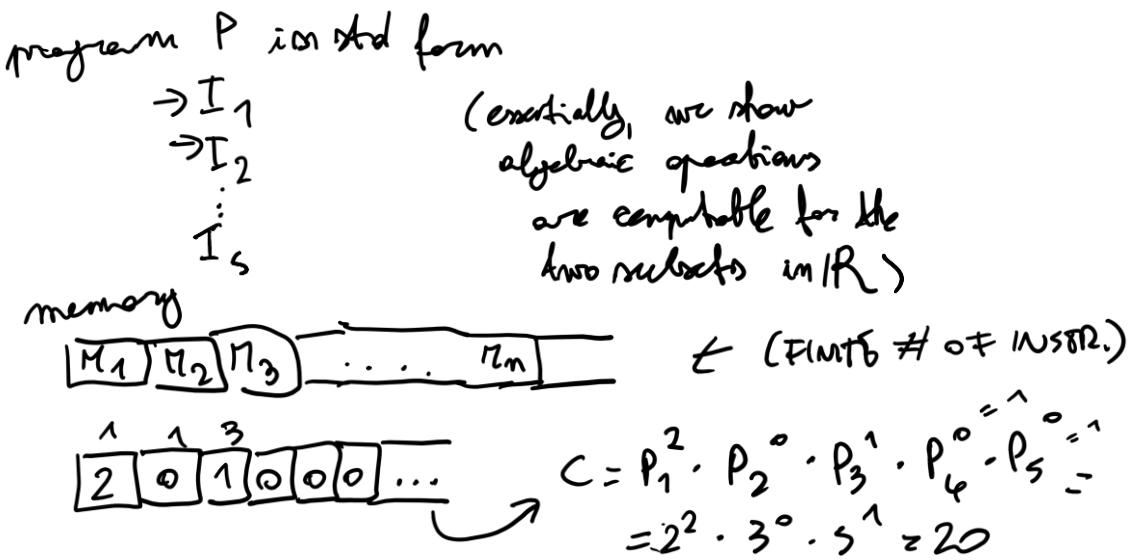
Written by Gabriel R.

If we knew $c_p^1, j_p \in \mathcal{R}$, we would conclude $f \in \mathcal{R}$. To resolve the problem and complete the proof, you need to demonstrate that the functions are indeed partial recursive functions.

Let's then try to c_p^1, j_p are in \mathcal{R} . We take a program P in standard form (composed by a list of instructions).

What we're going to do is working on sequences encoding representing the registers and program counter configurations (respectively, c_p and j_p in general form, so "computation" and "jump" defined by composition and primitive recursion), then manipulate these with functions that are also composite and recursive, obtaining as a result c_p^1 and j_p themselves.

We can see this as:



In the context of this memory encoding, we establish a representation for the configuration of registers. This representation, denoted as c , is calculated as the product of prime numbers, with each prime number raised to the power of the value stored in the corresponding register.

The product spans over all prime numbers, and the allocation position is indicated by the factorization of the prime numbers. In formal terms, this can be expressed as:

$$c = \prod_{i \geq 1} P_i^{r_i} = \prod_{i=1}^n P_i^{r_i}$$

with P_i representing the prime numbers and r_i representing the values stored inside registers.

Consider c_p and j_p with the recursive definition given before:

$$\left\{ \begin{array}{l} c_p^1: N^{k+1} \rightarrow N \\ c_p^1(\vec{x}, t) = \text{content of } R_1 \text{ after } t \text{ steps of computation of } P(\vec{x}) \end{array} \right.$$

$$\left\{ \begin{array}{l} J_P: \mathbb{N}^{k+1} \rightarrow \mathbb{N} \\ J_P(\vec{x}, t) = \begin{cases} \text{instruction to be executed after } t \text{ steps of } P(\vec{x}) \\ 0 \quad \text{if } P(\vec{x}) \text{ terminates in } t \text{ steps or fewer} \end{cases} \end{array} \right.$$

We define c_p, j_p by primitive recursion as follows (first, the base cases):

$$c_p(\vec{x}, 0) = \prod_{i=1}^k P_i^{x_i}$$



$$j_p(\vec{x}, 0) = 1$$

The recursion cases follow. We define:

$$c_p(\vec{x}, t + 1)$$

$$j_p(\vec{x}, t + 1)$$

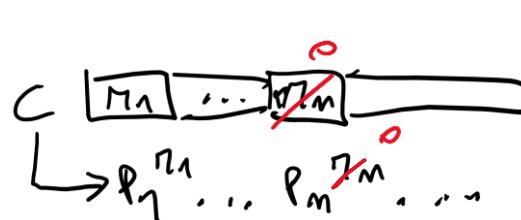
using a simplified notation:

$$c_p(\vec{x}, t) = c$$

$$j_p(\vec{x}, t) = j$$

↑ NOTATION

$$c_P(\vec{x}, t + 1) = \begin{cases} qt(p_n^{(c)_n}, c) & \text{if } 1 \leq j \leq l(P) \text{ & } I_j = Z(n) \\ p_n \cdot c & \text{if } 1 \leq j \leq l(P) \text{ & } I_j = S(n) \\ qt(p_n^{(c)_n}, c) \cdot p_n^{(c)_m} & \text{if } 1 \leq j \leq l(P) \text{ & } I_j = T(m, n) \\ c & \text{otherwise} \end{cases}$$



$(c_j = 0 \text{ or } 1 \leq j \leq l(P))$
and $I_j = J(m, n, k)$

In words, to explain properly everything that was defined above for each case in the same order:

- In case 1 the quotient of the number to be reset to zero is done so as to reset the exponent relative to the register in the numeric representation of the memory.
- In case 2 the multiplication of the prime number associated with the register is done to increase the exponent of the prime number associated with the register in the numeric representation of memory by 1.
- The case otherwise occurs when the program terminates or the jump instruction points outside the program, in either case the program memory does not change

We the define the transfer of said program:

$$j_P(\vec{x}, t + 1) = \begin{cases} j + 1 & \text{if } 1 \leq j < l(P) \text{ & } I_j = Z(n), S(n), T(m, n) \\ & \text{or } J(m, n, t) \text{ with } (c)_m \neq (c)_n \\ u & \text{if } 1 \leq j \leq l(P) \text{ & } I_j = J(m, n, u) \\ & \text{& } (c)_m = (c)_n \text{ & if } 1 \leq u \leq l(P) \\ 0 & \text{otherwise} \end{cases}$$

- In case 1, if the instruction is not jump, or it is a jump but with condition false, the instruction $t + 1$ is the next one.
 - o If the jump has condition is true and the instruction to jump to is internal to the program, the next instruction is q .
 - o Otherwise if the program is terminated or the jump instruction is finished outside the program 0 is returned.
- The two functions are then defined using a "per case" definition and combining previously defined functions with basic operations (recursive primitives), so these two functions are in \mathcal{PR} and therefore are also in \mathcal{R} .

Hence $j_p, c_p \in R$ and thus $f(\vec{x}) = (c_p(\vec{x}, \mu t. j_p(\vec{x}, t))_1$ and therefore $f \in \mathcal{R}$.

In this context, relying solely on unbounded minimalization and composition, we can confidently assert that the instruction pointer and program counter consistently yield valid outcomes. This assurance guarantees that the instruction pointer performs computations within defined boundaries, with a particular emphasis on the input for the first component (indicated by the subscript '1'), which is the register computation guaranteeing the successor instruction will be defined.

This way, the program will compute its code effectively, without going out of bounds.

9 PRIMITIVE RECURSIVE FUNCTIONS

We define the class of primitive recursive functions \mathcal{PR} as the smallest class of functions which:

- Includes the basic functions
 - o Zero function
 - o Successor function
 - o Projections
- Is closed under
 - o Composition
 - o Primitive recursion
 - o Minimalisation (not defined in all cases, because this is unbounded e.g. while loops)

Intuitively, \mathcal{PR} corresponds to *bounded* iterations, e.g. for loops. This way, we use a model to formalize *structured program* replacing jumps with for/while loops. This model can correspond to a class called $\mathcal{C}_{\text{for}, \text{while}}$, which coincides with $\mathcal{C} = \mathcal{R}$.

We also know that $\mathcal{PR} \not\subseteq \mathcal{R}$, because \mathcal{C}_{for} coincides with \mathcal{PR} only with the for construct, this way we can include all cases if well-defined.

- \mathcal{PR} does not contain all computable functions, because \mathcal{PR} has always total functions inside, obtained by composition and primitive recursion. We study this class to understand the expressive power of for/while loops.

One can still suppose \mathcal{PR} includes all total recursive functions, defining a set containing them all as Tot , so $\mathcal{PR} = \mathcal{R} \cap \text{Tot}$ (as was theorized by Hilbert). This is *false* (hence $\mathcal{PR} \not\subseteq \mathcal{R} \cap \text{Tot}$).

- The key reason for this is the while construct. Primitive recursive functions, as defined within the class \mathcal{PR} , are inherently based on bounded iterations, such as for loops
- Total functions always terminate, and while loops may not

Essentially, the while construct allows for both total/ending computations but also unbounded, even when considering total functions, potentially opening to unbounded results. As a consequence:

- only minimalization can be used to define non-total functions
- the \mathcal{PR} class is not able to define non-total functions and cannot define all total functions

9.1 THE ACKERMANN FUNCTION

A function which witnesses the inclusion $\mathcal{PR} \not\subseteq \mathcal{R} \cap \text{Tot}$ is the Ackermann function (the Greek letter below is "Psi" and an animation of the function [here](#) and my general summary on its point [here](#))

The function is an example of a total recursive function that is not primitive recursive (so, not primitive recursive but computable, only *not using the set of primitive recursive operations*, as it has more cases). Its definition involves unbounded recursion, which is not guaranteed to terminate.

The Ackermann's function is $\psi : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined as

$$\begin{cases} \psi(0, y) = y + 1 \\ \psi(x + 1, 0) = \psi(x, 1) \\ \psi(x + 1, y + 1) = \psi(x, \psi(x + 1, y)) \end{cases}$$

$$\begin{aligned} (x+1, 0) &\xrightarrow{\text{ex}} (x, 1) \\ (x+1, y+1) &\xrightarrow{\text{ex}} (x, y+1) \\ (x+1, y+1) &\xrightarrow{\text{ex}} (x, y+1) \end{aligned}$$

The function takes as argument two non-negative integers, so x and y . Specifically:

- Base case: the result is simply one more than the second argument y , which resembles the behavior of a successor function.
- In another base case where the second argument y is 0, the function behaves as follows: $\psi(x, 0) = \psi(x - 1, 1)$. Here, when y is 0, the function recursively calls itself with x decreased by 1 and y set to 1. So, first argument simply diminishes.
- In the most complex case, when both x and y are greater than 0, the Ackermann function proceeds with deep nested recursion. It evaluates as $\psi(x, y) = \psi(x - 1, \psi(p, y - 1))$. This case involves two levels of recursion. Initially, it calculates $\psi(x, y - 1)$, effectively decreasing the second argument y . Then, it calculates $\psi(x - 1, \psi(x, y - 1))$, resulting in a nested recursive structure.
 - o So, in this case, we have a case in which the first argument gets smaller, another when the second argument gets smaller

In other terms, the arguments of the function diminish in a *lexicographical order* on N^2 (two-dimensional, because it depends on both x and y) inside sequences of numbers: $(N^2, \leq_{lex}), (x, y) \leq_{lex} (x', y') \text{ if } x < x' \text{ or } (x = x') \text{ and } (y \leq y')$. We can show (N^2, \leq_{lex}) does not allow for infinite descending sequences.

This means that given two pairs (x, y) and (x', y') , the lexicographical order \leq_{lex} dictates that (x, y) is considered less than or equal to (x', y') if either x is less than x' or, in the case of equal x values, y is less than or equal to y' .

$$(1000, 1000000) <_{lex} (1001, 0)$$

$$(1000, 1000000) >_{lex} (1001, 0)$$

Here, when you compare $(1000, 1000000)$ and $(1001, 0)$:

- The lexicographical order first compares the first elements: 1000 and 1001. Here, 1000 is less than 1001, so $(1000, 1000000) <_{lex} (1001, 0)$ because the first element x in the first pair is smaller than the first element x in the second pair.
- If the first elements were equal, the lexicographical order would then compare the second elements y . In this case, the second elements are 1000000 and 0. Because the first elements x have already determined the order ($1000 < 1001$), there's no need to further compare the second elements y . In lexicographical order, if the first elements are different, the comparison stops at that point.

Concretely: the function grows enormously, but the argument diminish according to an order.

An example:

$$f: \mathbb{Z} \rightarrow \mathbb{Z}$$

$$f(z) = \begin{cases} 0, & z \geq 0 \\ f(z - 1), & z < 0 \end{cases}$$

Simply, we say that this has a finite recursion $\rightarrow f(-1) \rightarrow f(-2) \rightarrow f(-3) \dots$

In the example of $f(z)$, when the input is initially negative ($z < 0$), the function repeatedly reduces the value of z by 1. This process continues until z reaches a non-negative value.

Ackermann function has a logically sound recursion and to show so, we use need more notions.

9.2 PARTIALLY ORDERED/WELL FOUNDED POSETS

Definition (partially ordered set)

We define then a partially ordered set (abbreviated as "poset", elements defined in an increasing order) as:

$$\begin{aligned} (D, \leq) & \quad \leq \text{reflexive} \quad x \leq x \\ & \quad \leq \text{antisymmetric} \quad x \leq y \text{ and } y \leq x \Rightarrow x = y \\ & \quad \leq \text{transitive} \quad x \leq y \text{ and } y \leq z \Rightarrow x \leq z \end{aligned}$$

So basically, we analyze a binary relation in which elements are ordered (reflexive), there are no circular relations between elements because elements are different (antisymmetric) and elements have a predictable order (transitive).

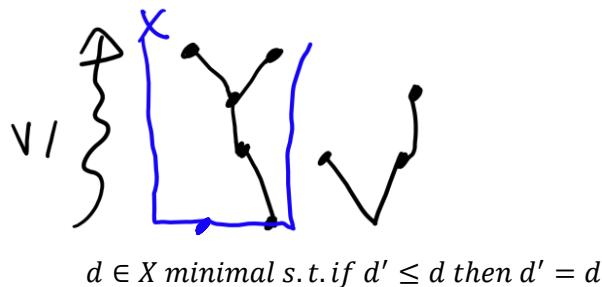
In a partially ordered set, some pairs of elements may be related, while others may not be related at all.

Definition (Well-founded posets)

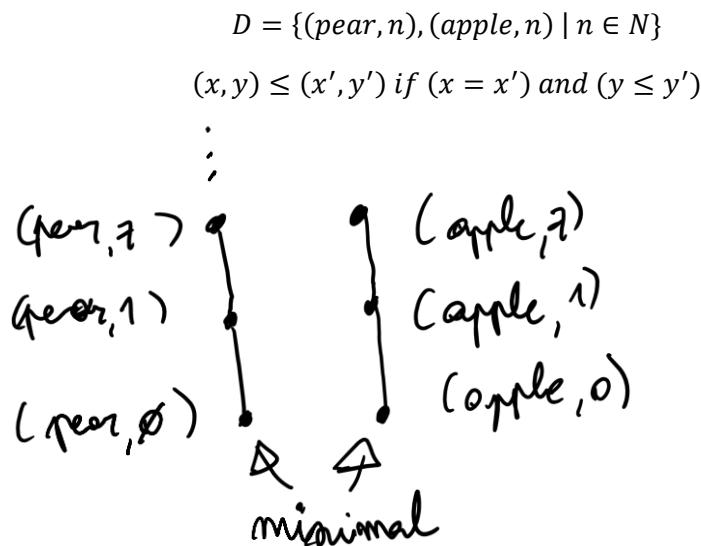
$$(D, \leq) \text{ is well-founded if } \forall x \subseteq D, x \neq 0, \text{ has a minimal element}$$

In other words, within any non-empty subset, you can always find an element that is minimal with respect to the given partial order (so, we have no infinite descending chains) – this is what happens in Ackermann.

- This means there's no other element in X that is strictly smaller than d in terms of the order relation (\leq).
- If the computation is well-founded, there is always a step which leads to a smaller value and eventually the program will terminate, given we will always find a minimal element.



(given the partial order, "we can't mix different things together" aka "you can't mix pears with apples").



In this context, we're dealing with a partially ordered set (poset) that consists of pairs of elements, where each element is associated with a label (such as "pear" or "apple") and a natural number (n).

The partial order on this set is defined such that elements are ordered first by their labels and then, within the same label, by their associated natural numbers. The key relationship here is " \leq ," which denotes the partial order on this set.

- An element (x, y) is considered less than or equal to (x', y') if and only if both the labels are the same ($x = x'$) and the natural number associated with the first element is less than or equal to the natural number associated with the second element ($y \leq y'$).

Here's the explanation for the example in the context of the partial order:

- Suppose you have two elements, (x, y) and (x', y') in D . These elements represent items labeled "pear" and "apple," along with natural numbers, respectively.
- The partial order specifies that you can't mix items of different labels, meaning you can't compare "pears" with "apples" in this order. So, comparing (x, y) and (x', y') only makes sense when x and x' are the same (both "pear" or both "apple").
- Once you've established that the labels match ($x = x'$), you can compare the natural numbers (y and y'). The element (x, y) is considered "minimal" if there is no other element (x', y') in D with the same label ($x = x'$) where y' is less than or equal to y .

We note also:

- Is \mathbb{Z} well-founded? No (we can't always find a minimal element)
- Is \mathbb{N} well-founded? Yes (given it's a well-ordered set, we can always find a minimal element)

Note: (D, \leq) is well-founded if and only if there is no infinite descending chain $d_0 > d_1 > d_2 \dots$ in D . This way our computation descends a decreasing sequence of values, which is necessarily finite.

(In words: This fact can be useful when dealing with termination problems. If we can conclude that the set of configurations is well-founded, we simply need to prove that inductively this is all defined. This works also here with Ackermann: given the computation is based on smaller values, at some point it will end)

Remember from before that $(\mathbb{N}^2, \leq_{lex})$ is well-founded.

Let $x \subseteq \mathbb{N}^2, x \neq 0$:

$$x_0 = \min\{x \mid \exists y. (x, y) \in X\}$$

$$y_0 = \min\{y \mid (x_0, y) \in X\} \rightarrow (x_0, y_0) = \min X$$

Essentially, what we just said simply means there is always a smallest element according to the lexicographical order and there is always a well-defined order.

To explain the further concepts, we need to properly define induction.

Given $P(n), n \in \mathbb{N}, P(0)$ and assuming $P(n)$ you can deduce $P(n + 1)$

↓

$$P(n) \text{ holds } \forall n$$

(essentially, given a case, if it holds for a base case, it will hold for all natural numbers). Let's give a simple reasoning by induction: a *binary tree* formation.

Statement: "A binary tree with height h has at most $2^{h+1} - 1$ nodes"

- Base case ($n = 0$) \rightarrow number of nodes = 1 $\leq 2^{0+1} - 1 = 2 - 1 = 1$
- Recursive case ($n \rightarrow n + 1$)



$n_1, n_2 < n+1$
 the inductive step
 is only on n ...
 you "can't" conclude
 actually, you "could"
 with a clever hypoth
 formulation

9.3 COMPLETE/WELL-FOUNDED INDUCTION AND ACKERMANN PROOF

As shown here, normal induction reaches cases where it can't conclude (basically, the height of a binary tree can vary, because it would involve proving that if statement holds for trees of different height using always the same k ; this is not linear, and the proof would require *bounding* the height to a value and inductively show the thing).

We then need the complete induction, in which this can be applied to any well-founded poset.

Specifically, to prove that $P(n)$ holds $\forall n \in \mathbb{N}$, show

$\boxed{\forall n, \text{assuming } P(n') \forall n' < n \text{ then } P(n)}$

All of this is a well-founded induction. We define here (D, \leq) well-founded order, $P(x)$ property over D , if $\forall d \in D$, assuming $\forall d' < d, P(d')$, we can conclude $P(d)$ "holds everywhere":

↓

$\forall d \in D P(d)$

All this tour leads to a conclusion: the Ackermann function is total and if a property holds for all numbers before, then it will hold for all those after.

Formally:

1) Ψ is total

- $\forall (x, y) \in N^2, \Psi(x, y) \downarrow$ proceed by well-founded induction of (N^2, \leq_{lex})

Proof

Let $(x, y) \in \mathbb{N}^2$, assume $\forall (x', y') <_{lex} (x, y), \Psi(x', y') \downarrow$, we want to show $\Psi(x, y) \downarrow$

We have three cases:

$$1) (x = 0) \rightarrow \Psi(x, y) = \Psi(0, y) = y + 1 \downarrow$$

$$2) (x > 0, y = 0) \rightarrow \Psi(x, 0) = \Psi(x - 1, 1)$$

$\xrightarrow{(x-1) <_{lex} (x, y)}$ hence $\Psi(x - 1, 1) \downarrow$ by inductive hypothesis

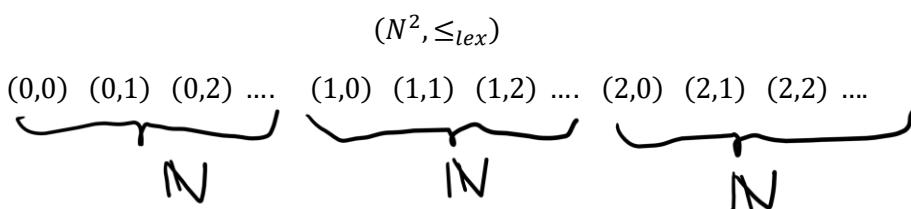
$$3) (x > 0, y > 0) \rightarrow \Psi(x, y) = \Psi(x - 1, y) = \Psi(x - 1, u) \downarrow \text{ (by ind. hyp.)}$$

$\xrightarrow{<_{lex} (x, y)}$ $\rightarrow \Psi(x, y - 1) \downarrow = u$ by inductive hypothesis

Essentially, we prove Ackermann is total given two non-negative numbers which are well-defined in their order. We consider three cases based on the values of x and y :

- *Case 1 ($x = 0$):* In this case, if x is 0, we know that $\Psi(x, y)$ is $\Psi(0, y)$. This leads to a straightforward result, which is $y + 1$. The function $\Psi(0, y)$ is guaranteed to terminate, so this case is covered.
- *Case 2 ($x > 0$ and $y = 0$):* When x is greater than 0 and y is 0, we have $\Psi(x, 0) = \Psi(x - 1, 1)$. We know that $\Psi(x - 1, 1)$ terminates because it's part of our induction hypothesis. This means that $\Psi(x, 0)$ also terminates.
- *Case 3 ($x > 0$ and $y > 0$):* In the most complex case, where both x and y are greater than 0, $\Psi(x, y)$ involves a nested recursion. It's defined as $\Psi(x - 1, \Psi(x, y - 1))$. Our induction hypothesis ensures that $\Psi(x, y - 1)$ terminates (denoted as " u "). Since $(x - 1, u)$ is smaller than (x, y) , and we've assumed that for all smaller pairs, Ψ terminates, we can conclude that $\Psi(x, y)$ also terminates.

If you want to discuss infinite things:



We are highlighting that, even though this set contains an infinite number of pairs, they are ordered in a systematic and predictable way. As you move through this set, the values in each pair follow a pattern, allowing us to compare and order them consistently.

Within this well-ordered set, the Ackermann function operates by moving through these pairs in a specific manner. It doesn't "jump out" of this structured order.

- The function goes through a process of descending values within this well-ordered set
- This way, it can compute values within the natural numbers (\mathbb{N}) without running into infinite or unbounded operations.

One could argue by using the Church-Turing thesis: the computation of $\Psi(x, y)$ is always reduced to the computation of ψ on smaller input values until we reach a base case where the successor is used. The above is unsatisfactory. Given it is always defined, it is total.

2) $\Psi \in \mathbb{R} = \mathcal{C}$

$$\Psi(1,1) = \Psi\left(0, \underbrace{\Psi(1,0)}_{\Psi(0,1)\wedge}\right) = \Psi(0,2) = 3$$

$$\begin{array}{c} 1 \\ | \\ 2 \end{array}$$

$$(1,1,3) \quad (0,2,3) \quad (1,0,2) \quad (0,1,2)$$

In words: we reach sets of values defined by recursion, in this case triples defined inside these sets. Especially, we characterize:

$$(x, y, z) \in N^3 \quad \rightarrow Z = \Psi(x, y)$$

$\rightarrow S$ contains all triples needed to compute $\Psi(x, y)$

A set $S \subseteq N^3$ is considered *valid* if, for all $(x, y, z) \in S$, it satisfies two conditions:

1. z equals $\Psi(x, y)$, ensuring that the results in the set are consistent with the Ackermann function.
2. S contains all the triples needed to compute $\Psi(x, y)$ for different values of x and y .

Formally, you just need to recall the function is defined (Ackermann system of equations definition), so $S \subseteq N^3$ is valid.

- $(0, y, z) \in S \rightarrow z = y + 1$
- $(x + 1, 0, z) \in S \rightarrow (x, 1, z) \in S$
- $(x + 1, y + 1, z) \in S \rightarrow \exists u (x + 1, y, u) \in S \text{ and } (x, u, z) \in S$

You can show:

$$\forall (x, y) \in N^2, z \in N \rightarrow \Psi(x, y) = z \text{ iff } \exists \text{ valid set of triples } S \in N^3 \text{ and } S \text{ finite s.t. } (x, y, z) \in S$$

(essentially, we have $\Psi(x, y) = z$ iff a valid finite set of triples is defined by complete induction, knowing the set is preserved under union)

Then (in words: every triple can be encoded as a set of numbers and then as a number using primes)

$$\Psi(x, y) = " \mu (S, z). (S \in N^3 \text{ finite valid set of triples and } (x, y, z)) "$$

$\overbrace{}^{\text{encode as a number}}$

$$\rightarrow \Psi \in \mathcal{R} = \mathcal{C}$$

(so, there exists the smallest finite number in which a valid set of triples minimizes correctly and gives values recursively enumerable inside the resulting function – aka computable and quantifiable).

3) $y \notin \mathcal{PR}$

This part of the proof wants to show that Ψ is not a primitive recursive function showing it grows faster than every other function in PR . We combine nested primitive recursion to show Ackermann cannot compute a finite number of nested primitive recursions.

By using primitive recursion you can define the sum via the successor:

- $x + y$

$$x + 0 = x$$

$$x + (y + 1) = (x + y) + 1$$

- $x * y$

$$x * 0 = 0$$

$$x + (y + 1) = (x * y) + x$$

- x^y

$$x^0 = 1$$

$$x^{(y+1)} = (x^y) * x$$

*nesting
primitive recursions
for loops*

(so, essentially, basic arithmetic operations under primitive recursion is defined).

Consider x as first parameter:

$$\Psi_x(y) = \Psi(x, y)$$

$$\begin{aligned} \Psi_{x+1}(y) &= \Psi_x(\Psi_{x+1}(y - 1)) = \Psi_x(\Psi_x(\Psi_{x+1}(y - 2))) = \underbrace{\Psi_x \dots \Psi_x}_{y \text{ times}} \Psi_{x+1}(0) \\ &= \underbrace{\Psi_x \dots \Psi_x}_{y \leftarrow 1}(1) = \Psi_x^{y+1}(1) \end{aligned}$$

Roughly: increasing x to $x + 1$ you need iterating Ψ_x , $y + 1$ times \rightarrow additional for loop

So, we need minimalization $\rightarrow \Psi \notin \mathcal{PR}$

(So, number of nested recursions is infinite and with primitive recursion we can't quantify it prior; it's necessary to use the unbounded minimalisation, hence Ackermann is not primitive recursive)

Yeah, that was quite a ride, wasn't it?

- Short explanation in words:

Intuitively, if x grows so does the level of nesting in the functions, which is equivalent to say that we need more nested for loops. Since x can grow to infinity and for loops cannot be nested to infinity, a while loop is needed. The for-loops nesting level won't be able to catch up in a bounded way, so is not in PR.

- Longer explanation in words:

The discussion transitions to $\Psi_{x+1}(y)$, representing the Ackermann function for the next value of x . The key insight here is that when you increase x by 1, you need to iterate $\Psi_x(y)$ a certain number of times, specifically $y + 1$ times. This represents additional loops or iterations in the computation.

The key insight is that you need a form of "minimalization" to determine how many additional iterations are required when x increases by 1, but primitive recursive functions lack this capacity.

For the final part, to account for this additional looping when increasing x , a process of minimalization is introduced. This is because the Ackermann function doesn't fit neatly into the framework of primitive recursion, as it requires also *unbounded iterations*, giving we need additional loop and iteration "to try to reach a finite value".

To properly move ahead with the function, you need to iterate the function multiple times, which goes beyond what primitive recursion can handle. This leads to the conclusion that Ψ is not a primitive recursive function, as it necessitates unbounded iteration and minimalization, making it a more powerful and complex function.

- At the end of the day:

To be able to define all total functions you also need minimalization: otherwise, some functions might be too powerful to express traditionally, like happens here.

Mathematically, we have:

$$\psi \in \mathcal{R} \cap \text{Tot}, \Psi \notin \mathcal{PR}$$

and so:

$$\mathcal{PR} \subsetneq \mathcal{R} \cap \text{Tot}$$

10 ENUMERATING URM PROGRAMS



We are postulating the existence of a *universal program* that can take programs (P) represented as numerical input (vector of inputs, so the usual \vec{x}) and produce computations as output by processing the given instructions (so $P(\vec{x})$). To do that, we will establish the set of all programs is effectively denumerable, having an effective coding of programs by the set of all natural numbers – aka compiler.

The goal is proving the enumeration of URM programs, so we give the following definition:

Definition (Countable set)

A set X is countable if $|X| \leq |\mathbb{N}|$ i.e. there is $f: \mathbb{N} \rightarrow X$ surjective (enumeration/denumerable), in which we can list all elements one after the other with no repetition (so, we can list all URM programs without repetition and without missing any) as:

$$\underbrace{f(0) \ f(1) \ f(2) \dots}_{x}$$

An enumeration is *without repetitions* if it is injective (we map elements from A to B distinctively) and thus bijective (complete mapping).

- If f is also injective, this is called *bijection enumeration*.
- We require this enumeration to be effective (so finite and computable) and this happens if f itself is effective; this doesn't talk about computability, instead we use already computable components arguing about effectiveness

Lemma

There are bijective enumerations of effective functions (bijective support functions)

- 1) $\pi: \mathbb{N}^2 \rightarrow \mathbb{N}$
- 2) $v: \mathbb{N}^3 \rightarrow \mathbb{N}$
- 3) $\tau: \bigcup_{k \geq 1} \mathbb{N}^k \rightarrow \mathbb{N}$

Proof

On each set, we give a function which effectively shows the lemma:

- 1) π

We saw already that the following function:

$$\begin{aligned} \pi: \mathbb{N}^2 &\rightarrow \mathbb{N} \\ \pi(x, y) &= 2^x(2y + 1) - 1 \end{aligned}$$

is computable and we give its inverse:

$$\pi^{-1}: \mathbb{N} \rightarrow \mathbb{N}^2$$

$$\pi^{-1}(n) = (\pi_1(n), \pi_2(n))$$

So, given:

$$\pi_1, \pi_2: \mathbb{N} \rightarrow \mathbb{N}$$

$$\pi_1(n) = (n+1)_1$$

(for your reference, remember the 1 in subscript represents the first component of output)

$$\pi_2(n) = \left(\frac{n+1}{2^{\pi_1(n)}} \right) - 1 = qt(2, qt(2^{\pi_1}, n+1)) - 1$$

2) v

Consider the function:

$$v: \mathbb{N}^3 \rightarrow \mathbb{N}$$

$$v(x, y, z) = \pi(x, \pi(y, z))$$

is computable and its inverse is built upon projections:

$$v^{-1}: \mathbb{N} \rightarrow \mathbb{N}^3$$

$$v^{-1}(n) = (v_1(n), v_2(n), v_3(n))$$

$$v_1(n) = \pi_1(n)$$

$$v_2(n) = \pi_1(\pi_2(n))$$

$$v_3(n) = \pi_2(\pi_2(n))$$

having v_1, v_2, v_3 computable (so, projections leverage multiple successor functions and allow us to map natural numbers and get back triples effectively).

2) τ (this letter is “tau”)

$$\tau: \bigcup_{k \geq 1} \mathbb{N}^k$$

$$\tau(x_1, \dots, x_k) = \prod_{i=1}^k P_i^{x_i} - 1$$

The encoding of tuple is *not injective*, given it probably won't always map distinct elements (because we calculate a product and if there are same values, we might have multiple tuples mapping to the same value and violate injectiveness); the presence of subtraction further elevates the chance of having a collision.

The idea is *incrementing* the last component in the following way:

$$(1, 0) \rightarrow p_1^1, p_2^0 - 1 = 2^1 * 3^0 - 1 = 1$$

$$(1, 0, 0) \rightarrow p_1^1 * p_2^0 * p_3^0 = 2^1 * 3^0 * 5^0 - 1 = 1$$

$$\tau(x_1, \dots, x_k) = (\prod_{i=1}^{k-1} P_i^{x_i}) * P_k^{x_k+1} = 2$$

The function τ is designed to increment the last component of the tuple by 1 and then compute the product of the other components.

We treat in a special way the last component (which is $P_k^{x_k+1}$) to determine the injectivity of the sum, given it will map finite components determined by the following inverse function:

$$\begin{aligned}\tau^{-1}: \mathbb{N} &\rightarrow \bigcup_{k \geq 1} \mathbb{N}^k \\ \tau^{-1}(n) &= \underbrace{a(n, 1) \ a(n, 2) \ \dots \ (n, l(n))}_{l(n)} \\ l(n) \ l: \mathbb{N} &\rightarrow \mathbb{N}\end{aligned}$$

The overall idea is to take a natural number n and determine how many components should be in the tuple (length of decoding), as determined by the function $l(n)$. Once you know the length, you can use the function $a(n, i)$ to compute each component of the tuple.

Now, given we can determine the length of the program, we want to determine the value of the single component (the last one $l(n)$ is computable, it's asked to show it by exercise; this is shown by inserting correctly values via injection):

$$\begin{aligned}n = \tau(\dots) &= \overbrace{\left(\prod_{i=1}^{k-1} P_i^{x_i} \right)}^{k-1} * P_k^{x_k+1} = 2 \\ k &= l(n)\end{aligned}$$

$$\begin{aligned}l(n) &= \text{largest } k \text{ such that } p_k \text{ divides } n + 2 \\ &\quad (\text{computable, "show it as exercise"})\end{aligned}$$

Solution (made by me, best to take with a grain of salt)

To show this is computable, we want to test some k values which will respect the property of $l(n)$.

So, given $n = 2$ and so $l(n) = 2$, we must find k :

- Start with $k = 1$. We check if P_1 (first prime number that we have at our disposal across all the possible ones, so 2) divides $2 + 2$, which is 4 (given $n + 2 = 2 + 2 = 4$). In this case, it happens.
- Continue with $k = 2$. Check if P_2 (second prime number, so 3) divides $(2 + 2)$. This does not happen.

We conclude the length of tuple is $k = 1$.

(Moving on)

We see another definition of $l(n)$ in notes, which uses bounded minimalization to exist.

In words, we can summarize its reasoning like this: observe there is always a smaller h which is defined as bound for x for which the division of the prime number for $(x + 2)$ will either give 1 or 0 (hence, the usual negated sign function).

So, given minimalisation is used, we can formalize it like:

$$x - \mu(h \leq x)$$

Usually, this is accompanied by usage inside the division function, so:

$$\overline{sg}(\text{div}(p_{x-h}, (x + 2)))$$

Finally:

$$l(n) = \max\{k : \text{div}(p_k, (x+2)) = 1\} = x - \mu(h \leq x) \cdot \overline{s\bar{g}}(\text{div}(p_{x-h}, (x+2)))$$

Let's give another function:

$$a(n, i) = \begin{cases} (n+2)_i, & i < l(n) \\ (n+2)_i - 1, & i = l(n) \end{cases}$$

$$l: \mathbb{N} \rightarrow \mathbb{N}, a: \mathbb{N}^2 \rightarrow \mathbb{N} \text{ computable}$$

We can now encode programs or instructions using a mapping that allows us to access program elements by computing the successor function i times, where i is determined by the length of the program. This approach enables effective encoding and decoding of programs using instructions, lists, or other data structures which compute elements continuously.

A more general but alternative encoding is this one (which works with composition on two elements):

- $\tau(x_1, \dots, x_k) = \pi(\prod_{i=1}^k p_i^{a_i}, k)$
- $l(n) = \pi_2(n)$
- $a(n, i) = (\pi_1(n))_i$

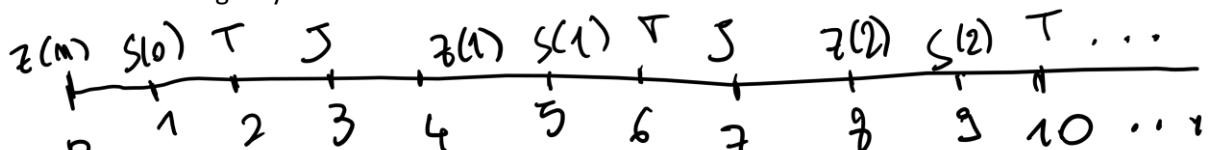
Observation: Let P the set of all URM programs. There is an "effective" enumeration which is bijective:

$$\gamma: P \rightarrow \mathbb{N} \text{ ("to every program assign a number")}$$

(the Greek letter is "gamma"). The previous one also uses $\beta: Istr_{URM} \rightarrow \mathbb{N}$, which "to every instruction assigns a number". Two key observations:

- Since programs are a sequence of instructions, γ is nothing more than the product of the power elevation of the encoding according to the instructions present.
- Since computable programs are enumerable and since computable functions are those that have a program that computes it, then such functions are also enumerable

Let $F = \{Z(n), S(n), T(m, n), J(m, n, t) : m, n, t \geq 1\}$, we consider $B: F \rightarrow \mathbb{N}$. We then put instructions in numbers in the following way:



What we want here is achieving a bijective effective correspondence using enumerations of pairs and triples of only computable functions sending. Multiplication by 4 is used to "make room" in the encoding so as to continue to have a biunivocal function. Final sums also serve the same purpose.

- $Z(n)$ instructions to multiples of 4
- $S(n)$ instructions to numbers congruent 1 mod 4
- $T(m, n)$ instructions to numbers congruent 2 mod 4
- $J(m, n, t)$ instructions to numbers congruent 3 mod 4

$$\left\{ \begin{array}{l} B(Z(n)) = 4 * (n - 1) \\ B(S(n)) = 4 * (n - 1) + 1 \\ B(T(m, n)) = 4 * \pi(m - 1, n - 1) + 2 \\ B(J(m, n, t)) = 4 * v(m - 1, n - 1, t - 1) + 3 \end{array} \right.$$

We can then define the inverse $B^{-1}: \mathbb{N} \rightarrow F$ such that $x \rightarrow r = rm(4, x)$ and $q = qt(4, x)$.

$$\beta^{-1}(x) = \begin{cases} Z(q+1), & \text{if } r = 0 \\ S(q+1), & \text{if } r = 1 \\ T(\pi_1(q)+1, \pi_2(q)+1), & \text{if } r = 2 \\ J(v_1(q)+1, v_2(q)+1, v_3(q)+1), & \text{if } r = 4 \end{cases}$$

And this way both β and β^{-1} are effective. Now $\gamma: P \rightarrow N$ can be defined as follows; if P URM program is composed by a list of instructions, then via composition we can apply β :

$$P \begin{cases} I_1 \\ I_2 \\ \dots \\ I_S \end{cases} \quad \gamma_p = \tau(\beta(I_1)) \dots \beta(I_S))$$

This way, putting the inverse γ^{-1} , we will get the number back from the corresponding program:

$$\text{inverse: } \gamma^{-1}: N \rightarrow P$$

$$\gamma^{-1}(n) = P = \{I_1 \dots I_{l(n)}\} \text{ and } I_1 = \beta^{-1}(a(n, i))$$

This ensures γ is bijective because of composition of bijective functions,

From now on, we have a fixed enumeration of URM programs, thus it's bijective given it comes from program composition and P is denumerable (mapping one-to-one with positive integers of programs).

To define the code of P , given γ the fixed enumeration of URM program, we define the *Gödel number* of P as $\gamma(P)$ (also called *code* of P), we write P_n to represent $\gamma^{-1}(n)$ as the n^{th} program of such enumeration. (formally, to a well-formed formula, it assigns a unique natural number).

Given this one, we can determine a fixed enumeration for programs, making us able to effectively compute their codes and to find the n^{th} program in a sequence.

Now, let's consider this program as an example, encoded by the β function:

$$P = \begin{cases} T(1,2) \rightarrow \beta(T(1,2)) = 4 * \pi(1 - 1, 2 - 1) + 2 = 4 * \pi(0,1) + 2 = 10 \\ S(2) \rightarrow \beta(S(2)) = 4 * (2 - 1) + 1 = 5 \\ T(2,1) \rightarrow \beta(T(2,1)) = 4 * \pi(2 - 1, 1 - 1) + 2 = 4 * \pi(1,0) + 2 = 6 \end{cases}$$

$$2^{(2-1+1)-1}$$

So, given the results here, the program will get these numbers as a list as shown before:

$$\begin{aligned} \gamma(P) &= \tau(10, 5, 6) \\ &= p_1^{10} * p_2^5 * p_3^{6+1} - 2 \\ &= 2^{10} * 3^5 * 5^7 - 2 \\ &= 19\,439\,999\,998 \end{aligned}$$

So, we want a machine taking in input this very big number and the only thing we care is, given the finite nature and effective enumeration, the program able to execute it and retrieve it as output (hence the well-formed reasoning behind).

This program computes because of $\mu x. x + 1$ (which means that using minimalisation, we're able to retrieve, from the last computation, the following numbers, hence composing a specific result. This can be mapped back to the corresponding program, given it's enumerated as "the one able to compute it")

The program P' able to execute the successor function (so $P': S(1)$) computes the same function (and the encoding follows):

$$\gamma(p') = \tau(\beta(S(1))) = \tau(4 * (1 - 1) + 1) = \tau(1) = p_1^{1+1} - 2 = 2^2 - 2 = 2$$

Basically, the numbers 19,439,999,998 and 2 represent two programs that calculate the same successor function. So, given $n = 100$, what is $P_{100} \rightarrow (\gamma^{-1}(100))$? Let's show it.

Remember the encoding for n was:

$$\left(\prod_{i=1}^{k-1} P_i^{x_i} \right) * P_k^{x_k+1} = 2$$

We can observe:

$$\begin{aligned} n + 2 &= 100 + 2 = 2^1 * 3^1 * 17^1 \\ &= p_1^1 * p_2^1 * p_3^1 * p_4^0 * p_5^0 * p_6^0 * p_7^1 \end{aligned}$$

And so the program will contain 7 instructions:

$$\begin{aligned} l(100) &= 7 \\ \beta^{-1}(1) &\rightarrow S(1) \\ \beta^{-1}(1) &\rightarrow S(1) \\ \beta^{-1}(0) &\rightarrow Z(1) \end{aligned}$$

All this program does (number 100) is calculating the constant 1.

Clearly, an enumeration of *URM* programs induces an enumeration of computable functions. So, fixed an effective enumeration $\gamma: P \rightarrow \mathbb{N}$, we define:

- $\phi_n^{(k)}: \mathbb{N}^k \rightarrow \mathbb{N}$ (as the function of k arguments (k -ary function) computed by the program $P_n = \gamma^{(-1)}(n)$ can be seen as $\phi_n^{(k)} = f_{P_n}^{(k)}$)
- $W_n^{(k)} = \text{dom}(\phi_n^{(k)}) = \{\vec{x} \in \mathbb{N}^k \mid \phi_n^{(k)}(\vec{x}) \downarrow\} \subseteq \mathbb{N}^k$
→ (as the domain, representing the set of argument vectors for which $\phi_n^{(k)}$ converges)
- $E_n^{(k)} = \text{cod}(\phi_n^{(k)}) = \{\phi_n^{(k)} \mid \vec{x} \in W_n^{(k)}\} \subseteq \mathbb{N}$
→ as the codomain, representing the set of values that $\phi_n^{(k)}$ can produce when applied to arguments in $W_n^{(k)}$

When $k = 1$, we omit it ϕ_n for $\phi_n^{(1)}$.

(This statement means that, when dealing with unary functions (functions that take only one argument), the subscript indicating the arity (in this case, k) is typically omitted. In other words, for functions of a single argument, you don't need to explicitly specify k because it's understood that k is 1. So, for $k = 1$, the function is commonly denoted as simply ϕ_n)

What this mess above means

- A function has a domain and a codomain or arguments which are defined over the k combinations of those
- Giving the function is surjective, either there are no programs that calculate it or there are infinite ones

Example

$$\phi_{100}: N \rightarrow N$$

$$\phi_{100}(x) = 0 \quad \forall x \in N$$

$$W_{100} = N \quad E_{100} = \{0\}$$

The following is an enumeration of all unary functions:

$\phi_0 \phi_1 \phi_2 \phi_3 \phi_4 \dots \phi_{19433999998}$

enumeration of all unary computable functions

↑ there are repetitions (infinite)

In this list, there are definitely repetitions, which indicate that certain unary computable functions result in the same output, particularly when we compute their successor values.

How many repetitions? Infinitely many (which means there is an unending occurrence of functions that yield the same results).

We're able to express, for every countable set, that the cardinality of every countable set is less than or equal the cardinality of the set of natural number (holds for set with one element but also any finite value of k):

$$|C^{(1)}| \leq |N|$$

$$|C^{(k)}| \leq |N| \quad \forall k$$

So, the union of all countable set C is countable too.

$$\bigcup_{K \geq 1}^n C^{(k)}$$

This expresses the idea that the union of finite or countable sets remains countable, and the cardinality of each set in the union is bounded by the cardinality of the natural numbers.

10.1 EXERCISES

Exercise

Take \mathcal{R} , the class of partial recursive functions, precisely the least rich class which:

- includes basic functions
- is closed under:
 - o composition
 - o primitive recursion
 - o minimalisation

We then consider the originally defined by Gödel-Kleene \mathcal{R}_0 , least rich class which:

- includes basic functions
- is closed under:
 - o composition
 - o primitive recursion
 - o minimalisation used only when result is total

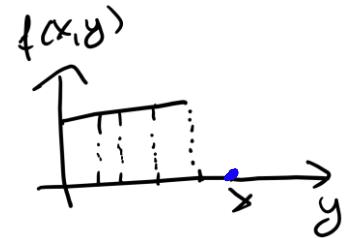
The question for us is trying to compare the original definition of \mathcal{R} with this one (*which has more constraints*). In math terms, it can be defined as $\mathcal{R}_0 \subseteq \mathcal{R} \cap \text{Tot}$ (\subseteq).

This is not obvious since one can obtain total functions from partial ones:

$$f: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$f(x, y) = \begin{cases} -1, & \text{if } y < x \\ 0, & \text{if } y = x \\ \uparrow, & \text{otherwise} \end{cases}$$

$$h(x) = \mu y. f(x, y) = x$$



So, we use minimalisation to find the smallest element here, which is x . Starting from the partial functions, you use combinators from R and still obtain total functions.

Solution (made by me, to take with a grain of salt)

To show the subset property, we're essentially trying to cover the case on which the partial recursive functions can use minimalisation, which happens if their result is defined. So, we want to give an example which is defined for "at least some elements" and get back something total.

The building blocks would be the usual ones, all the basic functions (e.g. zero, successor, projection, etc.) and operations (primitive recursion, composition, etc.). One can say for instance this could be easy; one finds a function which can't be defined in all cases and then call it quits. Thing is, to properly prove this we need a bound, which in partial recursive functions, we don't always have. Minimalisation allows us to use bounded operations and combine low-level operations like sg , $-$, \overline{sg} , etc.

To do this, we can consider a case in which we nest a finite function, which is total, inside a partial function. In this case, the minimalisation would formally ensure "we stay inside bounds as long as the situation is total". So, we define $f(x)$ as follows:

$$f(x) = \begin{cases} 1, & \text{if } \min(x, x - 1) \text{ is defined} \\ x, & \text{otherwise} \end{cases}$$

Then, we define the said total function $g(x)$ that utilizes $f(x)$ to ensure totality and checks the result of $f(x)$:

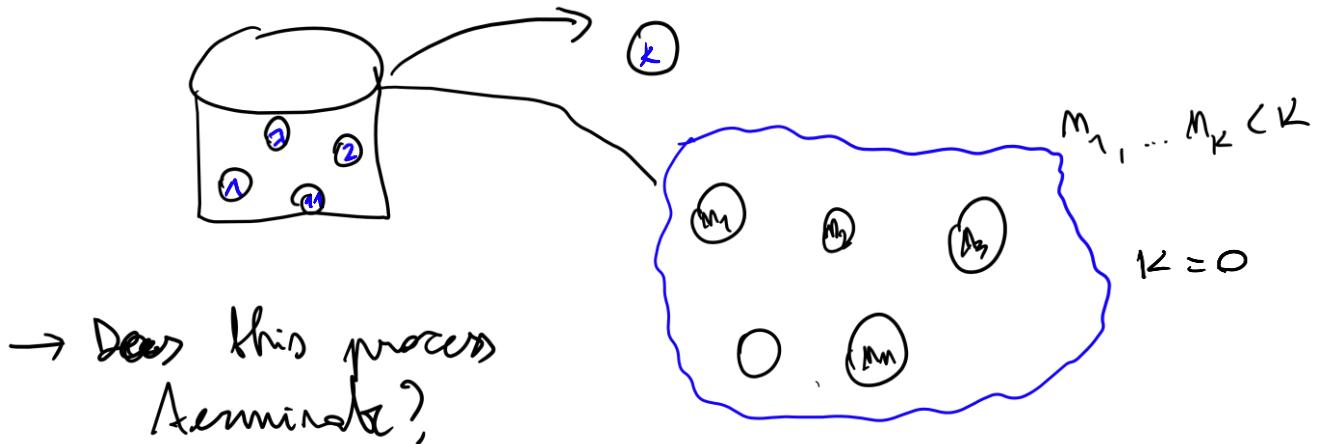
$$g(x) = \begin{cases} 0, & \text{if } f(x) = 0 \\ f(x), & \text{otherwise} \end{cases}$$

So, the minimalisation can be used like $g(x) = \mu y. f(x)$, finding the smallest y for which $f(x)$ is defined. Given the class definition, this is defined for the values of the subset for $f(x)$, when the function is total, minimalisation is allowed and allows us to get defined results.

Therefore, we have shown that \mathcal{R}_0 includes functions that can be derived from partial functions in \mathcal{R} (by applying minimalization) and that become total functions when their results are defined.

This confirms that \mathcal{R}_0 is indeed a subset of $\mathcal{R} \cap \text{Tot}$, as it includes functions that are both in \mathcal{R} and total, in accordance with the additional constraint of \mathcal{R}_0 .

Exercise



EXERCISE 8.9. Given a box with an arbitrary number of balls in it, each one with a number in \mathbb{N} , do the following:

- extract a ball;
- substitute the extracted ball with an arbitrary number of balls, each one with a label lower than the extracted one.

Prove that this process always terminates.

To prove it, we must define a well-founded order and decreasing it to prove the conjecture.

Solution (made by me, to take with a grain of salt)

Essentially, we want a set in which the minimalisation is always defined and recalling the well-founded poset definition. The well-founded order on \mathbb{N} ensures that there are no infinite descending chains, since the problem explicitly tells us we're dealing with natural numbers.

The property of order in \mathbb{N} is as follows:

- given two elements, call them a, b , we know that $a < b$ if a is less than b , leaving trace for a smaller natural number.

This respects the well-founded order definition, so, given a well-founded poset (P, \leq) every non-empty element has a minimal element d s.t. $\forall d' \in X, d' \leq d \Rightarrow d' = d$. This says that, from starting from N , we can take any subset of numbers and the well-foundedness will hold.

Given natural numbers are well-ordered, the well-foundedness of the order is trivial, given there is always the least element in the subset and minimalisation is always present. In other words, you can start with any natural number, and by repeatedly applying the "successor" operation (adding 1), you can generate all the natural numbers

Now, let's consider the process of extracting a ball and substituting it with balls labeled with lower numbers. At each step of the process, we extract a ball with a certain number, and by our well-founded order " $<$ " we know that the number of balls labeled with lower numbers is finite.

Let's provide a concrete example to illustrate the termination of the process. We can consider a box with an arbitrary number of balls labeled with natural numbers (\mathbb{N}). The process is as follows:

1. Extract a ball.
2. Substitute the extracted ball with an arbitrary number of balls, each labeled with a number lower than the one extracted.

Let's say we start with the following set of balls in the box (call it $S \subseteq \mathbb{N}$):

$$\{5, 3, 7, 1, 4\}$$

Now, let's apply the process step by step:

1. *Step 1:* We extract the ball labeled with 7.
2. *Step 2:* We substitute the ball with lower-labeled balls: $\{5, 3, 1, 4\}$

Now, we repeat the process:

1. *Step 1:* We extract the ball labeled with 5.
2. *Step 2:* We substitute the ball with lower-labeled balls: $\{3, 1, 4\}$

Once again:

1. *Step 1:* We extract the ball labeled with 4.
2. *Step 2:* We substitute the ball with lower-labeled balls: $\{3, 1\}$

Now, let's consider the well-founded order defined earlier: $a < b$ if a is less than b . In this case, the well-founded order is based on the natural numbers, where each number is less than the next. Since the extracted balls are labeled with natural numbers, and we always replace them with balls labeled with lower numbers, the process is guaranteed to terminate.

In this example, the process terminates when we have no more balls to extract because we always substitute them with balls labeled with lower natural numbers. This demonstrates that the process of extracting and substituting balls will always come to an end, regardless of the initial configuration of the box.

11 CANTOR DIAGONALIZATION TECHNIQUE

Roughly speaking, the diagonalization technique allows, starting from one object, to build an object of the same nature that differs from all values inside the collection, because the *object itself is meant to be different from all the set values*.

Why do we care then?

- Because it is a powerful argument to show functions are not computable, building them in a way and then using a partial function, the enumeration of such differs from all inputs and is not computable (source of this last one, which is pretty clear [here](#)).

The key is that it's different *by construction*, which means that you're choosing the digits of the different number, say d , specifically so that it will be different from every other item in the list (more [here](#) in case). This technique was created by Cantor to show that there are different levels of infinity and not all infinity levels are equal.

The idea behind is, given a countable set of objects, $x_i \in I$, we can build another object in the same nature of x s.t. $x \neq x_i$. The idea has the following structure:

$$x_0 \ x_1 \ x_2 \ x_3 \ \dots \ x_i \Leftarrow \text{position } k \text{ of } x_i$$

aim: build x s.t. $x \neq x_i$ x differs from x_i at position i

Cantor introduced the notion of *cardinality*, which is a measure of the size or "number" of elements in a set. He famously showed that not all infinite sets are of the same size. Here, we discuss Cantor's work on demonstrating different degrees of infinity, using the example of finite sets and power sets (sets of all subsets) as a basis for the argument.

For finite sets, the cardinality is clear for each case, as you can see here. This also holds for infinite sets, so this gives the idea there exist *more* infinite sets and *more* different orders of infinite.

$$\forall x \text{ set}, |X| < |2^x|, \quad 2^x = \{y \mid y \subseteq x\}$$

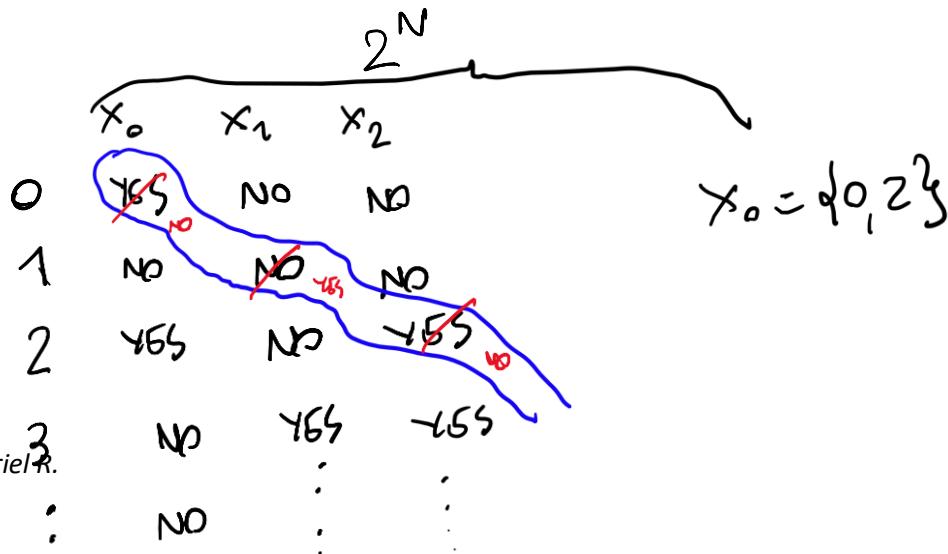
$$\text{if } x \text{ is finite } x = \{0, 1\}, 2^x = \{0, \{0\}, \{1\}, \{0, 1\}\}$$

$$|x| = 2 < |2^x| = 2^{|x|} = 2^2 = 4$$

Example: $|N| < |2^N|$

Proof Assume $|N| \geq |2^N|$, i.e. $|2^N|$ countable (so, the thesis is false – we assume the power set to be countable). This means that there exists an enumeration of 2^N ($\mathbb{N} \rightarrow 2^{\mathbb{N}}$), which is surjective.

We take the diagonal here, which is seen as a set. This is taken and then changed systematically; this will be different from all the sets listed above. I



We can define a diagonal D such that the i -th element differs from X_i on i , which means that we can always map an element in the diagonal such that “it will always be inside a set”.

$$\begin{aligned} D &= \{i \mid i \notin X_i\} \subseteq \mathbb{N} \\ \Rightarrow \exists k \in \mathbb{N} \text{ s.t. } D &= x_k \end{aligned}$$

Problem: $k \in D$?

- Yes: $k \in D \Rightarrow k \notin X_k = D \rightarrow$ contradiction
- No: $k \notin D \Rightarrow k \in X_k = D \rightarrow$ contradiction

$\Rightarrow 2^{\mathbb{N}}$ is not countable, $|\mathbb{N}| < |2^{\mathbb{N}}| \rightarrow$ the proof can't be done without contradictions.

In words, the diagonalization holds some logical implications (coming from a great answer [here](#)):

- It basically states “There exist some infinite sets that cannot be put into one-to-one correspondence with the set of natural numbers”
- There exist some infinite sets for which one of the following must be true: either such a set has no well-ordering, or it is larger than the set of natural numbers
- No matter what language or formal system one uses to describe mathematical objects, there exists some encoding that maps every different statement in that language or system to a unique natural number.
- Thus, the set of all possible unique descriptions of things is equal in size to the set of natural numbers. This implies that the set of all of the different mathematical things *that could possibly exist* is equal in size to the natural numbers
- Under other considerations, the diagonal argument implies that certain sets, those that are referred to as “uncountably infinite”, have no well-ordering. They are uncountable not because they are too large to be counted, but rather because there does not exist any way for them to be arranged in order to be counted.

In a nutshell: this shows there exists infinite sets that are uncountable (like real numbers set).

11.1 EXAMPLES

Exercise:

$$F = \{f \mid f: \mathbb{N} \rightarrow \mathbb{N}\}$$

$$|F| > |\mathbb{N}| \text{ (which means } ||\mathbb{N}| \rightarrow |\mathbb{N}|| > |\mathbb{N}|\text{)}$$

There are two approaches to consider:

1)

$$F_2 = \{f \in F \mid f: \mathbb{N} \rightarrow \mathbb{N} \text{ total}\} \subseteq F, \text{img}(F) \subseteq \{0,1\}$$

To be more formal, we need bijection, so $|F_2| = |2^{\mathbb{N}}|$ and for bijection $F_2 \rightarrow 2^{\mathbb{N}}, f \mapsto \{n \mid f(n) = 1\}$

$$\begin{array}{ccc} F_2 \subseteq F & \xrightarrow{\quad} & \mathbb{N} \\ F_2 \xrightarrow{\quad} \mathbb{P} & & |\mathbb{P}| \\ \text{injective} & & \\ f \xrightarrow{\quad} k & & \end{array}$$

Written by Gabriel R.

In words:

1. Define a subset of F , denoted as F_2 , which consists of functions in F that are "total" (defined for all natural numbers) and whose image (range) is a subset of $\{0, 1\}$.
2. The goal is to establish a bijection between F_2 and the power set of N , denoted as 2^N .
3. To define this bijection, for each function f in F_2 , you can associate it with a set of natural numbers where the set contains all n such that $f(n) = 1$. This association creates a function that maps functions in F_2 to subsets of natural numbers, or in other words, $F_2 \rightarrow 2^N$.

It's important to note that the identity function is needed as a component of this bijection in order to have actually enumerable elements; in any case, the function itself is uncountable.

(2nd possibility) $|F| > |\mathbb{N}|$

Consider an enumeration of elements in F :

	f_0	f_1	f_2	f_3	...
0	$f_0(0)$	$f_1(0)$	$f_2(0)$	$f_3(0)$...
1	$f_0(1)$	$f_1(1)$	$f_2(1)$	$f_3(1)$...
2	$f_0(2)$	$f_1(2)$	$f_2(2)$	$f_3(2)$...
.
.

Here the diagonalization tries to enumerate every single function over the i values, which are not finitely countable hence numerable, since we have more numbers than the natural set like the previous approach. We have $f \neq f_n \forall n$ since $f(n) \neq f_n(n)$ by construction.

Hence, there is no enumeration of all functions in $F \Rightarrow |F| > |\mathbb{N}|$ (since you cannot establish a one-to-one correspondence between F and the natural number).

Observation: There is a total non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$:

$$f(x) = \begin{cases} \phi_n(n) + 1, & \phi_n(n) \downarrow (n \in W_n) \\ 0, & \phi_n(n) \uparrow (n \notin W_n) \end{cases}$$

	φ_0	φ_1	φ_2	φ_3	...
0	$\varphi_0(0)$	$\varphi_1(0)$	$\varphi_2(0)$	$\varphi_3(0)$...
1	$\varphi_0(1)$	$\varphi_1(1)$	$\varphi_2(1)$	$\varphi_3(1)$...
2	$\varphi_0(2)$	$\varphi_1(2)$	$\varphi_2(2)$	$\varphi_3(2)$...
.
.

We can notice two things:

- f is total (defined for all natural numbers, thanks to diagonalization)
- f is not computable, $f \neq \phi_n \forall n \in N$ (letter here is "phi") – differs from all functions in the diagonal

Consequently, the enumeration does not contain all the functions of F , hence F not enumerable (bottom line: if you have infinite functions, it doesn't mean you have all of them) – so, if you take whatever enumeration of computable functions, you have a function always different from all other computable function.

This means it "exists" but cannot compute all inputs. Infact, $\forall n, f(n) \neq \phi_n(n)$:

- $\phi_n(n) \downarrow$ then $f(n) = \phi_n(n) + 1 \neq \phi_n(n)$
- $\phi_n(n) \uparrow$ then $f(n) = 0 \neq \phi_n(n)$

So, the use of diagonalization actually makes us establish a total function, but actually gives an input which *makes the enumeration incomplete*; in other words, it works based on the idea of creating a new element or object that is systematically distinct from all the elements in the list, thus showing that the list is incomplete. Given we can't count them, there exists an infinite list of non-computable functions.

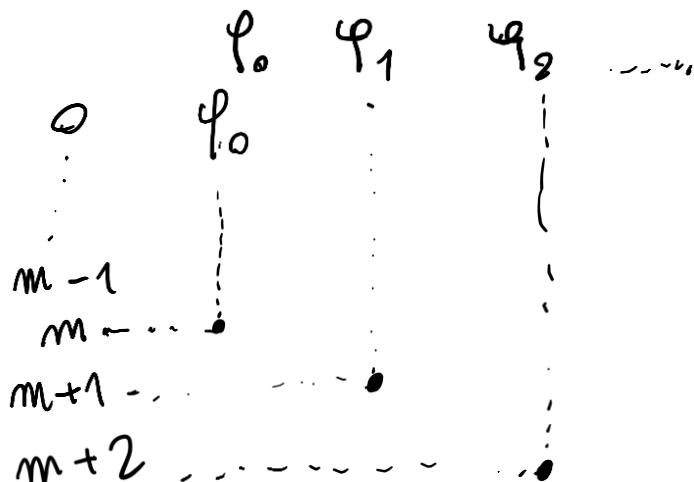
f is total since it is always defined. With the fact that the function $f(n) = \phi_n(n) + 1$ if $n \in W_n$ it will be by definition different from all computable functions, so it will be not computable.

Exercise

Let $F: \mathbb{N} \rightarrow \mathbb{N}$ be any function, $m \in \mathbb{N}$. Show that there is a non-computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$$g(n) = f(n), \forall n < m$$

The idea here is using a "translated diagonal", which is used to construct a non-computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ based on a given function $f: \mathbb{N} \rightarrow \mathbb{N}$ and a natural number m . This technique is often employed to show the existence of non-computable functions that are distinct from a given computable function f up to a certain point ($n < m$).



$$g(n) = \begin{cases} f(n), & n < m \\ \phi_{n-m}(n) + 1, & n \geq m \text{ and } \phi_{n-m}(n) \downarrow \\ 0, & n \geq m \text{ and } \phi_{n-m}(n) \uparrow \end{cases}$$

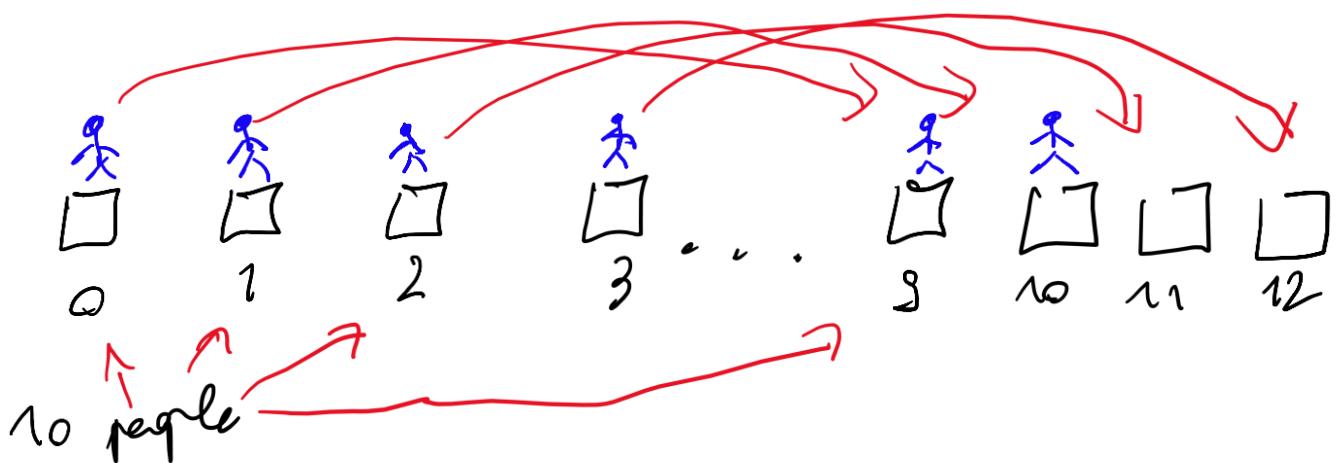
g is not computable since $\phi_n(n+m) \neq g(n+m)$, so $\forall n \phi_n \neq g$

Hilbert's Hotel is a famous thought experiment in the field of mathematics, particularly in set theory, proposed by the German mathematician David Hilbert.

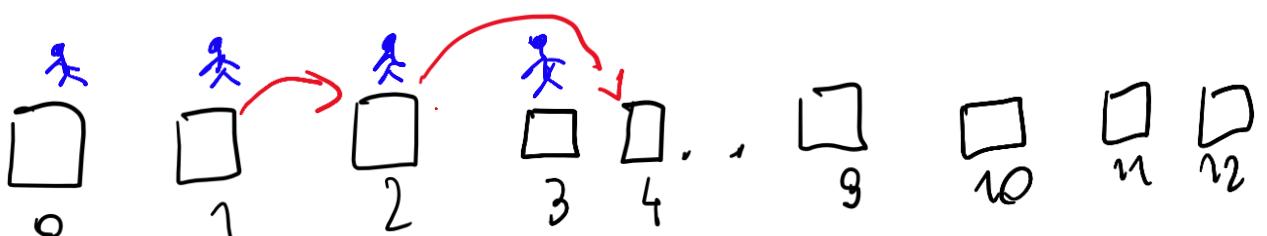
Here's a description of it: imagine a hotel with an infinite number of rooms, numbered 1, 2, 3, and so on, extending to infinity. This hotel is fully occupied, with every room containing a guest.

Now, let's consider several paradoxical situations:

- **New Guests Arriving:** Suppose a new guest arrives at the hotel and wants a room. In a typical finite hotel, this would be a problem because all the rooms are occupied. However, in Hilbert's Hotel, accommodating the new guest is not an issue.
 - Solution: The manager can simply ask every current guest to move to the room with a number one greater than their current room. So, the guest in room 1 moves to room 2, the guest in room 2 moves to room 3, and so on. This frees up room 1 for the new guest.
 - First drawing below shows this one.
- **Infinite New Guests:** Now, imagine an infinite bus filled with an infinite number of new guests arriving at the hotel. Each guest needs a room.
 - Solution: The manager can accommodate all the new guests. He asks the current guest in room 1 to move to room 2, the guest in room 2 to move to room 4, the guest in room 3 to move to room 6, and so on. In this way, all the odd-numbered rooms are now vacant, and the new guests can be placed in them.
- **Adding More Infinity:** If an additional infinite bus with infinitely many new guests arrives, you can continue this pattern to free up additional rooms for new guests. For example, by doubling the room numbers, all the rooms originally occupied by guests become vacant.
 - Second drawing shows this one.



If countably many new guests arrive?



Alternative:

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

$\delta \neq \varphi$

g is not computable:

$\varphi_0 \varphi_1 \dots \varphi_{m-1} \varphi_m \varphi_{m+1}$
 $\varphi \neq \varphi_m \forall m \geq M \nearrow$ infinitely many repetitions
 for all computable functions

for all computable functions $h \exists n \geq m \quad h = \phi_n \neq g$

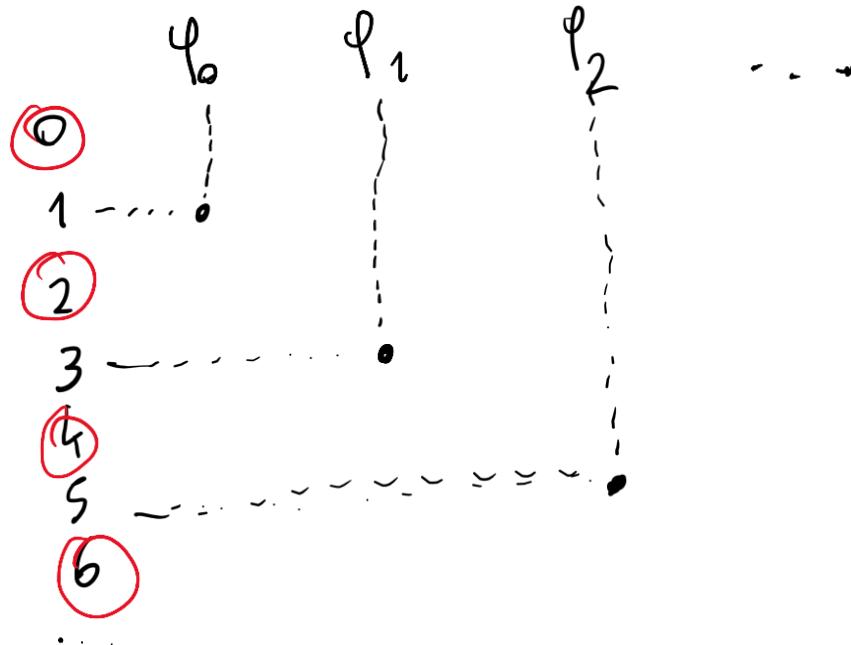
$\Rightarrow g$ is different from all computable functions $\Rightarrow g$ not computable

(Basically, since each function appears infinitely many times in the enumeration, skipping the first $m - 1$ steps does not create any problem. The contradiction arises because we've shown that there exists a computable function h such that $g(n)$ is different from h for some $n \geq m$.

The key insight here is that, by construction, g is designed to be distinct from all computable functions for n greater than or equal to m . This shows that g is non-computable, as there is no algorithm that can compute g for all inputs without contradiction).

Exercise

Show that there is a function $g: \mathbb{N} \rightarrow \mathbb{N}$ total, not computable s.t. $g(n) = 0, \forall n \text{ even}$ (returns 0 when the input is even, because there is no surjection – mapping; you can see the point in the following drawing)



$$g(n) = \begin{cases} 0, & \text{if } n \text{ is even} \\ \phi_{\frac{n-1}{2}}(n) + 1, & \text{if } n \text{ is odd and } \phi_{\frac{n-1}{2}}(n) \downarrow \\ 0, & \text{if } n \text{ is odd and } \phi_{\frac{n-1}{2}}(n) \uparrow \end{cases}$$

→ g is total

→ $g(n) = 0$ for all n even

→ g not computable since $g \neq \phi_n$ for all $n \in N$

$$g(n) = \phi_n(2n+1)$$

$$\left. \begin{array}{l} g(1) = \phi_0(1) \\ g(3) = \phi_1(3) \\ \vdots \end{array} \right\} g(n) = \phi_n(2n+1)$$

Solution and idea

It's total because it's defined for all natural numbers, but not computable. Consider, for W_n the set of natural numbers for which the function does halt, the following reasoning: if there are even numbers, *in any case* there will be natural numbers expressing them, *but we find at least a value* which we cannot compute, given the function was *designed to be different from all others from the start*. This can be seen as:

- If $2n+1 \in W_n \Rightarrow f(2n+1) = \phi_n(2n+1) + 1 \neq \phi_n(2n+1)$
- If $2n+1 \notin W_n \Rightarrow f(2n+1) = 0 \neq \phi_n(2n+1) \uparrow$

g differs for ϕ_n for all inputs regardless of whether the computation halts or not, so:

$$\forall n f(2n+1) \neq \phi_n(2n+1)$$

Exercise

$f_0, f_1, f_2 \dots (f_i)_{i \in N}$ given, define $f: N \rightarrow N$ s.t. $dom(f) \neq dom(f_i) \forall i \in N$

Solution and idea

To define a function $f: N \rightarrow N$ such that the domain of f is not equal to the domain of any of the functions f_i for all $i \in N$, you can use the concept of disjoint domain selection. The idea is to construct a function that selects values from a different set than the domains of all the f_i functions.

For each natural number $n \in N$, define $f(n)$ as follows:

1. If n is even, let $f(n) = 2n$. This ensures that f takes even numbers and maps them to even numbers.
2. If n is odd, let $f(n) = 2n + 1$. This ensures that f takes odd numbers and maps them to odd numbers.

The key idea here is to make f map even numbers to even numbers and odd numbers to odd numbers.

As a result, the domain of f is the set of all natural numbers (N), but the domain of each f_i is either the set of even natural numbers or the set of odd natural numbers, depending on the value of i .

Idea:

$$g(n) = \begin{cases} 0 & \text{if } n \notin dom(f_n) \\ \uparrow & \text{if } n \in dom(f_n) \end{cases}$$

In this way

$$\forall n n \in dom(g) \Leftrightarrow n \notin dom(f_n)$$

Similar to before, g differs for f_n for all inputs regardless of whether the computation halts or not.

12 PARAMETRISATION/SMN-THEOREM

Here I want to give a careful intuition on what the theorem is all about. In computability theory, we often want to express a function that takes a function as an argument.

The smn-theorem (also called “s-m-n-theorem” on the book, I’ll refer to the first notation) provides a way to represent such functions in a normalized form. It essentially provides a method for encoding and manipulating functions, which is particularly important in understanding the computability of functions.

- This is also called *parametrisation theorem/translation lemma* because it shows an index e for a computable function can be found effectively from a parameter
 - o some links useful to understand: [here](#), [here](#) and [here](#).
- The name of it comes from the three arguments, which in fact are S, m, n

Let $F: \mathbb{N} \rightarrow \mathbb{N}$ computable, so there exists $e \in N$ s.t. $f = \phi_e^{(2)} (P_e = \gamma^{-1}(e))$

$$f(x, y) = \phi_e^{(2)}(x, y)$$

(with P_e the program which computes F and γ^{-1} the inverse function to find e).

Let $x \in \mathbb{N}$ be fixed, we obtain a function f a single argument:

$$f_x: \mathbb{N} \rightarrow \mathbb{N}$$

$$f_x(y) = f(x, y) = \phi_e^{(2)}(x, y)$$

Given x is fixed (constant), for every value of x , you get a new function with one argument, which we'll call f_x . It's like plugging in x and leaving y as a variable, obtaining an index *effectively* from the unary function. The following are examples on this:

$$e.g. f(x, y) = y^x$$

$$f_0(y) = y^0 = 1$$

$$f_1(y) = y^1 = 1$$

$$f_2(y) = y^2$$

...

Since for all fixed $x \in \mathbb{N}, f_x$ is computable there is $d \in N$ s.t. $f_x = \phi_d$ *depends on e, x* $= P_{S(\ell, x)}$

Now, the key point here is that for every fixed x , the function f_x is computable. In other words, you can compute f_x using some program represented by a natural number d .

Hence there is a function $s: \mathbb{N}^2 \rightarrow \mathbb{N}, s(e, x) = d$

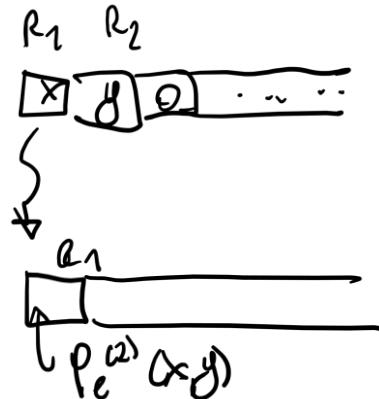
- The smn-theorem introduces a function s that takes two natural numbers, e and x , as inputs and returns another natural number, d . This function maps pairs of natural numbers to another natural number d which is itself computable.
- In simpler terms, there's a way to determine, using some program or procedure, which program computes the function f_x for any given x .

The smn-theorem additionally says that $s: \mathbb{N}^2 \rightarrow \mathbb{N}$ is computable.

$$f(x, y) = \begin{cases} \text{def } P_e(x, y) \\ \dots x, \\ y \\ \text{return} \dots \end{cases} \rightarrow \begin{cases} \text{def } P_e(\cancel{x}, y) \\ \dots \cancel{x} \leftarrow 1 \\ y \\ \text{return} \dots \end{cases}$$

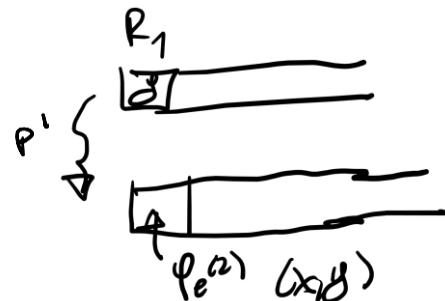
Idea:

Given $e \in \mathbb{N}$, we structure the program here on the right:



for each $x \in \mathbb{N}$ fixed,

we want a program P' as shown here:



What is P' doing?

$$P' = \begin{cases} \text{move } y \text{ to } R_2 \\ \text{write } x \text{ on } R_1 \text{ (set as constant during execution of } P') \\ \text{execute } P_e \text{ (} P' \text{ runs the program that corresponds to the index } e) \end{cases}$$

To get:

$$s(e, x) = \gamma(P')$$

Essentially, the computation of $s(e, x)$ involves:

- Get the program $P_e = \gamma^{-1}(e)$ that computes $\phi_e^{(2)}(x, y)$
 - o Means "find the program for which we can map back the original natural numbers"
- Get the program that computes $f_x = \lambda y. f(x, y)$ with fixed x from P_e
 - o Means the function will be bounded over a fixed parameter effectively

Intuitively, we get the program which will map back our couple of inputs and from that we can build a new program able to advance the computation. Even fixing an argument we take, from a transforming function over it, an effective program.

$$\phi_e^{(x)}(x, y) \text{ computed by } P_e = \gamma^{-1}(e)$$

For any fixed x , one obtains a function of y only. The following are all computable and the program which computes them all is obtained algorithmically.

$$x = 0 \quad y \mapsto \phi_e^{(2)}(0, y)$$

$$x = 1 \quad y \mapsto \phi_e^{(2)}(1, y)$$

...

The program which computes the functions above for each fixed x can be obtained algorithmically starting from P_e , starting with two arguments and hardcoding the indices, we will get the output effectively.

The name of theorem comes from working with function of form $f(\vec{x}, \vec{y}): \mathbb{N}^{m+n} \rightarrow \mathbb{N}$ starting from a total function called S , as seen before (hence, smn).

$$P_e(x, y)$$

$$\begin{matrix} x \\ y \\ \vdots \\ \end{matrix}$$

$$\xrightarrow{x = x_p}$$

$$\begin{matrix} x \\ x_0 \\ y \\ \vdots \\ \end{matrix}$$

$$\xrightarrow{x = x_0}$$

More generally: $f: \mathbb{N}^{m+n} \rightarrow \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{S(e, \vec{x})}^{(n)}(\vec{y})$$

12.1 SMN-THEOREM

Theorem (smn-theorem) (according to Kleene)

Given $m, n \geq 1$ there is a total computable function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ such that $\forall \vec{x} \in \mathbb{N}^m, \forall \vec{y} \in \mathbb{N}^n, \forall e \in \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

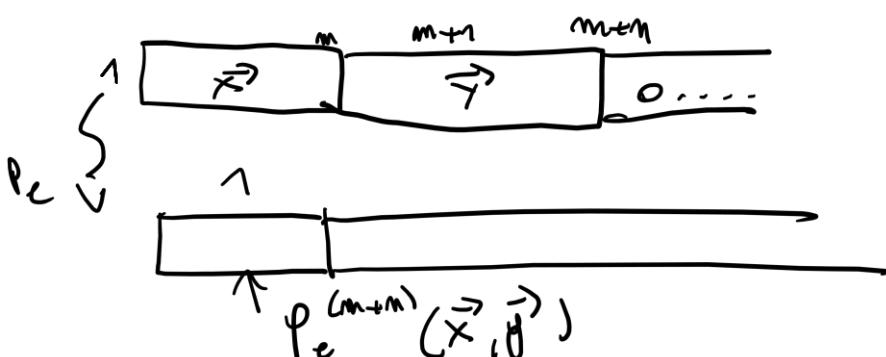
The smn-theorem states that given a function $g(x, y)$ which is computable, there exists a total and computable function s such that $\phi_{s(x)}(y) = g(x, y)$, basically "fixing" the first argument of g – usually, we fix x in favor of y . It's like partially applying an argument to a function. This is generalized over m, n tuples for x, y .

Basically, we have that the program over two indices m, n is the same as the transformed function over x inputs. Fixing an index, we obtain a program with the same features.

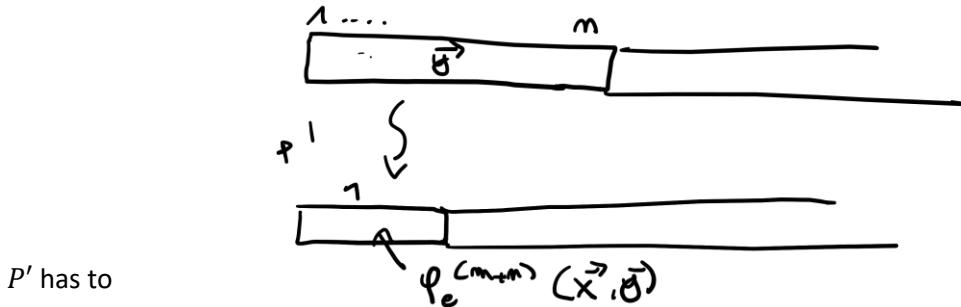
Proof

Intuitively, given $e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m$

- We get the program $P_e = \gamma^{-1}(e)$ in standard form that computes $\phi_e^{(m+n)}$, so starting from the first drawing (this below), in which we compute the $\phi_e^{(m+n)}$ over all inputs (\vec{x}, \vec{y}) (so $\phi_e^{(m+n)}(\vec{x}, \vec{y})$)



You want, for each $\vec{x} \in \mathbb{N}^m$ fixed, a program $P' \Leftarrow$ depending on e, \vec{x} (mapping back its inputs effectively and composing function *parameterizing* its values, this you can see below).



- Move \vec{y} to $m+1, \dots, m+n$ (so, move forward computation of m registers)
 - Write \vec{x} in $1 \dots m$ (so, load the value in the free m registers)
 - Execute P_e (so, execute the computation)

The program P' can be:

$T(m, m + n)$ // move y_n to R_{m+n}

.....

$T(1, m + 1)$ // move y_1 to R_{m+1}

$Z(1)$ } // write x_1 to R_1
 $S(1)$ }
...
 $S(1)$ } x_1 Lines

$Z(m)$ } // write x_m to R_m
 $S(m)$ }
...
 $S(m)$ } x_m , *A times*

Concatenation will update all the jump instructions, hence moving and writing values *for all function parameterized inside*, mapping back effectively with $P_e = \gamma^{-1}(e)$. Once the program P has been built, we have $S(e, \vec{x}) = \gamma(P')$. Given each function is effective, existence, totality and computability of s are informally proven.

In the context of the smn-theorem, ϕ_e^k is the e^{th} partial computable function of k variables. The theorem establishes that there exists a total computable function, denoted as $s_{(m,n)}$, which can effectively "translate" or encode the computation of $\phi_e^{(m+n)}(x, y)$ into the computation of $\phi_{s(e,\vec{x})}^n(\vec{y})$.

In summary, also given the book example (p. 90), we can establish given the effectiveness of γ and γ^{-1} , that s_m^n is effectively computable, given the bijective mapping and computability of both.

The formal proof of computability is long and involves the combination of many parametrized functions as auxiliary to construct the smn function (which is primitive recursive for all indices). Consider each for cases:

1) sequential composition of programs

Given two program codes (e_1, e_2) , we want the sequential encoding like $\rightarrow \gamma(P_{e_1} P_{e_2})$

Consider the *update* function:

$$upd: \mathbb{N}^2 \rightarrow \mathbb{N}$$

where $upd(e, h) = (\text{code of the program obtained from } P_e = \gamma^{-1}(e) \text{ by updating all jump instructions } J(m, n, t) \text{ to } J(m, n, t + h))$. Basically, this updates a program's jump instructions based on e and h .

We define an auxiliary function working on each single instruction encoded with β :

$$upd^\sim: \mathbb{N}^2 \rightarrow \mathbb{N}$$

where $upd^\sim(i, h) = \beta$ (code of the instruction obtained from $\beta^{-1}(i)$, updating the target if it is a jump)

Note: $\beta(j(m, n, t)) = v(m - 1, n - 1, t - 1) * 4 + 3$ (calculates an encoded value based on parameters m, n, t)

Given $i, h \in N, q = qt(4, i), r = rm(4, i)$, we formally define like:

$$\begin{aligned} upd^\sim(i, h) &= \begin{cases} i, & \text{if } rm(4, i) \neq 3 \\ v(v_1(q), v_2(q), v_3(q) + h) * 4 + 3, & \text{if } rm(4, i) = 3, q = qt(4, i) \end{cases} \\ &= i * sg(|rm(4, i) - 3|) + (v(v_1(q), v_2(q), v_3(q) + h) * 4 + 3) * \overline{sg}(|rm(4, i) - 3|) \end{aligned}$$

Overall, this function updates the encoded instruction i based on whether it's a jump instruction or not, adjusting the jump target when required, calculating each time a new value thanks to the present triplet (which consider the two sign functions in order to "encode the two different characteristic functions, covering all cases in which will be either 1 or 0, deciding if it's useful to jump or not").

Now:

$$\begin{aligned} upd(e, h) &= \tau(upd^\sim(a(e, 1), h)) \quad upd^\sim(a(e, 2), h) \quad upd^\sim(a(e, l(e)), h) \\ &= \left(\prod_{i=1}^{l(e)-1} p_i^{upd^\sim(a(e, i), h)} \right) * p_i^{upd^\sim(a(e, l(e)), h)+1} - 2 \end{aligned}$$

This, in words, represents a formal way to express the update of a program by modifying its jump instructions, considering the effect of the other two instructions when applied recursively.

Basically, we are doing a kind of pattern matching via parametrization and making it computable thanks to composition and primitive recursion. The goal is to make the entire program computable by ensuring that jump targets are correctly adjusted, reflecting the desired program behavior, also doing some "fine-tuning" (the minus 2).

We encode the input sequence of values like the following, where each instruction is adjusted to the current one, the next one and the fine-tuning that we quoted above:

$$\tau(y_1, \dots, y_m) = \prod_{i=1}^{n-1} P_i^{y_i} * P_m^{y_n+1} - 2$$

Consider $l(e) =$ length of the encoded sequence. For $1 \leq i \leq l(e)$, we get back the corresponding component, which is $a(e, i) = i^{\text{th}} \text{ component}$.

Now we will use the *concatenation of sequences* as a function.

$$c: \mathbb{N}^2 \rightarrow \mathbb{N}$$

The concatenation of sequences follows here, considering the mapping of each encoding with

$$c(e_1, e_2) = \tau(a(e_1, 1) \dots a(e_1, l(e_1)) \ a(e_2, 1) \dots a(e_2, l(e_2)))$$

The *concatenation of programs* can be defined as:

$$\text{seq}: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$\text{seq}(e_1, e_2) = \gamma \begin{pmatrix} P_{e_1} \\ P_{e_2} \end{pmatrix} = c(e_1, \text{upd}(a, l(e_2)))$$

Essentially, in words, we concatenate the first one with the update over the length of the second one.

Then we use the *transfer function*:

$$2) \text{transf}: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$\text{transf}(m, n) = \gamma \begin{pmatrix} T(n, n+m) \\ T(1, m+1) \end{pmatrix}$$

Technically, this one simply shift registers n positions forward.

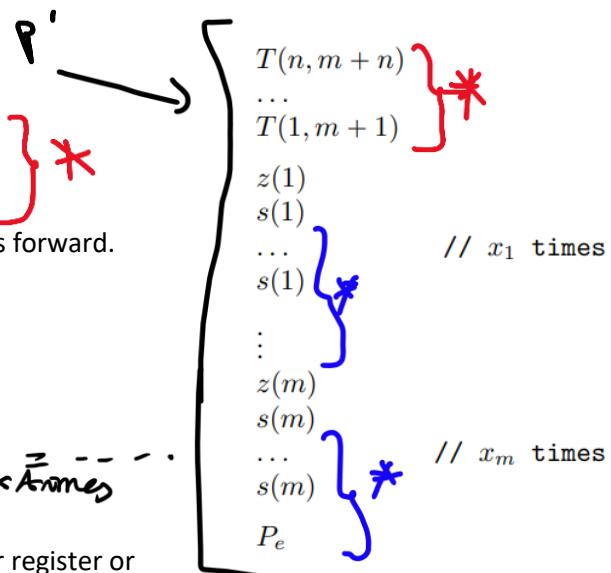
Continuing, we will use the *set function*:

$$3) \text{set}: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$\text{set}(i, x) = \gamma \begin{pmatrix} Z(i) \\ S(i) \\ S(i) \end{pmatrix}$$

x times

It allows you to set a specific value x into a particular register or location i and possibly perform some operations like incrementing the value.



Finally, this brings us to:

$$s_{m,n}(e, \vec{x}) =$$

$$\text{seq}(\text{transf}(m, n),$$

$$\text{seq}(\text{set}(1, x_1),$$

.....

$$\text{seq}(\text{set}(m, x_m), e) \dots)$$

All of this mess can refer to the following *prefix* function because it defines a sequence concatenating multiple subsequences “prefixing” other ones, hence obtaining the final function.

This proves that the smn-function is computable and total (actually, we can observe it's also primitive recursive) since it is a composition of primitive recursive functions, themselves effective.

In other words: $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$, $s_{m,n}(e, \vec{x}) = \text{seq}(\text{pref}_{m,n}(\vec{x}), e)$ which is in PR (remember the class definition given [here](#)).

12.2 SIMPLIFIED SMN-THEOREM

Corollary (Simplified smn-theorem): Let $f: \mathbb{N}^{n+m} \rightarrow \mathbb{N}$ be a computable function.

Then there is a total computable function $s: \mathbb{N}^m \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$f(\vec{x}, \vec{y}) = \phi_{s(\vec{x})}^{(n)}(\vec{y})$$

Note for the reader: You need to know the normal definition. This one is used “concretely” inside exercises (e.g. see recursiveness/reduction/second recursion theorem exercises to see how pattern-like it will be)

Proof

Since f is computable, there is $e \in \mathbb{N}$ s.t. $f = \phi_e^{(m+n)}$

$$f(\vec{x}, \vec{y}) = \phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{S_{m,n}(e, \vec{x})}^{(n)}(\vec{y}) \quad \forall \vec{x}, \vec{y}$$

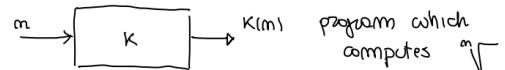
We conclude by setting $S(\vec{x}) = S_{m,n}(e, \vec{x})$

\uparrow *smn theorem*

In words: this is called simplified because you use a total computable function as index, simplifying the whole setting of the argument and all of its specifications.

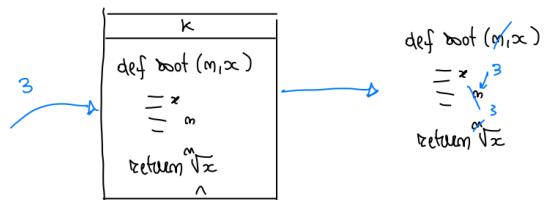
Example of usage of smn-theorem

Prove that there is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ such that $\forall n \in \mathbb{N}, \forall x \in \mathbb{N}$:



$$\phi_{K(n)}(x) = \lfloor \sqrt[n]{x} \rfloor$$

This means that ϕ_k is an enumeration of functions in form $\lfloor \sqrt[n]{x} \rfloor$; so, given n , it returns the program computing $\lfloor \sqrt[n]{x} \rfloor$



This is what right figure is saying.

Proof

The function $f: \mathbb{N}^2 \rightarrow \mathbb{N}$

$$\begin{aligned}
f(n, x) &= \lfloor \sqrt[n]{x} \rfloor \\
&= \max z. \ "z^n \leq x" \\
&= \min z. \ "(z+1)^n > x" \\
&= \mu z \leq x. \ x + 1 - (z+1)^n
\end{aligned}$$

is computable (because it is a bounded minimisation of a composition of known computable functions; as seen here, we put it and bound it correctly).

↓

by ([corollary](#) of) smn theorem there is $k: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$\phi_{K(n)}(x) = f(n, x) = \lfloor \sqrt[n]{x} \rfloor$$

12.3 EXAMPLES

Example

There is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall n, \phi_{K(n)}$ is defined only on n^{th} powers (on y^n for $y \in \mathbb{N}$), i.e.

$$W_{k(n)} = \{x \mid \exists y \text{ s.t. } x = y^n\}$$

We define

$$\begin{aligned} f(n, x) &= \begin{cases} \downarrow, & \text{if } \exists y \text{ s.t. } x = y^n \\ \uparrow, & \text{otherwise} \end{cases} \\ &= \mu y. "y^n = x" \\ &= \mu y. |y^n - x| \end{aligned}$$

computable.

By the ([corollary](#) of the) smn-theorem, $\exists k: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t. $\forall n, x \in \mathbb{N}$:

$$\phi_{K(n)}(x) = f(n, x)$$

$$\phi_{K(n)}(x) = f(n, x) = \begin{cases} \sqrt[n]{x}, & \text{if } \exists y \text{ s.t. } x = y^n \\ \uparrow, & \text{otherwise} \end{cases}$$

Observe that the domain of W is the set of powers:

$$W_{K(n)} = \{x \mid \exists y . x = y^n\}$$

In fact:

$$\begin{array}{c} \text{if } \exists y . x = y^n \\ \text{iff } \phi_{K(m)}(x) \downarrow \text{ iff } \exists y . x = y^m \end{array}$$

Exercise (homework)

Show that there is a total computable function $s: N \rightarrow N$ s.t. $W_{S(x)}^k = \{(y_1, \dots, y_k) \mid \sum_{i=1}^k y_i = x\}$

Solution

The smn-theorem states:

THEOREM 11.2 (smn theorem). Given $m, n \geq 1$ there is a computable total function

$$s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$$

such that $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\varphi_e^{(m+n)}(\vec{x}, \vec{y}) = \varphi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

We define a function:

$$f(x, \vec{y}) = \begin{cases} 0, & \sum_{i=1}^k y_i = x \\ \uparrow, & \text{otherwise} \end{cases}$$

We want essentially to bound the computation to $m = k - 1$ and $n = 1$ as we are dealing with k projections. We'll use the smn-theorem to find a function $s_{k-1,1}(e, x)$ s.t.

$$\phi_e^k(x) = \phi_{s_{k-1,1}}^{(1)}(y)$$

$$= \mu z. \ " \sum_{i=1}^k y_i - x "$$

$$= \mu z. \left| \sum_{i=1}^k y_i - x \right|$$

By the ([corollary](#) of the) smn-theorem, $\exists k: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t. $\forall n, x \in \mathbb{N}$

$$\phi_{S(n)}(x) = f(n, x)$$

$$\phi_{S(n)}(x) = f(n, x) = \begin{cases} \sum_{i=1}^n y_i, & \text{if } \sum_{i=1}^k y_i = x \\ \uparrow, & \text{otherwise} \end{cases}$$

13 UNIVERSAL FUNCTION

We now discuss how the theory developed up to now allows us to prove the computability of a universal function, i.e., a function which, roughly speaking, embodies every computable function of a given arity (which, for a specific value, makes capture all k -ary functions).

For instance, for arity 1, the universal function is $\Psi_U : \mathbb{N}^2 \rightarrow \mathbb{N}$

$$\Psi_U(x, y) = \varphi_e(y)$$

It captures all unary computable functions $\varphi_1, \varphi_2, \dots$. In fact, for all $e \in \mathbb{N}$

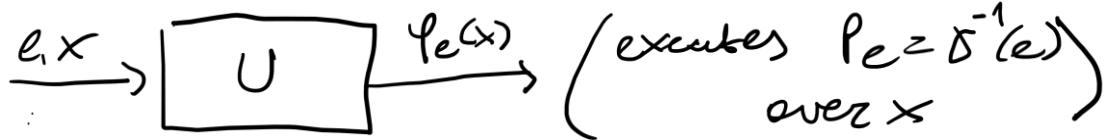
$$g(y) = \Psi_U(e, y) = \varphi_e(y) \rightsquigarrow g = \varphi_e$$

so Ψ_U represents all the computable functions of the form $\mathbb{N} \rightarrow \mathbb{N}$.

More generally, we have the following definition.

Let $\Psi_u : \mathbb{N}^2 \rightarrow \mathbb{N}, \Psi_u(e, x) = \phi_e(x)$ well-defined (where e is the index of the program, x is the input)

Is it computable?



When e varies on the natural numbers

$$\Psi_u(0, \dots) \quad \Psi_u(1, \dots) \quad \Psi_u(2, \dots)$$

$$\dots \quad \dots \quad \dots$$

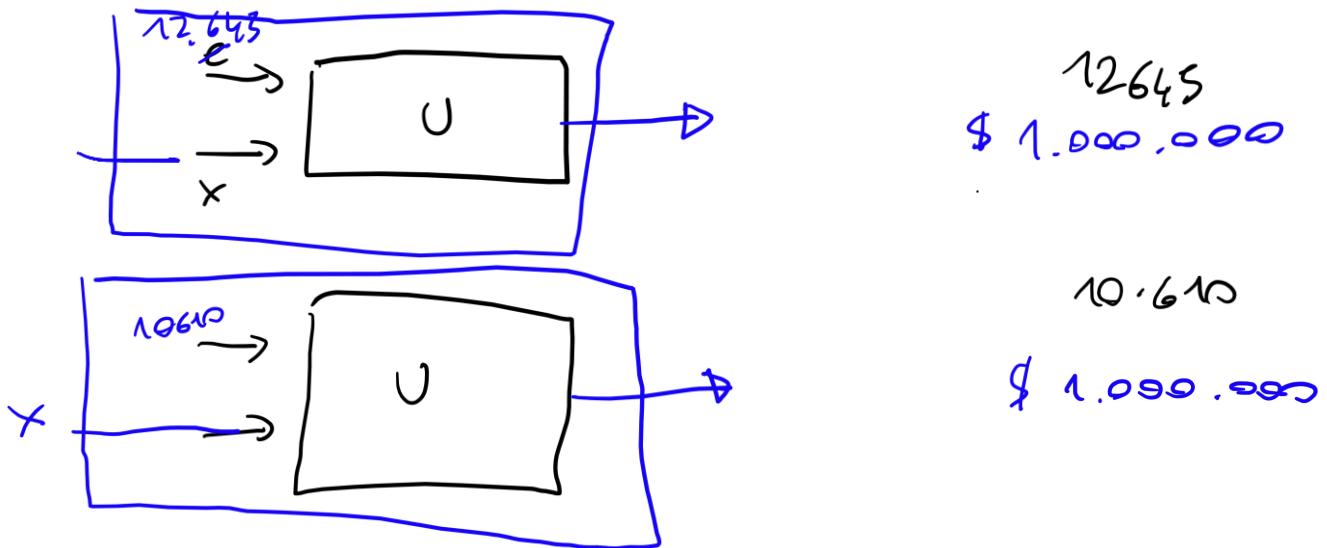
$$\phi_0 \quad \phi_1 \quad \phi_2$$

The function Ψ_u is indeed computable because it merely acts as a "translator" or "emulator" for other computable functions. It takes an index e for a computable program and an input x , and it calculates the result of applying the program ϕ_e to x . Since ϕ_e is a computable function by definition, the computation carried out by Ψ_u is also computable.

In the context of the example, it means it will be computable for any natural number. The key is that we just introduce the smn-theorem in order to modify arguments of functions and make them computable with introduction of additional parameters, hence computing the function output.

We take a metaphoric example here: a company called Turing s.p.a which is asked to produce a software able to recognize these numbers below.

Instead of programming, we take the universal machine, hardcoding the input instead of real input, giving the desired output:



The problem is the index; there is no precise way of working them in writing the program (so, if you put number x , you won't necessarily get number y universally over all indices). We write for this reason a general theorem working on any number of arguments.

13.1 DEFINITION

Theorem (Universal Program)

Let $k \geq 1$. Then the universal function $\Psi_u: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$, $\Psi(e, \vec{x}) = \phi_e^{(k)}(\vec{x})$ is computable.

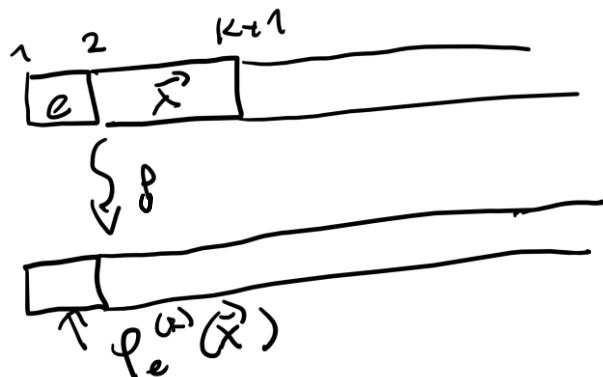
(consider e is the index of the program P_e , which we're given a description of what to run and the arguments \vec{x} ; in words here, the universal function takes the parameterized program inside and over all its values will be able to contain all programs and compute them effectively)

This here is important: it makes $\phi_x(x)$ computable, so the program is computable when we are able to effectively describe a computation over itself on its index; a universal program only needs the ability to decode any number e and hence mimic P_e .

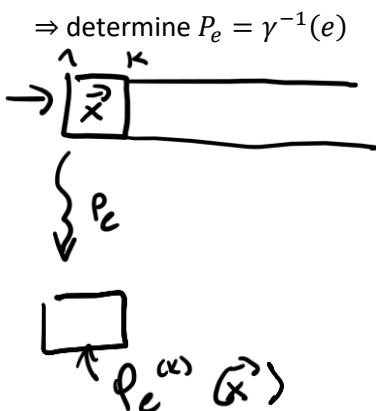
Proof

Fixed $k \geq 1$

given e, \vec{x}



How can P_u work



by Church-Turing Thesis
computable

unsatisfactory!

The idea here is:

- Getting the program $P_e = \gamma^{-1}(e)$
- Execute P_e on input \vec{x}
- If $P_e(\vec{x}) \downarrow$, the value $\Psi_U(e, \vec{x})$ is in R_1 otherwise the program correctly diverges

Since all operations are effective, that's why the Church-Turing thesis holds, and the function is computable. The above argument still gives the idea, but it's not satisfactory on formal terms. Let's try to give here the idea. We need to encode the content of memory, where the configuration of registers is

$c = \prod_{i \geq 1} p_i^{r_i}$ and from that we can obtain the value of each register as $r_i = (c)_i$



From the encoding, we can obtain the value of each register and we show how to simulate the execution steps of a program using only computable functions.

Given $c_k: \mathbb{N}^{k+2} \rightarrow \mathbb{N}$:

- $c_k(e, \vec{x}, t) =$ configuration of the memory after t steps of $P_e(\vec{x})$ (if $P_e(\vec{x})$ terminates in t steps or less \rightarrow final configuration)

Given $j_k: \mathbb{N}^{k+2} \rightarrow \mathbb{N}$:

- $j_k(e, \vec{x}, t) = \begin{cases} \text{index of instructions to be executed after } t \text{ steps of } P_e(\vec{x}) & \text{if } P_e(\vec{x}) \text{ does not halt in } t \text{ steps or fewer} \\ 0 & \text{if it does halt after } t \text{ or fewer steps} \end{cases}$

Observe:

- $P_e(\vec{x}) \downarrow$ then it stops in $t_0 = \mu t. j_x(e, \vec{x}, t)$ steps hence $\phi_e^{(k)}(\vec{x}) = (c_k(e, \vec{x}, \mu t. j_k(e, \vec{x}, t)))_1$
- $P_e(\vec{x}) \uparrow$ then $\mu t. j_x(e, \vec{x}, t) \uparrow$, hence $\phi_e^{(k)}(\vec{x}) \uparrow = (c_k(e, \vec{x}, \mu t. j_k(e, \vec{x}, t)))_1$

We essentially mean we're interested in the configuration obtained inside the first register.

Therefore, in all cases:

$$\Psi_u^{(k)}(e, \vec{x}) = \phi_e^{(k)}(\vec{x}) = (c_k(e, \vec{x}, \mu t. j_p(e, \vec{x}, t)))_1$$

(What we mean by that is the partial recursion gives as result the combination of every possible subinput, effectively – note: usually, composition of computable functions uses this 1, but it's often implied)

Written by Gabriel R.

If we show that c_k, j_k are computable, we can conclude that $\Psi_U^{(k)}$ is also computable.

Aim: show c_k, j_k computable (this allows us to show that, via composition and parametrization, this will prove the computability of c_p, j_p with a *fixed* program P). We build these functions out of the following smaller components:

- Given $i \in N$ instruction code i.e. $i = \beta(\text{Instr})$:

$$\begin{array}{ll} - Z_{\text{arg}}(i) = qt(4, i) + 1 & i = \beta(z(n)) = 4 * (n - 1) \\ - S_{\text{arg}}(i) = qt(4, i) + 1 & i = \beta(s(n)) = 4 * (n - 1) + 1 \\ - T_{\text{arg}}(i) = \pi_1(qt(4, i)) + 1 & i = \beta(T(m, n)) = 4 * \pi(n - 1, n - 1) + 2 \\ - T_{\text{arg}}(i) = \pi_2(\dots) + 1 & \text{computable} \end{array}$$

$j_{\text{arg}_1}, j_{\text{arg}_2}, j_{\text{arg}_3}$

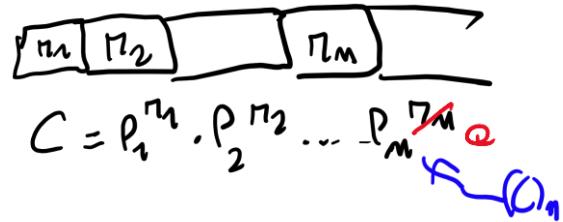
- Effect of executing some algebraic instruction on configuration c

$$\text{zero}(c, n) = qt(p_n^{(c)_n}, c)$$

$$\text{succ}(c, n) = c * p_n$$

$$\text{transf}(m, n) = \text{zero}(c, n) * p_n^{(c)_n}$$

computable



- Effect on configuration c of executing instruction with code i

$$\text{change}(c, i) = \begin{cases} \text{zero}(c, z_{\text{arg}}(i)), & \text{if } rm(4, i) = 0 \\ \text{succ}(c, s_{\text{arg}}(i)), & \text{if } rm(4, i) = 1 \\ \text{transf}(c, t_{\text{arg}_1}(i), t_{\text{arg}_2}(i)), & \text{if } rm(4, i) = 2 \\ c, & \text{if } rm(4, i) = 3 \end{cases} \quad \text{computable}$$

- Configuration of registers starting from configuration c and executing instruction number t in program P_e

$$\text{nextconf}(e, c, t) = \begin{cases} \text{change}(c, a(e, t)), & \text{if } 1 \leq t \leq l(e) \\ c, & \text{otherwise} \end{cases}$$

- Number of the next instruction to be executed after executing $i = \beta(\text{instr})$ and this is in position t in the program

$$\text{ni}(c, i, t) = \begin{cases} t + 1, & \text{if } (rm(4, i) \neq 3) \text{ or } (rm(4, i) = 3 \text{ and } (c)_{j_{\text{arg}_1}(i)} \neq (c)_{j_{\text{arg}_2}(i)}) \\ j_{\text{arg}_3}(i), & \text{otherwise} \end{cases} \quad \begin{matrix} A \rightarrow \\ \boxed{\begin{matrix} \text{---} \\ \boxed{z} \end{matrix}} \\ \downarrow t_m \end{matrix}$$

computable

- Next instruction if we execute instruction in position t of P_e in configuration c

$$\text{nextinstr}(e, c, t) = \begin{cases} \text{ni}(c, a(e, t), t), & \text{if } 1 \leq t \leq l(e) \wedge 1 \leq \text{ni}(c, a(e, t), t) \leq l(e) \\ 0, & \text{otherwise} \end{cases}$$

Now, by primitive recursion, we obtain the state of the program. We start from the initialization and content of the memory, then we apply recursively the next configuration and next instruction, both computable and defined. This way we apply a primitive recursion over single and defined components.

$$c_k(e, \vec{x}, 0) = \prod_{i=1}^k P_i^{x_i}$$



$$j_k(e, \vec{x}, 0) = 1$$

$$c_k(e, \vec{x}, t + 1) = \text{nextconf}(e, c_k(e, \vec{x}, t), j_k(e, \vec{x}, t))$$

$$j_k(e, \vec{x}, t + 1) = \text{nextinstr}(e, c_k(e, \vec{x}, t), j_k(e, \vec{x}, t))$$

c_k, j_k defined by primitive recursion from computable functions are computable (actually, they are in PR , given we never use minimalisation). Thus, we can now say:

$$\Psi_U^{(k)}(e, \vec{x}) = c_k \left(e, \vec{x}, \mu t. j_p(e, \vec{x}, t) \right)_1$$

is computable.

Corollary

The following predicates are decidable:

- $H_k(e, \vec{x}, t) = "P_e(\vec{x}) \downarrow \text{in } t \text{ steps or fewer}"$
- $S_k(e, \vec{x}, y, t) = "P_e(\vec{x}) \downarrow y \text{ in } t \text{ steps or fewer}"$

Proof

a)

The characteristic function:

$$x_{H_k}(e, \vec{x}, t) = \begin{cases} 1, & \text{if } H_k(e, \vec{x}, t) \\ 0, & \text{otherwise} \end{cases} = \overline{sg}(j_k(e, \vec{x}, t)) = \begin{cases} 0, & \text{if } P_e(\vec{x}) \downarrow \text{in } t \text{ steps} \\ \neq 0, & \text{otherwise} \end{cases}$$

is computable by composition.

b)

The characteristic function:

$$x_{S_k}(e, \vec{x}, y, t) = x_{H_k}(e, \vec{x}, t) \cdot \overline{sg}|y - (c_k(e, \vec{x}, t))_1|$$

is computable by composition.

Note: when $k = 1$, we often omit it $\rightarrow H(e, x, t)$ for $H_1(e, x, t)$

Exercise: Computability of the inverse, reprise

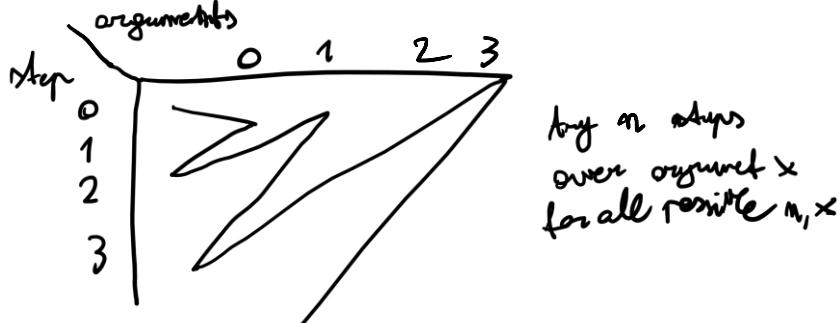
Let $f: N \rightarrow N$ total injective and computable then $f^{-1}: N \rightarrow N$

$$f^{-1}(y) = \begin{cases} x, & x \text{ s.t. } f(x) = y \text{ if it exists} \\ \uparrow, & \text{otherwise} \end{cases}$$

is computable

$$f^{-1}(y) \neq \mu x . |f(x) - y|$$

without totality:



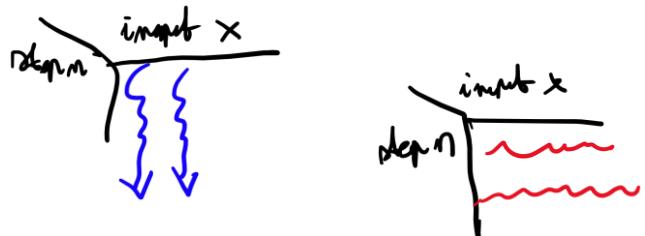
f is computable \rightarrow there is $e \in N$ s.t. $f = \phi_e$

$$\text{look for } \left\{ \begin{array}{l} \text{input } x \\ \text{number steps } n \end{array} \right. \text{ s.t. } P_e(x) \downarrow y \text{ in } t \text{ steps} \quad S(e, x, y, t)$$

(we are trying to say that given a program over underlying inputs, we are able to execute it and go to the following instruction, effectively halting the program over a finite number of steps)

$$f^{-1}(y) = \mu x . \mu n \cancel{S(e, x, y, t)}$$

$$\mu x . \mu n \cancel{S(e, x, y, t)}$$



(we are trying to see if given a pair, the mapping will be injective: infact, this does not happen, as you can see from following drawings and corresponding mappings)

$$f^{-1}(y) = (\mu w . S(e, (w)_1, y, (w)_2)_1$$

$$\pi^{-1}(w) = (\pi_1(w), \pi_2(w))$$

$$w \rightarrow (w)_1, (w)_2$$

$\times \quad \cap$

$$w = 3 = 2^0 * 3^1 \dots \rightarrow (0,1)$$

$$w = 6 = 2^1 * 3^1 \dots \rightarrow (1,1)$$

$$w = 30 = 2^1 * 3^1 * 3^2 \dots \rightarrow (1,1)$$

not injective

Instead of using this injective way, we can omit the totality option given there is at least more than one combination for which $|f(x) - y|$ is defined. We then use the minimalisation operator in order to find at least a solution for which it is computable, which does happen *at least for one value*. Given this assumption, *if it is bounded, then we can omit totality from the discussion because it will be computable anyway*.

Observation

Function which is total and not computable

$$f(x) = \begin{cases} \phi_x(x) + 1, & \text{if } \phi_x(x) \downarrow \\ 0, & \text{if } \phi_x(x) \uparrow \end{cases}$$

NOT A PROGRAM

$$= \begin{cases} \Psi_U(x, x) + 1, & \text{if } \phi_x(x) \downarrow \\ 0, & \text{otherwise} \end{cases}$$

We define this as total, and we argue the above is not a problem. Infact, for every x , if ϕ_x is total, then $\phi_x \neq f$ (where ϕ_x is the partial recursive k -ary function given by the x -th step of enumeration, so it can be different from the function). Since we said this:

$$f(x) = \phi_x(x) + 1 \neq \phi_x(x)$$

so f is not computable. It can be seen as computable as a combination of minimalisation and composition of computable functions, but since our function is not computable, it would be absurd anyway (below I put the example in question to see for yourself).

$$\text{Tot}(x) \equiv \text{"}\varphi_x\text{ is total"} \quad f(x) = (\mu w.(S(x, x, (w)_1, (w)_2)) \wedge \text{Tot}(x) \wedge (w)_3 = (w)_2 + 1) \vee ((w)_3 = 0 \wedge \neg \text{Tot}(x))$$

In words: we look for a w such that we get $\varphi_x(x)$ when x is total, 0 otherwise. We add 1, considering the encoding in tuples $((w)_1, (w)_2, (w)_3)$ will give us $\phi_x(x) + 1$ or 0 otherwise. It is decidable since only decidable predicates are used and thanks to S we will find the number of steps in which that function ends.

With S , if the function is total I will find the number of steps with which it ends.

13.2 RELATED EXERCISES

Exercise: Show that the predicate below is undecidable (this is the “infamous” Halting Problem)

$$\text{Halt}(x) = \begin{cases} \text{true}, & \text{if } \phi_x(x) \downarrow \text{ (i.e. } x \in W_x\text{)} \\ \text{false}, & \text{if } \phi_x(x) \uparrow \text{ (i.e. } x \notin W_x\text{)} \end{cases}$$

Idea: By contradiction: we show that assuming $\text{Halt}(x)$ decidable, we can prove f computable. To achieve this, we use the diagonal method constructing a total function which is different from every computable function, yet such that if Halt is computable, so is f . In doing so, we use the universal machine.

$$f(x) = \begin{cases} \phi_x(x) + 1, & \cancel{\text{if } \phi_x(x) \downarrow} \\ 0, & \text{otherwise} \end{cases} = \begin{cases} \Psi_U(\pi, x) + 1, & \text{if } \text{Halt}(x) \\ 0, & \text{otherwise} \end{cases}$$

*↑ 1 if Halt(x)
0 otherwise*

The equality just written is wrong: when $\phi_x(x) \uparrow$ then $\Psi_U(x, x) + 1 \Rightarrow$ the expression is \uparrow

Instead (basically; encoding of tuples of all the components, where using bounded minimalisation we express the computability of the current computation).

$$f(x) = \mu(t, y) . (S(x, x, y, t) \wedge z = y + 1 \wedge \text{Halt}(x)) \vee$$

$$\begin{aligned}
 & (z = 0 \wedge \neg Halt(x) \rightarrow 0)_z \\
 &= \mu w . (s(x, x, (w)_2, (w)_1) \wedge (w)_3 = (w)_2 + 1 \wedge Halt(x)) \vee \\
 & \quad ((w)_3 = 0 \wedge \neg Halt(x))_3
 \end{aligned}$$

$$(w)_1 = t, (w)_2 = y, (w)_3 = z$$

If you call (in words: the combination of the previous conditions in a function)

$$\begin{aligned}
 Q(x, w) \equiv & (s(x, x, (w)_2, (w)_1) \wedge (w)_3 = (w)_2 + 1 \wedge Halt(x)) \vee \\
 & ((w)_3 = 0 \wedge \neg Halt(x))
 \end{aligned}$$

decidable.

$$= (\mu w. |X_Q(x, w) - 1|)_3$$

(in words: bounding the minimalisation of the characteristic function, which seems computable because it combines also computable functions, but it is not, since from the beginning the function is not computable, hence the problem is not decidable)

computable as it arises as minimalisation of computable \Rightarrow contradiction $\Rightarrow Halt(x)$ not decidable.

Exercise: Let $Q(x)$ be a decidable predicate, $f_1, f_2: \mathbb{N} \rightarrow \mathbb{N}$ computable and define:

$$f(x) = \begin{cases} f_1(x), & \text{if } Q(x) \\ f_2(x), & \text{if } \neg Q(x) \end{cases}$$

We want to prove this is computable.

Proof

If f_1, f_2 total

$$\begin{array}{ccc}
 f(x) = f_1(x) * X_Q(x) + f_2(x) + X_{\neg Q}(x) & & \\
 \text{if } Q(x) & \downarrow & \downarrow \\
 & 1 & 0 \\
 \text{if } \neg Q(x) & \downarrow & \downarrow \\
 & 0 & 1
 \end{array}$$

$\Rightarrow f$ computable (in words: we use the bounded product combining the above functions)

In general, let $e_1, e_2 \in N$ s.t. $\phi_{e_1} = f_1$ and $\phi_{e_2} = f_2$

$$\begin{aligned}
 f(x) &= \mu(t, y). (S(e_1, x, y, t) \wedge Q(x)) \vee \\
 &\quad (S(e_2, x, y, t) \wedge \neg Q(x)) \rightarrow y \\
 &= \mu w. (s(e_1, x, (w)_2, (w)_1) \wedge Q(x)) \vee \\
 &\quad (s(e_2, x, (w)_1, (w)_2) \wedge \neg Q(x))_2 \\
 &\quad \text{decidable}
 \end{aligned}$$

$\Rightarrow f$ is computable

(in words: we define two components of the primitive recursion happening above and we bound the composition of the sum on the two combinations of such function, using the smn-theorem. Combining computable functions, basic operations and bounded minimalisation over the recursion, thanks to the universal machine, the indices will give us computable outputs, hence this is computable and decidable).

Exercise

$$f(x) = \begin{cases} 0, & \phi_x(x) \uparrow \\ \uparrow, & \phi_x(x) \downarrow \end{cases} \text{not computable}$$

If $Halt(x)$ is decidable, then f is decidable

Exercise

Let's start from totality and define this predicate:

$$Tot(x) = "\phi_x \text{ is total}" \equiv "P_x \text{ is terminating on every input}"$$

is undecidable.

In fact (in words: we use diagonalization as happened other times to give a computation different from all total computable function, given the primitive recursion is total and will be not equal to the following value. This leads to a contradiction, because if the predicate is decidable so it would be the function, which is not):

$$\begin{aligned} f(x) &= \begin{cases} \phi_x(x) + 1, & \text{if } Tot(x) \\ 0, & \text{otherwise} \end{cases} \\ \rightarrow f \text{ is total} &\quad \begin{cases} Tot(x) \Rightarrow f(x) = \phi_x(x) + 1 \Downarrow \\ \neg Tot(x) \Rightarrow f(x) = 0 \end{cases} \\ \rightarrow f \text{ is different from all total computable functions} & \\ (\text{if } \phi_x \text{ is total} \Rightarrow f(x) = \phi_x(x) + 1 \neq \phi_x(x)) & \\ \Downarrow & \\ f \text{ not computable} & \end{aligned}$$

If we assume that $Tot(x)$ is decidable, we derive f computable \rightarrow contradiction

$$f(x) = \begin{cases} f_1(x), & \text{if } Tot(x) \\ f_2(x), & \text{if } \neg Tot(x) \end{cases}$$

where:

$$\begin{aligned} f_1, f_2: N &\rightarrow N \\ f_1(x) &= \phi_x(x) + 1 = \Psi(x, x) + 1 \forall x \\ f_2(x) &= 0 \forall x \quad \text{computable} \end{aligned}$$

\Rightarrow by the previous exercise, f is computable, which is absurd $\Rightarrow Tot(x)$ not decidable.

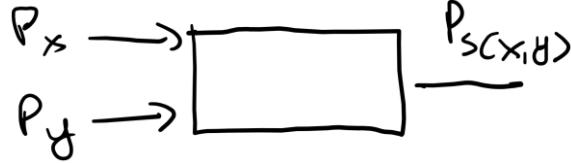
(in words: $f(x)$ is essentially saying, "If the Turing machine represented by ϕ_x halts on every input, return the value of the machine plus 1. Otherwise, return 0." Now, we assume that $Tot(x)$ is decidable, meaning we can reliably say whether ϕ_x halts on every input or not. We run into a contradiction because we can show that $f(x)$ is computable under this assumption).

13.3 EFFECTIVE OPERATIONS OF COMPUTABLE FUNCTIONS

The existence of the universal function, together with the smn-theorem, allows us to formalize operations that manipulate programs and derive their effectiveness. This allows us to combine domains of functions and get indices which can be computed and obtained effectively, allowing us to get effective operations.

- 1) *Effectiveness of product:* there exists a total computable function $S: \mathbb{N}^2 \rightarrow \mathbb{N}$

$$\forall x, y \quad \phi_{S(x,y)}(z) = \phi_x(z) * \phi_y(z) \quad \forall z$$



def $P_{S(x,y)}(z)$:
 $v_x = P_x(z)$
 $v_y = P_y(z)$
return $v_x * v_y$

Proof

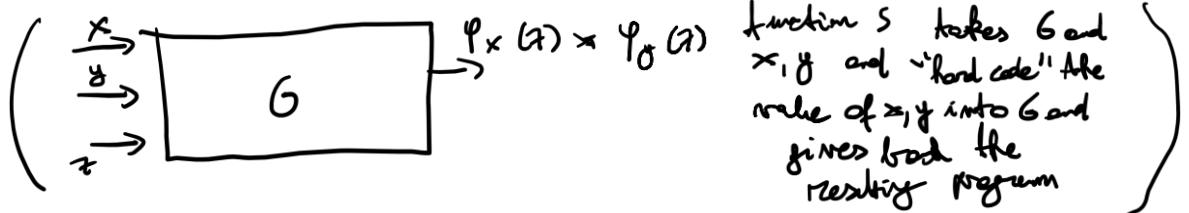
We define $g: \mathbb{N}^3 \rightarrow \mathbb{N}$

$$\begin{aligned} \phi_{S(x,y)}(z) &= g(x, y, z) = \phi_x(z) * \phi_y(z) \\ &= \Psi_U(x, z) * \Psi_U(y, z) \end{aligned}$$

g is computable (composition of computable functions)

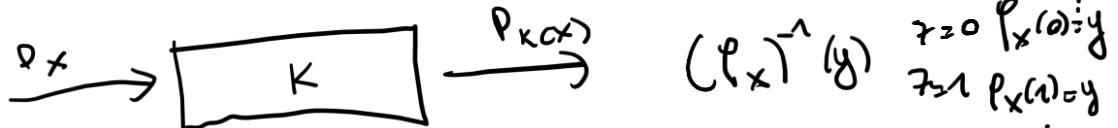
Hence (by [corollary](#) of) smn-theorem, there is $S: \mathbb{N}^2 \rightarrow \mathbb{N}$ total computable such that

$$\phi_{S(x,y)}(z) = g(x, y, z) = \phi_x(z) * \phi_y(z)$$



- 2) *Effectiveness of the inverse function*

There is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x$ if ϕ_x is injective then $\phi_{k(x)} = (\phi_x)^{-1}$



Proof

Define

$$\begin{aligned} g(x, y) &= (\phi_x)^{-1}(y) = \begin{cases} z, & \exists z \text{ s.t. } \phi_x(z) = y \\ \uparrow, & \text{otherwise} \end{cases} \\ &\quad (\text{if } \phi_x \text{ is injective}) \\ &= (\mu z. S(x, z, y, t))_2 \\ &= (\mu w. S(x, (w)_1, y, (w)_2)_1 \\ &= \mu w. (|X_s(x, (w)_1, y, (w)_2) - 1|)_1 \end{aligned}$$

computable

(in words: given the inverse function, we get a value from it, and we can map it back injectively to get the original value. Bounded minimalisation ensures, in combination with smn-theorem, that we can parametrize two functions over x and y , hence bounding a finite value via the characteristic function. This results in something computable).

Hence, by smn-theorem, there is $K: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$\phi_{K(x)}(y) = g(x, y) = (\phi_x)^{-1}(y) \quad \text{if } \phi_x \text{ injective}$$

What do we get when ϕ_x is not injective? $\phi_{K(x)}$ is one of the counterimages of y .

Question: Can you get the least counterimage? To put it more explicitly:

Given $f: \mathbb{N} \rightarrow \mathbb{N}$ computable, define $g(y) = \begin{cases} \min\{x \mid f(x) = y\}, & \text{if } \exists x \text{ s.t. } f(x) = y \\ \uparrow, & \text{otherwise} \end{cases}$

Is g computable?

Exercise to do here (tip: define minimalisation before finding the 0, for the current function and all arguments before the current one)

Possible solution (to take with a grain of salt)

We use minimalisation defining a computable function:

$$g(x, y) = \begin{cases} z, & \exists z \text{ s.t. } \phi_x(z) = y \\ \uparrow, & \text{otherwise} \end{cases}$$

And then we apply, defining a function S over the parameters via smn-theorem:

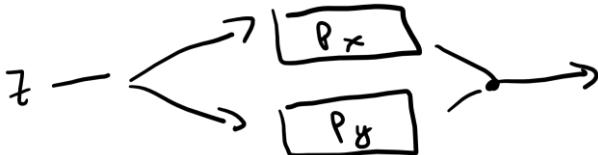
$$g(x, y) = (w.S(x, z, y, t))_2 = \mu(w. (|X_s(x, (w)_1, y, (w)_2)|)_1$$

which is computable and via smn-theorem corollary:

$$\phi_{S(x,y)}(z) = g(x, y) = \begin{cases} 1, & \text{if } \phi_x(z) = y \\ \uparrow, & \text{otherwise} \end{cases}$$

Exercise: There is a total computable function $S: \mathbb{N}^2 \rightarrow \mathbb{N}$ s.t. $W_{S(x,y)} = W_x \cup W_y$

$$\phi_{S(x,y)}(z) \downarrow \text{iff } \phi_x(z) \downarrow \text{or } \phi_y(z) \downarrow$$



We define a function $g: \mathbb{N}^3 \rightarrow \mathbb{N}$ s.t.

$$g(x, y, z) = \begin{cases} 1, & \text{if } \phi_x(z) \downarrow \text{ or } \phi_y(z) \downarrow \text{ (in other words } z \in W_x \vee z \in W_y) \\ \uparrow, & \text{otherwise} \end{cases}$$

$$= \mathbf{1}(\mu t . H(x, z, t) \vee H(y, z, t))$$

where $\mathbf{1}(x) = 1 \forall x$ (keep in mind the bold $\mathbf{1}$ is the "indicator function"; we use it because it can be true either for x or for y)

g is computable and thus by smn-theorem $\exists s: \mathbb{N}^2 \rightarrow \mathbb{N}$ total computable s.t.

$$\phi_{S(x,y)}(z) = g(x, y, z) = \begin{cases} 1, & \text{if } \phi_x(z) \downarrow \text{ or } \phi_y(z) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

S is the desired function (this part has borders drawn in red because quoting the professor "in exam you don't write this, it's clear S has the desired properties").

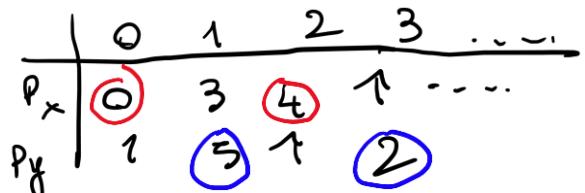
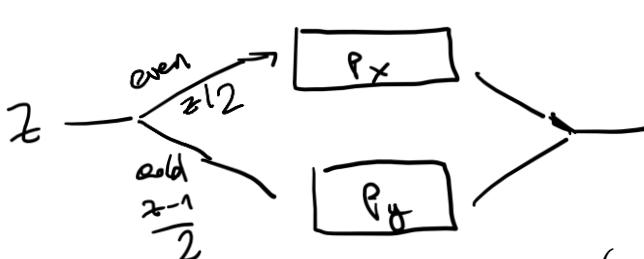
$$\begin{aligned} z \in W_{S(x,y)} &\text{ iff } \phi_{S(x,y)}(z) \downarrow && \text{iff } \phi_x(z) \downarrow \phi_y(z) \downarrow \\ &\text{ iff } z \in W_x \text{ or } z \in W_y && \\ &\text{ iff } z \in W_x \cup W_y && \end{aligned}$$

Exercise

There is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$$E_{S(x,y)} = E_x \cup E_y$$

($P_{S(x,y)}$ produces as outputs all values produced by P_x and P_y and to do this, we simulate ϕ_x on even numbers and ϕ_y on odd numbers – what the drawing wants to say). We define a function $g: \mathbb{N}^3 \rightarrow \mathbb{N}$:



$$g(x, y, z) = \begin{cases} \phi_x\left(\frac{z}{2}\right), & z \text{ even} \\ \phi_y\left(\frac{z-1}{2}\right), & z \text{ is odd} \end{cases}$$

$$\cancel{\Psi_U(x, qt(2, z)) * \overline{sg}(rm(2, z)) +}$$

$$\cancel{\Psi_U(y, qt(2, z)) * rm(2, z)}$$

$$= \mu(v, t). (S\left(x, \frac{z}{2}, v, t\right) \wedge z \text{ even}) \vee$$

$$(S\left(y, \frac{z-1}{2}, v, t\right) \wedge z \text{ odd})_v \vee$$

$$= (\mu w. (S(x, qt(2, z), (w)_1, (w)_2) \wedge z \text{ even}) \vee$$

$$(S(y, qt(2, z), (w)_1, (w)_2) \wedge z \text{ odd}))_1$$

desirable

One may be tempted to use composition over functions and decidable predicate this way, which does not work, because one or both functions can be undefined (hence, we mark them in red)

Here we express the program with bounded minimalisation and so we cover all input cases and express all possibilities for z even or odd. This way, we can ensure that is decidable, given the bounded minimalisation.

An alternative might include the use of \max function and remainder, which ensure we take the bounded maximum for the even and odd division, making it computable:

$$(\mu\omega . \quad | \max\{\chi_S(x, qt(2, z), (\omega)_1, (\omega)_2) \cdot \overline{sg}(rm(2, z)), \\ \chi_S(y, qt(2, z), (\omega)_1, (\omega)_2) \cdot sg(rm(2, z))\} - 1|)_1$$

By smn-theorem $\exists s: \mathbb{N}^2 \rightarrow \mathbb{N}$ total computable s.t. $\forall x, y, z$:

$$\phi_{S(x,y)}(z) = g(x, y, z) = \begin{cases} \phi_x\left(\frac{z}{2}\right), & z \text{ even} \\ \phi_y\left(\frac{z-1}{2}\right), & z \text{ odd} \end{cases}$$

I claim that s is the desired function, i.e. $E_{S(x,y)} = E_x \cup E_y$

$(\subseteq) v \in E_{S(x,y)}$

$$\exists z \text{ s.t. } \phi_{S(x,y)}(z) = v \\ \uparrow \phi(x, y, z)$$

hence two possibilities (whatever happens, we are in the codomain, and we still remain inside the union).

$$\left. \begin{array}{l} \bullet \quad v = \phi_x\left(\frac{z}{2}\right) \rightarrow v \in E_x \\ \bullet \quad v = \phi_y\left(\frac{z-1}{2}\right) \rightarrow v \in E_y \end{array} \right\} \rightarrow v \in E_x \cup E_y$$

$(\supseteq) v \in E_x \cup E_y \rightarrow v \in E_{S(x,y)}$

i.e. (1) $v \in E_x \rightarrow v \in E_{S(x,y)}$

(2) $v \in E_y \rightarrow v \in E_{S(x,y)}$

They are analogous and we prove only one; assume we are in the codomain of x :

1) $v \in E_x$ i.e. $\exists z \text{ s.t. } \phi_x(z) = v$

Therefore: $\phi_{S(x,y)} = (2z) = \phi_x\left(\frac{2z}{2}\right) = \phi_x(z) = v \rightarrow v \in E_{S(x,y)}$
even

2) identical (as said, the computations are the same here as last point)

Exercise: variant of URM machine in which you remove the successor, and you insert the predecessor:

$$\begin{array}{ll} URM^P & z(n) \\ \cancel{S(n)} & P(n) \quad r_m \leftarrow r_{m-1} \\ T(m, n) & \\ J(m, n, t) & \end{array}$$

e_p URM^P -computable function. What is the class of relationship between them?

$$C_p \stackrel{?}{=} C$$

Possible solution (to take with a grain of salt)

Consider the simulation of URM^P for URM normal machine to simulate the predecessor, which is not present in URM. In this case, you can consider a combination of both transfer and jump instructions. The predecessor should have the same configuration as current register minus one step of computation. So, we just need to jump back and transfer the content back:

$$J(m, n, t + 1)$$

$$T(m, n - 1)$$

We will be unlucky here; still, there will always be predecessor instructions, because the model is not powerful enough. So, the inclusion $C \not\subseteq C^P$ holds.

In the case of URM^P simulating the successor, we're basically doing the same thing forward:

$$J(m, n, t + 1)$$

$$T(m, n + 1)$$

Exercise: Are there $f, g: N \rightarrow N$ functions s.t.

- 1) f computable, g not computable, $f \circ g$ computable (composition of those)
- 2) f not computable, g not computable, $f \circ g$ computable

Tip: the answer is “yes” to both (computability is preserved by composition, non-computability is not preserved by composition).

Possible solution (to take with a grain of salt)

The function f can be something simple like:

$$f(x) = \begin{cases} \frac{(x+1)}{2}, & x > 0 \\ \uparrow, & \text{otherwise} \end{cases}$$

While g can be:

$$g(x) = \begin{cases} \phi_x(x) + 1, & x \in W_x \\ x, & x \notin W_x \end{cases}$$

In the second one, we have $\phi_x(x) \neq f(x) \forall x \in N$, hence this is not computable.

In this case, the composition holds computable, defining a predicate X_K which is $g = x_K$ and:

$$f \circ g(x) = \begin{cases} 0, & x \leq 0 \\ X_k, & \text{otherwise} \end{cases}$$

13.4 OTHER EXERCISES SOLVED IN LESSONS

Exercise

URM^P instructions

$$Z(n)$$

$$T(m, n)$$

$$J(m, n, t)$$

$$S(n) \quad P(n) \quad r_n \leftarrow r_n - 1 = \begin{cases} 0, & r_n = 0 \\ r_n - 1, & r_n > 0 \end{cases}$$

We conjecture if $C^P \subseteq C$ is true and holds or not.

Instead, we will understand from the first part of the following part that the inclusion is strict, so we'll show $C^P \subset C$. Let's structure the proof.

$$(C^P \subseteq C)$$

Given a program P in URM^P :

~~A $\rightarrow P(m)$~~
 ~~$S(1, 1, SUB)$~~
 :
 :



$SUB: J(m, m+1, k-m)$
 $S(m+2)$

$LOOP: J(m, m+2, 2SS)$
 $S(m+1)$
 $S(m+2)$
 $J(1, 1, LOOP)$

$RSS: T(m+1, m)$

Formal proof: using $Z(m), S(m), T(m, m), S(m, m+1), J(1, 1, t+1)$

For every program P of URM^P , for every $k \in N$, there is a URM-program P' s.t. $f_{P'}^{(k)} = f_P^{(k)}$ (proof by induction on the number of predecessors).

There is a problem: if you try to replace the predecessor instruction, you will still have other predecessor instructions recursively (a sort of hybrid between the two languages), so the full inclusion is not respected.

Note (my try on that, so could be wrong, but just to exercise myself):

There is this part on this exercise which makes the proof work not only for this machine but in general. This can be proven inductively too. In general, given as a base case a URM program P ' s.t. $f_{P'}^{(0)} = f_P^{(0)}$, this would be trivially true given the programs are equal.

Inductively, this would be working too, assuming we get $f_p'^m = f_p^m$, computing $f_p'^{(m+1)} = f_p^{m+1}$. Assuming the claim works for $k = m$, we will build a program P' using a register not referenced here, like $q = \max\{\rho(P), k\} + 1$, something like.

```

Start: I1
j: J(1,1,SUB)
l(P): Il(P)
J(1,1,END)
SUB: J(n,q,ZERO)
Z(n)
S(n)
T(m,n)
P(n)
ZERO: J(1,1,j + 1)

```

The program here will contain the h instructions of desired type and will be equally effective, then showing $f_p'^{(k)} = f_p^{(k)}$, which is the desired program.

(Coming back to the exercise)

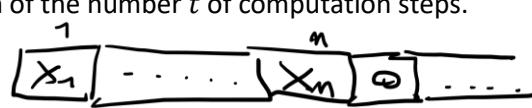
Now we try to prove the other part:



Given a program P of URM^P and $\vec{x} \in N^k$ the maximum value in memory after any number of computation of $P(\vec{x})$ is $\leq \max_{1 \leq i \leq n} x_i$

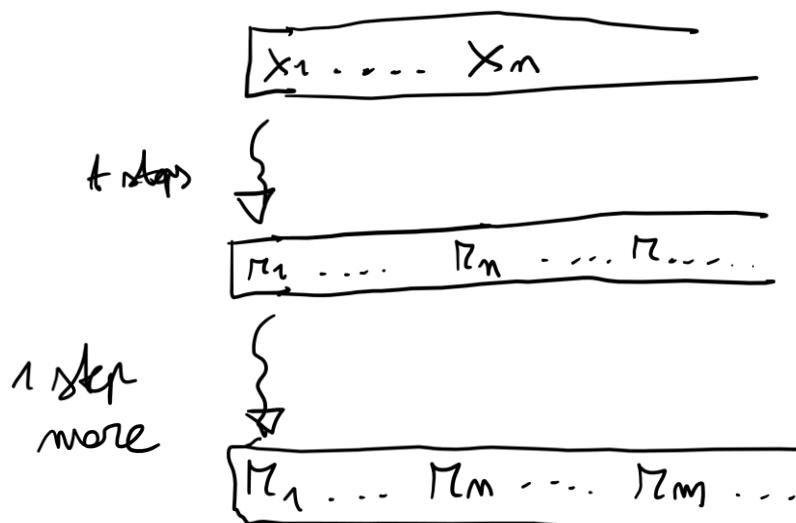
We proceed with a proof by induction of the number t of computation steps.

$(t = 0)$ the memory is:



$$\max_i M_i = \max_{1 \leq i \leq n} x_i$$

$(t \rightarrow t + 1)$ the content of memory after $t + 1$ steps is:



By inductive Hyp.
 $\max_i M_i \leq \max_{1 \leq i \leq k} x_i$

Several cases according to the instruction executed at step $t + 1$:

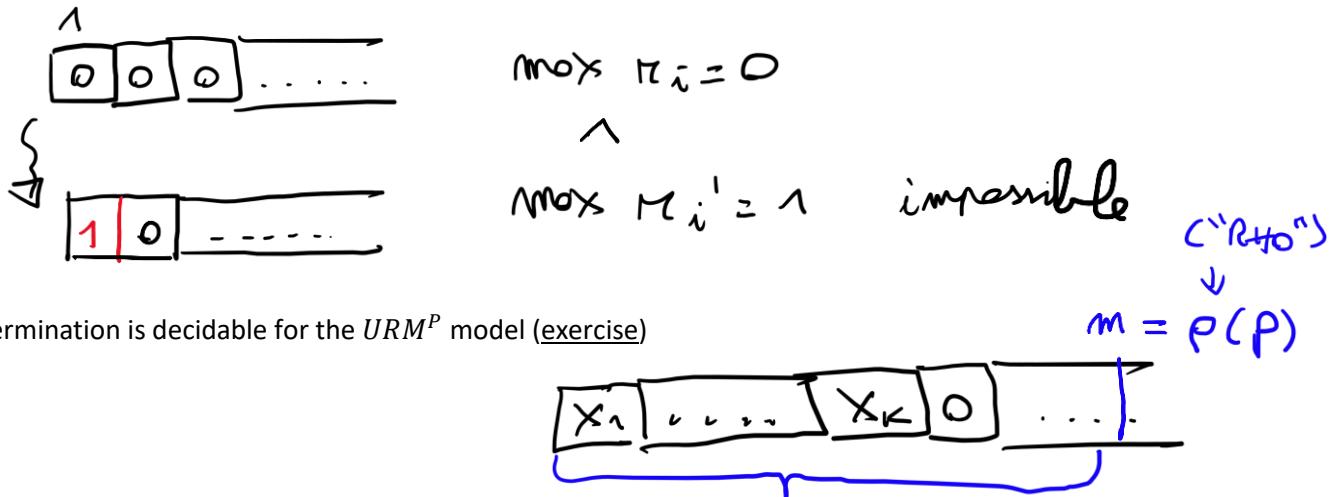
$Z(n)$

$$T(m, n) \quad \max_i r_i \leq \max_i r'_i \leq \max_{1 \leq i \leq n} x_i$$

$J(m, n, t)$

$P(n)$

* The successor $S: N \rightarrow N$ $S(x) = x + 1$ is not URM^P -computable.



My take on the solution (to take with a grain of salt)

To prove the termination is decidable for the URM^P model, we define a function assuming the model is computable, and we define W_x as the input set composed of the instructions we saw before:

$$f(x) = \begin{cases} f(x) + 1, & x \in W_x \\ 0, & x \notin W_x \end{cases} = \begin{cases} \Psi_U(x) + 1, & \text{if } x \in W_x \\ 0, & \text{if } x \notin W_x \end{cases}$$

We can see the problem as:

$$g(x, y) = \begin{cases} 1, & \text{if } \Psi_U(x+1) \\ 0, & \text{otherwise} \end{cases}$$

By the smn-theorem, there is $f: N \rightarrow N$ total and computable s.t.

$$\phi_{f(x)}(y) = g(x, y) = \mathbf{1}(\Psi_U(x+1))$$

We then constructed a total computable function deciding h , which will give as domain the values of the URM^P model and as range its outputs, effectively halting or not. Hence, W_x is decidable and so its model.

Exercise: Show that there is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$E_{K(x)} = W_x$ (in words: set of outputs of the function = values of its domain)

$P_x \rightarrow P_{K(x)}$ with set of outputs

= set of inputs where P_x terminates

$\text{def } P_{K(x)}(y)$
 $P_x(y)$
 $\text{return } y$

THIS IS WHAT
WE WANT
INTUITIVELY

Define $f: \mathbb{N}^2 \rightarrow \mathbb{N}$ as follows:

$$f(x), y = \begin{cases} y, & \text{if } \phi_x(y) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

$$\begin{aligned} &= \mathbf{1}_{\phi_x(y) * y} \\ &\quad \left\{ \begin{array}{ll} 1 & \text{if } \phi_x(y) \downarrow \\ \uparrow & \text{otherwise} \end{array} \right. \\ &= \mathbf{1}(\Psi_U(x, y)) * y \end{aligned}$$

$$\mathbf{1}(x) = 1 \forall x$$

computable
(composition of
computable functions)

Hence, by smn-theorem, there is $k: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t.

$$\phi_{K(x)}(y) = f(x, y) = \begin{cases} y, & \text{if } \phi_x(y) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

(in words: partial recursion allows us to cover all cases for true/false thanks to characteristic function, then the smn-theorem allows us to get the function in two parameters as above, effectively terminating).

* k is the desired function $W_x = E_{K(x)}$

$(W_x \subseteq E_{K(x)})$ Let $y \in W_x \Rightarrow \phi_x(y) \downarrow \Rightarrow \phi_{K(x)}(y) = f(x, y) = y$

hence $y \in E_{K(x)}$

$(E_{K(x)} \subseteq W_x)$ Let $y \in E_{K(x)}$ i.e. there is $z \in N$ s.t. $\phi_{K(x)}(z) = y$

" $\Psi(x, z)$ "

$\Rightarrow z = y$ and $\phi_x(y) \downarrow \Rightarrow y \in W_x$

(in words: all cases, thanks to partial recursion over ϕ_x allow us to get to y as output, this thanks to smn-theorem)

Exercise: There is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$$W_{K(x)} = P \quad (P \text{ is the set of even numbers})$$

$$E_{K(x)} = \{y \in N \mid y \geq x\}$$

Define $f: \mathbb{N}^2 \rightarrow \mathbb{N}$ as:

$$f(x, y) = \begin{cases} x + \frac{y}{2}, & \text{if } y \text{ is even} \\ \uparrow, & \text{otherwise} \end{cases}$$

$$= x + qt(z, y) + \underbrace{\mu w. rm(z, y)}_{\begin{array}{l} \textcircled{0} \text{ when } y \text{ is even} \\ \uparrow \text{otherwise} \end{array}}$$

computable.

Hence, by the smn-theorem, there is $k: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$\phi_{K(x)}(y) = f(x, y) = \begin{cases} x + \frac{y}{2}, & \text{if } y \text{ even} \\ \uparrow, & \text{otherwise} \end{cases}$$

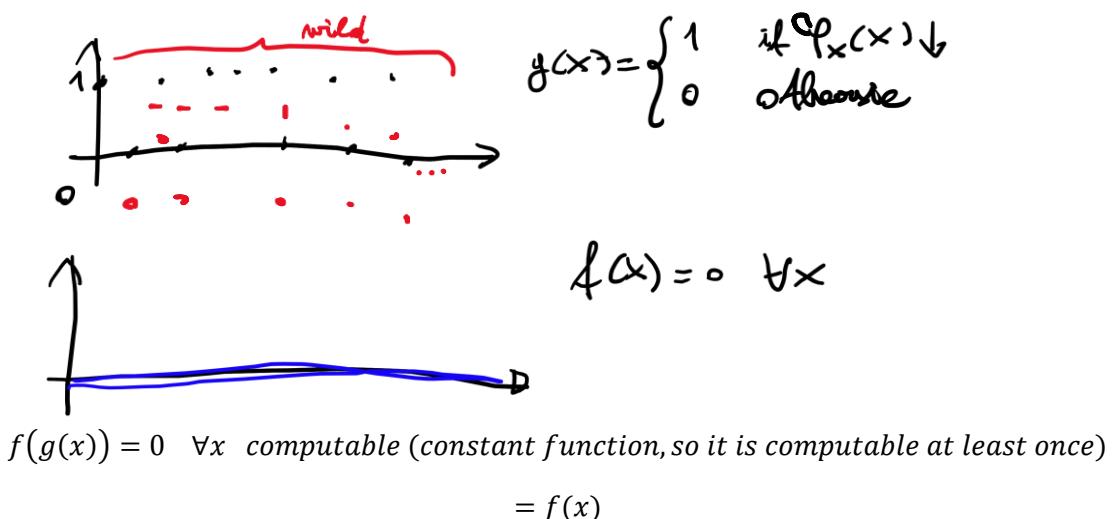
k is the desired function:

$$\begin{aligned} \rightarrow W_{K(x)} &= P \quad (\text{ok}) \\ \rightarrow E_{K(x)} &\doteq \{y \mid y \geq x\} \\ E_{K(x)} &= \{\phi_{K(x)}(y) \mid y \in W_x\} \\ &= \{\phi_{K(x)}(y) \mid y \in P\} \\ &= \{\phi_{K(x)}(2z) \mid z \in N\} \\ &= \left\{x + \frac{y}{2} \mid z \in N\right\} \\ &= \{x + z \mid z \in N\} \\ &= \{y \mid y \geq x\} \end{aligned}$$

Exercise

Are there f, g with f computable, g not computable s.t. $f \circ g$ computable?

The usual mistake here is constructing a function with some non-computable parts and assuming this will be not computable. The function is not computable because it's wildly irregular, as shown here:



Then for the second part of the exercise:

- Are there f, g with f not computable, g not computable s.t. $f \circ g$ computable?

$$- g(x) = \begin{cases} 1, & \phi_x(x) \downarrow \\ 0, & \text{otherwise} \end{cases} \quad \text{not computable}$$

$$f(x) = \begin{cases} 0, & \text{if } x \leq 1 \\ \phi_x(x) + 1, & \text{if } x > 1 \text{ and } \phi_x(x) \downarrow \\ 0, & \text{if } x > 1 \text{ and } \phi_x(x) \uparrow \end{cases}$$

not computable. This is intuitive, but to be more formal:

$$\begin{aligned} f &\neq \phi_x \quad \forall x > 1 \\ (\forall y \quad \exists x > 1 \text{ s.t. } \phi_x = \phi_y \rightarrow f \neq \phi_x = \phi_y \\ \rightarrow f &\text{ different from all computable functions}) \end{aligned}$$

but

$$f(g(x)) = 0, \quad \forall x \text{ computable!}$$

Exercise: Show that every computable function f can be obtained as the composition of two non-computable functions g, h .

My take on the solution (to take with a grain of salt)

Let's assume $h(x) = f(g(x))$ as computable, having the underlying function $g: N \rightarrow N$ as follows:

$$g(x) = \begin{cases} \phi_x(x) + 1, & \text{if } x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

The function is not computable, given $\phi_x(x) + 1 \neq \phi_x$, so the composition will be

$$h(x) = \begin{cases} \phi_{g(x)} + 1, & \text{if } x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

Both will not be computable; define an arbitrary computable function such as:

$$f(x) = \begin{cases} 2x + 1, & x > 0 \\ 0, & \text{otherwise} \end{cases}$$

Since g ignores its inputs, $(h \circ g)$ just effectively applies f to some constant y .

So by choosing a simple computable f , we get a computable function from the composition of non-computable g and h .

Exercise: Prove that $\text{pow2}: N \rightarrow N$, $\text{pow}_2(x) = 2^x$ is PR

by using only the definition of PR.

(least class of functions including basic functions (zero, successor, projections) and closed under composition and primitive recursion)

One way can be observing the sum is primitive recursive and using:

$$\begin{aligned} x+y &\quad \begin{cases} x+0=x \\ x+(y+1)=(x+y)+1 \end{cases} \\ x*y &\quad \begin{cases} x*0=0 \\ x*(y+1)=(x*y)+x \end{cases} \\ x^y &\quad \begin{cases} x^0=1=\text{succ}(0) \\ x^{y+1}=(x^y)*x \end{cases} \quad \text{succ}: N \rightarrow N \\ \text{pow2}(x) &= 2^y = \text{succ}(\text{succ}(0))^y \quad \text{succ}(x) = x+1 \end{aligned}$$

Alternatively, one can use primitive recursion directly:

$$\begin{cases} \text{pow2}(0) = 2^0 = 1 = \text{succ}(0) \\ \text{pow2}(y + 1) = 2^{y+1} = 2^y + 2^y = \text{pow2}(y) + \text{pow2}(y) \end{cases}$$

and observe that $+$ (the sum) is in PR.

Alternatively, you don't need the full sum:

$$\begin{aligned} & \begin{cases} \text{pow2}(0) = 2^0 = 1 = \text{succ}(0) \\ \text{pow2}(y + 1) = 2^{y+1} = \text{twice}(2^y) \end{cases} & \text{twice: } N^1 \rightarrow N^1 \\ & \begin{cases} \text{twice}(0) = 0 \\ \text{twice}(y + 1) = \text{twice}(y) + 2 = \text{succ}(\text{succ}(\text{twice}(y))) \end{cases} \end{aligned}$$

Exercise

$$X_P \in \mathcal{PR}$$

$$X_P(x) = \begin{cases} 1, & \text{if } x \text{ is even} \\ 0, & \text{otherwise} \end{cases}$$

$$X_P(x) = \overline{\text{sg}}(\text{rm}(2, x))$$

: ↙ complex!

directly

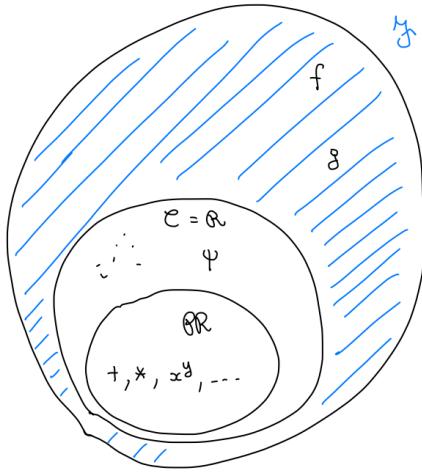
$$\begin{cases} X_P(0) = 1 \\ X_P(y + 1) = \overline{\text{sg}}(X_P(y)) \end{cases}$$

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

14 RECURSIVE SETS

Until now, we defined classes of computable/decidable properties and techniques for proving computability, specifically characterizing:

- recursive properties (easy – decidable)
- recursively enumerable properties (less easy – semidecidable [can only answer in positive case])
- non-recursive properties (hard – undecidable [not always you can give an answer])



This we know
are not
computable

$$f(x) = \begin{cases} 1 & \text{if } \varphi_x(x) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

$$g(x) = \begin{cases} 1 & \text{if } w_x \in N \\ 0 & \text{otherwise} \end{cases}$$

We are trying to characterise mathematically classes of undecidable predicates/non-computable functions, giving a structure to the class of non-computable functions and sort out general classes of problems which do not admit an algorithmic solution. Every interesting property of program behavior is not computable (correctness, absence of bugs, termination).

Specifically, we are focusing on the sets of numbers $x \subseteq \mathbb{N}$ and on the corresponding membership problem " $x \in X?$ "

\uparrow programs

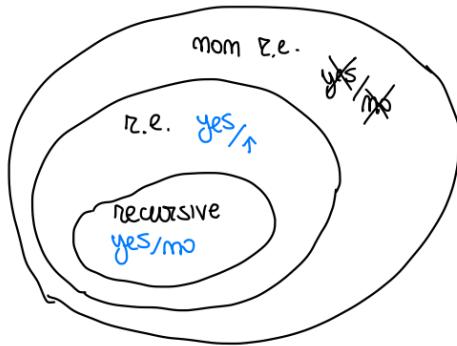
In most cases X will be seen as a set of program codes and thus it can be seen as a program property, e.g.:

- $X = \{x \mid \phi_x = \text{fact}\}$ (factorial function)
- $X = \{x \mid P_x \text{ has linear complexity}\}$ (the program underneath has linear complexity)
- $X = \{x \mid P_x \text{ does not modify } R_j\}$ (the program does not modify the registers)
- $X = \{x \mid P_x \text{ executes each of its instructions for at least one input}\}$
- :

We will discuss and distinguish between:

- *decidable properties/recursive sets*, in which it is possible to answer "yes" or "no" when the property holds or not;
 - o These are those problems for which there exists a corresponding Turing machine that halts on every input with an answer – yes (accepting) or no (rejecting).
- *semidecidable properties/recursively enumerable sets*, in which it is possible to answer "yes" when the property holds or \uparrow when the property does not hold.
 - o These are those problems for which a Turing machine halts on the input accepted by it but can either loop forever or halt on the input which is rejected by the Turing Machine.

Graphically, you can see the correspondence between sets like:



14.1 DEFINITION

Definition (recursive sets)

A set $A \in N$ is defined as recursive if the characteristic function:

$$\chi_A: N \rightarrow N$$

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases} \text{ is computable}$$

(\Leftrightarrow in other words if predicate " $x \in A$ " is decidable) - so, a set A is recursive if and only if there is a verifier program, returning "true" on A and "false" on its complement \bar{A} (partial verifier/semi-verifier)

If $\chi_A \in \mathcal{PR}$ we will say A is *primitively* recursive and the notion can be extended to subsets of N^k .

Examples

The following sets are recursive:

- N recursive, since $\chi_N = 1 \forall x$ computable
- \emptyset recursive, since $\chi_\emptyset(x) = 0 \forall x$ computable
- P (prime numbers set) recursive, given $\chi_P(x) = \overline{\text{sg}}(\text{rm}(2, x)) = \begin{cases} 1, & \text{if } x \text{ is prime} \\ 0, & \text{otherwise} \end{cases}$
- :

* *Observation:* All finite sets $A \subseteq \mathbb{N}$ are recursive.

Proof

Let $A = \{x_0, x_1, \dots, x_k\}$, with $A \subset \mathbb{N}$ and $|A| < \infty$

$$\chi_A(x) = \overline{\text{sg}}\left(\prod_{i=0}^n |x - x_i|\right) \text{ computable}$$

The following set is not recursive:

$$K = \{x \in \mathbb{N} \mid \phi_x(x) \downarrow\} = \{x \in \mathbb{N} \mid x \in W_x\}$$

(from now on, K means the halting set, where programs will terminate). Hence:

$$\chi_K(x) = \begin{cases} 1, & \text{if } \phi_x(x) \downarrow \\ 0, & \text{otherwise} \end{cases} \text{ not computable}$$

Also: $\{x \mid \phi_x \text{ total}\}$ is not recursive.

Observation: Let $A, B \subseteq \mathbb{N}$ recursive sets. Then:

$$1) \overline{A} = \mathbb{N} \setminus A$$

$$2) A \cap B$$

$$3) A \cup B$$

are recursive.

Proof

$$1) X_{\overline{A}}(x) = \begin{cases} 1, & \text{if } x \in \overline{A} \\ 0, & \text{otherwise} \end{cases} = \overline{\text{sg}}(X_A(x)) \text{ computable}$$

$x \notin A$

(2) – (3) [see decidable predicates – you can describe a bounded product/sum of both predicates and will be computable]

14.2 REDUCTION AND RELATED PROBLEMS

Now we will define reduction, a simple but powerful tool when studying the decidability status of problems, formalizing the idea of having a problem A which you argue is “simpler” than another problem, calling it B . Intuitively:

A reduces to B

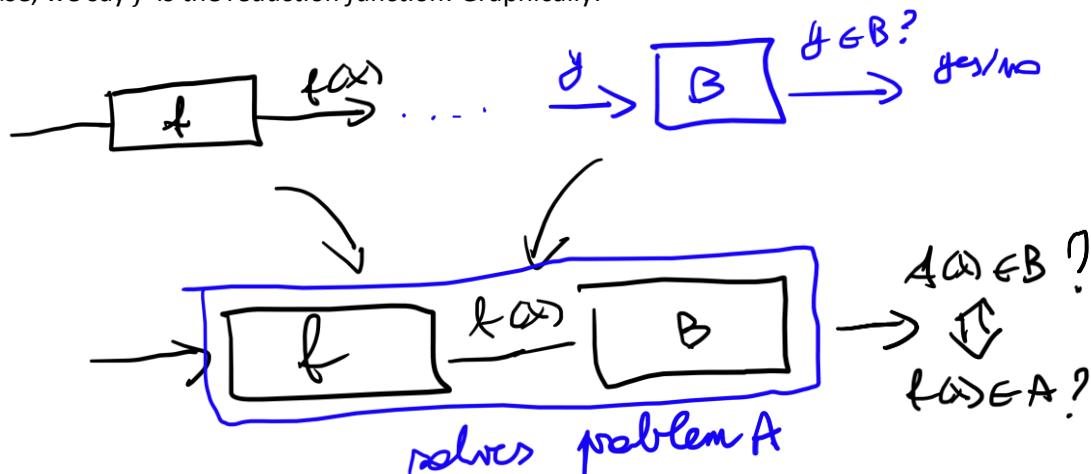
(every instance of A can be transformed **easily** into an instance of B)

Definition (reduction)

Given $A, B \subseteq \mathbb{N}$, we say the problem $x \in A$ reduces to " $x \in B$ " (or simply that A reduces to B) if there is a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$

$$x \in A \text{ iff } f(x) \in B$$

In this case, we say f is the *reduction function*. Graphically:



In this case $A \leq_m B$ (the symbol is read like “which reduces to”, with “m” standing for “mapping”: more [here](#) on notation and meaning. Translate it as “many-one-reducible” or “m-reducible” [also look [here](#)])

Observation

Let $A, B \subseteq \mathbb{N}$, $A \leq_m B$

- 1) if B is recursive then A is recursive
- 2) if A not recursive then B is not recursive

Proof

- 1) Let B recursive

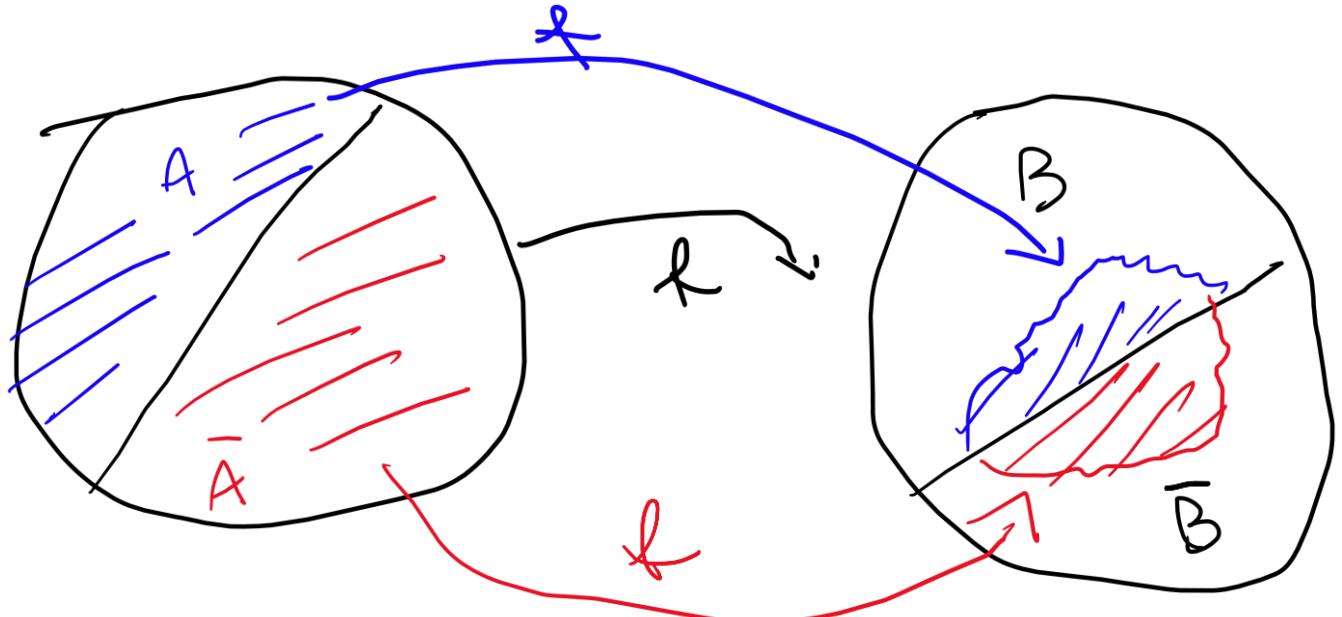
$$\chi_B(x) = \begin{cases} 1, & \text{if } x \in B \\ 0, & \text{otherwise} \end{cases} \text{ computable}$$

since $A \leq_m B$, there is a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \Rightarrow x \in A \iff f(x) \in B$. Then:

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases} = \chi_B(f(x)) \text{ computable by composition}$$

$\rightarrow A$ is recursive (in other words; we simply observe that $\chi_A = \chi_B \circ f$).

- 2) this part is equivalent to (1), given the composition works in both ways.

Example

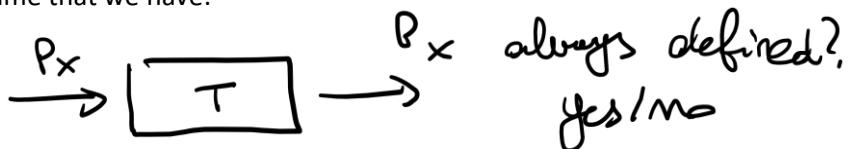
$$k = \{x \mid x \in W_x\} = \{x \mid \phi_x(x) \downarrow\} \text{ not recursive}$$

$$T = \{x \mid W_x = N\} = \{x \mid \phi_x \text{ total}\}$$

$$k \leq_m T ?$$

Proof

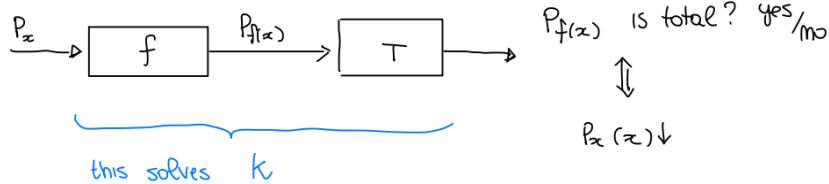
Assume that we have:



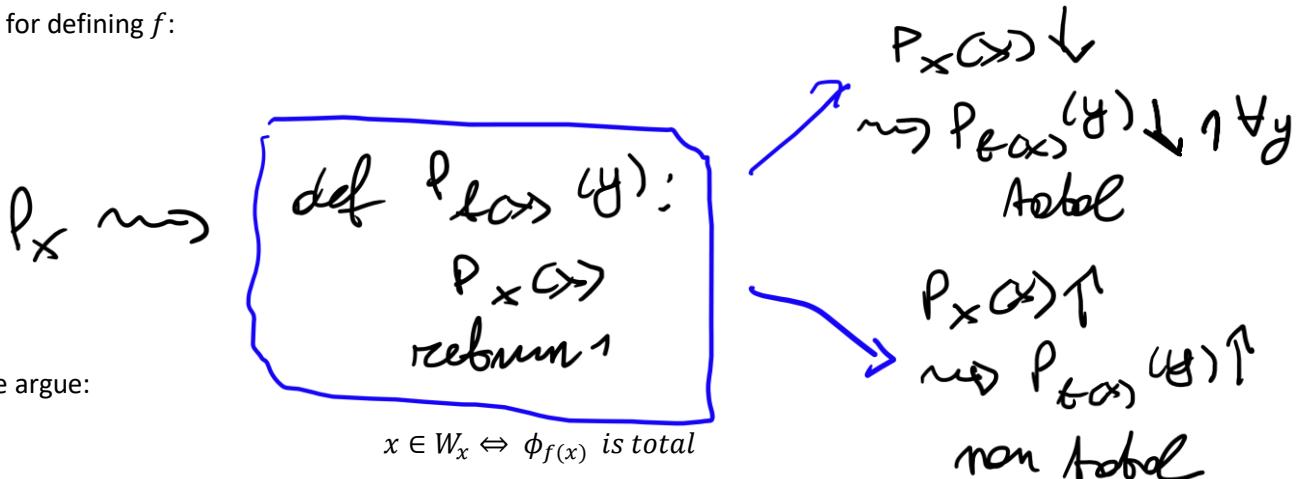
Given P_x we construct $P_{f(x)}$ s.t.

$$P_x(x) \downarrow \text{ iff } P_{f(x)} \text{ is defined everywhere}$$

then we could construct:



The idea for defining f :



Given we argue:

We define:

$$g(x, y) = \begin{cases} 1, & x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

Formally:

$$\begin{aligned} g(x, y) &= \mathbf{1}(\phi_x(x)) & \mathbf{1}(x) &= 1 \quad \forall x \\ &= \mathbf{1}(\Psi_U(x, x)) \end{aligned}$$

By the smn-theorem, there is $f: N \rightarrow N$ total and computable s.t.

$$\phi_{f(x)}(y) = g(x, y) = \mathbf{1}(\phi_x(x)) \quad \forall x, y$$

We claim that $f: N \rightarrow N$ is the reduction function for $K \leq_m T$ i.e. $\forall x, x \in K \iff f(x) \in T$

- if $x \in K \rightarrow f(x) \in T$

$$\begin{aligned} \text{if } x \in K \rightarrow \phi_x(x) \downarrow &\Rightarrow \phi_{f(x)}(y) = 1 \quad \forall y \Rightarrow \\ &\Rightarrow \phi_{f(x)} \text{ total i.e. } f(x) \in T \end{aligned}$$

- if $x \notin K \rightarrow f(x) \notin T$

$$\begin{aligned} \text{if } x \notin K \rightarrow \phi_x(x) \uparrow &\Rightarrow \phi_{f(x)}(y) \uparrow \quad \forall y \Rightarrow \\ &\Rightarrow \phi_{f(x)} \text{ not total i.e. } f(x) \notin T \end{aligned}$$

Therefore, f is the reduction function for $K \leq_m T$ hence, since K not recursive then T is not recursive.

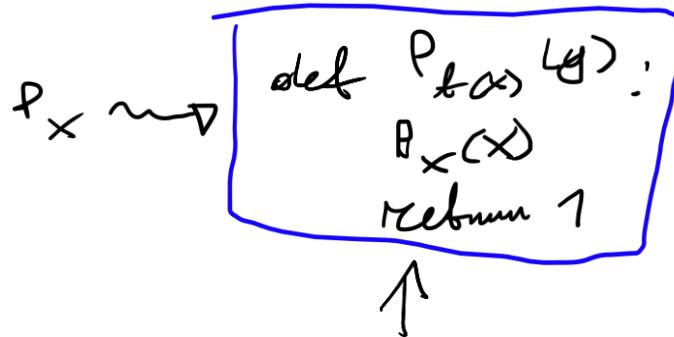
Example (input problem)

Let $n \in N$ fixed. Consider $A_n = \{x \mid \phi_x(n) \downarrow\}$ which is not recursive.

Proof

We will prove $K \leq_m A_n$

We define a function f s.t. $x \in K \Leftrightarrow f(x) \in A_n$ i.e. $x \in W_x \Leftrightarrow \phi_{f(x)}(n) \downarrow$. Ideally:



defined on n iff $P_x(x) \downarrow$

- $P_x(x) \downarrow \Rightarrow P_{f(x)} \downarrow \forall y \text{ in particular } P_{f(x)}(n) \downarrow$
- $P_x(x) \uparrow \Rightarrow P_{f(x)}(y) \uparrow \forall y \text{ in particular } P_{f(x)}(n) \uparrow$

Define $g: N^2 \rightarrow N$:

$$g(x, y) = \mathbf{1}(\phi_x(x)) = \mathbf{1}(\Psi_U(x, x)) \text{ computable}$$

By the smn-theorem there is $f: N \rightarrow N$ total computable s.t.

$$\phi_{f(x)}(y) = g(x, y) = \mathbf{1}(\phi_x(x))$$

The function f is the reduction function for $K \leq_m A_n$

* $x \in K \rightarrow f(x) \in A_n$

if $x \in K$ then $\phi_x(x) \downarrow$. Therefore, $\phi_{f(x)}(y) = \mathbf{1}(\phi_x(x)) = 1 \forall y$. In particular $\phi_{f(x)}(n) \downarrow$, thus $f(x) \in A_n$.

* $x \notin K \rightarrow f(x) \notin A_n$

If $x \notin K$ then $\phi_x(x) \uparrow$. Therefore, $\phi_{f(x)}(y) = \mathbf{1}(\phi_x(x)) \uparrow \forall y$. In particular $\phi_{f(x)}(n) \uparrow$, thus $f(x) \notin A_n$

 $K \leq_m A_n$ since K not recursive, A_n is not recursive

Exercise $A_n \leq_m K$ (home)

My take on the solution (to take with a grain of salt)

Let $n \in N$ fixed. Consider $A_n = \{x \mid \phi_x(n) \downarrow\}$ which is not recursive.

We will prove $A_m \leq_m K$

We define a function f s.t. $x \in A_m \Leftrightarrow f(x) \in W_x \Rightarrow f(x) \in K$. We define f as a function running over a program P_x s.t. $P_x(y) = 1$ if $y = m, \uparrow$ otherwise.

Define $g: N^2 \rightarrow N$:

$$g(x, y) = \mathbf{1}(\phi_x(x)) = \mathbf{1}(\Psi_U(x, x)) \text{ computable}$$

By the smn-theorem there is $f: N \rightarrow N$ total computable s.t.

$$\phi_{f(x)}(y) = g(x, y) = \mathbf{1}(\phi_x(x))$$

The function f is the reduction function for $A_n \leq_m K$

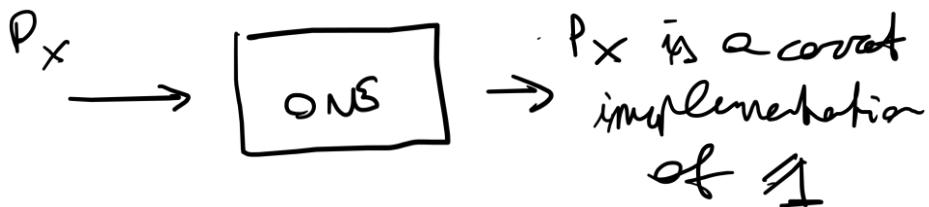
- $x \in A_n \rightarrow f(x) \in K$

If $x \in K$ then $\phi_x(m) \downarrow$. Therefore, $\phi_{f(x)}(y) = \mathbf{1}(\phi_x(m)) = x \forall x$. In particular $\phi_{f(x)}(m) \downarrow$, thus $f(x) \in K$.

- $x \notin A_n \rightarrow f(x) \notin K$

If $x \notin K$ then $\phi_x(m) \uparrow$. Therefore, $\phi_{f(x)}(y) = \mathbf{1}(\phi_x(m)) \uparrow \forall x$. In particular $\phi_{f(x)}(m) \uparrow$, thus $f(x) \notin K$

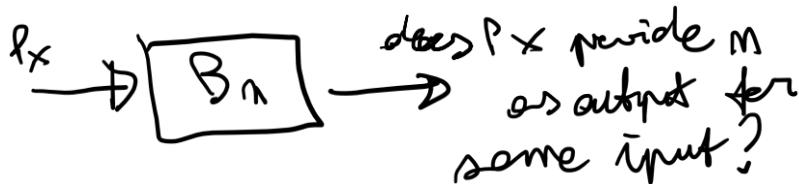
Example: $ONE = \{x \mid \phi_x = \mathbf{1}\}$



$K \leq_m ONE$ same reduction function as before

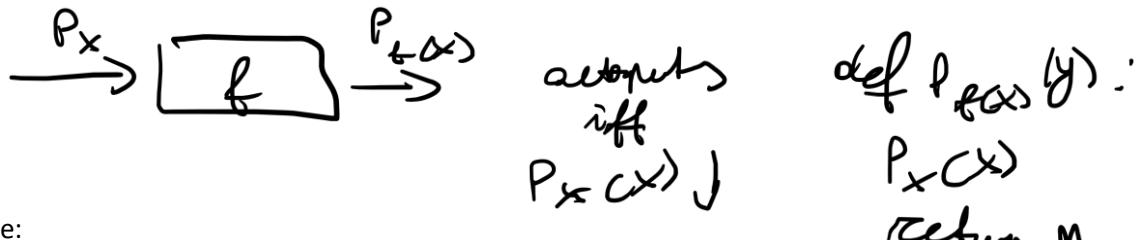
Example (output problem)

Let $n \in N$. Consider $B_n = \{x \mid n \in E_x\}$ not recursive



Show

$$k \leq_m B_n$$



Define:

$$g(x, y) = n * \mathbf{1}(\phi_x(x)) = n * \mathbf{1}(\Psi(x, x)) \text{ computable}$$

By the smn-theorem there is $f: N \rightarrow N$ total computable s.t. $\phi_{f(x)}(y) = g(x, y) = n * \mathbf{1}(\phi_x(x)) \forall x, y$

f is the reduction function for $k \leq_m B_n$

- if $x \in K$ then $\phi_x(x) \downarrow$.

So:

$$\phi_{f(x)}(y) = n * \mathbf{1}(\phi_x(x)) = n \quad \forall y$$

Thus:

$$n \in E_{f(x)} = \{n\}$$

hence $f(x) \in B_n$

- if $x \notin K$ then $\phi_x(x) \uparrow$. Thus

$$\phi_{f(x)}(y) = n * \mathbf{1}(\phi_x(x)) \uparrow \quad \forall y$$

Thus:

$$n \notin E_{f(x)} = \emptyset$$

hence $f(x) \notin B_n$

We conclude $k \leq_m B_n$, hence B_n not recursive.

Exercise

- 1) There exists $f: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$|W_{f(x)}| = 2x$$

$$|E_{f(x)}| = x$$

$\forall x$

Solution

Solution: We can define, for instance,

$$f(x, y) = \begin{cases} qt(2, y) & \text{if } y < 2x \\ \uparrow & \text{otherwise} \end{cases}$$

Observe that $f(x, y) = qt(2, y) + \mu z. (y + 1 - 2x)$ is computable and finally use the smn theorem to get function $s(x)$. \square

- 2) Functions computed by programs which can only jump forward

$$I_i: J(m, n, t) \quad t > i$$

are all total (what if we allow only for backward steps?)

My take on the solution (to take with a grain of salt)

We will prove this inductively on a program P by induction on the instruction i^{th} over the index t .

- Base case: The first instruction has the instruction $t > i > 0$ (at least 1)
- Inductive case: At step $t + 1^{th}$ step, the previous instruction has an index $> t$, so at step $t + 1$, the jump address must be $> t$, therefore the instruction will have index $> t + 1$.

After t steps, the index would definitely be greater than the program length, at $> l(P)$, so the program must halt at the $(t + 1)^{th}$ step of computation, effectively halting the program.

15 SATURATED SETS AND RICE'S THEOREM

Rice's theorem helps showing program properties are not decidable or not recursive, basically.

More in detail, it roughly states that no property of the behaviour of programs which is related to the input/output (besides the obvious ones) is decidable or, in other words, that no non-trivial property of computable functions is decidable. We can see it like this:



We consider properties we already know they are not decidable:

- “ P is terminating on every input ”
 - “ P has some fixed $x \in \mathbb{N}$ as an output ”
 - “ P computes a function f ”
 - ⋮
 - “ the length of program P is ≤ 10 ”
 - ⋮
- }
- decidable
- }
- undecidable

What is a *behavioral property* of a program?

$$\begin{array}{c}
 A \subseteq \mathbb{N} \\
 \uparrow \\
 \text{set of programs (program property)}
 \end{array}$$

Let's give then:

$$\begin{aligned}
 T &= \{n \mid P_n \text{ is terminating on every input}\} \\
 &= \{n \mid \phi_n \text{ is total}\} \\
 &= \{n \mid \phi_n \in T\} \\
 ONE &= \{n \mid P_n \text{ is a sound implementation of } \mathbf{1}\} \\
 &= \{n \mid \phi_n \text{ is } \mathbf{1}\} \\
 &= \{n \mid \phi_n \in \{\mathbf{1}\}\}
 \end{aligned}$$

$A \subseteq \mathbb{N}$ (program property) is a behavioral property if, for all programs $n \in \mathbb{N}$ the facts that $n \in A$ or $n \notin A$ only depends on ϕ_n

(the computed function in I/O, nothing else; so, we might say a behavioral property would be a set of programs that share a common behavioral trait during their execution and their behavior reflects how a program behaves during runtime).

15.1 SATURATED SETS

Definition

A subset $A \subseteq \mathbb{N}$ is saturated (or extensional) if $\forall m, n \in \mathbb{N}$

$$\text{if } m \in A \text{ and } \phi_m = \phi_n \text{ then } n \in A$$

(in words:

- given two programs, if the first program is in the set of programs satisfying the property and two programs are computing the same thing, then also the second program satisfies the property
- this means that if one program with a certain property is in the set, all programs computing the same function must also be in the set)

The property does not depend on the program but on the function it computes. A saturated set contains all the indices (which are infinite) of programs that compute functions with a particular common characteristic.

\Updownarrow

A saturated if $A = \{n \mid \phi_n \text{ satisfies a property of functions}\} = \{n \mid \phi_n \in A\}$

where $A \subseteq F$
 ↓ ↓
 property of functions set of all functions

Examples

- $T = \{n \mid P_n \text{ is terminating on every input}\}$

$$= \{n \mid \phi_n \text{ is total}\}$$

$$= \{n \mid \phi_n \in T\}$$

Where $T = \{f \in F \mid f \text{ total}\}$

- $ONE = \{n \mid P_n \text{ is a sound implementation of } \mathbf{1}\}$ (which simply means “computes **1**”)

$$= \{n \mid \phi_n = 1\}$$

$$= \{n \mid \phi_n \in \{\mathbf{1}\}\}$$

- $LEN10 = \{n \mid P_n \text{ has length } \leq 10\}$

$$m \in LEN10 \quad \text{and } \phi_m = \phi_n$$

$$m \notin LEN10$$

e.g.

$m = \gamma(Z(1)) \in LEN10$ (program with 10 lines; the next one has more lines, but they do the same thing)

$$m = \gamma \left(\begin{array}{c} Z(1) \\ Z(1) \\ \vdots \\ Z(1) \end{array} \right) \geq 11 \notin LEN10$$

$\phi_m = \phi_n = 0$
 constant zero

- $K = \{n \mid \phi_n(n) \downarrow\}$ (this is the halting problem, checking if it terminates over the program code)
 $= \{n \mid \phi_n \in \mathcal{K}\}$

$\mathcal{K} = \{f \mid f(\text{?}) \downarrow\}$??? (here we would like to have a function able to compute the halting problem in the same set, but this does not happen)

It seems that K is not saturated (we can't conclude the proof, because we are not able to define something which "behaves as the halting problem" itself and then given K does not depend on the underlying function, we would like to have "something able to prove it universally").

Formally, I should find $m, n \in N$ s.t.

$$\begin{array}{ll} m \in K & \phi_m(m) \downarrow \\ m \notin K & \phi_n(n) \uparrow \end{array} \quad \text{and } \phi_m = \phi_n$$

(they have different values, but they are computing the same function).

If we were able to show there exists a program $m \in N$ s.t.

$$\phi_m(x) = \begin{cases} 1, & \text{if } x = m \\ \uparrow, & \text{otherwise} \end{cases}$$



we can conclude:

- 1) $m \in K \quad \phi_m(m) \downarrow$
- 2) for a computable function there are infinitely many programs hence there is $n \neq m$ s.t. $\phi_m = \phi_n$
- 3) $n \notin K$

$$\begin{array}{ccc} \phi_n(n) = \phi_m(n) \uparrow & & \\ \swarrow & & \uparrow \\ \phi_n = \phi_m & & n \neq m \end{array}$$

K is not saturated! (hence, they don't compute the same thing)

What about (*)?

$\text{def } P(x):$



*if $x \in P$ then 1
if $x \notin P$ then \uparrow*

*if $x = \text{"def } P(x)}$:
... "*

We check if x corresponds to the program we are defining.

15.2 RICE'S THEOREM

Definition (Rice's theorem)

Let A be a set and $A \neq \emptyset, A \neq \mathbb{N}, A \subseteq \mathbb{N}$. If A is saturated, then A is not recursive.

In simpler terms:

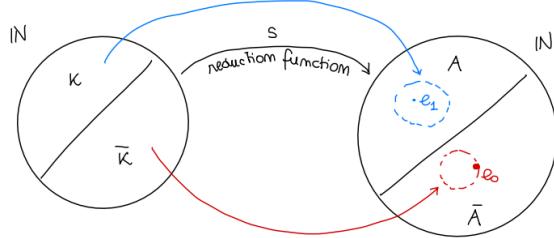
- the saturation property implies that A contains all the indices of programs that compute functions with a common characteristic
 - o this property holds extensionally, meaning it solely depends on the elements within the set without consideration for their internal representations.
- the significance of this in proving non-recursiveness lies in the inherent uncertainty: we cannot definitively determine whether a program possesses a specific property precisely

Proof

We start from the halting problem, making it reducible to A . So:

$K \leq_m A$ (since K is not recursive $\rightarrow A$ is not recursive)

To remember, this happens with reduction “behind the scenes”:



Let e_0 be an index $s.t. \phi_{e_0}(x) \uparrow \forall x$ (program for the function always undefined – consider, as note, we assume $\phi_{e_0} \neq 0$, which could be not true). We distinguish two cases depending on $e \in A$ or $e \notin A$.

1) Assume $e_0 \notin A$

Let $e_1 \in A$ (it exists since $A \neq \emptyset$)

and define:

$$\begin{aligned}
 g(x, y) &= \begin{cases} \phi_{e_1}(y), & \text{if } x \in K \\ \phi_{e_0}(y), & \text{if } x \in \bar{K} \text{ (or } x \notin K\text{)} \end{cases} \\
 &= \begin{cases} \phi_{e_1}(y), & \text{if } x \in K \quad [\phi_x(x) \downarrow] \\ \uparrow, & \text{if } x \in \bar{K} \quad [\phi_x(x) \uparrow] \end{cases} \\
 &= \phi_{e_1}(y) * \mathbf{1}(\phi_x(x)) \\
 &\quad \uparrow \begin{cases} 1 & \text{if } \phi_x(x) \downarrow \\ 0 & \text{otherwise} \end{cases} \\
 &= \phi_{e_1}(y) * \mathbf{1}(\Psi_U(x, x))
 \end{aligned}$$

computable!

By the smn-theorem, there is $s: N \rightarrow N$ total and computable s.t. $\forall x, y:$

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} \phi_{e_1}(y), & \text{if } x \in K \\ \phi_{e_0}(y), & \text{if } x \in \bar{K} \end{cases}$$

s is the reduction function for $K \leq_m A$

- $x \in K \Rightarrow s(x) \in A$

if $x \in K$ then $\phi_{s(x)}(y) = g(x, y) = \phi_{e_1}(y) \quad \forall y$

i.e. $\phi_{s(x)} = \phi_{e_1}$ since $e_1 \in A$ and A saturated $\rightarrow s(x) \in A$

- $x \notin K \Rightarrow s(x) \notin A$

if $x \notin K$ then $\phi_{s(x)}(y) = g(x, y) = \phi_{e_0}(y) \quad \forall y$

i.e. $\phi_{s(x)} = \phi_{e_0}$ since $e_0 \notin A$ and A saturated $\rightarrow s(x) \notin A$

Hence, as expected by our construction, s is the reduction function and since K is not recursive, we deduce A is not recursive either.

2) if instead $e_0 \in A$,

we have $e_0 \notin \bar{A}$

$\bar{A} \subseteq N$ is saturated (since A is)

$\bar{A} \neq \emptyset$ (since $A \neq \emptyset$)

$\bar{A} \neq N$ (since $A \neq N$)

\rightarrow by (1) applied to \bar{A} we deduce \bar{A} not recursive $\rightarrow A$ not recursive either

15.3 EXAMPLES

Example (Output problem)

We proved that $B_n = \{x \mid n \in E_x\}$ and we observed it was not recursive by showing $K \leq_m B_n$. We can conclude the same using:

- B_n is saturated (hence, given codomain=image, they compute the same values)

$$B_n = \{x \mid \phi_x \in B_n\}$$

$$B_n = \{f \mid n \in \text{cod}(f)\}$$

- $B_n \neq \emptyset$ (we get at least one element from natural set, hence it is well-defined and total)

e.g. let $e_1 \in N$ be s.t. $\phi_{e_1}(y) = y \quad \forall y \rightarrow n \in E_{e_1} = N$

$\rightarrow e_1 \in B_n \neq \emptyset$ (so, using the identity, we find all possible numbers as output)

- $B_n \neq N$ (there are always different elements we can map)

e.g. let $e_2 \in N$ s.t. $\phi_{e_2}(y) = m (\neq n) \quad \forall y$

$e_2 \in B_n$ (since $n \neq E_{e_2} = \{m\}$)

Written by Gabriel R.

\Rightarrow by Rice's theorem, B_n is not recursive

Example

$$I = \{x \in \mathbb{N} \mid P_x \text{ has infinitely many possible outputs}\}$$

$$= \{x \in \mathbb{N} \mid E_x \text{ is infinite}\}$$

Is it saturated? Yes, it is. We are not making assumptions over the underlying program, we're only interested in the property of functions and its sets. We will argue what we just wrote:

- I saturated

$$I = \{x \mid \phi_x \in I\}$$

with $I = \{f \mid \text{cod}(f) \text{ infinite}\}$

- $I \neq \emptyset$ (it is not empty)

if e_1 is as previous exercise $\rightarrow E_{e_1} = N$ infinite $\Rightarrow e_1 \in I$

- $I \neq \mathbb{N}$

if e_2 is as before $\rightarrow E_{e_2} = \{m\} \rightarrow e_2 \notin I$

$\Rightarrow I$ not recursive, by Rice's theorem

Example

$$A = \{x \mid x \in W_x \cap E_x\} \text{ (programs halting on their own description)}$$

Is it saturated?

$$A = \{x \mid \phi_x \in A\}$$

$$A = \{f \mid ? \in \text{dom}(f) \cap \text{cod}(f)\}$$

 we do not know what to put here; essentially, the function is computing its own code, or the syntax, halting on the given input in a predictable way. It doesn't mean it isn't saturated, only there is no such function "able to do it universally"

Hence, we will not use Rice. We show $K \leq_m A$ i.e. that there is a total computable function $s: N \rightarrow N$ s.t.

$$x \in K \text{ iff } s(x) \in A$$

\Updownarrow

$$s(x) \in W_{s(x)} \dots \phi_{s(x)} \downarrow$$

and

$$s(x) \in E_{s(x)} \dots \phi_{s(x)}(y) = s(x) \text{ for some } y$$

We define, by the smn-theorem, a function of two arguments as follows (we know nothing about the function discussed before; to be sure, we use a function defined for every natural number and produces code as an output without knowing the code using also universal function; then the opposite case).

$$\begin{aligned} g(x, y) &= \begin{cases} y, & x \in K \\ \uparrow, & \text{otherwise} \end{cases} \\ &= y * \mathbf{1}(\phi_x(x)) \end{aligned}$$

$$= y * \mathbf{1}(\Psi_U(x, x)) \text{ computable}$$

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$\phi_{s(x)} = g(x, y) = \begin{cases} y, & \text{if } x \in K \\ \uparrow, & \text{otherwise} \end{cases} \quad \forall x, y$$

s is the reduction function

$$\rightarrow x \in K \text{ then } \phi_{s(x)}(y) = g(x, y) = y \quad \forall y$$

Hence, $W_{s(x)} = N, E_{s(x)} = N$ and so $s(x) \in W_{s(x)} \cap E_{s(x)} = \mathbb{N}$

Thus, $s(x) \in A$.

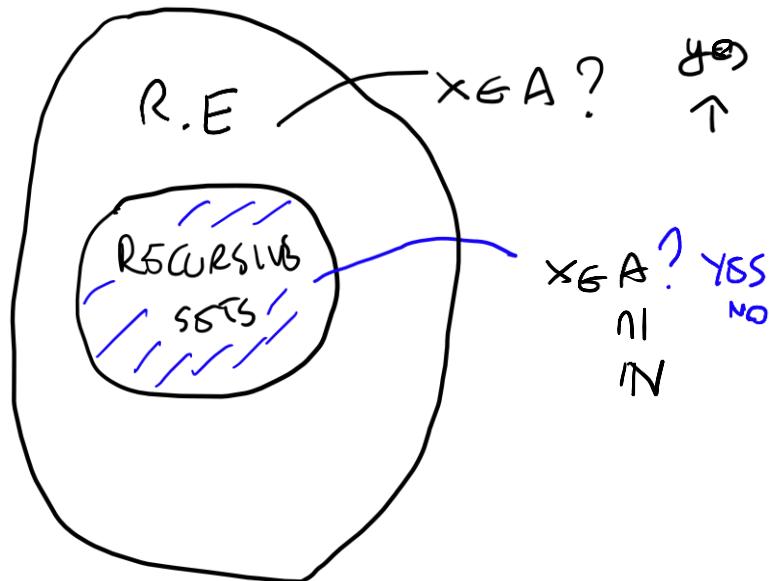
$$\rightarrow x \notin K \text{ then } \phi_{s(x)}(y) = g(x, y) \uparrow$$

Hence, $W_{s(x)} = \emptyset, E_{s(x)} = \emptyset$ and so $s(x) \notin W_{s(x)} \cap E_{s(x)} = \emptyset$

Thus, $K \leq_m A$ and since K is not recursive, also A is not recursive.

16 RECURSIVELY ENUMERABLE SETS

We started looking at recursive sets, which have properties completely satisfiable, either "yes" or "no", with a set of numbers associated to code satisfying the property (and understanding if a number is inside the specific set or not). We then move to the larger class, responding "yes" or "does not exist" for all possible elements.



16.1 DEFINITION

A set $A \subseteq N$ is recursively enumerable (called from now on "r.e.") if the semi-characteristic function $sc_A: N \rightarrow N$:

$$sc_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases} \text{ is computable}$$

- Enumerable since there is surjective function total and computable $f: N \rightarrow N$ s.t. $\text{cod}(f) = A$
- Recursive since the enumeration can be done via a computable function

A property/predicate $Q(\vec{x}) \subseteq N^k$ is semi-decidable if

$$sc_Q: N^k \rightarrow N$$

$$sc_Q(\vec{x}) = \begin{cases} 1, & \text{if } Q(\vec{x}) \\ \uparrow, & \text{otherwise} \end{cases} \text{ computable}$$

Keep in mind that a recursive set is said to be decidable, a r.e. set is said to be semidecidable.

Note

If $Q(x) \subseteq N$

$Q(x)$ semidecidable if and only if $\{x \in N \mid Q(x)\}$ r.e.

(we could define also recursive/r.e. sets $A \subseteq N^k$)

Saying that A is r.e. is like saying the predicate $Q(x) = "x \in A"$ is semidecidable and we're also generalizing to subsets of N^k and k -ary predicates.

In practice:

- there is an algorithm such that the set of input numbers for which the algorithm halts is exactly A

or equivalently:

- there is an algorithm that enumerates the members of A . That means that its output is simply a list of all the members of A : $a_1, a_2, a_3 \dots$. If A is infinite, this algorithm will run forever.

A recursively enumerable set is a set where you can write a program that will output each element in the set: $E_1, E_2, E_3 \dots$ it's okay if this program never stops. People usually talk about this in the context of languages. A recursively enumerable language is a language where you could write a program that writes out every valid string in that language. A language is just a set of strings, so "the set of all prime numbers in base 10" is a valid language.

Also, if a set is recursive, it's also recursively enumerable.

Observation

Let $A \subseteq \mathbb{N}$ be a set.

$$A \text{ recursive} \Leftrightarrow A, \bar{A} \text{ are r.e.}$$

Proof

(\Rightarrow) Let $A \subseteq \mathbb{N}$ be recursive, i.e.

$$\chi_A = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{otherwise} \end{cases} \text{ is computable}$$

We want to show A r.e., i.e.

$$sc_A = \begin{cases} 1, & \text{if } x \in A \\ \uparrow, & \text{otherwise} \end{cases} \text{ is computable}$$

Intuitively:



(we're defining the semi-characteristic function to show it is r.e.)

Formally:

$$sc_A(x) = \mathbf{1}(\mu w. | \chi_A(x) - 1 |)$$

$\underbrace{\quad}_{\text{o if } x \in A}$
 $\underbrace{\quad}_{\text{1 if } x \notin A}$
 $\underbrace{\quad}_{\text{o if } x \in A}$
 $\uparrow \text{otherwise}$

computable, since it is composition/minimalisation of
computable functions

(in words: we use minimalisation to give a finite computation if the argument is in its domain, which will be for the characteristic function minimized and the indicator function, which will express all the possibilities for the expression to be valid or not)

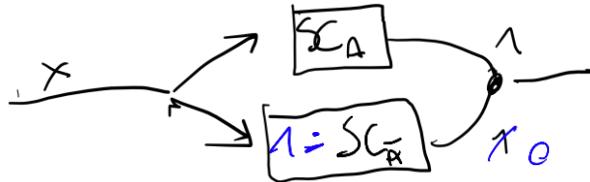
Hence, A is r.e.

Concerning \bar{A} , note that since A recursive, also \bar{A} recursive. Hence, by the argument above, \bar{A} is r.e. (just simply do the same and substitute for both characteristic/semi-characteristic function $\chi_{\bar{A}}$ and it will work).

(\Leftarrow) Let A, \bar{A} be r.e., i.e. the semi-characteristic functions are computable:

$$sc_A(x) = \begin{cases} 1, & \text{if } x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

$$1 - sc_{\bar{A}}(x) = \begin{cases} 1, & \text{if } x \in \bar{A} \\ \uparrow, & \text{otherwise} \end{cases}$$



Here, we combine two machines, in which both are computable, given they are recursive and r.e. This means they will both terminate and since $x \in A$ or $x \in \bar{A}$, the process will terminate for sure. Infact, now we consider the results of both computations, then building the characteristic function.

Let $e_1, e_0 \in N$ s.t. $sc_A = \phi_{e_1}$ and $1 - sc_{\bar{A}} = \phi_{e_0}$

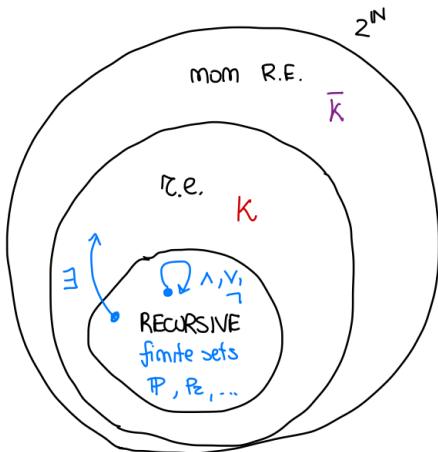
Idea (combining minimalization and encoding in pairs which will terminate):

$$\left(u(y, t). S(e_1, x, y, t) \vee S(e_0, x, y, t) \right)_{\downarrow y}$$

Formally (we apply projection and then use negated sign to represent the binary computation over the projection on first component, hence obtaining a computable thing):

$$\begin{aligned} \chi_A(x) &= \mu w. S(e_1, x, (w)_1, (w)_2 \vee S(e_0, x, (w)_1, (w)_2) \\ &\quad \uparrow \\ &\quad (w)_1 = y \quad (w)_2 = t \\ &= \left(\mu w. \overline{sg}(\max(X_S(e_1, x, (w)_1, (w)_2), \chi_S(e_0, x, (w)_1, (w)_2))) \right)_1 \end{aligned}$$

computable. Hence, A is recursive.



* K not recursive, it is r.e.

$$\begin{aligned} sc_K(x) &= \begin{cases} 1, & \text{if } x \in K (\phi_x(x) \downarrow) \\ \uparrow, & \text{otherwise} \end{cases} \\ &= \mathbf{1}(\phi_x(x)) \\ &= \mathbf{1}(\Psi_U(x, x)) \\ * \bar{K} &\text{ is not r.e.} \end{aligned}$$

Otherwise if \bar{K} r.e., since K r.e., we would have K recursive

16.2 EXISTENTIAL QUANTIFICATION

$$Q(t, \vec{x}) \subseteq \mathbb{N}^{k+1} \quad \text{decidable}$$

$$P(\vec{x}) \equiv \exists t. Q(t, \vec{x}) \quad \text{semi-decidable}$$

So, in words: if a decidable predicate is universally quantified existentially, it can become semi-decidable.

Written by Gabriel R.

An example might help on this:

Consider $Q(t, \vec{x})$ be the statement "There exists a natural number t such that the sum of t and the first component of \vec{x} is equal to the second component of \vec{x} ."

Mathematically, this could be represented as:

$$Q(t, \vec{x}) \equiv (t + x_1 = x_2)$$

In this example, Q asserts that there is a natural number t such that adding t to the first component of \vec{x} results in the second component of \vec{x} . The solutions to $Q(t, \vec{x})$ would be tuples (t, x_1, x_2) where this condition holds true (in our cases, in general, t is the number of steps).

So, we mean, looking at [this](#) one, " $\exists x P(x)$ is true when $P(x)$ is true for at least one value of x ". Given it is semidecidable, *eventually* some value might be found and there would be an algorithm for that. If you consider [this](#) one, you will see quantification it's basically "minimalisation for predicates".

16.3 STRUCTURE THEOREM

Structure theorem (Structure of semi-decidable predicates)

$$\begin{aligned} & \text{there is } Q(t, \vec{x}) \subseteq \mathbb{N}^{k+1} \text{ decidable predicate} \\ P(\vec{x}) \text{ semi-decidable} \Leftrightarrow & \quad \text{s.t. } P(\vec{x}) = \exists t. Q(t, \vec{x}) \end{aligned}$$

(in words: the predicate P is a generalization of a decidable predicate that is computed over multiple points. In general, existentially quantifying transforms a decidable predicate into a semidecidable one)

Proof

(\Rightarrow) Let $P(\vec{x}) \subseteq \mathbb{N}^k$ be semi-decidable:

$$sc_P(\vec{x}) = \begin{cases} 1, & \text{if } P(\vec{x}) \text{ is computable} \\ \uparrow, & \text{otherwise} \end{cases}$$

i.e. there is $e \in \mathbb{N}$ s.t. $sc_P = \phi_e^{(k)}$

Observe:

$$\begin{aligned} P(\vec{x}) & \text{ iff } sc_P(\vec{x}) = 1 \\ & \text{ iff } sc_P(\vec{x}) \downarrow \\ & \text{ iff } P_e(\vec{x}) \downarrow \\ & \text{ iff } \exists t. H^{(k)}(e, \vec{x}, t) \end{aligned}$$

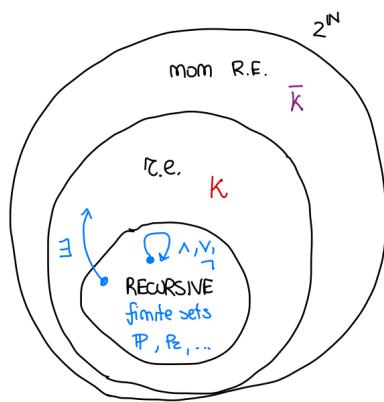
If we let $Q(t, \vec{x}) = H^{(k)}(e, \vec{x}, t)$ decidable and $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$

(\Leftarrow) We assume $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$ with $Q(t, \vec{x})$ decidable

$$\begin{aligned} sc_P(\vec{x}) & = \begin{cases} 1, & \text{if } P(\vec{x}) \Leftrightarrow \exists t. Q(t, \vec{x}) \Leftrightarrow \exists t. X_Q(t, \vec{x}) = 1 \\ \uparrow, & \text{otherwise} \end{cases} \\ & = \mathbf{1}(\mu t. |X_Q(t, \vec{x}) - 1|) \\ & \quad \underbrace{\qquad\qquad\qquad}_{\substack{t \text{ s.t. } Q(t, \vec{x}) \text{ if it exists} \\ \uparrow \text{ otherwise}}} \end{aligned}$$

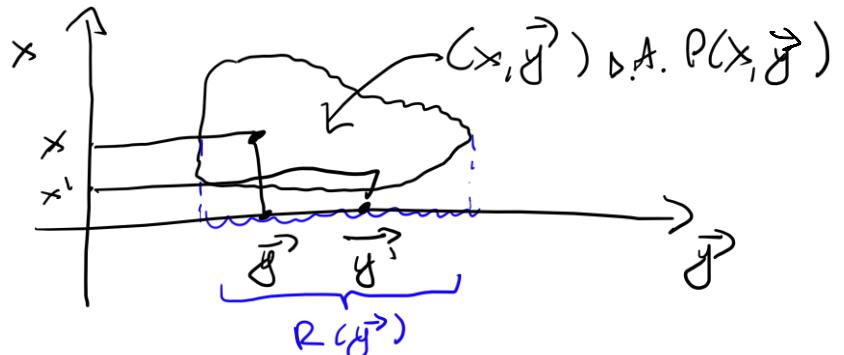
16.4 PROJECTION THEOREM

Definition (projection theorem) – closure by existential quantification



Let $P(x, \vec{y}) \subseteq \mathbb{N}^{k+1}$ semi-decidable

Then $R(\vec{y}) \equiv \exists x. P(x, \vec{y})$ is semi-decidable



In essence, the projection theorem tells us that if we have a property that is semi-decidable for pairs of numbers, then we can define another property about the second part of those pairs, and it will also be semi-decidable. It establishes a connection between the semi-decidability of properties involving pairs and the semi-decidability of properties involving only one part of those pairs.

It also shows RE class is shown with respect to existential quantification.

Proof

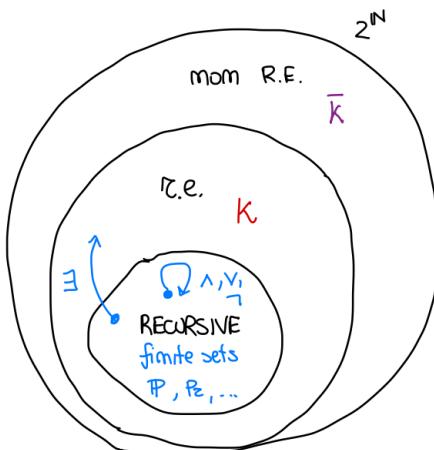
Let $P(x, \vec{y}) \subseteq \mathbb{N}^{k+1}$ semi-decidable. Hence, by structure theorem, there is $Q(t, x, \vec{y}) \subseteq \mathbb{N}^{k+2}$ decidable s.t.

$$P(x, \vec{y}) \equiv \exists t. Q(t, x, \vec{y})$$

Now:

$$\begin{aligned} R(\vec{y}) &\equiv \exists x. P(x, \vec{y}) \equiv \exists x. \exists t. Q(t, x, \vec{y}) \\ &\equiv \exists w. \underbrace{Q((w)_1, (w)_2, \vec{y})}_{\text{decidable}} \end{aligned}$$

Hence R is the existential quantification of a decidable predicate \Rightarrow by structure theorem, it is semi-decidable.



Theorem (Closure under conjunction/disjunction – and/or)

Let $P(\vec{x}), Q(\vec{x}) \subseteq \mathbb{N}^k$ semi-decidable predicates. Then:

- 1) $P(\vec{x}) \wedge Q(\vec{x})$ semi-decidable
- 2) $P(\vec{x}) \vee Q(\vec{x})$

Proof

Since $P(\vec{x}), Q(\vec{x})$ are semi-decidable, by structure theorem, there are two decidable predicates such that:

$$\begin{aligned} P(\vec{x}) &\equiv \exists t. P'(t, \vec{x}) && \text{with } P'(t, \vec{x}), Q'(t, \vec{x}) \text{ decidable} \\ Q(\vec{x}) &\equiv \exists t. Q'(t, \vec{x}) \end{aligned}$$

$$1) P(\vec{x}) \wedge Q(\vec{x}) \equiv \exists t. P'(t, \vec{x}) \wedge \exists t. Q'(t, \vec{x})$$

$$\equiv \exists w. \underbrace{\left(P'((w)_1, \vec{x}) \wedge Q'((w)_2, \vec{x}) \right)}_{\text{decidable}}$$

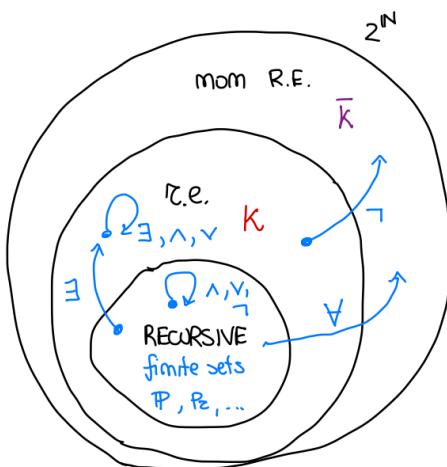
(here, the projection theorem was used, thanks to structure theorem and minimalisation over decidable predicates)

Hence, by the structure theorem, $P(\vec{x}) \wedge Q(\vec{x})$ is semi-decidable.

$$2) P(\vec{x}) \vee Q(\vec{x}) \equiv \exists t. P'(t, \vec{x}) \vee \exists t. Q'(t, \vec{x})$$

$$\equiv \exists t. \underbrace{\left(P'(t, \vec{x}) \wedge Q'(t, \vec{x}) \right)}_{\text{decidable}},$$

Hence, by the structure theorem, $P(\vec{x}) \vee Q(\vec{x})$ is semi-decidable.



* Negation ?

$$Q(x) \equiv "x \in K" \equiv "\phi_x(x) \downarrow"$$

semi-decidable

$$\neg Q(x) \equiv "x \notin K" \equiv "\phi_x(x) \uparrow"$$

not semi-decidable

Theorem (Universal quantification)

$R(t, x) \equiv \neg H(x, x, t)$ decidable

" $x \in \bar{K}$ " $\equiv \forall t. R(t, x) \equiv \forall t. \neg H(x, x, t)$ non semi-decidable

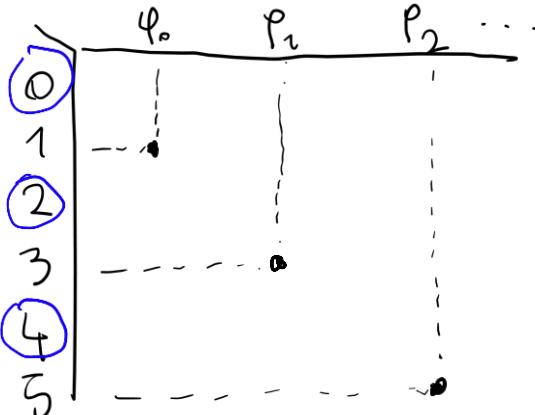
This means that the set of semi-decidable predicates is closed under \vee, \wedge, \exists but not under \forall and \neg

- Universal quantification is many to deal with because even if a decidable predicate it is universally quantified can become non-semi-decidable. Intuitively this is true because it is indefinite to go and test a predicate on infinite values.
- This is essentially saying that there exists a property R involving universal quantification over terms t and a variable x s.t. R is decidable but you universally quantity over t in the context of $\neg H(x, x, t)$, the resulting property is non-semi-decidable, indicating that determining membership in the complement of set K is not always computationally possible.

16.5 OTHER EXERCISES FROM LESSONS

Exercise: Define a function total and non-computable $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $f(x) = x$ on infinitely many $x \in \mathbb{N}$

1st idea



$$f(x) = \begin{cases} x, & \text{if } x \text{ is even} \\ \phi_{\frac{x-1}{2}}(x) + 1, & \text{if } x \text{ is odd and } \phi_{\frac{x-1}{2}}(x) \downarrow \\ 0, & \text{if } x \text{ is odd and } \phi_{\frac{x-1}{2}}(x) \uparrow \end{cases}$$

- f total
- $f(x) = x \ \forall x$ even (infinite set)
- f not computable (total and \neq from all total computable functions) ($\forall x$ if ϕ_x is total, $f(2x+1) = \phi_x(2x+1) + 1 \neq \phi_x(2x+1)$)

2nd idea

$$f(x) = \begin{cases} \phi_x(x) + 1, & \phi_x(x) \downarrow \\ x, & \phi_x(x) \uparrow \end{cases}$$

- total
- not computable ($\forall x$ if ϕ_x is total, $f(x) = \phi_x(x) + 1 \neq \phi_x(x)$), hence f is different from all total computable functions
- $f(x) = x, \forall x \in \bar{K}$ (\bar{K} is infinite, otherwise it would be recursive and so it will be computable)

3rd idea

$$f(x) = \begin{cases} x + 1, & \text{if } \phi_x(x) \downarrow \\ x, & \text{otherwise} \end{cases}$$

- f total
- $f(x) = x \forall x \in \bar{K}$
- f not computable (exercise)

My take on the solution

According to the last specification:

- the function is indeed total, because is defined over the natural numbers, hence $\forall x$ over $x + 1$ and defined whe ϕ_x is defined, undefined otherwise
- the function computes correctly the identity function, so for each value of $f(x)$, it should halt and output a value for x . Since we are outside \bar{K} ($x \notin K$), this means $\phi_x(x)$ does not halt, however according to the definition, in this case it should be undefined.
- this contradicts the fact $f(x)$ should halt for all inputs and have it undefined for such x , given in this case $\phi_x(x) \uparrow$ should happen, but it doesn't. Given also \bar{K} is not r.e. $f(x)$ is not computable

Exercise:

If f is computable, and g coincides with f almost everywhere (except for a finite set of inputs) then g is computable.

My take on the solution

If f is computable, there exists a function able to compute it, may it be the identity function, say you have:

$$f(x) = \begin{cases} x, & x \in W_x \\ 0, & \text{otherwise} \end{cases}$$

We define $g(x) = f(x) \forall x$ except those elements inside a set S ; $g(x)$ can take any value and the set will remain finite. Say for example:

$f(x) = 2x \forall x \in N$, $g(x)$ to be the same for $f(x)$ except for $x = 5$; for $x = 5$, let $g(5) = 100$. In this case, $g(x)$ coincides for all $x \neq 5$ and the overall behavior of g is computable because it's based on the computable function f .

Because $g(x)$ coincides with $f(x)$ for almost all x , the algorithm that computes f can also be used to compute g and by composition it remains computable.

EXERCISE 15.9. Prove that if $P(\vec{x})$ is semi-decidable and is not decidable then $\neg P(\vec{x})$ is not semi-decidable.

My take on the solution

To prove this, we can use diagonalization. Assume $P(\vec{x})$ is semi-decidable but not decidable. This means there exists a semi-decidable predicate $Q(t, \vec{x})$ s.t. $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$.

Suppose at this point $\neg P(\vec{x})$ is semi-decidable and this means there exists a semi-decidable predicate $R(u, \vec{x})$ s.t. $\neg P(\vec{x}) \equiv \exists u. R(u, \vec{x})$.

Using diagonalization, we define $S(v) \equiv \neg R(v, v)$ which says this does not satisfy the property $R(v, v)$. Now we consider the relationship between $S(v)$ and $Q(t, \vec{x})$. If we could semi-decide $\exists v. S(v)$, we would have a semi-decision for $\neg P(\vec{x})$. But this leads to a contradiction.

By the smn-theorem, we define $\forall w. Q(h(w), w) \equiv S(w)$ which maps w to a term $t = h(w)$ s.t. $Q(t, w) \equiv \neg R(w,)$. When $w = h(w)$ we have $Q(h(w), h(w)) \equiv \neg R(h(w), h(w)) \equiv S(h(w))$.

This leads to a contradiction because $S(h(w)) \equiv \neg R(h(w), h(w))$ and $\neg P(h(w)) \equiv \neg R(h(w), h(w)) \equiv R(h(w), h(w))$. Since those cannot be simultaneously true, we conclude that if $\neg P(\vec{x})$ is not semi-decidable.

16.6 RECURSIVELY ENUMERABLE SETS AND REDUCIBILITY

We want to adapt the reduction as a tool for recursive enumerability as we did already for recursiveness.

Given two sets $A, B \subseteq \mathbb{N}$ and $A \leq_m B$:

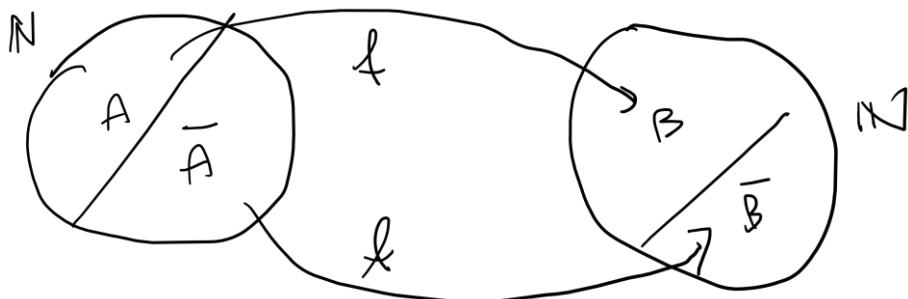
- 1) if B is r.e. then A is r.e.
- 2) if A is not r.e. then B is not r.e.

Proof

Let $A \leq_m B$ there is $f: \mathbb{N} \rightarrow \mathbb{N}$ total computable

$$\forall x, x \in A \text{ iff } f(x) \in B$$

This is what the reduction is doing:



(1) Let B r.e.

$$sc_B(x) = \begin{cases} 1, & x \in B \\ \uparrow, & x \notin B \end{cases} \text{ computable}$$

then

$$sc_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & x \notin A \end{cases} = sc_B(f(x))$$

computable computable
 computable computable
 computable computable

hence sc_A computable
 \Downarrow A is r.e.

(2) equivalent to (1)

- Why recursively enumerable?

We know what enumerable/countable means: our set has the cardinality of natural numbers (or smaller)

enumerable / countable $|A| \leq |\mathbb{N}|$

i.e. there is $f: \mathbb{N} \rightarrow A$ surjective



We want to show *recursively enumerable sets are enumerable by a computable function.*

Proposition (in old Italian notes indicated as "Etymology theorem", because it explains why we say r.e.)

Let $A \subseteq \mathbb{N}$ be a set.

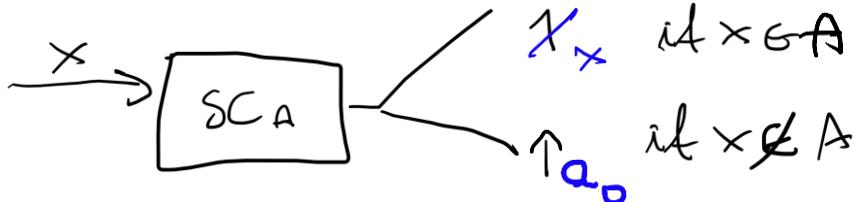
A is r.e. iff $A = \emptyset$ or there exists $f: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t. $A = \text{cod}(f)$.

Proof

(\Rightarrow) Let $A \subseteq \mathbb{N}$ be r.e., i.e.

$$sc_A(x) = \begin{cases} 1, & \text{if } x \in A \\ \uparrow, & \text{otherwise} \end{cases} \text{computable}$$

We can see it graphically as:



$$f(x) = x * sc_A(x) \text{ computable}$$

$$\text{img}(f) = \{f(x) \mid x \in \mathbb{N}\} = A$$

NOT total (which is not, because $sc_A(x)$ is not total, given it is only if $x \in A$, which is a limited case)

Assume $A \neq \emptyset$, fix $a_0 \in A$:

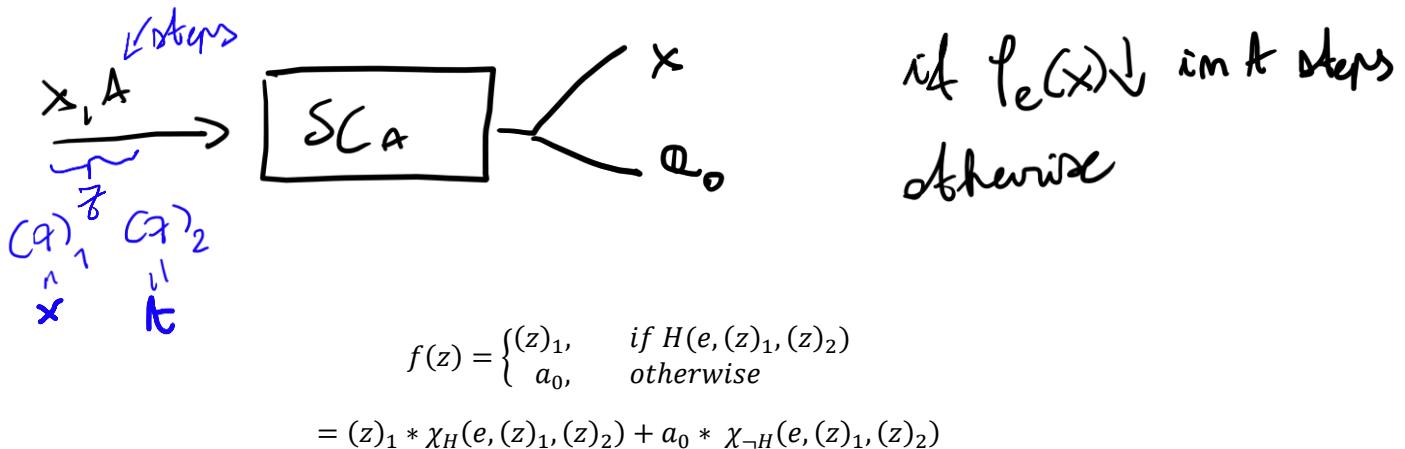
$$f(x) = \begin{cases} x, & \text{if } x \in A \\ a_0, & \text{otherwise} \end{cases}$$

is total, given $\text{img}(f) = A$, but it's not computable given we are distinguishing two cases, it's only defined "for some values" (only semi-decidable).

not computable

We proceed as follows: fix $e \in \mathbb{N}$ s.t. $\phi_e = sc_A$

We provide the input (x), the number of steps (t). Basically, if the steps are enough to terminate on x (the function is defined), we give x as output (hence, it provides a definite output).



We check whether the program P_e terminates over x in t steps, otherwise provide as output a_0 . This is computable given it is defined by cases, using composition and encoding in pairs over the halting set.

f is:

- computable (composition of computable functions)
- total (composition of total functions)
- $\boxed{\text{img}(f) = A}$

This last fact is not completely clear, we will prove it in two ways:

(\subseteq) let $x \in \text{img}(f) \xrightarrow{?} x \in A$

\downarrow there is z s.t. $x = f(z)$ (this is just what the image is), hence there are two possibilities:

- $x = f(z) = (z)_1$ with $H(e, (z)_1, (z)_2)$

hence $P_e((z)_1) \downarrow$, thus $sc_A(z)_1 \downarrow 1$

therefore $x = (z)_1 \in A$

- $x = f(z) = a_0 \in A$

(\supseteq) let $x \in A \xrightarrow{?} x \in \text{img}(f)$

\downarrow $sc_A(x) = 1 \downarrow$ and thus $P_e(x) \downarrow$ for a suitable number of steps t

i.e. $H(e, x, t)$ is true

Therefore, if we take $z \in \mathbb{N}$ s.t. $(z)_1 = x, (z)_2 = t$

$$f(z) = (z)_1 = x \quad (\text{e.g. } z = 2^x * 3^t \dots)$$

thus $x \in \text{img}(f)$

(\Leftarrow)

- if $A = \emptyset$ then A is r.e. (since \emptyset is finite, hence recursive)
- if $A = \text{img}(f)$, where the function f is total computable

$$x \in A \text{ iff there exists } z \in \mathbb{N} \text{ s.t. } f(z) = x$$

(we search for this input and to do so, we use minimalization) then:

$$sc_A(x) = \mathbf{1}(\mu z. |f(z) - x|)$$

if $x \in \text{img}(f) = A$
otherwise

computable
↓

A is r.e. (because sc_A is computable)

Observation Let $A \subseteq \mathbb{N}$

A is r.e. iff $A = \text{dom}(f)$, f computable

(hence

$W_0, W_1, W_2, \dots \dots \dots \dots \dots$ enumeration of r.e. sets)

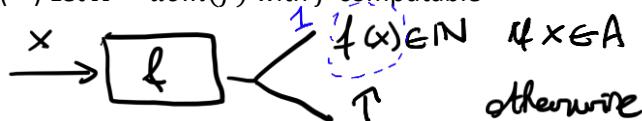
Proof

(\Rightarrow) Let $A \subseteq \mathbb{N}$ be r.e., i.e.

$$sc_A(x) = \begin{cases} 1, & \text{if } x \in A \\ \uparrow, & \text{otherwise} \end{cases} \text{ computable}$$

hence, $A = \text{dom}(sc_A)$, as desired.

(\Leftarrow) Let $A = \text{dom}(f)$ with f computable



$$sc_A(x) = \mathbf{1}(f(x)) \text{ computable}$$

hence A r.e.

Exercise Let $A \subseteq \mathbb{N}$

A r.e. iff $A = \text{img}(f)$ computable

My take on the solution

(\Rightarrow) Let $A \subseteq \mathbb{N}$ be r.e. and we consider:

$$sc_A(x) = \begin{cases} 1, & \text{if } x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

We consider for example $f(x) = x * sc_A(x)$ which is computable since it involves basic arithmetic operations on the semicharacteristic function.

For any $x \in \mathbb{N}$, if $x \in A$, then $f(x) = x$ and if $x \notin A$, $f(x)$ is undefined.

(\Leftarrow) Supposing $A = \text{img}(F)$, we consider $sc_A(x)$ to be as follows:

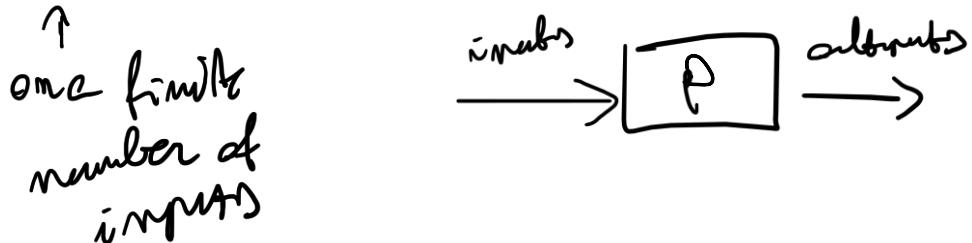
$$sc_A(x) = \begin{cases} 1, & \text{if } f(y) = x \\ \uparrow, & \text{otherwise} \end{cases}$$

This one here respects the definition of being in the image and given the semicharacteristic function is computable, this is r.e.

17 RICE-SHAPIRO'S THEOREM

The Rice-Shapiro theorem helps in proving that a set is not r.e.; in particular, the theorem says that any observation made about computable functions can be done by testing the values of the functions at finitely many arguments.

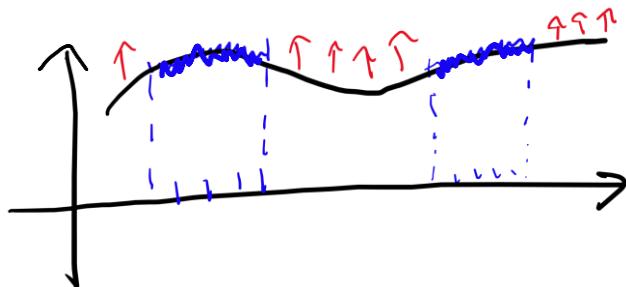
More precisely, the only properties of the behavior of programs which can be semi-decidable are the "finitary properties" (properties which depend on the behaviour on a finite number/amount of inputs).



Examples

- the program P on input \emptyset outputs value 1 finitary
- program P is defined on at least two inputs finitary
- program P is defined on every input not finitary
- program P produces infinitely many values as outputs not finitary
- the program P computes the factorial not finitary

To formalize the notion of a finitary property, we can see from the function plot “we care on what the program is doing only for some inputs” (the blue part), the other parts (the red ones) we don’t care.



We need some more tools, like:

→ finite function (function defined only on a finite domain)

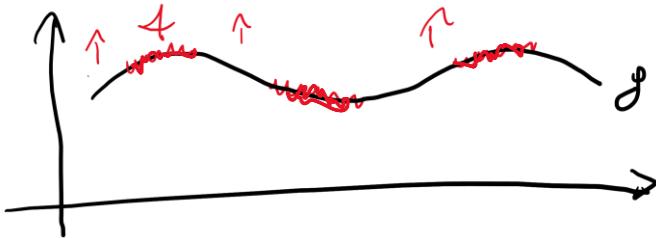
A function $\theta: \mathbb{N} \rightarrow \mathbb{N}$ (this is “theta”) is finite if $\text{dom}(\theta)$ is finite (considering only the finite inputs, the blue ones, while for the rest it is undefined):

$$\theta(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x = x_2 \\ \vdots, & \\ y_n, & \text{if } x = x_n \\ \uparrow, & \text{otherwise} \end{cases}$$

→ subfunction (part of the original function)

We say that f is a subfunction of g , written $f \sqsubseteq g$,

if $\forall x$ if $f(x) \downarrow$ then $g(x) \downarrow$ and $f(x) = g(x)$ (whenever f is defined/undefined, so is g)



17.1 DEFINITION

Theorem (Rice-Shapiro) – consider [this](#) wonderful definition to get the idea, learn it well please

Let $\mathcal{A} \subseteq \mathcal{C}$ (where A is a property of functions) be a set of computable functions and let $A = \{x \mid \phi_x \in \mathcal{A}\}$

Then if A is r.e. then ~~$\exists \theta \subseteq f, \theta$ finite s.t. $\theta \in \mathcal{A}$~~

$$\Downarrow \boxed{\forall f (f \in \mathcal{A} \Leftrightarrow \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A})}$$

↑ property is finitary

(the validity of a property depends only on a finite part of the function)

- We use it to prove only the way is not r.e./recursive (the blue arrow), not the contrary (red arrow going reverse in way), starting from a finitary property
- So, if a property is r.e. then it is finitary, but the converse does not hold

Note: on Wikipedia it was written wrong the definition, because it was $f \in A \Rightarrow$ and not in the way "if and only if", which is \Leftrightarrow as above, was corrected by prof. Baldan there.

(I also did some contributions using terms of the course in Rice/Rice-Shapiro/smn-theorem English definitions in Wikipedia and also following ones like the recursion theorems, just to clarify the concepts if possible – of course, feel free to discredit/add your comments on those voices)

Exercise

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a computable, let $g = f$ almost everywhere (except for a finite set $\{x \mid f(x) \neq g(x)\}$ finite, then g is computable.

Proof

Assume f computable

and $g(x) = f(x) \quad \forall x \neq x_0 \quad f(x_0) \neq g(x_0)$

(1) if $g(x_0) \uparrow$ hence $f(x_0) \downarrow$
 $\underbrace{\quad}_{\text{then } g(x) = f(x) + \mu w. \overline{sg} |x - x_0|}$
 $\uparrow \text{if } x \neq x_0$
 $\uparrow \text{otherwise}$

computable.

2) if $g(x_0) = y_0 \in \mathbb{N}$

let $e \in N$ be s.t. $f = \phi_e$

$$g(x) = \begin{cases} \mu w. (S(e, x, (w)_1, (w)_2) \wedge (x \neq x_0)) \\ (((w)_1 = y_0) \quad \wedge (x = x_0)) \end{cases}$$

computable.

An inductive reasoning allows to conclude in the general case.

Alternatively:

$$D = \{x \in \mathbb{N} \mid f(x) \neq g(x)\} \text{ finite}$$

$$\theta(x) = \begin{cases} g(x), & x \in D \\ \uparrow & \text{otherwise} \end{cases} \text{ finite function} \rightarrow \text{computable}$$

observe

$$g(x) = \begin{cases} f(x), & x \notin D \\ \theta(x), & x \in D \end{cases}$$

computable
decidable (D finite)

computable since it is defined by cases using a decidable predicate and computable function.

Exercise

Define a total non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\text{img}(f) = \{2^n \mid n \in \mathbb{N}\}$

Solution

On this, there are two ideas than can be given by diagonalization; one which is a bit long and the other one which uses a simple/clever idea. Graphically, we are seeing if the image is the set of powers of 2, which corresponds to:

	φ_0	φ_1	φ_2	φ_3
0	-	.	1	;
1	--	-	1	1
2	-	-	1	1
3	-	-	-	1

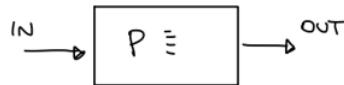
We can use diagonalization on this based on the power of 2, which may be represented by:

$$f(x) = \begin{cases} 2^{\phi_x(x)}, & \text{if } \phi_x(x) \downarrow \\ 1, & \text{if } \phi_x(x) \uparrow \end{cases}$$

- f total
- f not computable, $f \neq \phi_x(x) \neq f(x) + 1$
- $\text{img}(f) = \{2^n \mid n \in \mathbb{N}\}$
 - o $(\subseteq) \forall x, f(x)$
 - $x + 1 \rightarrow 2\phi_x(x)$
 - $x = 0, 2^0 = 1$

- (⊇) $\exists x \text{ s.t. } f(x) = 2^n, x \text{ s.t. } \phi_x(z) = n \forall z$
 - $\phi_x(x) = n \downarrow$

We are interested, considering a program on I/O, for Rice-Shapiro's theorem on different things:



- program properties concerning I/O
- properties of computable functions $\mathcal{A} \subseteq \mathcal{C}$
 - $T = \{f \in \mathcal{C} \mid f \text{ is total}\} = \{f \in \mathcal{C} \mid \text{dom}(f) = \mathbb{N}\}$
 - $ONE = \{\mathbf{1}\}$
 - :
- program properties (extensional/saturated = all the problems computing the same function extend a property, that's why they are called extensional) as sets of programs $\mathcal{A} \subseteq \mathbb{N}$
 - $T = \{x \mid \phi_x \in \tau\}$
 - $P_{ONE} = \{x \mid \phi_x = \mathbf{1}\}$
 - :

On this aspect:

- Rice's theorem explicitly says "no meaningful I/O program property is decidable – only trivial extensional properties are decidable" – no property is decidable, but we can relax the hypotheses a bit with the following one
- Rice-Shapiro tries to figure out "if properties of programs (extensional) can be semidecidable only if they are *finitary* (it talks about the behaviour of the program *only* on a finite set of inputs)"
 - Rice-Shapiro is used to check if something is not semidecidable, not "if it is"

Remember the definition given [here](#). Now we will give the full proof.

For your reference in this section to not go up:

If A is r.e. \Leftrightarrow $\forall f (\exists \theta \subseteq f \text{ finite } \theta \in A)$

17.2 PROOF

We want to show that $(*) \Rightarrow (**)$

For this we show $\neg(**) \Rightarrow \neg(*)$

This amounts to:

(note: \mathcal{A} corresponds to the "calligraphic A" (subset of computable functions), A is the set)

- 1) $\exists f \text{ s.t. } f \notin \mathcal{A} \text{ and } \exists \theta \subseteq f, \theta \text{ finite}, \theta \in \mathcal{A} \Rightarrow A \text{ not r.e.}$
- 2) $\exists f \text{ s.t. } f \in \mathcal{A} \text{ and } \forall \theta \subseteq f, \theta \text{ finite}, \theta \notin \mathcal{A} \Rightarrow A \text{ not r.e.}$

Now for the proofs:

- 1) $\exists f, f \notin \mathcal{A} \text{ and } \exists \theta \subseteq f, \theta \text{ finite with } \theta \in \mathcal{A} \Rightarrow A \text{ not r.e.}$

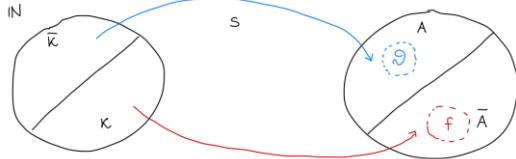
Let $f \notin \mathcal{A}$ be s.t. where there is $\theta \subseteq f, \theta \text{ finite}, \theta \in \mathcal{A}$.

We consider the set not r.e., which is the complement of halting set, specifically:

$$\overline{K} = \{x \mid x \notin W_x\} \leq_m A$$

↓
not r.e.

To do this, we need a reduction function, considering the halting set and its complement compared with the set itself:



Define as usual a function of two arguments (we do this to parametrize it sooner or later, which will happen thanks to smn-theorem):

$$g(x, y) = \begin{cases} \theta(y), & x \in \overline{K} \\ f(y), & x \in K \end{cases}$$

This seems not computable, but actually it is. We “explode” the cases this way:

$$\begin{aligned} g(x, y) &= \begin{cases} \theta(y), & \text{if } x \in \overline{K} \text{ and } y \in \text{dom}(\theta) \\ f(y), & \text{if } x \in \overline{K} \text{ and } y \notin \text{dom}(\theta) \\ \uparrow, & \text{if } x \in K \end{cases} \\ &= \begin{cases} \theta(y), & \text{if } x \in K \text{ or } y \in \text{dom}(\theta) \\ \uparrow, & \text{otherwise} \end{cases} \end{aligned}$$

decidable

semi decidable

finite

semi decidable

undecidable

$$Q(x, y) \equiv "x \in K \vee y \in \text{dom}(\theta)"$$

$$\hookrightarrow sc_Q(x, y) = \begin{cases} 1 & \text{if } Q(x, y) \\ \uparrow & \text{otherwise} \end{cases}$$

computable

$$= f(y) * sc_Q(x, y) \quad \text{computable by composition}$$

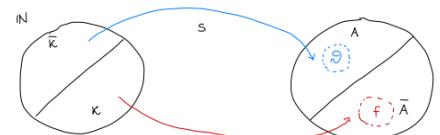
By the smn-theorem, there is a total computable function $s: N \rightarrow N$ s.t. $\forall x, y$:

$$\phi_{s(x)} = g(x, y) = \begin{cases} \theta(y), & x \in \overline{K} \\ f(y), & x \in K \end{cases}$$

We show that s is the reduction function for $\overline{K} \leq A$ (proving the picture is correct)

- if $x \in \overline{K} \rightarrow s(x) \in A$

let $x \in \overline{K}$ then $\forall y, \phi_{s(x)}(y) = g(x, y) = \theta(y)$ hence $\phi_{s(x)} = \theta \in A \Rightarrow s(x) \in A$ (functions same on every input and the program is in the same set)



- if $x \in K \rightarrow s(x) \in \bar{A}$

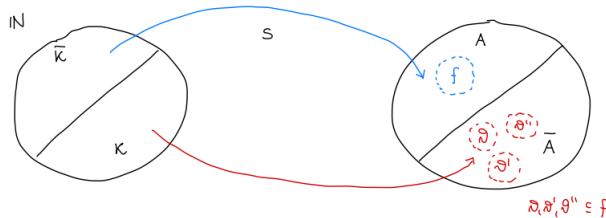
let $x \notin \bar{K}$ i.e. $x \in K$ then $\phi_{s(x)}(y) = g(x, y) = f(y)$ hence $\phi_{s(x)} = f \notin A \Rightarrow s(x) \notin \bar{A}$

Hence $\bar{K} \leq_m A$ and since \bar{K} not r.e. we conclude A not r.e.

2) if there is $f \in A$ s.t. $\forall \theta \subseteq f, \theta$ finite, $\theta \notin A \Rightarrow A$ not r.e.

Let $f \in A$ and assume $\forall \theta \subseteq f, \theta$ finite, $\theta \notin A$ (all functions have not the property)

The reduction considers the complement of halting set gives the function, then the halting set gives the finite subfunctions. So, we show $\bar{K} \leq_m A$ and we will conclude.



The following is the intuition, considering the second argument exists if the index is fixed over the first argument:

$$g(x, y) = \begin{cases} f(y), & x \in \bar{K} \\ \theta(y), & x \in K \end{cases} \quad \begin{array}{l} P_x(x) \uparrow \leftarrow \text{infinite} \\ P_x(x) \uparrow \leftarrow \text{finite} \end{array}$$

intuition \nearrow *some $\theta \subseteq f$ finite*

We use y as counter, checking if program does not or does terminate in y steps.

$$g(x, y) = \begin{cases} f(y), & \text{if } \neg H(x, x, y) \\ \uparrow, & \text{if } H(x, x, y) \end{cases}$$

(if we are in first case, the computation will always continue and always be in first case, obtaining f , otherwise if it stops in the defined number of steps, this is undefined – we use a characteristic function for the halting set property just defined here)

$$= f(y) + \mu z. \chi_H(x, x, y)$$

$\underbrace{\begin{array}{ll} 0 & \text{if } \neg H(x, x, y) \\ 1 & \text{if } H(x, x, y) \end{array}}_{\text{computable}}$

$\underbrace{0}_{\uparrow} \text{ if } \neg H(x, x, y)$
otherwise

(we use a fake minimization just to say “it may halt or not according to the underlying predicate”)

Hence, by the smn-theorem, $\exists s: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t. $\forall x, y$:

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} f(y), & \text{if } \neg H(x, x, y) \\ \uparrow, & \text{if } H(x, x, y) \end{cases}$$

We show that s is the reduction function for $\bar{K} \leq_m A$.

- $x \in \overline{K} \rightarrow s(x) \in A$

if $x \in \overline{K}$, then $\phi_x(x) \uparrow$ i.e. $P_x(x) \uparrow$

Then $\forall y \neg H(x, x, y)$.

Thus $\forall y, \phi_{s(x)}(y) = g(x, y) = f(y)$

Therefore, $\phi_{s(x)} = f \in A$ and thus $s(x) \in A$

- $x \in K \rightarrow s(x) \in \overline{A}$

if $x \in K$ then $\phi_x(x) \downarrow$ i.e. $P_x(x) \downarrow$

Then $\exists y_0 \in N$ s.t. $\forall y < y_0 \neg H(x, x, y)$ (the computation is still going)

$y \geq y_0 H(x, x, y)$ (the computation has stopped)

thus

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} f(y), & \text{if } y < y_0 \\ \uparrow, & \text{otherwise} \end{cases}$$

Then $\phi_{s(x)} \subseteq f$

$\text{dom}(\phi_{s(x)}) \subseteq [0, y_0] \quad \phi_{s(x)}$ finite

i.e. $\phi_{s(x)}$ is a finite subfunction of f , hence $\phi_{s(x)} \in \overline{A}$ hence $s(x) \in \overline{A}$

Thus $\overline{K} \leq_m A$ and, since \overline{K} not r.e., A not r.e.

Typical use of Rice-Shapiro: Prove that $A \subseteq N$ not r.e.

1) $\exists f$ s.t. $f \notin A$ and $\exists \theta \subseteq f, \theta$ finite, $\theta \in A \Rightarrow A$ not r.e.

(there exists a function which is not in the set and there is a finite subfunction which is in the set)

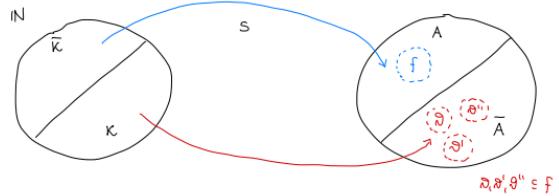
2) $\exists f$ s.t. $f \in A$ and $\forall \theta \subseteq f, \theta$ finite, $\theta \notin A \Rightarrow A$ not r.e.

(there exists a function which is in the set and, for all finite subfunctions (be precise here, don't do the mistake I did) there is a finite subfunction which is not the set)

So, here that I wrote it, you see what's the point of this; use subfunctions to prove your own thing each time working *always for saturated sets, given we think about extensional/saturated properties*.

Rice-Shapiro provides a general framework for proving undecidability based on properties of functions. It doesn't require constructing intricate functions for specific cases each time.

- The key idea is to leverage the existence of finite subfunctions to show undecidability.
- The theorem captures the essence that for certain properties, whether a function is total or not, there always exists a finite subfunction over which the property can be defined, or vice versa.



17.3 EXAMPLES

Example (Totality)

Consider the following example, for set of total functions:

$$\tau = \{f \mid f \text{ is total}\}$$

$$T = \{x \mid x \in \tau\} = \{x \mid \phi_x \text{ is total}\}$$

We can use Rice-Shapiro (1) and (2) present above; find finite subfunctions over which can be defined even if set is not totally defined or viceversa.

- T is not r.e.

Consider the identity or any other total function (the identity is always defined)

$$id \in \tau \quad \text{dom}(id) = \mathbb{N}$$

$$\forall \theta \subseteq id \quad \theta \text{ finite } \quad \text{dom}(\theta) \text{ finite} \neq \mathbb{N} \Rightarrow \theta \notin \tau$$

$\Rightarrow T$ is not r.e. (by Rice-Shapiro)

- \bar{T} is not r.e.

$$id \notin \bar{\tau} \text{ and } \theta = \emptyset \text{ finite } \quad \emptyset(x) \uparrow \forall x \quad \theta \subseteq id \quad \theta \in \bar{\tau}$$

$\Rightarrow \bar{T}$ is not r.e.

Example

$$ONE = \{x \mid \phi_x = \mathbf{1}\}$$

$$= \{x \mid \phi_x \in \{\mathbf{1}\}\}$$

ONE is not r.e.

$\mathbf{1} \in \{\mathbf{1}\}$ and $\forall \theta \subseteq \mathbf{1}, \theta \text{ finite}, \theta \notin \{\mathbf{1}\} \Rightarrow$ by Rice-Shapiro, this is enough to say ONE is not r.e.

\overline{ONE} is not r.e.

$\mathbf{1} \notin \{\mathbf{1}\}$ and $\theta = \emptyset \subseteq \mathbf{1}, \theta \text{ finite}, \theta \in \{\mathbf{1}\} \Rightarrow$ by Rice-Shapiro, \overline{ONE} is not r.e.

Observation: The converse implication of Rice-Shapiro is false

$$A \subseteq C \quad A = \{x \mid \phi_x \in A\}$$

$$\forall f (f \in A \Leftrightarrow \exists \theta \subseteq f, \theta \text{ finite}, \theta \in A)$$

~~↓~~ *false*

A r.e.

Counterexample

$A \subseteq C$ s.t.

a) $\forall f (f \in A \Leftrightarrow \exists \theta \subseteq f, \theta \text{ finite}, \theta \in A)$

b) $A = \{x \mid \phi_x \in A\}$ not r.e.

We claim $A = \{f \mid \text{dom}(f) \cap \bar{K} \neq \emptyset\}$ satisfies (a) and (b)

1) let f be a function

- $f \in A \Rightarrow \text{dom}(f) \cap \bar{K} \neq \emptyset$ i.e. $\exists x_0 \in \text{dom}(f) \cap \bar{K}$

if we define $\theta(x) = \begin{cases} f(x), & x = x_0 \\ \uparrow, & \text{otherwise} \end{cases}$

$$\theta \subseteq f, \theta \text{ finite}, \quad \text{dom}(\theta) = \{x_0\}$$

$$\rightarrow \text{dom}(\theta) \cap \bar{K} = \{x_0\} \neq \emptyset$$

- if there is $\theta \subseteq f, \theta \text{ finite}, \theta \in A$

$$\text{dom}(\theta) \cap \bar{K} \neq \emptyset$$

$$\underset{\text{def}}{\cup} \text{dom}(f)$$

$$\rightarrow \text{dom}(f) \cap \bar{K} \subseteq \text{dom}(\theta) \cap \bar{K} \neq \emptyset$$

$$\rightarrow \text{dom}(f) \cap \bar{K} \neq \emptyset \rightarrow f \in A$$

b) $A = \{x \mid \phi_x \in A\} = \{x \mid \text{dom}(\phi_x) \cap \bar{K} \neq \emptyset\}$ not r.e.

Intuition: Assume that you can semidecides if $x \in A$

in order to check $x \in \bar{K}$ create

<pre>def P(y): if y=x return 0 else loop</pre>	and check if $\text{dom}(P) \cap \bar{K} \neq \emptyset$? $\{\infty\}$
--	--

We show $\bar{K} \leq_m A$

define $g(x, y) = \mu z. |y - x| = \begin{cases} 0, & y = x \\ \uparrow, & \text{otherwise} \end{cases}$ computable

by the smn-theorem, there exists a function $s: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$g(x, y) = \phi_{s(x)}(y)$$

s is the reduction function for $\bar{K} \leq_m A$

$$x \in \bar{K} \Leftrightarrow \text{dom}(\phi_{s(x)}) \cap \bar{K} \neq \emptyset \Leftrightarrow s(x) \in A$$

\uparrow
 $\{\infty\}$

since \bar{K} not r.e. then A is not r.e.

18 FIRST RECURSION THEOREM

In programming languages, we have higher-order functions that take other functions as arguments and produce functional results.

For example consider this *functional type*, in which a function takes in input a function of same type and gives as output another function of same type. Our computational model is not able to represent that, considering the output function are likely to be infinite objects and hence incapable of being given in a finite time. Consider the following example, written in Go language:

```
type T = func (int) int
func succ (f:T) T
res = func (x: int) int
    return f(x) + 1
return res
```

These in computability can be defined as *operators* (as in the book) or *functionals* (as the professor prefers, this is used here). Specifically here, we take an integer, and we map the sum of the same type, as integer, incrementing by 1 the function as you can see.

This means a functional takes “functions as input” and “produces functions as output”. A key characteristic is its *effectiveness*, meaning it can handle infinite inputs and outputs. They calculate in finite time “using only a finite part of the finite function” (hence, they are called operators because of this – the definition comes from Cutland, more later)

Consider functionals as higher-order functions that take functions as arguments and returns functions as results.

$$\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^h)$$

in which F is defined as the set of functions this way over k/h tuples:

$$F(\mathbb{N}^k) = \{f \mid f: \mathbb{N}^K \rightarrow \mathbb{N}\}$$

Both present above are total.

[What is a functional \$\Phi\$ recursive \(computable\)?](#)

Example: successor (from now on, functionals are written in mathematical language)

$$\begin{aligned} \text{succ}: F(\mathbb{N}^1) &\rightarrow F(\mathbb{N}^1) \\ f &\mapsto \text{succ}(f) \\ \text{where } \text{succ}(f)(x) &= f(x) + 1 \end{aligned}$$

Example: factorial

$$\begin{aligned} \text{fact}: \mathbb{N} &\rightarrow \mathbb{N} \\ \underline{\text{fact}}(x) &= \begin{cases} 1, & \text{if } x = 0 \\ x * \underline{\text{fact}}(x - 1), & \text{if } x > 0 \end{cases} \\ \Phi_{\text{fact}}: F(\mathbb{N}^1) &\rightarrow F(\mathbb{N}^1) \end{aligned}$$

$$f \mapsto \Phi_{fact}(f)$$

where

$$\Phi_{fact}(f)(x) = \begin{cases} 1, & \text{if } x = 0 \\ x * f(x - 1), & \text{if } x > 0 \end{cases}$$

in this case the fixpoint exists unique

then the factorial $fact: \mathbb{N} \rightarrow \mathbb{N}$ is a fixed point/fixpoint of Φ_{fact} (a function which is not changed from the transformation and is an element mapped to itself by the function), i.e. $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\Phi_{fact}(f) = f$.

Looking [here](#), a fixed point x in a set X s.t. $x \in X$ is a fixed point with a map to itself such that $f(x) = x$.

We won't prove the fixed point yet; assume, in this case, the fixpoint exists and it is unique.

Example

$$f: \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = \begin{cases} 0, & \text{if } x = 0 \\ f(x + 1), & \text{if } x > 0 \end{cases}$$

$$f(0) = 0$$

$$f(2) = ?$$

(this can go on, so $f(2)$ is $f(3)$, $f(3)$ is $f(4)$... so it will be defined for all natural numbers or just undefined).

$$\text{functional } \Phi: F(\mathbb{N}^1) \rightarrow F(\mathbb{N}^1)$$

$$\Phi(f)(x) = \begin{cases} 0, & \text{if } x = 0 \\ f(x + 1), & \text{if } x > 0 \end{cases}$$

there are (infinitely) many fixed points for Φ

$$f(n) = \begin{cases} 0, & \text{if } x = 0 \\ \uparrow, & \text{if } x > 0 \end{cases}$$

This is what
a programmer
means

$$f_k(n) = \begin{cases} 0, & \text{if } x = 0 \\ k, & \text{if } x > 0 \end{cases} \quad \text{for } k \in \mathbb{N}$$

Example (Ackermann's function – just to refresh, it was defined [here](#) – definition by cases recursively):

$$\Psi: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$\begin{cases} \Psi(0, y) = y + 1 \\ \Psi(x + 1, 0) = \Psi(x, 1) \\ \Psi(x + 1, y + 1) = \Psi(x, \Psi(x + 1, y)) \end{cases}$$

(I do remember you by courtesy, Ψ is uppercase psi). We introduce the corresponding functional:

$$\text{functional: } \Psi: F(\mathbb{N}^2) \rightarrow F(\mathbb{N}^2)$$

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

Ψ Ackermann's function is some "special" fixpoint of Ψ .

In simpler terms, the statement is asserting that when you apply the functional transformation Ψ to Ackermann's function, the result is Ackermann's function itself. This makes Ackermann's function a fixed point of Ψ , which is special because it behaves as the fixed point of itself.

So, we ask: What is a recursive (computable) functional?

Idea: Given $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^h)$ we ask that $\forall \vec{x} \in \mathbb{N}^h$, (all the tuples)

$\Phi(f)(\vec{x})$ is computable (so, value of transformed function is computable)

→ using a finite amount of information on f

i.e. values of f over a finite number of inputs

→ the finite amount of information is processed in an "effective way", so in a computable way

More precisely, in order to compute $\Phi(f)(\vec{x})$

→ we use a finite subfunction $\theta \subseteq f$

in a computable way i.e. there is ϕ computable (in the old sense)

$$\begin{aligned}\Phi(f)(x) &= \phi(\theta, \vec{x}) \\ &= \phi(\tilde{\theta}, \vec{x})\end{aligned}$$

encoding of Θ

(we refer to a number of points which can be referred to as finite subfunctions in order to compute other computable functions; here, the problem is that we have functions, not numbers)

Note: finite functions can be encoded as numbers (*encoding of finite functions*)

$$\theta \rightarrow \tilde{\theta} \in \mathbb{N}$$

$$\theta(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x = x_2 \\ \vdots & \\ y_n, & \text{if } x = x_n \\ \uparrow, & \text{otherwise} \end{cases}$$

$$\tilde{\theta} = \prod_{i=1}^n P_{x_i+1}^{y_i+1}$$

(the product represents uniqueness of the composition of numbers into prime functions)

given the above encoding, we can check if values are or not inside the domain:

$$x \in \text{dom}(\theta) \text{ iff } (\tilde{\theta})_{x+1} \neq 0$$

$$\text{if } x \in \text{dom}(\theta) \text{ then } \theta(x) = (\tilde{\theta})_{x+1} - 1$$

18.1 RECURSIVE FUNCTIONALS

Definition (Recursive functional)

A functional $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^h)$ is recursive if there is a total computable function

$$\phi: \mathbb{N}^{h+1} \rightarrow \mathbb{N} \text{ s.t. for all } f \in F(\mathbb{N}^k)$$

$$\text{for all } \vec{x} \in \mathbb{N}^h$$

$$\Phi(f)(\vec{x}) = y \quad \text{iff there exists } \theta \subseteq f \text{ s.t. } \phi(\tilde{\theta}, \vec{x}) = y$$

In simpler terms, recursive functionals essentially produce outputs of the same type as a finite part of the input function, acting as both input and output themselves.

All the functionals that we considered above are recursive (given they are finite only on limited number of points).

- In simple terms, a recursive functional is a higher-order function that can be effectively computed by a total computable function.
- You only need the value of the function on a single point and then, given it is defined on a finite number of points, it is recursive only from specified set of functions to other sets of the same kind

Observation: Let $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^h)$ be a recursive functional and $f \in F(\mathbb{N}^k)$.

If f is computable then $\Phi(f)$ is computable.

Observation: Let $\Phi: F(\mathbb{N}^1) \rightarrow F(\mathbb{N}^1)$ be a recursive functional and let $f \in F(\mathbb{N}^k)$ be computable.

If $f: \mathbb{N} \rightarrow \mathbb{N}$ is computable, then $\Phi(f): \mathbb{N} \rightarrow \mathbb{N}$ is computable

$$\begin{array}{ccc} \xi & & \xi \\ f = p_e, e \in \mathbb{N} & \xrightarrow{\Phi} & \Phi(f) = \phi_a, a \in \mathbb{N} \\ \vdash p_e, e \in \mathbb{N} & & \vdash \phi_a, a \in \mathbb{N} \end{array}$$

(everything can be computed by programs, from a starting to a target function)

hence Φ induces a function over programs

$$h_\Phi: \mathbb{N} \rightarrow \mathbb{N}$$

$$e \mapsto h_\Phi(e) = a \text{ s.t. } \Phi(\phi_e) = \phi_{h_\Phi(e)}$$

(so, from the image of the functional of program computed by program e , what you get is a function computed by the transformed property)

This is defined as extensional function: $\forall e, e' \in \mathbb{N} \text{ s.t. } \phi_e = \phi_{e'} \text{ then } \phi_{h_\Phi(e)} = \phi_{h_\Phi(e')}$

(from programs which compute the same function, if you apply the functional transformation, the two programs will compute the same function)

It should be noted that the book defines precisely the functional as continuous and monotone; I add this because the professor notions given up until this last one precisely state this.

18.2 MYHILL-SHEPHERDSON'S THEOREM

Definition

(1) Let $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^i)$ be a recursive function. Then, there exists a total computable function $h_\Phi: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \Phi(\phi_e^{(k)}) = \phi_{h_\Phi(e)}^{(i)}$ and h_Φ is extensional.

(For this first part - intuitively, the behaviour of the recursive functional on computable functions is captured by a total extensional function on the indices)

(2) Let $h: \mathbb{N} \rightarrow \mathbb{N}$ be a total computable function and h extensional.

Then, there is a unique recursive functional $\Phi: F(\mathbb{N}^k) \rightarrow F(\mathbb{N}^i)$ s.t. for all $e \in \mathbb{N}$ (possible programs)

$$\Phi(\phi_e^{(k)}) = \phi_{h(e)}^{(i)}$$

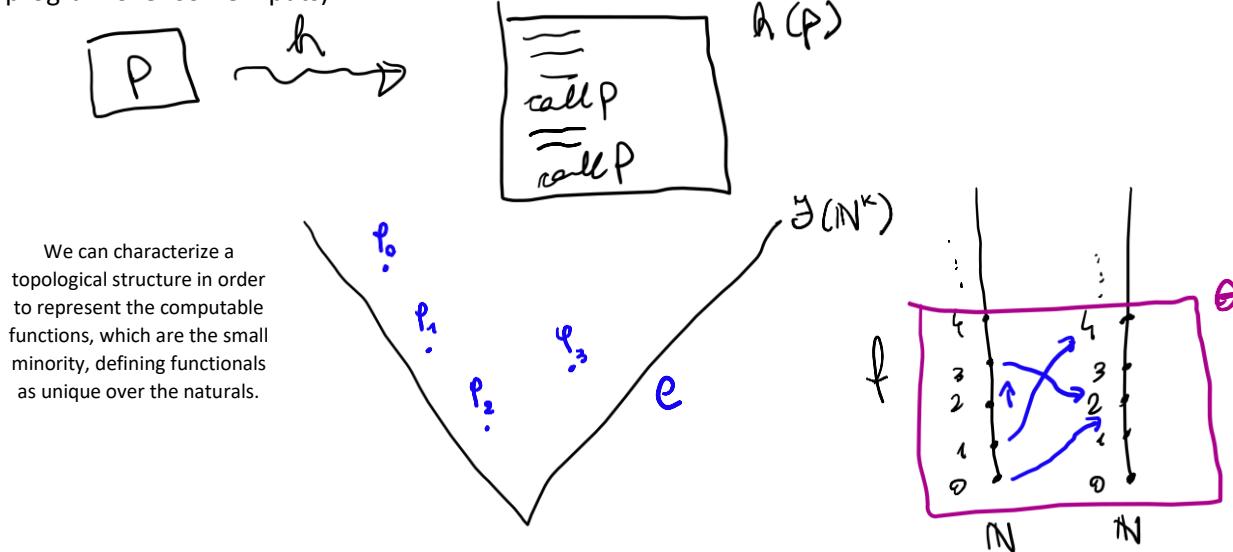
(For this second part - Computable extensional functions uniquely identify computable functions through program transformations. In contrast, recursive functionals extend this identification to non-computable functions, highlighting that all functions, even non-computable ones, can be approximated precisely by computable functions, like finite subfunctions).

In Wikipedia, I see this can generally be extended as "Myhill isomorphism theorem", which provides a characterization for two numberings (assignments of natural numbers to sets of similar objects) to induce the same notion of computability on a set. Basically, there exists a total computable bijection, which maps elements reducible to each other in both directions, given the functions are extensional.

The Myhill-Shepherdson Theorem, stemming from the Rice-Shapiro Theorem, defines the computable type 2 functionals. These functionals operate on computable partial functions, yielding numbers as results in cases of termination. Notably, they adhere to a specific effectiveness criterion and exhibit continuity as functionals. This can be also found [here](#).

Transforming a function means transforming the program in an effective way, basically.

Consider then the extensional program transformation h (which never uses the syntax, only calls the program over some inputs)



18.3 DEFINITION

Definition (First Recursion Theorem – Kleene)

Let $\Phi: F(\mathbb{N}^k) \rightarrow (\mathbb{N}^k)$ be a recursive functional. Then Φ has a least fixed point $f_\Phi: \mathbb{N}^k \rightarrow \mathbb{N}$ which is computable i.e.

$$(i) \Phi(f_\Phi) = f_\Phi$$

$$(ii) \forall g \in F(\mathbb{N}^k) \text{ s.t. } \Phi(g) = g \text{ it holds that } f_\Phi \leq g$$

(ii) f_Φ is computable

This is also called “Kleene’s First Recursion Theorem” or Fixed-point theorem (of recursion theory). The Cutland Computability book specifies it is used to give “meaning” to programs, computing a recursive program, ensuring implementing the program will be defined rigorously over its inputs in a correct way.

Example: Ackermann function

$$\Psi: F(\mathbb{N}^2) \rightarrow F(\mathbb{N}^2)$$

$$\begin{cases} \Psi(f)(0, y) = y + 1 \\ \Psi(f)(x + 1, 0) = f(x, 1) \\ \Psi(f)(x + 1, y + 1) = f(x + 1, f(x, y + 1)) \end{cases} \quad \text{recursive functional}$$

the Ackermann function ψ is the least fixed point of Ψ which exists and is computable by the First Recursion Theorem (fixpoint is unique since it is total and means applying the functional a certain number of times to get eventually to a base case).

Example

$$f(x) = \begin{cases} 0, & \text{if } x = 0 \\ f(x + 1), & \text{if } x > 0 \end{cases}$$

functional $\Phi: F(\mathbb{N}^1) \rightarrow F(\mathbb{N}^1)$

$$\Phi(f)(x) = \begin{cases} 0 \\ f(x + 1) \end{cases}$$

there are many fixed points for Φ :

$$\boxed{f(n) = \begin{cases} 0, & \text{if } x = 0 \\ \uparrow, & \text{if } x > 0 \end{cases}} \quad \leftarrow$$

$$f_k(n) = \begin{cases} 0, & \text{if } x = 0 \\ k, & \text{if } x > 0 \end{cases} \quad \text{for } k \in \mathbb{N}$$

We want this because it is the least fixpoint!! ($f \leq f_k \forall k$)

Example: minimalisation

$$f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$$

$$\mu y. f(\vec{x}, y): \mathbb{N}^k \rightarrow \mathbb{N}$$

can be seen as a least fixed point (because it uses f on a finite number of points). The important thing is that the recursive functional is defined using only a finite number of times the function it receives as an argument.

$$\Phi: F(\mathbb{N}^{k+1}) \rightarrow F(\mathbb{N}^{k+1})$$

$$\Phi(g)(\vec{x}, y) = \begin{cases} y, & \text{if } f(\vec{x}, y) = 0 \\ g(\vec{x}, y + 1), & \text{if } f(\vec{x}, y) \downarrow \text{and } \neq 0 \\ \uparrow, & \text{otherwise} \end{cases}$$

least fixed point is $m: N^{k+1} \rightarrow N$

$$m(\vec{x}, y) = \mu z \geq y. f(\vec{x}, z)$$

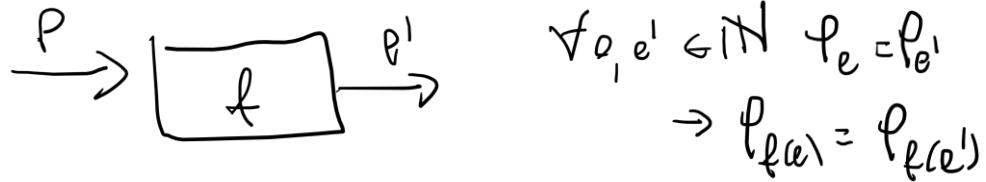
computable
by P Rec. T Heorem

hence

$$m(\vec{x}, 0) = \mu z. f(\vec{x}, z)$$

19 SECOND RECURRENCE THEOREM

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be computable, total and extensional (which takes in input a program and provides in output a transformed program – if the starting program computes a function, the output program computes again the same function, as you can see here):



by Myhill-Shepherdson's theorem, there exists a (unique) recursive functional $\Phi: F(\mathbb{N}) \rightarrow F(\mathbb{N})$ which has the same behavior as f :

$$\forall e \in \mathbb{N} \quad \Phi(\phi_e) = \phi_{f(e)}$$

By the First Recurrence Theorem, Φ has a least fixed point $f_\Phi: \mathbb{N} \rightarrow \mathbb{N}$ computable. Therefore, there is a program $e_0 \in \mathbb{N}$ s.t.

$$\begin{cases} \Phi(f_\Phi) = f_\Phi \\ \exists e_0 \in \mathbb{N} \text{ s.t. } f_\Phi = \phi_{e_0} \end{cases}$$

$$\underline{\Phi_{e_0}} = f_\Phi = \Phi(f_\Phi) = \Phi(\phi_{e_0}) = \underline{\Phi_{f(e_0)}}$$

In summary, given $f: \mathbb{N} \rightarrow \mathbb{N}$ computable total ~~extensional~~, there is $e_0 \in \mathbb{N}$ s.t. $\phi_{e_0} = \Phi_{f(e_0)}$

(we are saying that we take a program, we apply an effective transformation in an extensional way [say, replace instructions such as successor, jump, remove lines], there will always be a program which is not changed by the transformation of the function, even before and after)

2nd recursion
theorem

In practice:

- same hypotheses as the first recurrence theorem, but without the assumption of extensionality
- first recurrence theorem gives meaning to a program over a certain number of inputs which are confined finitely on some sets/number of points (this is the point of Myhill-Shepherdson)

19.1 DEFINITION AND PROOF IDEA

Definition (2nd Recurrence Theorem)

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a total computable function. Then, there exists a program $e_0 \in \mathbb{N}$ s.t. $\phi_{e_0} = \phi_{f(e_0)}$

(the key is that the program is transformed, so not the same before and after the transformation; the function, instead, remains the same – this theorem states that this holds also when f is not extensional)

It is also called Kleene's Second Fixed Point Theorem (aka Second Recurrence Theorem) and:

- If two programs compute the same thing, that is, with the same input they give the same output, then both programs compute the same function
- As Cutland writes, it has its name because it justifies very general definitions "by recursion"

Proof

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be total computable.

Observe $x \mapsto \phi_x(x)$ computable

$$\Psi_U(x, x)$$

$x \mapsto f(\phi_x(x))$ computable

define

$$\begin{aligned} g(x, y) &= \phi_{f(\phi_x(x))}(y) && \text{convention } \phi_\top = \uparrow \\ &= \Psi_U(f(\phi_x(x)), y) \\ &= \Psi_U(f(\Psi_U(x, x)), y) && \text{computable} \end{aligned}$$

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall x, y$

$$\phi_{s(x)}(y) = g(x, y) = \phi_{f(\phi_x(x))}(y) \quad (*)$$

Since s is computable, there is $m \in \mathbb{N}$ s.t. $S = \phi_m$.

Substituting in $(*)$

$$\phi_{\phi_m(x)}(y) = \phi_{f(\phi_x(x))}(y) \quad \forall x, y$$

In particular, for $x = m$

$$\phi_{\phi_m(m)}(y) = \phi_{f(\phi_m(m))}(y) \quad \forall y$$

Hence

$$\phi_{\phi_m(m)} = \phi_{f(\phi_m(m))}$$

If we let $e_0 = \phi_m(m) \downarrow$ and replace in the previous equation, we conclude

$$\phi_{e_0} = \phi_{f(e_0)}$$

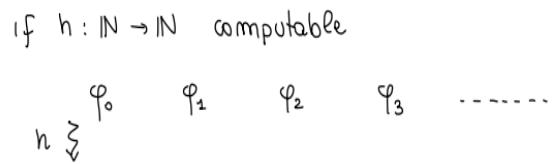
(note that $\phi_m = s$ total, hence $\phi_m(m) \downarrow$)

This theorem can therefore be interpreted in the following manner “given any effective procedure to transform programs, there is always at a program such that, when modified by the procedure, it does exactly what it did before, or it is impossible to write a program that changes the extensional behaviour of all programs”.

Idea

(We discuss the idea behind the proof given, according to the professor, the “mysterious” nature of the statement and the proof itself – a possible interpretation comes from a diagonalization argument.

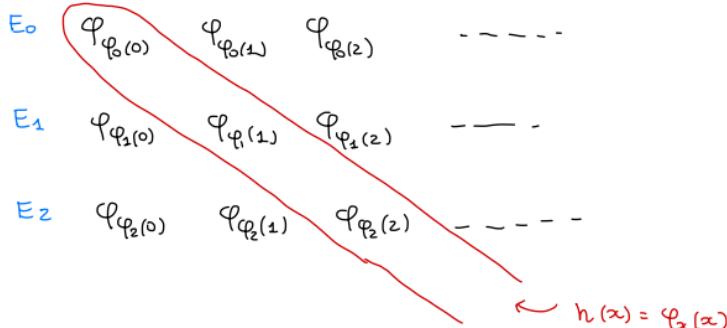
This theorem is pretty deep and like the other recursion theorem by Kleene itself, is a really important result)



$$\phi_{h(0)} \quad \phi_{h(1)} \quad \phi_{h(2)} \quad \phi_{h(3)} \quad \dots$$

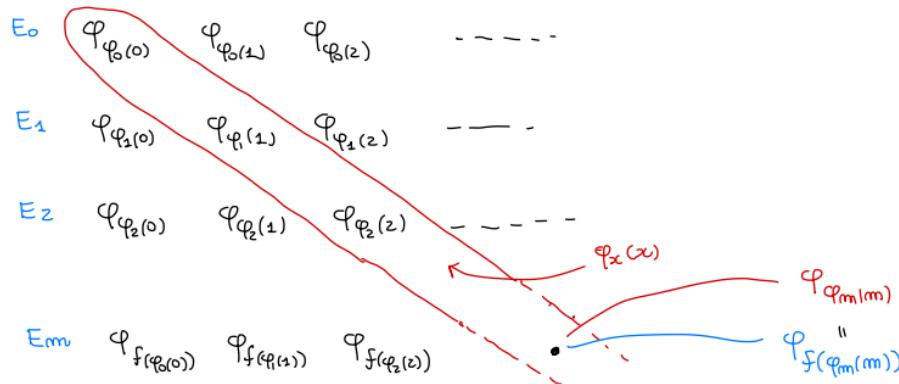
(you can use this enumeration to transform it to a different enumeration; the second one might not be an enumeration of computable functions, but only some of them)

you can do the above for $h = \phi_i \quad i = 0, 1, 2, \dots$ (doing it for all the possible computable functions)



In the proof we took the diagonal transformed by f :

$$h(x) = f(\phi_x(x)) = f(\Psi_U(x, x)) = \phi_m(x)$$



(Since everything is computable, also is the enumeration and so the program must be somewhere, defined as E_m – via the diagonal transformation, we find the right element because the sets are saturated and so the element is the same)

Let's consider again Rice's theorem; what the next part is essentially saying is that using Second Recursion Theorem, the proof is much shorter, proving it directly in a few lines.

19.2 APPLICATION EXAMPLES

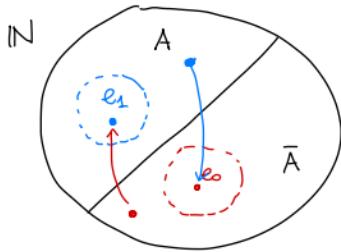
Rice's Theorem

Let $A \subseteq \mathbb{N}$ saturated, $A \neq \emptyset, A \neq \mathbb{N}$, then A is not recursive

Proof (alternative proof using 2nd Recursion Theorem)

Let $A \subseteq \mathbb{N}, A \neq \emptyset, A \neq \mathbb{N}$ saturated

(it means all programs inside a set are computing the same functions, so if you have a program, in either sets each one computes the same thing)



$$\begin{array}{ll} A \neq \emptyset & \rightsquigarrow \exists e_1 \in A \\ A \neq \mathbb{N} & \rightsquigarrow \exists e_0 \in \bar{A} \end{array}$$

Assume by contradiction A is recursive and define $f: \mathbb{N} \rightarrow \mathbb{N}$

$$\begin{aligned} f(x) &= \begin{cases} e_0, & x \in A \\ e_1, & x \notin A \end{cases} \\ &= e_0 * \chi_A(x) + e_1 * \chi_{\bar{A}}(x) \\ &\left(\begin{array}{ll} \text{if } x \in A \rightarrow \chi_A(x) = 1 & \chi_{\bar{A}}(x) = 0 \\ \text{if } x \notin A \rightarrow \chi_A(x) = 0 & \chi_{\bar{A}}(x) = 1 \end{array} \right. \\ &\quad \left. \begin{array}{ll} e_0 * 1 + e_1 * 0 = e_0 \\ e_0 * 0 + e_1 * 1 = e_1 \end{array} \right) \end{aligned}$$

If A recursive, f computable total but for all $e \in N$, $\phi_e \neq \phi_{f(e)}$

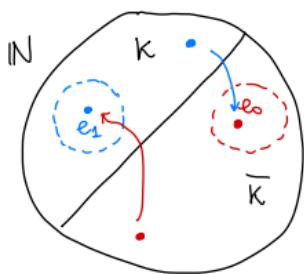
- $e \in A \Rightarrow f(e) = e_0 \notin A$ and since A is saturated (they can't compute the same function, otherwise they would be both in the same set), so $\phi_e \neq \phi_{f(e)}$
- $e \notin A \Rightarrow f(e) = e_1 \in A$, thus since A is saturated, $\phi_e \neq \phi_{f(e)}$

This is absurd, given it contradicts the 2nd Recursion Theorem $\rightarrow A$ not recursive

(and this arises by the fact we considered A recursive, but actually it is not).

Proposition: The halting set $K = \{x \in \mathbb{N} \mid \phi_x(x) \downarrow\}$

Proof (alternative proof using 2nd Recursion Theorem – which again is shorter and easier)



$$\begin{array}{l} \text{if } e_0 \in \mathbb{N} \text{ s.t. } \varphi_{e_0}(x) \uparrow \forall x \\ \text{we have } e_0 \in \bar{K} \\ \\ \text{if } e_1 \in \mathbb{N} \text{ s.t. } \varphi_{e_1}(x) = 1 \forall x \\ \text{we have } e_1 \in K \end{array}$$

Define $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$$\begin{aligned} f(x) &= \begin{cases} e_0, & \text{if } x \in K \\ e_1, & \text{if } x \notin K \end{cases} \\ &= e_0 * \chi_K(x) + e_1 * \chi_{\bar{K}}(x) \end{aligned}$$

If K were recursive, then $\chi_K, \chi_{\bar{K}}$ would be computable and f would be computable.

Since f is total (it also means it has a fixed point), by 2nd Recursion Theorem there is $e \in \mathbb{N}$ s.t. $\phi_e = \phi_{f(e)}$ (which is impossible this last one, but we assume it is):

$\rightarrow e \in K \Rightarrow f(e) = e_0$, so $\phi_e(e) \downarrow \neq \phi_{f(e)}(e) = \phi_{e_0}(e) \uparrow$

$\rightarrow e \notin K \Rightarrow f(e) = e_1$, so $\phi_e(e) \uparrow \neq \phi_{f(e)}(e) = \phi_{e_1}(e) = 1 \downarrow$

Written by Gabriel R.

there is a contradiction again because we assume K to be recursive.

Hence, K is not recursive.

* K is not saturated

$$K = \{x \in \mathbb{N} \mid \phi_x(x) \downarrow\}$$

We want to show that there are $e, e' \in N$ s.t.

$$\begin{aligned}\phi_e &= \phi_{e'} \\ e &\in K, \quad e' \notin K\end{aligned}$$

* Assume that there is $e \in N$ s.t.

$$\phi_e(x) = \begin{cases} 0, & \text{if } x = e \\ \uparrow, & \text{otherwise} \end{cases}$$

then

- $e \in K$ since $\phi_e(e) = 0 \downarrow$
- there exists $e' \in \mathbb{N}, e' \neq e$ s.t. $\phi_e = \phi_{e'}$
- $e' \notin K$ $\phi_{e'}(e') = \phi_e(e') \uparrow \Leftrightarrow e \neq e'$

* We need to show that there exists $e \in N$ s.t.

$$\phi_e(x) = \begin{cases} 0, & \text{if } x = e \\ \uparrow, & \text{otherwise} \end{cases}$$

(the notes here try to observe that there is an e_0 s.t. $\phi_{e_0} = \{(e_0, e_0)\}$, which I think is clear)

We think of a possible program which checks if the input is the program itself, otherwise loop (this is called "quine" if you look in Wikipedia – where I also wrote more notes about the two recursion theorems, basically extending the notion of computable functions and writing the definition of our course).

In words, this theorem essentially proves the existence of such a program, considering it will match any input as its code with no problems.

Intuition

Kleene.py

```
def P(x):
    if x == "":           program we are defining
        return 0           read ("Kleene.py")
    else: loop
```

Formally, we define a function parametrized:

$$\begin{aligned}g(e, x) &= \begin{cases} 0, & \text{if } x = e \\ \uparrow, & \text{otherwise} \end{cases} \\ &= \mu z. |x - e| \text{ computable}\end{aligned}$$

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total computable s.t.

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} 0, & \text{if } x = e \\ \uparrow, & \text{otherwise} \end{cases}$$

since s is total computable, by the 2nd Recursion Theorem, there is $e_0 \in \mathbb{N}$ s.t. $\phi_{e_0} = \phi_{s(e_0)}$. Hence:

$$\phi_{e_0}(x) = \phi_{s(e_0)}(x) = g(e_0, x) = \begin{cases} 0, & \text{if } x = e_0 \\ \uparrow, & \text{otherwise} \end{cases}$$

Observe that $e_0 \in K$ and we also know that there are infinitely many indices for the same function. Thus, let $e \neq e_0$ s.t. $\phi_e = \phi_{e_0}$. Then $\phi_e(x) = \phi_{e_0}(x) \uparrow$.

So, e_0 is the desired program and $e \notin K$. Thus, K is not saturated.

Exercise: Random Numbers (from the 1st Lesson)

→ $n \in \mathbb{N}$ is random if all programs producing n in output are “larger” than n

There were specifically two questions:

→ there are infinitely many random numbers

→ the property of being random is not decidable

Try again to solve this one:

→ size of a program? $|P_e| = e$ (combinatorial, many programs computing the same function)

→ define $n \in \mathbb{N}$ random if for all $e \in \mathbb{N}$ s.t. $\phi_e(0) = n$ it holds $e > n$

(here we need 2nd Recursion Theorem)

Solution

It can be found [here](#).

Exercise

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a function and consider $B_f = \{e \in \mathbb{N} \mid \phi_e = f\}$

Are $B_f, \overline{B_f}$ recursive/r.e.?

1) f not computable

$$B_f = \emptyset, \quad \overline{B_f} = \mathbb{N} \text{ recursive (hence r.e.)}$$

2) f computable

B_f is saturated, $B_f \neq \emptyset, B_f \neq \mathbb{N} \Rightarrow$ by Rice, B_f and $\overline{B_f}$ not recursive

Can it be r.e.? Yes, sometimes it is.

Suppose we have the always undefined function:

$$f = \emptyset \quad (f(x) \uparrow \forall x)$$

$$\overline{B_f} = \{e \mid \phi_e \neq \emptyset\}$$

$$= \{e \mid \exists x. \phi_e(x) \downarrow\}$$


$$sc_{\overline{B_f}}(x) = \mathbf{1}(\mu w. H(x, (w)_1, (w)_2))$$

Complete solution (exercise completion and conclusions)

- More generally, if $f = \theta$ finite, take any total g such that $\theta \subseteq g$

$g \neq B_f, f: \theta \subseteq g$ finite $\theta \in B_f \Rightarrow$ by Rice-Shapiro, B_f is not r.e.

- If f is infinite

$f \in B_f = \{f\} \quad \forall \theta \subseteq f, \theta \neq f, \theta$ finite, $\theta \notin B_f \Rightarrow B_f$ is not r.e.

Also, $\overline{B_f}$ is not r.e.

- if $f = \emptyset$ then $\overline{B_f}$ is r.e. (because of what was said above)
- if $f \neq \emptyset$ then $f \neq \overline{B_f} = \{\overline{f}\}, \theta = \emptyset \subseteq f, \theta \in \overline{B_f} \rightarrow$ by Rice-Shapiro, $\overline{B_f}$ is not r.e.

20 ENDING LESSONS – EXERCISES

20.1 EXAM OF 19/01/2022

A typical exam...

Computability Jan 19 2022

Exercise 1

definitions
proofs
small variations

- Provide the definition of reducibility, i.e., given sets $A, B \subseteq \mathbb{N}$ define what it means that $A \leq_m B$.
- Show that if A is not recursive and $A \leq_m B$ then B is not recursive.
- Show that if A is recursive then $A \leq_m \{1\}$.

Exercise 2

constructions of PR / R
diagonalisation
smm

Is there a non-computable total function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(x) = f(x+1)$ on infinitely many inputs x , i.e., such that the set $\{x \in \mathbb{N} \mid f(x) = f(x+1)\}$ is infinite? Provide an example or show that such a function cannot exist.

Exercise 3

classify sets ("recursive") , saturatedness

Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is quasi-total if it is undefined on a finite number of inputs, i.e., $\text{dom}(f)$ is finite. Classify the set $A = \{x \in \mathbb{N} \mid \varphi_x \text{ quasi-total}\}$ from the point of view of recursiveness, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Exercise 4

Classify the set $B = \{x \in \mathbb{N} \mid \exists y > 2x. y \in E_x\}$ from the point of view of recursiveness, i.e., establish whether B and \bar{B} are recursive/recursively enumerable.

On this exam just taken as an example, some comments in general (each exercise weights 8 points):

- this is the structure more or less
- you have to try to be as much precise as possible, taking nothing for granted or saying, "this is obvious", says Baldan – try to articulate proofs and exercises

In case you are (masochist and insane) interested in the oral exam:

ORAL EXAM: optional, needed for distinction (grade)
focused on theory / proofs
range: ± 4

Exercise 1

- Provide the definition of reducibility, i.e., given sets $A, B \subseteq \mathbb{N}$ define what it means that $A \leq_m B$.
- Show that if A is not recursive and $A \leq_m B$ then B is not recursive.
- Show that if A is recursive then $A \leq_m \{1\}$.

(a) We say $A \leq_m B$ if there exists a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}, x \in A \text{ iff } f(x) \in B$ (*)

(b) We prove the contrapositive, i.e. if $A \leq_m B$ and B recursive, then A is recursive.

Assume B recursive, i.e.

$$\chi_B(x) = \begin{cases} 1, & x \in B \\ 0, & \text{otherwise} \end{cases} \text{ is computable}$$

observe that

$$\begin{aligned} \chi_B(x) &= \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases} \stackrel{(*)}{=} \begin{cases} 1, & f(x) \in B \\ 0, & \text{otherwise} \end{cases} \\ &= \chi_B(f(x)) \end{aligned}$$

since χ_A is the composition of computable functions, it is computable $\Rightarrow A$ is recursive

(c) A is recursive $\Rightarrow A \leq_m \{1\}$

If A is recursive, then

$$\begin{aligned} \chi_A: \mathbb{N} &\rightarrow \mathbb{N} \\ \chi_A(x) &= \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases} \\ &= \chi_B(x) \end{aligned}$$

is computable and total

$$x \in A \text{ iff } \chi_A(x) = 1 \text{ iff } \chi_A(x) \in \{1\}$$

hence χ_A is the reduction function for $A \leq_m \{1\}$

Extra question: Does the converse hold?

$$A \leq_m \{1\} \text{ then } A \text{ is recursive}$$

Yes, since $\{1\}$ is finite, hence it is recursive.

Alternatively: let $f: \mathbb{N} \rightarrow \mathbb{N}$ be the reduction function for $A \leq_m \{1\}$.

Then, $\forall x$

$$x \in A \text{ iff } f(x) = \{1\} \text{ iff } f(x) = 1$$

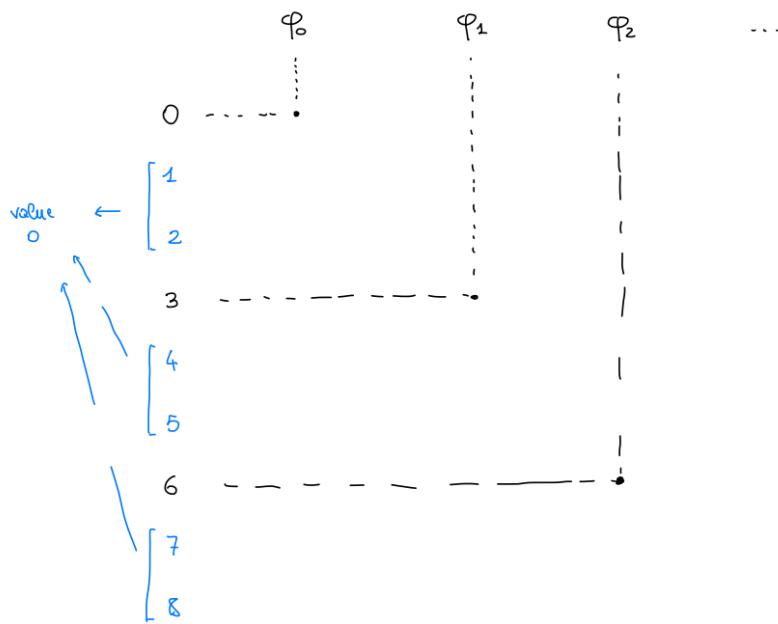
thus

$$\chi_A(x) = \overline{sg}(|f(x) - 1|) = \begin{cases} 1, & \text{if } f(x) = 1 \\ 0, & \text{otherwise} \end{cases} = \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases}$$

Exercise 2

Is there a non-computable total function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(x) = f(x+1)$ on infinitely many inputs x , i.e., such that the set $\{x \in \mathbb{N} \mid f(x) = f(x+1)\}$ is infinite? Provide an example or show that such a function cannot exist.

Idea (bruteforce one (1 = diagonalization) – more elegant one (2))



$$g(x) = \begin{cases} \phi_y(x) + 1, & \text{if } x = 3y \text{ for some } y \text{ and } \phi_y(x) \downarrow \\ 0, & \text{if } (x = 3y \text{ for some } y \text{ and } \phi_y(x) \uparrow) \text{ or } x \neq 3y \quad \forall y \\ x/3 \end{cases}$$

Note that g is:

→ total

→ not computable since $\forall y, \phi_y(3y) \neq g(3y)$

- if $\phi_y(3y) \downarrow$ then $g(3y) = \phi_y(3y) + 1$
- if $\phi_y(3y) \uparrow$ then $g(3y) = 0$

→ there are infinitely many x s.t. $g(x) = g(x+1)$

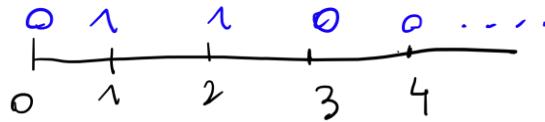
$$\forall y \quad \text{if } x = 3y + 1$$

neither x nor $x+1$ are multiples of 3, hence $g(x) = g(x+1) = 0$ by construction.

(Trying the same argument with multiples of 2 can work – but actually, it's easier to define the function but more difficult to prove it correct, since it would require showing the function differs each time from an even number of arguments – arguably more difficult to prove)

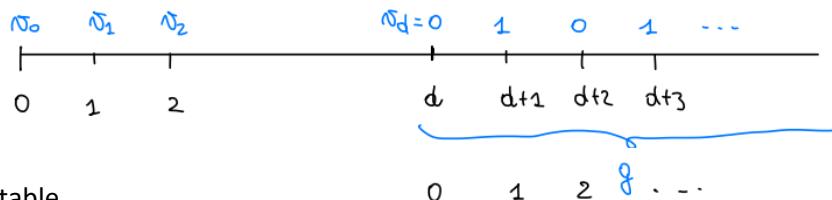
2) alternative solution

Can I use χ_K ? (characteristic function of the halting set)



Observation:

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a function s.t. $\text{cod}(f) = \{0,1\}$ and there is $d \in \mathbb{N}$ s.t. $\forall x > d, f(x) \neq f(x+1)$



Then, f is computable.

In fact, let:

$$f(x) = v_x \quad x \leq d \quad \text{and } v_d = 0$$

(assuming it stops until a point and then it will start alternating)

and define $g: \mathbb{N} \rightarrow \mathbb{N}$ by primitive recursion:

$$\begin{cases} g(0) = 0 \\ g(y+1) = \overline{sg}(g(y)) \end{cases} \quad \text{computable}$$

then

$$f(x) = \prod_{i=0}^{d-1} \overline{sg}(|x-i|) * v_i + g(x-d)$$

computable.

(trick we used for all the finite functions is to use the productory; the sign function tells us "I want to find exactly x " and then, one multiplies by the right value, subtracting the upper bound)

Hence, the desired function in the exercise can be $f = \chi_K$

- χ_K total
- χ_K non-computable
- $\forall d, \exists x \geq d \text{ s.t. } \chi_K(x) = \chi_K(x+1)$ (otherwise, it would be computable)

Again, a diagonalization argument – there are infinitely many inputs and there will always be fixed points such that you can't possibly compute them all, so the condition $x \leq d$ will totally hold, but not effectively)

$$\Rightarrow \{x \in \mathbb{N} \mid \chi_K(x) = \chi_K(x+1)\} \text{ is infinite}$$

Exercise 3

Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is quasi-total if it is undefined on a finite number of inputs, i.e., $\overline{\text{dom}(f)}$ is finite. Classify the set $A = \{x \in \mathbb{N} \mid \varphi_x \text{ quasi-total}\}$ from the point of view of recursiveness, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Conjecture:

Being quasi-total means it has to be defined on an infinite number of inputs (because it is undefined on a finite number of inputs); because of this, it is not r.e. since you can't write the semicharacteristic function – the complement will be undefined on an infinite number of inputs; on the complement, again for same conclusion on normal set, it is not r.e.

$$A \text{ not r.e. } \bar{A} \text{ not r.e.}$$

A is saturated:

$$A = \{x \in \mathbb{N} \mid \phi_x \in A\}$$

$$A = \{f \mid f \text{ is quasi total}\} = \{f \mid \overline{\text{dom}(f)} \text{ finite}\}$$

- A is not r.e.

Observe that $\text{id} \in A$ since $\text{dom}(\text{id}) = \mathbb{N}$ and so $\overline{\text{dom}(\text{id})} = \overline{\mathbb{N}} = \emptyset$ is finite and for all $\theta \subseteq \text{id}$, θ finite, $\theta \notin A$ (so, we're saying this is not quasi-total), since $\text{dom}(\theta)$ is finite $\Rightarrow \overline{\text{dom}(\theta)}$ infinite.

Hence, A is not r.e., by Rice-Shapiro.

$$*\bar{A} \text{ is not r.e. } (\bar{A} = \{f \mid f \text{ not quasi - total}\} = \{f \mid \overline{\text{dom}(f)} \text{ infinite}\})$$

note that $\text{id} \notin \bar{A}$ and $\theta = \emptyset \subseteq \text{id}$ finite and $\theta \in \bar{A}$ since $\overline{\text{dom}(\emptyset)} = \overline{\emptyset} = \mathbb{N}$ infinite.

Hence, by Rice-Shapiro, \bar{A} is not r.e.

(Professor suggests to not proceed by reduction, given you would have to prove Rice-Shapiro again, definitely making the proof longer)

Exercise 4

Classify the set $B = \{x \in \mathbb{N} \mid \exists y > 2x. y \in E_x\}$ from the point of view of recursiveness, i.e., establish whether B and \bar{B} are recursive/recursively enumerable.

Conjecture: B is r.e., not recursive $\Rightarrow \bar{B}$ not r.e. (otherwise, B recursive) and thus \bar{B} not recursive.

- B is r.e.

In fact

$$\begin{aligned} sc_B(x) &= \mathbf{1}(\mu(z, y, t). (S(x, z, y, t) \wedge y > 2x))) \\ &= \mathbf{1}(\mu(z, d, t). (S(x, z, 2x + 1 + d, t))) \\ &= \mathbf{1}(\mu w. S(x, (w)_1, 2x + 1 + (w)_2, (w)_3)) \\ &= \mathbf{1}(\mu w. |X_S(x, (w)_1, 2x + 1 + (w)_2, (w)_3) - 1|) \end{aligned}$$

$$\begin{array}{c} y = 2x + 1 + d \\ \downarrow \end{array}$$

computable, hence B is r.e.

- B is not recursive

We should that $K \leq_m B$ we need a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $x \in K$ iff $S(x) \in B$

$$\exists z \quad \phi_{s(x)}(z) > 2S(x)$$

define

$$g(x, z) = \begin{cases} z, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

$$= z * sc_K(x)$$

hence, g is computable.

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall x, z$

$$\phi_{s(x)}(z) = g(x, z) = \begin{cases} z, & \text{if } x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

We claim that s is the reduction function for $K \leq_m B$.

- if $x \in K$ then $s(x) \in B$.

Let $x \in K$. Then, $\phi_{s(x)}(z) = z \ \forall z$

hence $\phi_{s(x)}(2s(x) + 1) = 2s(x) + 1 > 2s(x)$

thus $s(x) \in B$

- if $x \notin K$ then $s(x) \notin B$

Let $x \notin K$. Hence $\phi_{s(x)}(z) \uparrow \ \forall z$

Thus we have $E_{s(x)} = \emptyset$, hence there is no $y \in E_{s(x)}$ s.t. $y > 2s(x)$ hence $S(x) \notin B$

Consequently, B is not recursive.

Since B is r.e. and not recursive, then \overline{B} not r.e. (otherwise, if B, \overline{B} r.e. we would have B recursive). In turn, this implies that \overline{B} not recursive.

* Extra question (not part of the exam)

If $B = \{x \in N \mid \exists y > 2x. y \in E_x\}$ saturated?

Apparently, it is not since it "refers to x in the property". Let's prove it by showing that there are

$$e \in B \quad e' \notin B \quad \text{with } \phi_e = \phi_{e'}$$

We show that there is $e \in \mathbb{N}$ s.t.

$$\phi_e(x) = 2e + 1$$

Define:

$$g(n, x) = 2n + 1$$

computable, hence by smn-theorem about there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall n, x$:

$$\phi_{s(n)}(x) = g(n, x) = 2n + 1$$

Since s is total and computable, there is $e \in \mathbb{N}$ s.t.

$$\phi_{s(e)} = \phi_e$$

Thus

$$\phi_e(x) = \phi_{s(e)}(x) = 2e + 1$$

Hence

$$\begin{array}{l} e \in B \quad \text{since } 2e + 1 \in E_e \\ \text{---} \\ 2e \end{array}$$

Now, there are infinitely many indexes for ϕ_e , thus we can take $e' \in N, e' > e$ s.t. $\phi_e = \phi_{e'}$.

Note that $E'_e = E_e = \{2e + 1\}$ and $2e + 1 < 2e'$ thus $e' \notin B$.

Summing up, $e, e' \in N, e \in B, e' \notin B, \phi_e = \phi_{e'} \Rightarrow B$ is not saturated

20.2 VARIOUS EXERCISES SOLVED (1/2)

Exercise

Given $f: \mathbb{N} \rightarrow \mathbb{N}$ a fixed function and define $B_f = \{e \in \mathbb{N} \mid \phi_e = f\}$. Classify this set from the point of view of recursiveness

Solution

The set B_f is saturated, because $B_f = \{e \in \mathbb{N} \mid \phi_e \in B_f\}, B_f = \{f\}$

We have two cases:

1) f is not computable, so $B_f = \emptyset, \overline{B_f} = \mathbb{N}$ recursive

2) f is computable

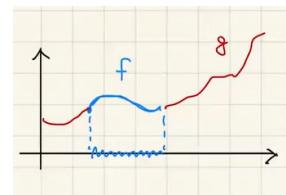
We are using Rice-Shapiro in this case:

- B_f is not r.e.

2.a) if f is finite, $f = \theta$. Let g be a total function s.t. $f \subseteq g$.

We are assuming it is a finite function and then we take another function defined in the same points the other one is defined, only on a subset of points, like you see in figure.

Then, we have $g \notin B_f$ and $f = \theta \subseteq g$ and $f \in B_f$



(so, we define a finite subfunction which is in the set, the function is not in there – this holds when the function is finite, otherwise we use the other part of Rice-Shapiro).

hence by Rice-Shapiro, B_f not r.e. (hence not recursive)

2.b) if f is not finite, not that $f \in B_f$ and $\forall \theta \subseteq f, \theta \text{ finite}, \theta \notin B_f$ then B_f not r.e. by Rice-Shapiro (hence, B_f not recursive)

- $\overline{B_f}$ is not r.e.

Again we have two cases

$$1) f = \emptyset (f(x) \uparrow \forall x)$$

$\overline{B_f}$ is r.e. since $e \in \overline{B_f}$ iff there is some input $\phi_e(x) \downarrow$ hence $sc_{\overline{B_f}}(e) = \mathbf{1}(\mu(x, t).H(e, x, t)) = \mathbf{1}(\mu w.H(e, (w)_1, (w)_2))$

(hence, we look for a single point on which the program terminates in t steps, then characterizing it with the encoding in tuples).

This is computable, hence $\overline{B_f}$ is r.e.

$$2) f \neq \emptyset$$

$\overline{B_f}$ is not r.e by Rice-Shapiro, $f \in \overline{B_f}$ and $\theta = \emptyset \subseteq f, \theta \notin B_f$ hence $\theta \in \overline{B_f}$ hence by Rice-Shapiro, $\overline{B_f}$ is not r.e.

Exercise

Show that $gcd: \mathbb{N}^2 \rightarrow \mathbb{N}$, defined as:

$gcd(x, y) = \text{greatest common divisor of } x \text{ and } y$

is computable (primitive recursive)

Solution

Define:

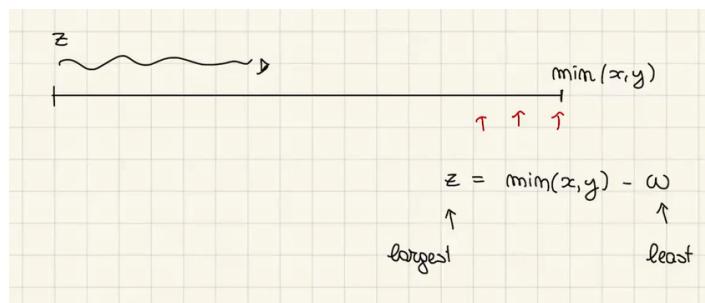
$$gcd(x, y) = \max z . z \text{ divisor of } x \text{ and } z \text{ divisor of } y$$

This can be expressed as $rm(x, z) = 0$ for "z divisor of x" and $rm(z, y) = 0$ for "z divisor of y".

$$= \max z \leq \min(x, y) . (rm(x, z) + rm(z, y) = 0)$$

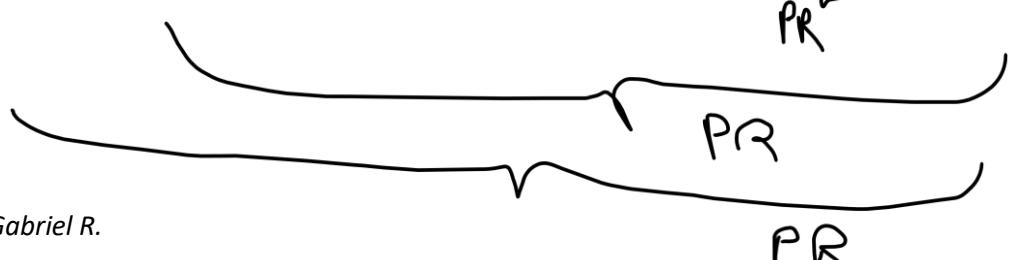
(we know that this is bounded, given there exists a minimum in order to find the largest value)

What we would want is to find a minimum value subtracted to the bound to find the right value:



$$= \min(x, y) - (\mu w \leq \min(x, y) . (z = \min(x, y) - w \wedge rm(z, x) + rm(z, y) = 0))$$

$$= \min(x, y) - (\mu w \leq \min(x, y) . (\underbrace{rm(\min(x, y) - w, x) + rm(\min(x, y) - w, y)}_{PR})) = 0$$



Written by Gabriel R.

hence gcd is primitive recursive.

Exercise

Show there are $m, n \in N$ such that

$$1) \phi_n = \phi_{n+1}$$

$$2) \phi_m \neq \phi_{m+1}$$

Solution

(We just need to apply the Second Recursion Theorem, observing that at least in one case, the program behavior won't change, and this happens for the successor).

1) Observe that $s(n) = n + 1$ is total and computable, hence by the second recursion theorem, there is $n \in N$, $\phi_n = \phi_{s(n)} = \phi_{n+1}$

2) (one can just try to negate here)

if it were that $\forall m, \phi_m = \phi_{m+1}$ then inductively $\phi_0 = \phi_1 = \phi_2 = \phi_3 = \dots$

i.e. all computable unary functions would be the same and this is not the case (e.g. $\mathbf{1} \neq \text{succ}$)

Exercise

Define the class of PR and using only the definition show that

$\max_2: N \rightarrow N, \max_2(x) = \max(2, x)$ is PR

Two ways:

1) Rebuild \max

Define the sum by primitive recursion:

$$\begin{aligned} & \text{sum } (x + y) \\ & \left\{ \begin{array}{l} x + 0 = x \\ x + (y + 1) = (x + y) + 1 \end{array} \right. \end{aligned}$$

Then, define the predecessor by primitive recursion:

$$\begin{aligned} & \text{predecessor } y - 1 \\ & \left\{ \begin{array}{l} 0 - 1 = 0 \\ y + 1 - 1 = y \end{array} \right. \end{aligned}$$

Then, define the difference by primitive recursion:

$$\begin{aligned} & \text{difference } x - y \\ & \left\{ \begin{array}{l} x - 0 = x \\ x - (y + 1) = (x - y) - 1 \end{array} \right. \\ & \max \quad \max(x, y) = x + (y - x) \\ & \max_2(x) = \max(2, x) = \max((0 + 1) + 1, x) \end{aligned}$$

2) Define what you really need

$$\max_2(0) = 2$$

$$\max_2(y+1) = \begin{cases} 2, & \text{if } y = 0 \\ y+1, & \text{if } y > 0 \end{cases} = y+1 + \overline{sg}(y)$$

sum as above

$$sg(y)$$

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

3) An even faster way: if you consider 2 as $2 = 1 + 1$ you can write:

$$\begin{cases} \max_2(0) = 2 \\ \max_2(y+1) = \max_1(y) + 1 \end{cases}$$

$$\max_1 = \max(1, y)$$

$$\begin{cases} \max_1(0) = 1 \\ \max_1(y+1) = y+1 \end{cases}$$

Exercise

1) $A = \{x \mid \phi_x(x) = x^2\}$

2) $B = \{x \mid \phi_x(y) = y^2 \text{ for infinitely many } y's\}$

Questions:

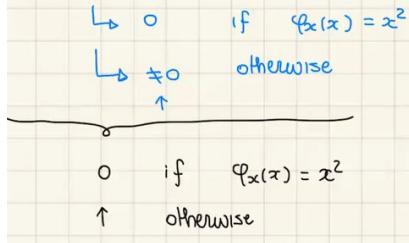
1) Classify A, B according to recursiveness

2) Are A, B saturated?

(1) Conjecture A r.e. and not recursive $\rightarrow \overline{A}$ not r.e. (hence not recursive)

- A r.e.

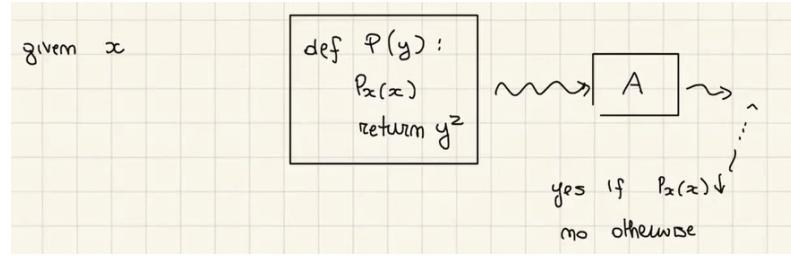
$$sc_A(x) = \mathbf{1}(\mu z. |\phi_x(x) - x^2|) = \mathbf{1}(\mu z. |\Psi_U(x, x) - x^2|)$$



computable $\rightarrow A$ r.e.

- A is not recursive

Define a function over itself: when it terminates, it will always work and then use the reduction from the halting problem.



define

$$g(x, y) = \mathbf{1}(\phi_x(x)) * y^2 = \mathbf{1}(\Psi_U(x, x)) * y^2 = \begin{cases} y^2, & \text{if } \phi_x(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

computable, hence by the smn-theorem, there is $s: N \rightarrow N$ total and computable s.t.

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} y^2, & \phi_x(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

and s is the reduction function for $K \leq_m A$.

- if $x \in K$ then $\phi_{s(x)} = y^2 \forall y$, hence in particular $\phi_{s(x)}(s(x)) = (s(x))^2 \rightarrow s(x) \in A$
- if $x \notin K$ then $\phi_{s(x)}(y) \uparrow \forall y$, hence $\phi_{s(x)}(s(x)) \neq (s(x))^2 \rightarrow s(x) \notin A$

hence $K \leq_m A$ and since K not recursive, A is not recursive

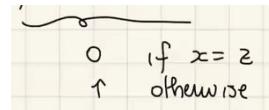
2)

2.a) Is $A = \{x \mid \phi_x(x) = x^2\}$ saturated?

No, it's not. Let $e \in N$ s.t. $\phi_e(x) = \begin{cases} e^2, & \text{if } x = e \\ \uparrow, & \text{otherwise} \end{cases}$

In fact define

$$g(z, x) = \begin{cases} z^2, & \text{if } x = z \\ \uparrow, & \text{otherwise} \end{cases} = z^2 \cdot \mu w. |x - z|$$



computable. By the smn-theorem, there exists $s: N \rightarrow N$ total computable function s.t.

$$\phi_{s(x)} = g(z, x) = \begin{cases} z^2, & \text{if } x = z \\ \uparrow, & \text{otherwise} \end{cases}$$

By the 2nd recursion theorem, there is $e \in N$ s.t. $\phi_e = \phi_{s(e)}$

$$\phi_e(x) = \phi_{s(e)}(x) = g(e, x) = \begin{cases} e^2, & \text{if } x = e \\ \uparrow, & \text{otherwise} \end{cases}$$

Given this:

- $e \in A$ (since $\phi_e(e) = e^2$)
- let $e' \neq e$ s.t. $\phi_e = \phi_{e'}$

$$\phi_{e'}(e') = \phi_e(e') \uparrow \neq e'^2 \Rightarrow e' \notin A$$

So, A is not saturated.

2.b) Is $B = \{x \mid \phi_x(y) = y^2 \text{ for infinitely many } y's\}$ saturated?

B is saturated, infact

$$B = \{x \mid \phi_x \in B\}$$

with $B = \{f \mid f(y) = y^2 \text{ for infinitely many } y's\}$

conjecture: B, \overline{B} not r.e. (hence not recursive)

- B is not r.e.

Let $f: N \rightarrow N, f(y) = y^2$ then $f \in B$ (since $\{y \mid f(y) = y^2\} = N$ infinite)

for all $\theta \subseteq f, \theta \text{ finite}, \{y \mid \theta(y) = y^2\} = \text{dom}(\theta) \text{ finite} \rightarrow \theta \notin B$

hence, by Rice-Shapiro, B not r.e.

- \overline{B} is not r.e.

Note that f as defined above $f \notin \overline{B}$ and $\theta = \emptyset \subseteq f, \theta \in \overline{B}$

hence, by Rice-Shapiro, \overline{B} is not r.e.

20.3 VARIOUS EXERCISES SOLVED (2/2)

Exercise

Classify from the point of view of recursiveness set $A = \{x \in \mathbb{N} \mid W_x \subseteq E_x\}$

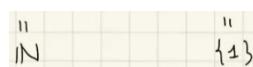
Solution

- A is saturated

$$A = \{x \mid \phi_x \in A\} \quad A = \{f \mid \text{dom}(f) \subseteq \text{cod}(f)\}$$

- A is not r.e.

Observe $\mathbf{1} \notin A, \text{dom}(\mathbf{1}) \not\subseteq \text{cod}(\mathbf{1})$



but $\theta = \emptyset \subseteq \mathbf{1}$ and $\theta \in A$

hence A is not r.e. by Rice-Shapiro.

- \overline{A} is not r.e.

Take $\text{pred}(x) = x - 1$

$$\text{dom}(\text{pred}) = \text{cod}(\text{pred}) = N$$

hence $\text{pred} \in A \rightarrow \text{pred} \notin \overline{A}$

but if you take $\theta \subseteq \text{pred}$

$$\theta(x) = \begin{cases} 0, & x \leq 1 \\ \uparrow, & \text{otherwise} \end{cases}$$

$$\text{dom}(\theta) = \{0, 1\} \not\subseteq \text{cod}(\theta) = \{0\}$$

hence $\theta \notin A \rightarrow \theta \in \overline{A}$

Therefore, by Rice-Shapiro, \overline{A} not r.e. (hence \overline{A} not recursive)

Exercise

Call $f: \mathbb{N} \rightarrow \mathbb{N}$ injective if $\forall x, y \in \text{dom}(f) f(x) = f(y) \text{ then } x = y$

$$A = \{x \mid \phi_x \text{ is injective}\}$$

Conjecture: \overline{A} r.e., not recursive $\rightarrow A$ not r.e. (hence not recursive)

- \overline{A} is r.e.

$$\begin{aligned} sc_{\overline{A}}(x) &= \text{look for } y, z \text{ s.t. } \phi_x(y) = \phi_x(z) \\ &= \mathbf{1}(\mu(y, z, t, v). S(x, y, v, t) \wedge S(x, z, v, t)) \\ &\quad \begin{array}{c} | \\ (\omega)_1 \\ | \\ (\omega)_2 \\ \backslash \\ (\omega)_3 \\ \diagup \\ (\omega)_4 \end{array} \quad \begin{array}{c} \text{y} \neq \text{z} \\ \boxed{} \end{array} \\ &= \mathbf{1}(\mu w. S(x, (w)_1, (w)_4, (w)_3) \wedge S(x, (w)_2, (w)_4, (w)_3) \wedge (w)_1 \neq (w)_2) \\ &= \mathbf{1}(\mu w. S(x, (w)_1, (w)_4, (w)_3) \wedge S(x, (w)_1 + 1 + (w)_2, (w)_4, (w)_3)) \end{aligned}$$

computable $\rightarrow \overline{A}$ r.e.

- \overline{A} not recursive

(1st possibility)

Reduction $K \leq_m \overline{A}$

define

$$\begin{aligned} g(x, y) &= \begin{cases} \text{not injective in } y & \text{if } x \in K \\ \text{injective} & \text{if } x \notin K \end{cases} \\ &= sc_K(x) \text{ computable} \end{aligned}$$

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total computable such that $\forall x$

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} 1, & \text{if } x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

Now s is the reduction function for $K \leq_m \overline{A}$

- if $x \in K$ then $\phi_{s(x)}(y) = 1 \quad \forall y$ hence $\phi_{s(x)} = \mathbf{1} \in \overline{A}$ and thus $s(x) \in \overline{A}$
- if $x \notin K$ then $\phi_{s(x)}(y) = \uparrow \quad \forall y$ hence $\phi_{s(x)} = \emptyset \notin \overline{A}$ and thus $s(x) \notin \overline{A}$

Since $K \leq \overline{A}$ and K not recursive then \overline{A} not recursive.

(2nd possibility)

Observe that \overline{A} is saturated and not trivial:

- if e_1 is s.t. $\phi_{e_1} = \mathbf{1}$ then $e_1 \in \overline{A} \neq \emptyset$
- if e_2 is s.t. $\phi_{e_2} = \emptyset$ then $e_2 \notin \overline{A} \neq N$

by Rice's theorem \overline{A} not recursive.

Exercise

Say $f: N \rightarrow N$ is monotone if f is total and $\forall x, y \in N$ if $x \leq y$ then $f(x) \leq f(y)$

Question

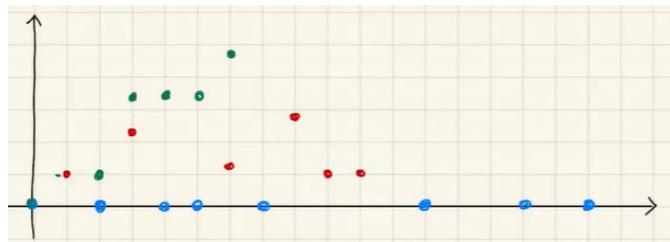
Is there a monotone non-computable function?

Consider

$$f(x) = \begin{cases} \phi_x(x) + 1, & \text{if } \phi_x(x) \downarrow \\ 0, & \text{otherwise} \end{cases}$$

we know that it is total and not computable.

We can plot the function considering points where function exists and others where it doesn't. So, anyway, it will stop when defined having only a finite number of points where the sum is finite again.



Define $g(x) = \sum_{y < x} f(y)$

- total
- not computable $\forall x \quad g(x) \neq \phi_x(x)$
 - $\phi_x(x) \downarrow \quad g(x) = \sum_{y \leq x} f(y) \geq f(x) = \phi_x(x) + 1 \rightarrow g(x) > \phi_x(x)$
 - $\phi_x(x) \uparrow \quad g(x) \uparrow \neq \phi_x(x)$

→ g is monotone, $\forall x, y \quad x \leq y$

$$g(x) = \sum_{z \leq x} f(z) \leq \sum_{z \leq x} f(z) + \sum_{x < z \leq y} f(z) = \sum_{z \leq y} f(z) = g(y)$$

- Alternative solution

$g: \mathbb{N} \rightarrow \mathbb{N}$

$$g(x) = \begin{cases} x + 1, & \text{if } \phi_x(x) \downarrow \text{ and } \phi_x(x) \neq x + 1 \\ x, & \text{otherwise (if } \phi_x(x) = x + 1 \text{ or } \phi_x(x) \uparrow) \end{cases}$$

- g total

- g not computable $\forall x \phi_x(x) \neq g(x)$

- o if $\phi_x(x) \downarrow$ $\begin{cases} \phi_x(x) \neq x + 1 \text{ then } g(x) = x + 1 \neq \phi_x(x) \\ \phi_x(x) = x + 1 \text{ then } g(x) = x \neq \phi_x(x) \end{cases}$

- o if $\phi_x(x) \uparrow$ then $g(x) \downarrow \neq \phi_x(x)$

- g is monotone $\forall x, y \quad x < y \Rightarrow g(x) \leq x + 1 \leq y \leq g(y)$

Even simpler



$$g(x) = \begin{cases} x + 1, & \text{if } \phi_x(x) \downarrow \\ x, & \text{otherwise} \end{cases}$$

- total and monotone
- not computable

$$\chi_K(x) = g(x) - x = \begin{cases} 1, & \text{if } \phi_x(x) \downarrow \\ 0, & \text{otherwise} \end{cases}$$

If g were computable then χ_K , composition of computable function would be computable $\rightarrow g$ not computable.

Exercise

Is there a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x) = \sum_{y \leq x} f(y)$ is computable?

Solution

No: $f(x) = g(x+1) - g(x) = \sum_{y < x+1} f(y) - \sum_{y < x} f(y)$

$$(f(0) + f(1) + \dots + f(x)) - (f(0) + f(1) + \dots + f(x-1)) = f(x) \leftarrow \text{this is what the sum is doing}$$

hence, if g were computable, also f would be computable, by composition.

What about the case in which f is not total?

$$f(x) = \begin{cases} \uparrow, & \text{if } x = 0 \\ \chi_K(x), & \text{if } x > 0 \end{cases} \quad \text{not computable (exercise)}$$

$$g(x) = \sum_{y < x} f(y) = \begin{cases} 0, & \text{if } x = 0 \\ \uparrow, & \text{otherwise} \end{cases}$$

$$= \mu z. x \text{ computable}$$

Exercise

Show that there is $x \in N$ s.t. $\phi_x(y) = x - y$

Solution

State precisely the Second Recursion Theorem and define $g(x, y) = x - y$ computable

By the smn-theorem, $g(x, y) = \phi_{s(x)}(y)$ for $s: N \rightarrow N$ total computable and using the 2nd recursion theorem there is x_0 s.t. $\phi_{x_0} = \phi_{s(x_0)}$

$$\phi_{x_0}(y) = \phi_{s(x_0)}(y) = g(x_0, y) = x_0 - y$$

20.4 SOLUTION OF THE EXERCISE ON RANDOM NUMBERS

There is a video explaining that on Moodle of this year; for convenience, I'll put this year, commented if possible.

The informal way of having a number $n \in \mathbb{N}$ which is random if for every program which outputs n , P is larger than n show that

- there are infinitely many random numbers
- the property of being random is undecidable

Formal view

- program size $|P_e| = e$
- $n \in \mathbb{N}$ is random if for all programs $e \in \mathbb{N}$ s.t. $\phi_e(0) = n$ it holds $e > n$

1) there are infinitely many random numbers

Recall that each computable function is computed by infinitely many programs. Hence, for each $k \in \mathbb{N}$ there is $e_1 < e_2 < \dots < e_k$ s.t. $\phi_{e_i} = \emptyset$ $i = 1, \dots, k$

$$|\{\phi_i(0) \mid i \leq e_k \wedge \phi_i(0) \downarrow\}| \leq e_k - k$$

hence there are at least k numbers $n \leq e_k$ which can't be generated by programs $e < n \rightarrow$ these numbers are random. Since this holds for every k , there are infinitely many random numbers.

2) $R = \{n \mid n \text{ is random}\}$ is not recursive

Assume R to be recursive, i.e.

$$\chi_R(n) = \begin{cases} 1, & \text{if } n \in R \\ 0, & \text{otherwise} \end{cases}$$

Define:

$g(n, x) = \text{least random number } > n = \mu z. z \in R \text{ and } z > n = n + \mu z. (n + 1 + z \in R)$
computable.

Written by Gabriel R.

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable $s.t. g(n, x) = \phi_{s(n)}(x)$

least random number > m

By the 2nd recursion theorem there is $n_0 \in N$ s.t. $\phi_{n_0} = \phi_{s(n_0)}$

$$\phi_{n_0}(0) = \phi_{s(n_0)}(0) = g(n_0, 0) = (\text{least random number} > n_0)$$

hence n_0 generates a random number $> n_0$, contradiction!

$\Rightarrow R$ not recursive

Note \overline{R} is r.e.

$$sc_{\overline{R}}(n) = \mathbf{1}(\mu t. \bigvee_{e=0}^n S(e, 0, n, t)) \text{ computable}$$

$\rightarrow R$ is not r.e.

\hookrightarrow
 $\left\{ \begin{array}{l} \text{check if some} \\ \text{program } e < m \text{ outputs } m \text{ on } 0 \end{array} \right.$

21 TUTORING LESSONS 2023-2024

This year tutoring lessons were recorded and can all be found [here](#).

21.1 TUTORING 1: PRIMITIVE RECURSION EXERCISES

$$\text{isqrt}(x) = \lfloor \sqrt{x} \rfloor$$

$$x = 25 \quad |\sqrt{25}| = |5| = 5$$

Note: This is based on this one exam (2nd appeal of 2022-2023 present on MEGA)

Exercise 2

Give the definition of the class \mathcal{PR} of primitive recursive functions. Show that the following functions are in \mathcal{PR}

1. $\text{isqrt} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{isqrt}(x) = \lfloor \sqrt{x} \rfloor$;
2. $\text{lp} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{lp}(x)$ is the largest prime divisor of x (Conventionally, $\text{lp}(0) = \text{lp}(1) = 1$.)

You can assume primitive recursiveness of the basic arithmetic functions seen in the course.

1)

1. $\text{isqrt} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{isqrt}(x) = \lfloor \sqrt{x} \rfloor$;

This is based for primitive recursive functions (recursion, but kinda limited).

$$Z(x) = 0$$

$$S(x) = x + 1$$

$$x_1, x_2, x_n \rightarrow N$$

This is the partial recursion (defined for zero and successor; every value depends on the previous one, because it's defined by induction):

$$f(\vec{x}, 0) = \dots$$

$$f(\vec{x}, y + 1) = \dots$$

We want to write $\lfloor \sqrt{x} \rfloor$ with primitive recursive functions. The square root function mathematically can be defined as: $y = \sqrt{x}$, $y^2 = x$. We want to find the y value eventually and do to that we use minimalisation (in this case it's not unbounded, so there is no finite number of steps).

Is there a way to bound values of y ?

Let's say $x = 24, y = 0, y^2 = 0, y = 1, y^2 = 1 \dots y = 4, y^2 = 16 < 24, y = 5, y^2 = 25 > 24$

We know $\sqrt{x} \leq x$ because we're working with positive values. We search every value until we find x . We think for example, $x = 25, y = 5$.

We observe that when x changes, y does too.

$$x = 23, y = 4$$

$$x = 24, y = 4$$

$$x = 25, y = 5$$

$$x = 26, y = 5$$

To bound the search for y , consider the relationship between y and x . As x changes, y changes as well. When x increases, y also increases, and vice versa. We want to see what happens after the current y . For example, $(y + 1)^2 = 25$ (the same for 4 for 25 and 5 for 36). So, because it's recursive, the square root it's between y^2 and $(y + 1)^2$.

Essentially, we need to cover a case $y^2 \leq x < (y + 1)^2$. In this case, you can see that $(y + 1)^2$ changes when y changes. So, the square root is between y^2 and $(y + 1)^2$. This means we need to find a y where y^2 is less than or equal to x , but $(y + 1)^2$ is greater than x .

So, we want to prove:

$$(y + 1)^2 > x$$

$$(y + 1) = x \rightarrow \{0 \text{ } (y + 1)^2 \leq x, x_0 \text{ otherwise}\}$$

To formally define the *isqrt* function, we can use minimalization. The minimalization process finds the smallest y such that $(y + 1)^2$ is greater than x . This can be expressed as:

$$\text{isqrt}(x) = \mu y < x + 1. ((y + 1)^2 > x)$$

Another step is introducing the negated sign function, so \overline{sg} which allows us to turn this condition into a binary value (1 for true and 0 for false).

So, by using the negated sign function within the minimalization operator, you are effectively searching for the smallest y for which the condition $(y + 1)^2 > x$ becomes true, and this condition ensures that you find the largest y such that y^2 is less than or equal to x , which is the definition of the *isqrt* function. To combine everything properly:

$$\text{isqrt}(x) = \mu y < x + 1. \overline{sg}((y + 1)^2 > x)$$

One can use division and we can assume they're defined to use "You can define primitive recursiveness...", for example:

$$(y + 1)^2 > x \text{ (so, not the square root of } x)$$

$$\frac{x}{(y+1)^2} < 1 \text{ (so, the result of square root exists and is less than 1)}$$

$$\text{div}(x, (y + 1)^2) \text{ (we use this to check if result is less than 1)}$$

We don't need to use directly primitive recursion, but bounded minimalisation implies there is always a finite number which implies "everything is defined \geq than the bound".

So, we write:

$$\begin{aligned} f(x, y) &\rightarrow f(x, 0) = x \\ f(x, y + 1) &= \{ f(x, y) \text{ if } f(x, y) \neq x, \quad y \text{ if } (y + 1)^2 > x, \quad x \text{ otherwise} \} \end{aligned}$$

(so, base case simply means x , the second case is “we didn’t find the square root yet”, the third one is just x with no root found).

To define with bounded minimalisation, we write:

$$isqrt(x) = f(x, x)$$

(You’re essentially saying, “Let y be x in the function f .” This is equivalent to saying you’re looking for the value of f when y is x)

To assure to handle all cases for composition and make sure everything is computable, we use:

$$f(x, y) = sg(x - f(x, y)) + \overline{sg}(x - f(x, y)) * (y * sg((y + 1)^2 - x) + x * \overline{sg}((y + 1)^2 - x))$$

(so, the sign function is just there to mean “binary variable” on the difference if it holds or not, while the negated sign does the same with opposite values. The middle product ensures that the function returns the value of y , while the last part does the opposite, so just checks if we have found the right root yet).

2)

2. $lp : \mathbb{N} \rightarrow \mathbb{N}$ such that $lp(x)$ is the largest prime divisor of x (Conventionally, $lp(0) = lp(1) = 1$.)

$$lp \text{ biggest prime divisor, } P_x \text{ } x\text{-th prime}$$

The problem is that we’re using minimalisation. We need to find, given $P_x < x$, a number y s.t. $y < P_y < x$. So, we note that minimizing to maintain the property of primes, such that x is always bigger than x , so:

$$\mu i < x. (\dots)$$

With

$$y = x - i$$

$$P_y \text{ divide } x$$

This approach explicitly checks the prime divisors of x starting from p_x , then p_{x-1} , and so on, stopping at the first prime divisor found.

We use the remainder function to obtain the prime number to check if x is divisible by the prime number P_y . When the remainder is zero, it means P_y divides x .

$$rem(x, P_y) = 0$$

This is the case for minimalisation needed and it happens. Thanks to this, as soon as we find the number, it simply stops. So, we’re looking for:

$$x = \mu i < x. rem(x, P_{x-1})$$

(note $P_{x-1} = P(x - 1)$ so this is a function call). Everything is defined for P by recursion (this is crucial for the approach because you need a way to generate prime numbers and ensure that they are available for the minimization process):

$$P(x - \mu i < x. rem(x, P_{x-1}))$$

Again, we use the sign to flip and obtain 1/0 for the edge cases, so:

$$lp(x) = P(x - \mu i < x.\text{rem}(x, P_{x-1})) - sg(x - 1)$$

Actually, the program P is defined for the recursive case, so $x - i$ is the subscript; a proper form is:

$$lp(x) = P_{x-i(x)}(x - \mu i < x.\text{rem}(x, P_{x-1})) - sg(x - 1)$$

The official solution counts the prime divisors, restricting the search for all the prime numbers (defines a function for counting these). Then, simply consider the cases where the count is zero and when the length is not, so using a negated sign function. This is as follows:

2. Observe that, for $x > 1$, $lp(x)$ is surely smaller or equal to p_x . Hence one can count the prime divisors of x , restricting the search to p_1, \dots, p_x :

$$\text{count}(x) = \sum_{i=1}^x \text{div}(p_i, x)$$

and then $lp(x) = p_{\text{count}(x)}$. The function needs to be adjusted for $x = 0$ and $x = 1$, where $\text{count}(x) = 0$ and thus $p_{\text{count}(x)} = 0$ while $lp(x) = 1$. This is easily done as follows:

$$lp(x) = p_{\text{count}(x)} + \overline{sg}(x \dashv 1).$$

Since we use only known primitive recursive functions, bounded sum and composition we conclude that lp is primitive recursive.

Alternatively, a similar idea to the one we used is checking explicitly by recursion the prime divisors of x :

Alternatively, the idea can be to check explicitly the prime divisors of x , starting from p_x , then p_{x-1} and so on, stopping at the first. In detail, look for the smaller y , call it $i(x)$, such that p_{x-y} is a divisor of x .

This way, we define the smaller $i(x)$ as an existing value (hence, minimalisation) of the possibility (binary combination) of finding a prime number respecting this property by recursion dividing each time (hence, the division as said, but also the negated sign).

$$i(x) = \mu y \leqslant x. \overline{sg}(\text{div}(p_{x-y}, x))$$

Then, whenever $x > 1$, $lp(x) = p_{x-i(x)}$ and the cases $x \leqslant 1$ must be treated separately as before:

$$lp(x) = p_{x-i(x)} \cdot sg(x \dashv 1) + \overline{sg}(x \dashv 1).$$

Exercise 2.1(p). Give the definition of the set \mathcal{PR} of recursive primitive functions and, using only the definition, prove that the function $\text{pow2} : \mathbb{N} \rightarrow \mathbb{N}$, defined by $\text{pow2}(y) = 2^y$, is primitive recursive.

Give the definition of $2^n : N \rightarrow N$, proving that this is primitive recursive. This exercise is based on the one above. So, define (thanks to zero and successor function):

$$\text{double}(0) = 0$$

$\text{double}(n + 1) = s(s(\text{double}(n)))$ (doubling the value of 2^n , correctly stating this is equivalent of multiplying by 2)

$$2^0 = s(z, 0)$$

$$2^{n+1} = \text{double}(2^n)$$

We need to decompose the function in natural terms. Specifically, $2^0 = 1$, $2^{n+1} = 2 * 2^n$

The basic building blocks for primitive recursive functions include:

1. Zero function ($Z(x)$): $Z(x) = 0$ for all x .
2. Successor function ($S(x)$): $S(x) = x + 1$ for all x .
3. Projection functions ($Proj(x, i)$): $Proj(x, i) = x[i]$ for all x and for each natural number i

The right solution is simply multiplying continuously by 2 recursively.

Solution: We define $pow2 : \mathbb{N} \rightarrow \mathbb{N}$:

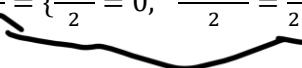
$$\begin{cases} pow2(0) = 1 \\ pow2(y + 1) = double(pow2(y)) \end{cases}$$

where $double(x)$ can be defined by primitive recursion as

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

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

We define the basic functions:

- $\frac{0}{2} = 0$
 - $\frac{n+1}{2} = \left\{ \begin{array}{l} \frac{0+1}{2} = 0, \quad \frac{n+1+1}{2} = \frac{n}{2} + 1 \end{array} \right.$
- 

Again: given the basic operations, we know we're dividing, so it can be a good idea to use the remainder function for the division by 2 and then probably the negated sign one to check if the division correctly holds and gives a result.

First we define the function $\overline{sg} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\overline{sg}(x) = 1$ if $x = 0$ and $\overline{sg}(x) = 0$ otherwise:

$$\begin{cases} \overline{sg}(0) = 1 \\ \overline{sg}(x + 1) = 0 \end{cases}$$

Then the function $rm_2 : \mathbb{N} \rightarrow \mathbb{N}$ which returns the remainder of the division of x by 2:

$$\begin{cases} rm_2(0) = 0 \\ rm_2(x + 1) = \overline{sg}(rm_2(x)) \end{cases}$$

Finally the function $half : \mathbb{N} \rightarrow \mathbb{N}$ can be defined as:

$$\begin{cases} half(0) = 0 \\ half(x + 1) = half(x) + rm_2(x) \end{cases}$$

Exercise 2.2(p). Give the definition of the set \mathcal{PR} of primitive recursive functions and, using only the definition, prove that the characteristic function χ_A of the set $A = \{2^n - 1 : n \in \mathbb{N}\}$ is primitive recursive. You can assume, without proving it, that sum, product, sg and \overline{sg} are in \mathcal{PR} .

$$A = \{2^n - 1 \mid n \in \mathbb{N}\}$$

$$X_A(x) = \{1 \text{ if } x \in A, 0 \text{ otherwise}\}$$

We first define a primitive recursive function for A , something like:

$$f(0) = 1$$

$$f(2^y) = 1 \text{ for } y > 0$$

More generally, given this is defined over n values, we observe $a(n) \in N$, so the function is defined as follows:

$$a(0) = 0, a(n + 1) = 2 * a(n) + 1$$

given for the last part this is formalizing $2^n = 2 * a(n)$ and adding one to form the recursive case.

One can possibly define a function which simply checks if we find exactly the n^{th} value needed to prove this property (sign/negated sign, here we use this one to immediately discard if we don't find the right value) and simply check if the recursive value is different from the one before, this way proving the recursion to be consistent.

Now define $chk : \mathbb{N}^2 \rightarrow \mathbb{N}$, in a way that $chk(x, m) = 1$ if there exists $n \leq m$ such that $x = a(n)$ and 0 otherwise. It can be defined by primitive recursion as follows:

$$\begin{cases} chk(x, 0) &= \bar{s}\bar{g}(x) \\ chk(x, m + 1) &= chk(x, m) + eq(x, a(m + 1)) \end{cases}$$

Hence we can deduce that $chk \in \mathcal{PR}$ by the fact that $y\dot{-}1$ and $x\dot{-}y$ are in \mathcal{PR} , and observing that $eq(x, y) = \bar{s}\bar{g}(x\dot{-}y + y\dot{-}x)$, hence also such function is in \mathcal{PR} . We conclude by noting that $\chi_A(x) = chk(x, x)$. \square

21.2 TUTORING 2: EXERCISES ON DIAGONALIZATION AND PARTIAL RECURSIVE FUNCTIONS

We have the set of partial recursive functions R which has composition/primitive recursion and zero-successor/transfer function.

Also the set R_0 has all of this but also in unbounded in n . So, $R_0 \subseteq \supseteq ?R \cap \text{Tot}$

It would be nice to have minimalisation in two ways, so:

$C = R, R \rightarrow C \rightarrow R$. We start with something total, and we end with something total, which is the minimalisation.

$C_P^1(\vec{x}, \mu t, j_p^n(\vec{x}, t)) \in R_0$ (we need to recall this one, which allows us to define a vector of values which can be minimized and be bounded definitely)

The general idea is to use step by step with $f^{-1}(y) = \mu x. f(x) = y$.

So the idea is creating a function in C and showing via inverse this is also in C .

The idea for diagonalization is creating something that is total but shown as not computable. We want to:

$f(x) = f(x + 1) \rightarrow$ find a function which has the property of an infinite amount not computable, but total

What we're saying is: there exists a number ϕ_n which is different from the other inputs, and it's taken to show non computability.

Take something like: $0 = 0 \neq \phi_n(2) 0 0 \dots$

$$f(x) = \begin{cases} 0, & \text{if } \text{rem}(x, 3) \neq 2 \\ 0, & \text{if } \phi_{\frac{x-3}{2}} \uparrow \text{ and } \text{rem}(x, 3) = 0 \\ \phi_{\frac{x-3}{2}} + 2, & \text{otherwise} \end{cases}$$

$$\phi_{\frac{x-3}{2}}(x) = \uparrow 0 \text{ and } y, \text{ for } y + 1$$

$$f(3n) = f(3n + 1) = 0$$

We're simply saying: "we want to make the input different over a specific value".

- The function is designed so that when x is a multiple of 3 ($3n$), $f(x)$ is set to 0, making it distinct.
- This distinct value (0) for input $x = 3n$ is used to demonstrate non-computability.

The general concept is to create a function with certain inputs (in this case, multiples of 3) that are distinctly different from the rest. These distinct inputs are chosen in such a way that their behavior showcases non-computability.

It's a clever use of diagonalization to introduce distinctness into the function, making it appear non-computable while ensuring its total definition.

Exercise

f total non computable, $\text{img}(f) = \{2^n \mid n \in N\} \Leftrightarrow$ (1) non computable, (2) total (easy), (3) $f(x)$ needs to be a power of 2.

$$f(x) = \begin{cases} 2 \text{ (any value power of 2 anyway)}, & , \text{if } \phi_x(x) \uparrow \\ 2, & \text{if } \phi_x(x) \downarrow \neq 2 \\ 4, & \text{if } \phi_x(x) \downarrow = 2 \end{cases}$$

$$\rightarrow \text{img}(f) \subseteq \{2^n \mid n \in N\}$$

$$\rightarrow \text{img}(f) = \{2^n \mid n \in N\}$$

$$2^n \neq \phi_n(x), \phi_x(x) = n$$

$$2^n = n$$

The values in the image grow faster and are different from each other: $\frac{1}{0}, \frac{2}{1}, \frac{4}{2}, \text{etc.}$ This distinctness and the non-computable nature of f make it challenging to list or generate the entire image of f .

The exercise defines $f(x)$ based on the behavior of $\phi_x(x)$, which is the computation of a function indexed by x on input x . If $\phi_x(x)$ does not halt, $f(x)$ is set to 2. If $\phi_x(x)$ halts but doesn't equal 2, $f(x)$ is set to 2^n , where n is the value of $\phi_x(x)$. If $\phi_x(x)$ equals 2, $f(x)$ is set to 4.

Exercise

f not computable, total, $g(x) = \prod_{y < x} f(y)$ computable.

(E.g. $f(0), f(1), f(2) \dots$ not computable, $f(0) * f(1), f(0) * f(1) * f(2)$ computable)

Say for example: $\frac{g(x)}{g(x-1)} = f(x)$

$$f(0) = 0 \rightarrow f(x) = \begin{cases} 0, & \text{if } x < 0 \\ 0, & \text{if } \phi_{x-1}(x) \uparrow \\ \phi_{x-1}(x) + 1, & \text{otherwise} \end{cases}$$

The key element related to diagonalization is the use of the expression $\phi_{x-1}(x)$ in the definition of $f(x)$. This expression represents the computation or evaluation of a function indexed by $x - 1$ on input x . Specifically, it represents the behavior of some program ϕ indexed by $x - 1$ when given the input x .

The use of diagonalization is a critical part of the definition. It ensures that $f(x)$ is different from any computable function. This is achieved by defining $f(x)$ in a way that depends on whether the program $\phi_{x-1}(x)$ halts or not. The fact that it includes a condition for when the program halts (adding 1 to the result) and when it doesn't (remaining 0) is what makes $f(x)$ non-computable.

21.3 TUTORING 3: SMN-THEOREM EXERCISES

The smn-theorem allows us to partially apply arguments to function.

The formal definition is:

$$\forall m, n \geq 1, \quad \exists s : N^{m+1} \rightarrow N \text{ s.t. } \forall e \in N, x \in N^m, y \in N^n$$

You get a function s total and computable

$$|W_{s(x)}| = 2x$$

$$|E_{s(x)}| = x$$

Usually, we can think a function with two arguments, something like

$$f(x, y) = \begin{cases} qt(2, y), & y < 2x \\ \uparrow, & \text{otherwise} \end{cases}$$

Remember to be able to handle edge cases (e.g. when y holds a value, being careful it is $< 2x$). Let's try to limit ourselves to the first x values. Let's try to define "something for all x ", which in this case might be the quotient function. This should work as follows:

$$\frac{2x}{2} = x \text{ for the } y \text{ part inside } f(x, y) \text{ and the function becomes } \frac{2x-1}{2} = x - 1$$

The smn-theorem will give us, by construction:

$$S(x)(y) = f(x, y)$$

By construction we have:

$$|W_{s(x)}| = |\{y \mid f(x, y) \downarrow\}| = |\{y \mid y < 2x\}| = 2x$$

$$|E_{s(x)}| = |\{qt(2, y) \mid y < 2x\}| = |\{z \mid z < x\}| = x$$

Exercise S function total and computable, with

$$W_{s(x,y)} = \{z \mid x * z = y\}$$

There is no condition the co-domain.

We can define a function on three arguments in which we can bound all values here.

$$f(x, y, z) = \begin{cases} 0, & \text{if } x * z = y \\ \uparrow, & \text{otherwise} \end{cases}$$

In this case, one can also try to take the definition by cases with partial recursion. It also uses unbounded minimalisation. We're not defining a specific w value, but just to get to our condition.

$$\mu w. (x * z -\cdot y) + (y -\cdot x * z)$$

Let's give another example to define unbounded minimalisation:

$$g(x) = \begin{cases} 2x, & \text{if } x \bmod 3 = 1 \\ \uparrow, & \text{otherwise} \end{cases} = 2x + \mu w. rm(3, x)$$

Now for a slightly different function:

$$\begin{aligned}
 g(x) &= \begin{cases} x, & \text{if } x \bmod 2 = 0 \\ 2x, & \text{if } x \bmod 3 = 1 \\ \uparrow, & \text{otherwise} \end{cases} \\
 &= x * \overline{sg}(rm(2, x)) + 2x(rm(2, x) * eq(\dots)) + \mu w (rm(2, x) * \overline{sg}(eq \dots))
 \end{aligned}$$

$x \text{ if mod } 2 = 0$ $2x \text{ if } x \bmod 2 \neq 0 \text{ and } x \bmod 3 = 1$
 0 otherwise

Exercise

$$W_{S(x)} = P \text{ (even numbers; in other words } \{2n \mid n \in N\}$$

$$|E_{S(x)}| = 2x$$

$$f(x) = \begin{cases} rm(2x, \frac{y}{2}), & \text{if } rm(2, y) = 0 \\ \uparrow, & \text{otherwise} \end{cases}$$

We want to consider y can be an even number and we want to consider all cases in which is defined for y and $< 2x$. What we want to achieve:

$$y \text{ even} \Rightarrow y = 2n \forall n, \quad \frac{y}{2} = \frac{2n}{2} = n \forall n$$

We use the remainder to actually limit (thinking of a function plot) "all the possibilities to have it under $2x$ ".

$$W_{S(x)} = \{y \mid rm(2, y) = 0 \text{ and } x \neq 0\} = \{y \mid y \text{ even}\} = P$$

$$|E_{S(x)}| = \left| \left\{ m, n \left(\frac{y}{2}, 2x - 1 \right) \mid y \right\} \right| = |\{m, n(m, 2x - 1) \mid n\}| = |\{m \mid m \leq 2x - 1\}| = 2x$$

We use m, n just because we want to reach values over the codomain.

21.4 TUTORING 4: R.E. SETS

We want to show the following is r.e.

$$A = \{x \in N \mid |W_x| \geq 2\}$$

We define the set A , which is the set of functions in which the domain is ≥ 2 :

$$A = \{f \mid |\text{dom}(f)| \geq 2\}$$

We're saying that is A is saturated, then it is r.e., which means $A = \{x \mid \phi_x \in A\}$. Given it's a finite property, the set is r.e., so \overline{A} is not r.e.

We prove the program halts in t steps in two inputs (x, y) , so we give a program H halting in those steps and the computation over an hypothetical z , which we minimize $S(x, y, z, t)$. We're minimizing all the values at the same time, searching for the minimum after t steps on the w tuple.

$$sc_A(x) = \mu w. \overline{sg}(H(x, (w)_1, (w)_2) * H(x, (w)_3, (w)_4))$$

We minimize for 0, that's why we use the negated sign function, so we get:

$$= \mu w. \overline{sg}(H(x, (w)_1, (w)_2) * H(x, (w)_3, (w)_4))$$

We want to show the following is r.e.

$$A = \{x \in N \mid W_x \leq E_x\} = \{x \mid \phi_x \in A\}$$

$$A = \{f \mid \text{dom}(f) \subseteq \text{cod}(f)\} \text{ saturated} \Rightarrow \text{not recursive}$$

In words: decide if a set is recursive or not depends on the nature of domain/codomain and possible reductions over the characteristic function of the halting problem K , which is not recursive.

$$\overline{K} \leq A \quad x \in \overline{K} \Leftrightarrow f(x) \in A \Rightarrow A \text{ not r.e.}$$

$A \leq K \rightarrow A$ is r.e.

$$x \in \overline{K} (\phi_x(x) \uparrow) \Leftrightarrow g(x) \in A \quad W_{g(x)} \leq E_{g(x)}$$

We are looking for an index, so this is the smn-theorem, which is $h(x, y) \rightarrow \phi_{g(x)}(y)$

$$h(x, y) = \begin{cases} y, & \neg H(x, x, y) \\ y + 1, & \text{otherwise} \end{cases}$$

The point is that it differs “on at least one index”:

$$\phi_x(x) [0 \ 1 \ 2 \ \dots \ n]$$

$$W_{g(x)} = N$$

$$E_{g(x)} = N \setminus \{m\}$$

and eventually we get an index n which was computed by the original program.

$$\phi_x(x) \uparrow \Rightarrow W_{g(x)} \subseteq E_{g(x)}$$

- $\phi_x(x) \uparrow \Rightarrow H(x, x, y)$ always finite

$$\Rightarrow h(x, y) = \phi_{g(x)}(y) = y$$

$$W_{g(x)} = N \subseteq N = E_{g(x)}$$

- $\phi_x(x) \downarrow \Rightarrow W_{g(x)} \not\subseteq E_{g(x)}$

$$\exists m \forall m \geq n, H(x, x, m)$$

$$W_{h(x)} = N$$

$$E_{g(x)} = \{0 \leq j \leq n + 1\} \cup \{z + 1 \mid z \geq n\}$$

$$n \in W_{g(x)}, n \notin E_{g(x)}$$

We want to know if the following is recursive or not (spoiler: it's not)

$$A = \{x \in N \mid W_x \cap E_x = \emptyset\}$$

In a less fancy way, we can write:

$$A = \{f \mid \text{dom}(f) \cap \text{cod}(f) = \emptyset\} \quad A = \{x \in N \mid \phi_x \in A\}$$

If we can prove i.e. A is r.e., \overline{A} won't be r.e (and viceversa holds).

$$sc_{\overline{A}}(x) = \mathbf{1}(\mu w. H(x, y, t) \wedge S(x, z, y, t))$$

also written, using the encoding in tuples,

Written by Gabriel R.

$$sc_{\bar{A}}(x) = \mathbf{1}(\mu w. H(x, (w)_1, (w)_2) \wedge S(x, (w)_3, (w)_4, t))$$

if $\exists y$ s.t. then $y \in W_x$ and s.t. $y \in E_x$

$$\Rightarrow y \in W_x \cap E_x \Rightarrow W_x \cap E_x \neq 0$$

$\Rightarrow \bar{A}$ is r.e. and so A is not recursive and is not r.e.

21.5 TUTORING 5: R.E. SETS AND REDUCTION

We want to know if the set is recursive/r.e. (this exercise is from the exam of last year):

B non empty set, finite, $A = \{x \mid E_x \cap B \neq 0\}$

The set is not recursive, because the codomain intersected with x is not defined.

$$A = \{f \mid \text{cod}(f) \cap B \neq 0\}$$

$$A = \{x \mid \phi_x \in A\}$$

Using Rice's theorem, the set is not recursive.

We can also use an index $b \in B$ having $g(x) = b = \phi_b(x), e \in A, A \neq 0$

We give the empty function, which always diverges $h(x) \uparrow. \phi_e = h, e_1 \notin A, A \neq N$.

Now: the set A is not recursive. We want to understand if it is r.e., for example checking if x is in the set.

Proving it is r.e., we show the semi-characteristic function, searching for a specific value:

$$sc_A(x) = \mu z(\dots)$$

There's gotta be an element in the intersection: $y \in E_x$ and $y \in B$. We use the function if another one halts in a number of steps H and S , computed on the same value (which has as a first argument the function computed, specifically the index, on which input

$$\phi_x(w) = y$$

$$S(x, w, y, t)$$

Use the problem conditions: given the set is finite, we use the conditions in our search to find eventually some elements (for example $(y = b_1) \vee (y = b_2) \vee \dots (y = b_n)$). We use w as a tuple of three elements $(h = (z, y, t))$

$$sc_A(x) = \mathbf{1}(\mu w. S(x, (w)_1 y, (w)_3) \wedge (y \in B))$$

This will terminate "if there is such y ". We put the 1 because the semicharacteristic function has to return 1. Given there is semicharacteristic function, A is r.e.

We need to think still about \bar{A} . This is not recursive. Also it is not r.e., then A would be recursive, but it is not. Starting from finite and total, we have everything done.

Other exercise:

f total computable, $A = \{x \mid x \in f(W_x) \cup E_x\}$

(where $f(W_x)$ can mean that, having a set B , $f(B) = \{f(y) \mid y \in B\}$)

We cannot apply Rice theorem, given there are not only domain and codomain, but also x .

Let's start from the definition: $x \in f(W_x) \vee x \in E_x$ and so it is computable as a search.

We rewrite the condition simplifying bit by bit:

$$x \in \{f(y) \mid y \in W_x\}$$

$$x = f(y) \wedge y \in W_x$$

So, the function would be computable step by step.

Consider something that gives x in output, so $x \in E_x$, with $\phi_x(z) = x$

We want the function x to stop with input z with t steps, so $S(x, z, x, t)$.

The function will stop eventually, so $y \in W_x$, given x is total, so $\phi_x(y) \downarrow$, so $H(x, y, t)$. Again, w represents a tuple. We check the y (so, the third element in tuple, given $w = (z, t, y)$, halting effectively in t steps over the search of y).

$$sc_A(x) = \mathbf{1}(\mu w. S(x, (w)_1, x, (w)_2) \vee ((x = f((w)_3)) \wedge H(x, (w)_3, (w)_2)))$$

The set is r.e.; we don't know if it is recursive or not. We ask then about the complement of A , so \overline{A} . If is it not r.e., we use a reduction or use Rice-Shapiro.

We assume the function is computable $x = f(y)$, and we use an index e such that $f = \phi_e$, in this way $\phi_e(y) = x$.

We know \overline{A} not r.e., so C not r.e. know $C \leq \overline{A}$. This way $x \in C \Leftrightarrow s(x) \in \overline{A}$.

There is a function which returns the index of another function, which is in this case $\phi_{s(x)}(y)$, which uses the smn-theorem. So, there is a function of two arguments which exists, call it s , $\exists s$ total computable $s.t. \phi_{s(x)}(y) = g(x, y)$. To represent it, we use the negation of K , so negation of the halting set, which is \overline{K} . If $x \in \overline{K}$ it means that $\phi_x(x) \uparrow \Rightarrow s(x) \in \overline{A} \Leftrightarrow s(x) \notin E_{s(x)} \wedge s(x) \notin f(W_{s(x)})$.

The idea is $E_{s(x)} = W_{s(x)} = \emptyset$. So $x \notin \overline{K}$ (so $x \in K$), we have $\phi_x(x) \downarrow \Rightarrow s(x) \in A$

This says $s(x) \in f(W_{s(x)}) \cup E_{s(x)}$. The idea is $E_{s(x)} = N$, so it does not compute when undefined, otherwise there is a function which is defined for all natural numbers (so, identity function).

$$g(x, y) = \begin{cases} \uparrow, & \phi_x(\overline{x}) \uparrow \\ y, & \phi_x(x) \downarrow \end{cases} = \begin{cases} y, & \text{if } \phi_x(x) \downarrow = sc_K(x) \\ \uparrow, & \text{otherwise} \end{cases}$$

The other condition $s(x) \in \overline{A} \Rightarrow \phi_x(x) \uparrow$ is not so easy to treat and does not make us conclude anything special. Given the function is defined and it is 1 as a value, we will get $y * sc_K(x)$.

$\phi_{s(x)}$ can converge and $\phi_x(x) \downarrow x \in K = y$ otherwise diverges and $\phi_x(x) \uparrow, x \in K = \uparrow$, proving this is the actual reduction function.

If the function is in the complement set, it is not in the other and we want to understand if it is in the union or not. So, $x \in \overline{A}, x \notin A, \neg(x \in f(W_x) \cup E_x), x \notin f(W_x) \wedge x \notin E_x$

The search “will never end”: $\mu y. \phi_x(y) = x, \forall y, \phi_x(y) \uparrow \text{or } \phi_x(y) \neq x$

Other exercise:

$A = \{x \mid \phi_x \text{ total}\}$. Is this recursive/r.e. and also the complement.

We get the set of computable functions $A = \{f \mid f \text{ total}\}$ and we have $A = \{x \mid \phi_x \in A\}$. Because of Rice, this is not recursive. To show it is total, we would need to do it on every possible input.

Again, we use reduction.

$$\overline{K} \leq_m A$$

$$x \in \overline{K} \Leftrightarrow s(x) \in A$$

$$\phi_{s(x)} = g(x, y) = \begin{cases} 0, & \neg H(x, x, y) \\ \uparrow, & \text{otherwise} \end{cases} = \mu w. H(x, x, y)$$

by smn-theorem

$$x \in \overline{K}, \phi_x(x) \uparrow \Rightarrow s(x) \in A, \phi_{s(x)} \text{ total}$$

$$x \notin \overline{K} (x \in K) \Rightarrow s(x) \notin A, \phi_{s(x)} \text{ not total and } \uparrow \text{ after some point}$$

The deduction from the first of the two hypotheses is that the function does not halt and when it does, there is an element only after some point t_0 , so $\neg H(x, x, y)$ and $H(x, x, t_0)$ and so $g(x, y) \uparrow$.

(This is only the first direction: A is not r.e.)

For the complement, we want to understand if the set does or does not halt form some input and the idea is to leverage then \overline{A} not r.e., because the search requires \forall steps $\neg H(\dots)$.

Again, we use as a reduction $\overline{K} \leq_m \overline{A}$, so:

- $x \in \overline{K}, \phi_x(x) \uparrow s(x) \in \overline{A}, \phi_{s(x)} \uparrow \text{ always undefined}$
- $x \in K, \phi_x(x) \downarrow, s(x) \in A, \phi_{s(x)} \downarrow \text{total}$

And so $g(x, y) = sc_K(x)$.

Consider also, then if A is r.e., also \overline{A} is r.e., then both are recursive. In all other combinations, they will not be recursive.

21.6 TUTORING 6: R.E./RICE-SHAPIRO EXERCISES

We start from this exercise asked by the audience:

Exercise 8.5. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \exists y, z \in \mathbb{N}. z > 1 \wedge x = y^z\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

If we use bounded minimalization, there will always be an answer (correct bound) for the set:

$$\mu w < x^2. z > 1 \wedge x = y^z$$

Considering: $y < x, z < x$ and we see $z = \text{rem}(x, w)$ and $y = \frac{w}{x}$

Substituting $x = y^{z+2}$ to have only mathematical operations, we have:

$$\begin{aligned} &= \mu w < x^2 \\ x &= \left(\frac{w}{x}\right)^{\text{rem}(x,w)+2} \\ &\left| x - \left(\frac{w}{x}\right)^{\text{rem}(x,w)+2} \right| \end{aligned}$$

The important part is that we are trying to find an encoding such that there is only one value which can be found.

Given we are looking for it, we define:

$$sg\left(x^2 - \left(\frac{w}{x}\right)^{\text{rem}(x,w)+2}\right)$$

This approach above it is linear logic, this following one is mathematic:

$$\begin{aligned} f(x, z) &= x - \mu z \leq x. \neg(x = y^2) \\ g(x) &= x - \mu z \leq x. \overline{sg}(f(x, z + 2)) \end{aligned}$$

Exercise 7.11. Let $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$ be the function encoding pairs of natural numbers into the natural numbers. Prove that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is computable if and only if the set $A_f = \{\pi(x, f(x)) | x \in \mathbb{N}\}$ is recursively enumerable.

We define, given f is computable, we say A_f is r.e.

We write the semicharacteristic function as follows:

$$\begin{aligned} sc_A(y) &= \begin{cases} 1, & \text{if } \exists x. \phi(x, f(x)) = y \\ \uparrow, & \text{otherwise} \end{cases} \\ &= \mathbf{1}(\mu x. \pi(x, f(x)) = y) \end{aligned}$$

Trying the proof on the other side:

A_f is r.e. $\Rightarrow f$ computable

$$\begin{array}{c} \swarrow \quad \searrow \\ f(x) = y \Leftrightarrow \pi(x, y) \in A_f \end{array}$$

\rightarrow search $y \rightarrow$ compute $sc_A(\pi(x, y)) \rightarrow$ if it halts, return y

Written by Gabriel R.

The search can actually be phrased like (index of function, input of the function, how many steps and take the first component

$$f(x) = \mu w. H(e, \pi(x, (w)_1), (w)_2)_1$$

sc_A computable and there exists e s.t. $\phi_e = sc_A$

Exercise 8.71. Classify the following set from the point of view of recursiveness

$$A = \{x \mid W_x \cup E_x = \mathbb{N}\},$$

i.e., establish if A and \bar{A} are recursive/recursively enumerable.

The set is saturated, hence this is not recursive. Specifically:

$$A = \{f \mid \text{dom}(f) \cup \text{cod}(f) = N\} \text{ with } A = \{x \in N \mid \phi_x \in A\}$$

A saturated \Rightarrow not recursive

Idea: This is not r.e. and we need to look at the complement

$$\begin{aligned} \bar{A} &= \{x \in N \mid W_x \cup E_x \neq N\} \\ &\rightarrow \exists y \notin W_x \cup E_x \Rightarrow y \notin W_x \wedge y \notin E_x \end{aligned}$$

A is r.e. $\Rightarrow \forall f (f \in A \Leftrightarrow \exists \theta \subseteq f, \theta \text{ finite}, \theta \in A)$

1) $\exists f \notin A \wedge \theta \subseteq f, \theta \text{ finite}, \theta \in A$

2) $\exists f \in A, \forall \theta \subseteq f, \theta \text{ finite}, \theta \notin A$

Case 1 does not work: $f \notin A, \theta \subseteq f \rightarrow \exists y \in \text{dom}(f) \wedge y \notin \text{cod}(f)$

Let's try with the second case:

$f \in A, \theta \subseteq f, \text{dom}(f) \cup \text{cod}(f) = N$, where $\text{dom}(f)$ is infinite, while $\text{dom}(\theta)$ is finite

Let $f = id, \text{dom}(f) = N \Rightarrow \text{dom}(f) \cup \text{cod}(f) = N \Rightarrow f \in A$

Consider $\theta \subseteq f, \theta \text{ finite} \Rightarrow \text{dom}(\theta) \text{ finite}, \text{cod}(\theta) \text{ finite} \Rightarrow \text{dom}(\theta) \cup \text{cod}(\theta) \text{ finite} \neq N \Rightarrow \theta \notin A$

Let's prove it for all conditions for the second part.

$$f \notin \bar{A} \quad (f \in A) \quad \theta \subseteq f, \theta \text{ finite}, \theta \in \bar{A}$$

$$f = id \in A$$

We use the empty function, $\theta = \emptyset$ (a function which “as much undefined as possible”), $\theta \in \bar{A}$ and so by Rice-Shapiro, this is not r.e.

Exercise 8.55. Classify the following set from the point of view of recursiveness

$$B = \{x \mid \varphi_x(0) \uparrow \vee \varphi_x(0) = 0\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

We argue this is not r.e. and we will use Rice-Shapiro. \bar{A} r.e., $\theta = \emptyset$

21.7 TUTORING 7: R.E. AND REDUCTIONS

Consider:

$$f(x) = \begin{cases} \phi_x(x) + 1, & \text{if } \phi_y(y) \downarrow \forall y \leq x \\ 0, & \text{otherwise} \end{cases}$$

To be complete, this can be written as $\mu w. (\overline{s}\bar{g}(x, x, (w)_1, (w)_2) * sg(1 - x))$ (not necessary)

So, the reasoning would be:

$$\begin{aligned} x = 0 \quad & \phi_0(0) \downarrow \\ x = 1 \quad & \phi_0(0) \downarrow \wedge \phi_1(1) \\ x = 2 \quad & \phi_0(0) \downarrow \wedge \phi_1(1) \wedge \phi_2(2) \end{aligned}$$

Let i s.t. $\phi_j(i) \downarrow \forall j < i$ and $\phi_i(i) \uparrow$ and you can write:

$$f(x) = \begin{cases} \phi_x(x) + 1, & \text{if } x < i \\ 0, & \text{otherwise} \end{cases}$$

Consider the reduction $\bar{K} \leq_m B$

$$x \in \bar{K} \Leftrightarrow f(x) \in B$$

This can be used say with $B = \{x \mid \phi_x \text{ inc}\}$ which means $y < z$ in $\text{dom}(\phi_x)$ and $\phi_x(y) < \phi_x(z)$

$$f(x) = \begin{cases} y, & \text{if } \neg H(x, x, y) \\ 0, & \text{otherwise} \end{cases}$$

Is there a total computable function f such that $D = \{x \in \mathbb{N} \mid f(x) \neq \phi_x(x)\}$ is finite?

We can see it as:

$$f(x) = \begin{cases} a_1, & \text{if } x = x_1 \\ 2, & \text{if } x = x_2 \\ \dots \\ a_n, & \text{if } x = x_n \\ \phi_x(x), & \text{otherwise} \end{cases}$$

With $a_i \neq \phi_x(x)$ because it's built to be different from every value.

We have two functions f, g total and computable and find

$$f(x) \neq \phi_x(x) \quad \forall x \in K$$

$$g(x) \neq \phi_x(x) \quad \forall x \notin K$$

Consider $f = \phi_e$ and $f(e) \neq \phi_e(e)$ and so f cannot be computable. When $f(e) \uparrow$ it means $f \in K$ and so $f(e)$ diverges, because f is total.

For $g(e) \downarrow \neq \phi_e(e) \uparrow e \notin K \quad g(x) \downarrow \forall x \in K \Rightarrow g$ any total function

Written by Gabriel R.

f almost total $f(x) \uparrow$

$f(x) \uparrow$ only if $x \in D$ finite

$A = \{x \mid \phi_x \text{ almost total}\}$

A saturated so you have $A = \{x \mid \phi_x \text{ almost total}\} = \{x \mid \phi_x \in \mathcal{A}\}$

By Rice-Shapiro:

- A is not r.e. because $\text{id} \in A$ but $\theta = \emptyset \notin A$.

$\overline{A} = \{x \mid \phi_x \text{ not almost total}\} = \{x \mid \phi_x(y) \uparrow \text{ for infinite } y\}$

- \overline{A} is not r.e.

$\text{id} \in A, \theta \text{ finite} \subseteq \text{id} \rightarrow \text{infinite } y, \theta(y) \uparrow \Rightarrow \theta \in \overline{A}$

21.8 TUTORING 8: ALL KINDS OF EXERCISES

Let's try to define the following function by primitive recursion:

$\text{And}(x, y)$

$\text{And}(0, 0) = 0, \text{And}(0, 1) = 0, \text{And}(1, 0) = 0, \text{And}(1, 1) = 1$

We can define:

$$\begin{cases} \text{And}(x, 0) = 0 \\ \text{And}(x, y + 1) = x \end{cases}$$

Underlying, there would be two functions, as follows:

$$\begin{cases} f(0) = 0 \\ f(x + 1) = 0 \end{cases} \text{ and } \begin{cases} g(0) = 0 \\ g(x + 1) = 1 \end{cases}$$

We can take for example the least power of two, then representing it as a set:

$$\text{lsPow2}(x) = \chi_A = \mu y$$

$$A = \{2^y \mid y \in \mathbb{N}\}$$

It depends on the value you are trying to bound upon, consider:

$$x = 2^y < 2^x$$

$$1 \cdot 2^0 < 2^1 \quad 2 \cdot 2^1 < 2^2 \quad 4 \cdot 2^2 < 2^3$$

Consider $|2^y - x|$ is 0 if $2^y = x$, 1 if $2^y \neq x$.

We can also have $\mu y < x \mid 2^y - x \mid$ which can be seen as y if $2^y = x \rightarrow 1$, x if no $y \rightarrow 0$

So, to consider all cases, we also subtract the value we are looking for, which is x and it becomes, using the sign function.

$$\chi_A = sg|(\mu y < x) * |2^y - x|) - x \text{ or also } sg(x - \mu y < x * |2^y - x|)$$

Is the function:

$$f(x) = \begin{cases} \phi_x(x+1) + 1, & \text{if } \phi_x(x+1) \downarrow \\ 0, & \text{otherwise} \end{cases}$$

computable?

Nope, because it's a basic diagonalization argument; consider for instance function $h(x) = f$ and by usual diagonalization $h(x+1) = f(x) \neq \phi_x(x+1)$ and also $f(x+1) \neq \phi_{x+1}(x)$

Define a total non-computable function f s.t. $f(x) \neq \phi_x(x)$ on a single x .

$$f(x) = \begin{cases} \phi_x(x), & \text{if } x \neq x_0 \\ y_0, & \text{if } x = x_0 \end{cases}$$

Considering $y_0 = \phi_x(y_0)$. Also define:

$$f(x) = \begin{cases} \phi_x(x), & \text{if } \phi_x(x) \downarrow \wedge x \neq x_0 \\ y_0, & \text{if } \phi_x(x) \downarrow \wedge x = x_0 \\ g(x), & \text{if } \phi_x(x) \uparrow \end{cases}$$

(so, a diagonalization argument)

22 MANY SOLVED EXERCISES WITH FULL COMMENTARY

Premise

The exercises solved here are carefully revised, often with professor solution, other times from old tutors, other times solved by me and revised as precisely as possible, both in notation and writing. Here, many more exercises were added looking at past exams, so this can serve as a complement to the good “exercises.pdf” given by the professor.

Here, many exercises were either solved thanks to old tutorings or translated from Italian solutions (mostly complete). Some exercises solved by the old tutor until 21/22 are present (just enter with @unipd mail) [here](#) (with full tutoring lessons and solved exercises). Tutorings of 23/24 are instead present [here](#).

All sections and subsections are named following the same names in the Moodle PDF subsections. Given the overall quality of this file and a single person behind of it, I think you would not complain anyway.

22.1 URM MACHINES

Note: this one is for partial exams. Useful to understand how to use induction, not for much else.

Exercise 1.1(p). Consider a variant, denoted URM^- , of the URM machine obtained replacing the successor instruction $S(n)$ with a predecessor instruction $P(n)$. Executing $P(n)$ replaces the content r_n of register n with $r_n - 1$. Determine the relation between the set C^- of the functions computable by a URM^- machine and the set C of functions computable by a standard URM machine. Is one contained in the other? Is the inclusion strict? Justify your answer.

Solution: It holds that $C^- \subseteq C$ because predecessor is URM-computable. Inclusion is strict because it is possible to prove, inductively on the number of steps, that the maximum of the values contained in the registers at any time is bounded by the maximum value in the initial configuration. As a consequence the successor function is not URM^- computable. \square

It holds that $C^- \subseteq C$ because predecessor is URM-computable. Inclusion is strict because it is possible to prove, inductively on the number of steps, that the maximum of the values contained in the registers at any time is bounded by the maximum value in the initial configuration. As a consequence, the successor function is not URM' computable.

Let us denote by URM' the modified machine. We observe that the URM' machine instructions can be encoded as programs of the standard URM machine.

The instruction $I_j : S(p, n)$ in a standard URM machine can be replaced with a jump to the following routine:

```

SUB : J(p, q, n)
    S(p, n)
    S(q, q)
    J(1, 1, SUB)

```

Similarly, for an instruction $I_j : P(p, n)$ in URM' , we can replace it with:

```

SUB : J(p, q, n)
    P(p, n)
    S(q, q)
    J(1, 1, SUB)

```

The proof proceeds by induction on the number h of S and P instructions in the program.

Base Case ($h = 0$): A program P with 0 instructions of S and P is already a URM program. Trivially, the base case holds.

Inductive Step ($h \geq 1$): Suppose the result holds for h ; we aim to prove it for $h+1$. Assume the program P contains at least one S or P instruction, and let j be the index of an instruction:

$$\begin{aligned} 1 &: I_1 \\ &\vdots \\ j &: S(p, n) \quad \text{or} \quad P(p, n) \\ &\vdots \\ \ell &: I_\ell \end{aligned}$$

We construct a program P_2 using an unused register q ($q = \max(\rho(pP), k) + 1$):

$$\begin{aligned} 1 &: I_1 \\ &\vdots \\ j &: J(1, 1, \text{SUB}) \\ &\vdots \\ \ell &: I_\ell \\ \\ \text{SUB} &: J(p, q, n) \\ &S(p, n) \quad \text{or} \quad P(p, n) \\ &S(q, q) \\ &J(1, 1, \text{SUB}) \end{aligned}$$

The program P_2 , containing h instructions of type S or P , satisfies $f(\mathbf{p}_k)P_2 = f(\mathbf{p}_k)P$. By the inductive hypothesis, there exists a URM program P_1 such that $f(\mathbf{p}_k)P_1 = f(\mathbf{p}_k)P_2$, completing the inductive step.

The inclusion is strict (CC'). For example, the successor function is not URM' computable. Starting from a configuration with all registers at 0, any URM' program, after any number of steps, will produce a configuration with all registers still at 0. This is formally proven by induction on the number of steps.

Exercise 1.2(p). Consider a variant of the URM machine where the jump and successor instructions are replaced by the instruction $JI(m, n, t)$ which compare the content r_m and r_n of registers R_m and R_n and then:

- if $r_m = r_n$, increment register R_m and jump to the address t (it is intended that if t is outside the program, the execution of the program halts).
- otherwise, continue with the next instruction.

Describe the relation between the set \mathcal{C}' of the functions computable by the new machine and the set \mathcal{C} of the functions that can be computed by a standard URM machine. Is one included in the other? Is the inclusion strict? Justify your answers.

Reading the text carefully, it seems the only difference present is a jump only when a certain condition happens: this is intended, as jump and successor are replaced with a new instruction which increments a register only if something happens. This can also be encoded as a jump to a subroutine which actually combines the full jump/successor instruction all in one.

Let's check this one by one:

- $\mathcal{C} \subseteq \mathcal{C}'$: trivial, as every single instruction of \mathcal{C} can be easily encoded in the other, simply not using $JI(m, n, t)$. This can be alternatively encoded as a jump to a subroutine which combines, as said, both jump and successor

$$i_1 : S(m)$$

$$i_1 + 1 : J(m, n, t)$$

- $\mathcal{C}' \subseteq \mathcal{C}$: as they both contain the same instruction length, given they're both close under composition and primitive recursion, the jump will make us intuitively "stay under the \mathcal{C} class", simply by replacing the next instruction with another jump. Because the successor function is defined, this can be easily encoded with the new jump instruction, so instead of having $I_j : S(n)$, you can have $I_j : JI(n, n, j + 1)$.

Exercise 1.3(p). Consider a variant URM^s of URM machine obtained by removing the successor $S(n)$ and jump $J(m, n, t)$ instructions, and inserting the instruction $JS(m, n, t)$, which compares the contents of register 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 \mathcal{C}^s of functions computable by a URM^s machine and the set \mathcal{C} of functions computable by a standard URM machine. Is one included in the other? Is the inclusion strict? Justify your answers.

Any URM^* machine can be simulated by a standard URM machine. To do this, we simply replace each $JS(m, n, t)$ instruction with the following sequence of instructions:

$$J(m, n, l)$$

$$S(m)$$

$$J(l, t, t)$$

Or even simply:

$$J(m, n, t)$$

$$S(m)$$

This sequence of instructions will compare the contents of registers m and n , and if they coincide, it will jump to instruction t . Otherwise, it will increment the m^{th} register and continue with the next instruction.

Therefore, any function that can be computed by a URM^* machine can also be computed by a standard URM machine.

The inclusion is not strict, because while you can somehow make the jump, you cannot make the successor, because you will have to both modify the content of register and then moving to next instruction; we can make the first one, but not the second one. So, URM^* cannot properly encode URM .

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. \square

Proof:

Consider a modified machine denoted by URM'. We assert that URM' machine instructions can be encoded as programs of the standard URM machine.

Let $I_j : A(p, n)$ be an instruction in URM'. We can replace it with a jump to the following routine, where q represents the index of the first register not used by the program (initially containing 0):

```
SUB : J(p, q, j + 1)
S(p, q)
S(q, q)
J(1, 1, SUB)
```

Similarly, for an instruction $I_j : C(p, n)$, we can replace it with a jump to the subroutine:

```
SUB : J(p, q, ZERO)
Z(p, q)
S(p, n)
ZERO J(1, 1, j + 1)
```

In a more formal manner, we prove that $C' \subseteq C$, demonstrating that for any number of arguments k and any program P using both sets of instructions, we can obtain a URM program P_1 such that $f(\mathbf{p}_k)P_1 = f(\mathbf{p}_k)P$.

The proof proceeds by induction on the number h of A and C instructions in the program:

Base Case ($h = 0$): A program P with 0 instructions of A and C is already a URM program, trivially satisfying the base case.

Inductive Step ($h \geq 1$): Suppose the result holds for h ; we aim to prove it for $h+1$. Assume the program P contains at least one A or C instruction, and let j be the index of a C instruction:

```
1 : I1
:
j : A(p, n)
:
ℓ : Iℓ
```

We construct a program P_2 using an unused register q ($q = \max(\rho(pP), k) + 1$):

```
1 : I1
:
j : J(1, 1, SUB)
:
ℓ : Iℓ
J(1, 1, END)
SUB : J(p, q, ZERO)
Z(p, q)
S(p, n)
ZERO J(1, 1, j + 1)
END :
```

The program P_2 , containing h instructions of type A or C , satisfies $f(\mathbf{p}_k)P_2 = f(\mathbf{p}_k)P$. By the inductive hypothesis, there exists a URM program P_1 such that $f(\mathbf{p}_k)P_1 = f(\mathbf{p}_k)P_2$, completing the inductive step.

The inclusion is strict ($C \subseteq C'$). For instance, the successor function is not URM' computable. Starting from a configuration with all registers at 0, any URM' program, after any number of steps, will produce a configuration with all registers still at 0. This is formally proven by induction on the number of steps.

Exercise 1.5. Consider a variant of the URM machine, which includes the jump and transfer instructions and two new instructions

- $A(m, n)$ which adds to register m the content of register n , i.e., $r_m \leftarrow r_m + r_n$;
- $C(n)$ which replaces the value in register n by its sign, i.e., $r_n \leftarrow sg(r_n)$.

Determine the relation between the set C' of the functions computable with the new machine and the set C of the functions that can be computed with the URM machine. Is one included in the other? Is the inclusion strict? Justify your answers.

Let's make this exercise discussing the usual two-way implication.

- Let's start considering $C' \subseteq C$, which tries to encode the new URM machine, call it URM_{AC} inside the normal URM. The addition can be easily replaced by a jump to the next instruction in which we are trying to reach the successor of m and for the first value possible after the sum.

We consider replacing $A(m, n)$ with jump to the following subroutine, where x is the following index yet to be reached by computation:

```
SUB: J(n, x, i + 1)
      S(m)
      S(q)
      J(1,1,SUB)
```

- For the sign instruction, this can be encoded to a jump to the following subroutine

```
SUB: J(n, x, j + 1)
      Z(n)
      S(n)
      J(1,1,j + 1)
```

Formally, we are trying to prove that the program P' computes the same function of C' s.t. $f_p'^{(k)} = f_p^{(k)}$ and it holds for induction.

- The base case $h = 0$ is trivial, given $h = h + 1$ and it is trivial to conclude
- The inductive case is such that the length of the program is able to contain inductively all of the instructions, such as:

```
START: I1
      J(1,1,SUB)
      l(P): Il(P)
      END
```

This holds for each subroutine is encoded inside of it and the program P'' is s.t. $f_p''^{(k)} = f_p^k$ and contains the analogous instructions as the first one.

Now, for the remaining implication, $C \not\subseteq C'$, because there's potentially the risk, thanks to the sign instruction, that all instructions could become 0, and so every single register.

Basically the same of this one from 2018-11-12:

Exercise 1.5. Consider a variant of the URM machine, which includes the jump and transfer instructions and two new instructions

- $A(m, n)$ which adds to register m the content of register n , i.e., $r_m \leftarrow r_m + r_n$;
- $C(n)$ which replaces the value in register n by its sign, i.e., $r_n \leftarrow \overline{sg}(r_n)$.

Determine the relation between the set C' of the functions computable with the new machine and the set C of the functions that can be computed with the URM machine. Is one included in the other? Is the inclusion strict? Justify your answers.

Solution: Let us denote by URM^* the modified machine. We observe that the URM^* machine instructions can be encoded as programs of standard URM machine.

The instruction $I_j : A(m, n)$ can be replaced with a jump to the following routine (where we denote by q the index of the first register not used by the program, hence such register initially contains 0)

```
SUB   : J(n, q, j + 1)
      S(m)
      S(q)
      J(1, 1, SUB)
```

Similarly, by indicating again with q the index of an unused register, an instruction $I_j : C(m)$ can be replaced by a jump to the subroutine

```
SUB   : J(n, q, ZERO)
      Z(n)
      S(n)
ZERO  : J(1, 1, j + 1)
```

More formally, we can prove that $C^* \subseteq C$ showing that, for each number of arguments k and for each program P using both sets of instructions we can obtain a URM program P' which computes the same function, i.e., such that $f_{P'}^{(k)} = f_P^{(k)}$.

The proof proceeds by induction on the number h of A and C instructions in the program. The base case $h = 0$ is trivial, since a program P with 0 instructions A and C is already a URM program. Suppose that the result holds for h , let us prove it for $h + 1$. The program P certainly contains at least one A or C instruction. Assume it is a C instruction and call j its index.

```
1     : I1
...
j     : A(m, n)
...
ℓ(P) : Iℓ(P)
```

We build a program P'' , using a register not referenced in P , say $q = \max\{\rho(P), k\} + 1$

```
1     : I1
...
j     : J(1, 1, SUB)
...
ℓ(P) : Iℓ(P)
      J(1, 1, END)
SUB   : J(n, q, ZERO)
      Z(n)
      : S(n)
ZERO  : J(1, 1, j + 1)
END   :
```

The program P'' is such that $f_{P''}^{(k)} = f_P^{(k)}$ and it contains h instructions of type A or C . By inductive hypothesis, there exists a URM program P' such that $f_{P'}^{(k)} = f_{P''}^{(k)}$, which is the desired program.

If the instruction I_j is of type A , we proceed in a completely analogous way, replacing the instruction with its encoding and using the inductive hypothesis.

For the converse implication $\mathcal{C} \subseteq \mathcal{C}^*$ observe, analogously, $Z(n)$ and $S(n)$ can be encoded inside the modified machine. More precisely, given a program P and q_1, q_2 index of registers not used by the program (so initially at 0) consider:

```
 $C(q_1) // fa in modo che  $q_1$  contenga 1$   
 $P'$ 
```

(the text says “to make q_1 contain 1) where P' is obtained by P substituting $Z8M$ with $T(q_2, m)$ and each $S(m)$ with $A(m, q_1)$

Exercise 1.6(p). Consider a variant URM^m of the URM machine obtained by removing the successor instruction $S(n)$ and adding the instruction $M(n)$, which stores in the n th register the value $1 + \min\{r_i \mid i \leq n\}$, i.e., the successor of the least value contained in registers with index less than or equal to n . Determine the relation between the set \mathcal{C}^m of functions computable by the URM^m machine and the set \mathcal{C} of the functions computable by the ordinary URM machine. Is one included in the other? Is the inclusion strict? Justify your answers.

Solution: Observe that the instruction $M(n)$ can be simulated in the URM machine as follows: store in an “unused” register k , an increasing number, which starts from zero. Such a number is

compared with all registers R_1, \dots, R_n until it coincides with one of them. Then the value in register k will be the minimum of registers R_1, \dots, R_m . Its successor is the value to be stored in R_n

$Z(k)$	
$LOOP :$	$J(1, k, END)$
	$J(2, k, END)$
	...
	$J(n, k, END)$
	$S(k)$
	$J(1, 1, LOOP)$
$END :$	$S(k)$
	$T(k, n)$

Conversely, the instruction $S(n)$ can be simulated in the URM^m machine as follows. Assume again that k is the number of a register not used by the program. Then the encoding can be the following:

$T(1, k)$
$T(n, 1)$
$M(1)$
$T(1, n)$
$T(k, 1)$

□

Exercise 1.7(p). Define the operation of primitive recursion and prove that the set \mathcal{C} of URM-computable functions is closed with respect to this operation.

PROOF. Let $f : \mathbb{N}^k \rightarrow \mathbb{N}$ and $g : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ be computable functions. We want to prove that $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ defined through primitive recursion

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

is computable.

Let F, G programs in standard form for f, g . We want a program H for h . We proceed as suggested by the definition.

We start from

x_1	\dots	x_k	y	0	\dots
-------	---------	-------	-----	-----	---------

we save the parameters and we start to compute $h(\vec{x}, 0)$ using F .

If $y = 0$ we are done, otherwise we save $h(\vec{x}, 0)$ and compute $h(\vec{x}, 1) = g(\vec{x}, 0, h(\vec{x}, 0))$ with G . We do the same for $h(\vec{x}, i)$ until we arrive at $i = y$.

As usual we need registers not used by F and G , $m = \max\{\rho(F), \rho(G), k + 2\}$ and we build the program for h as follows:

1	\dots	$m + 1$	\dots	$m + k$	$m + k + 1$	\dots	$m + k + 3$	
\dots	\dots	\dots	\vec{x}	\dots	i	$h(\vec{x}, 2)$	y	0

```

 $T(1, m + 1)$ 
...
 $T(k, m + k)$ 
 $T(k + 1, m + k + 3)$ 
 $F[m + 1, \dots, m + k \rightarrow m + k + 2] \quad // \text{ compute } h(\vec{x}, 0)$ 
 $LOOP : J(m + k + 1, m + k + 3, END) \quad // \text{ i=y?}$ 
 $G[m + 1, \dots, m + k + 2 \rightarrow m + k + 2]$ 
 $S(m + k + 1) \quad // \text{ i = i+1}$ 
 $J(1, 1, LOOP)$ 
 $END : T(m + k + 2, 1)$ 

```

Another exercise of the same kind (2015-09-03)

Define unbounded minimization and define \mathcal{C} is closed with respect to this operation

DEFINITION 6.31. Let $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ be a function. Then the function $h : \mathbb{N}^k \rightarrow \mathbb{N}$ defined through **unbounded minimisation** is:

$$h(\vec{x}) = \mu y. f(\vec{x}, y) = \begin{cases} \text{least } z \text{ s.t.} & \begin{cases} f(\vec{x}, z) = 0 \\ f(\vec{x}, z) \downarrow \quad f(\vec{x}, z') \neq 0 \quad \text{for } z < z' \end{cases} \\ \uparrow & \text{otherwise, if such a } z \text{ does not exist} \end{cases}$$

THEOREM 6.32 (Closure under minimalisation). *Let $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ a computable function (not necessarily total). Then $h : \mathbb{N}^k \rightarrow \mathbb{N}$ defined by $h(\vec{x}) = \mu y. f(\vec{x}, y)$ is computable.*

PROOF. Let F be a program in standard form for f .

Idea: for $z = 0, 1, 2, \dots$ we compute $f(\vec{x}, z)$ until we find zero.

We need to save the argument \vec{x} in a register R_m ($m = \max\{\rho(F), k + 1\}$) such that it is not used by the program F .

So the program for h is obtained as follows:

1	...	k	...	$m + 1$...	$m + k$	$m + k + 1$
		\vec{x}		\vec{x}		z	

$T(1, m + 1)$
 ...

$T(k, m + k)$	$F[m + 1, \dots, m + k + 1 \rightarrow 1]$	$// f(\vec{x}, z) \rightarrow R_1$	□
$J(1, m + k + 2, END)$		$// f(\vec{x}, z) = 0?$	
$S(m + k + 1)$		$// z = z + 1$	
$J(1, 1, LOOP)$			

$LOOP :$ $END :$ $T(m + k + 1, 1)$

Exercise (2019-11-18-solved)

Consider a variant of the URM machine, in which the zero instruction is replaced by the $P(n)$ instruction whose effect is to replace the contents of the register n with its predecessor, so $r_n \leftarrow r_n - 1$.

Show what relationship there is between C' the set of computable functions of the new machine and the set C of computable function with the URM machine. Are one contained inside the other? Is the inclusion strict? Motivate your answers.

Solution

We have that $C' = C$ given the instructions of a machine can be encoded inside the other and to prove $C' \subseteq C$ we show there is a URM program P' and an index $k \in N$ for which there is a URM program P which computes the same function, s.t. $f_P^{(k)} = f_{P'}^{(k)}$ as follows.

Observe $P(n)$ is each instruction, with j its ordering number and $m = \max\{\rho(P'), k\} + 1$ the first register not used by P' which can be encoded by a jump to subroutine:

```

SUB:    J(n,m,j+1)
        S(m)
LOOP:   J(n,m,END)
        S(m)
        S(m+1)
END    T(m+1,n)
        J(1,1,j+1)

```

More formally, we show that for each P' of URM' machine, we obtain a program P s.t. $f_P^{(k)} = f_{P'}^{(k)}$ which does not use $P(n)$ instructions. Proof goes on by induction:

- $h = 0$ which is trivial, because P' is already good
- $h \rightarrow h + 1$, in which case P' has for sure at least a $P(n)$ instruction and consider as before j the index of said instruction. P' will have shape:

	1	:	I_1
	...		
j	$P(n)$		
	...		
$\ell(P')$:	$I_{\ell(P')}$	

We build a program P'' using an index $m = \max\{\rho(P'), k\} + 1$:

```

1      :  I1
...
j      :  J(1,1,SUB)
...
ℓ(P') :  Iℓ(P')
        J(1,1,END)
SUB   :  J(n,m,j+1)
        S(m)
LOOP  :  J(n,m,RES)
        S(m)
        S(m+1)
        J(1,1,LOOP)
RES   :  T(m+1,n)
        J(1,1,j+1)
END   :

```

P'' by hypotheses contains h instructions of type $P(n)$ and is s.t. $f_{P''}^{(k)} = f_P^{(k)}$ which is the desired program.

For the opposite inclusion, we proceed similarly, noting that the instruction $Z(n)$ could be encoded in the URM' machine as follows, where m is, as above, the index of a register beyond the area used by the program (and thus to 0).

```
SUB:    J(n,m,j+1)
        P(n)
        J(1,1,SUB)
```

Similar to this one we have ([2020-11-23](#))

Consider a URM^p variant of the URM machine in which the zero instruction $Z(n)$ is replaced by the predecessor instruction $P(n)$ that decrements the contents of register n , like $r_n \leftarrow r_n - 1$. Stating C^p the set of functions computable by the URM^p machine, establish the relationship between C^p and C

Are they one contained in the other? Is the inclusion tight? Motivate the answers.

Solution

Given $P(n)$ can be encoded inside the URM machine, clearly $C^p \subseteq C$. More precisely, $I_j: P(n)$ can be replaced by a jump to the following subroutine, using q as the index of first register not used by the program (initially at 0):

<i>SUB</i>	: $J(n,q,RIS)$ $S(q+1)$
<i>LOOP</i>	: $J(n,q+1,RIS)$ $S(q)$ $S(q+1)$ $J(1,1,LOOP)$
<i>RIS</i>	: $T(q,n)$ $J(1,1,j+1)$

The routine checks if n contains 0. When it does, there is nothing to do. Otherwise, with R_q starting from 0 and R_{q+1} from 1, it continues to increment the two registers. When R_{q+1} is equal to R_n , we have R_q contains the predecessor.

More formally, $C_p \subseteq C$ so for a number of arguments k , we have a program P' which computes the same function, so $f_P'^{(k)} = f_P^{(k)}$.

The proof goes on by induction. When $h = 0$ is trivial, and when $h = h + 1$ for sure the program will have at least a P instruction and for an index j instruction:

1	: I_1
	...
j	: $P(n)$
	...
$\ell(P)$: $I_{\ell(P)}$

We build a program P'' using a register not used by p , so $q = \max\{\rho(P), k\} + 1$.

1	:	I_1
		...
j	:	$J(1, 1, SUB)$
		...
$\ell(P)$:	$I_{\ell(P)}$
		$J(1, 1, END)$
SUB	:	$J(n, q, RIS)$
		$S(q + 1)$
$LOOP$:	$J(n, q + 1, RIS)$
		$S(q)$
		$S(q + 1)$
		$J(1, 1, LOOP)$
RIS	:	$T(q, n)$
		$J(1, 1, j + 1)$
END	:	

The program P'' is a URM^p program s.t. $f_P''^{(k)} = f_P^{(k)}$ and contains h instructions of P . So, $P?$ is the desired program.

The opposite inclusion and thus equality also applies. In fact, the instruction $Z(n)$ can simply be replaced by an instruction $T(q, n)$, where q is any register not used by the program and thus 0. More precisely, given a URM program P and a fixed number of arguments $k \in \mathbb{N}$, called $q = \max\{\rho(P), k\} + 1$ the index of the first unused register and thus initially 0, replacing in P each instruction $Z(n)$ by instruction $T(q, n)$, is a URM^p program that computes exactly the same function.

22.2 PRIMITIVE RECURSIVE FUNCTIONS

Note: I suggest doing only practice here; useful link to see some primitive recursive functions [here](#). Then proceed by primitive recursion (base case – recursive step) or by bounded minimalization (so, consider both cases in which the variable exists and also it doesn't)

Exercise 2.1(p). Give the definition of the set \mathcal{PR} of recursive primitive functions and, using only the definition, prove that the function $pow2 : \mathbb{N} \rightarrow \mathbb{N}$, defined by $pow2(y) = 2^y$, is primitive recursive.

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

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

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

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

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

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

We can define the primitive recursion “manually”, by crafting for $pow2 : \mathbb{N} \rightarrow \mathbb{N}$ the following:

$$\begin{cases} pow2(0) = 1 \\ pow2(y + 1) = 2 * 2^y = 2 * pow2(y) = double(pow2(y)) \end{cases}$$

where $double(x)$ is a function defined by \mathcal{PR} itself, such that $double : \mathbb{N} \rightarrow \mathbb{N}$:

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

Exercise 2.2(p). Give the definition of the set \mathcal{PR} of primitive recursive functions and, using only the definition, prove that the characteristic function χ_A of the set $A = \{2^n - 1 : n \in \mathbb{N}\}$ is primitive recursive. You can assume, without proving it, that sum, product, sg and \overline{sg} are in \mathcal{PR} .

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

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

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

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

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

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

To prove the characteristic function χ_A is primitive recursive, we define it $\in PR$, considering a as an element inside recursion s.t. $a(n) : n \in \mathbb{N}$:

$$\begin{cases} a(0) = 0 \\ a(n+1) = 2^{n+1} + 1 = 2 * a(n) + 1 \end{cases}$$

This is the general case for $a(n)$, so we want to handle all cases in which $x = a(n)$ (and for which the power of $n - 1$ is defined).

This can be defined by primitive recursion as a new function called $val : \mathbb{N}^2 \rightarrow \mathbb{N}$ s.t.:

$$\begin{cases} val(x, 0) = \overline{sg}(x) \\ val(x, n+1) = val(x, n) + \overline{sg}(x, x-1) \end{cases}$$

(where the negated sign makes you obtain 0, while the other considers the recursive definition of the function and basically defines the actual value x with the previous one ($x - 1$), making it defined for all cases).

Definitely more correctly (I tried here a different solution, given the one by the teacher comes out of nowhere for this second part at least):

Now define $chk : \mathbb{N}^2 \rightarrow \mathbb{N}$, in a way that $chk(x, m) = 1$ if there exists $n \leq m$ such that $x = a(n)$ and 0 otherwise. It can be defined by primitive recursion as follows:

$$\begin{cases} chk(x, 0) = \overline{sg}(x) \\ chk(x, m+1) = chk(x, m) + eq(x, a(m+1)) \end{cases}$$

Hence we can deduce that $chk \in \mathcal{PR}$ by the fact that $y-1$ and $x-y$ are in \mathcal{PR} , and observing that $eq(x, y) = \overline{sg}(x-y + y-x)$, hence also such function is in \mathcal{PR} . We conclude by noting that $\chi_A(x) = chk(x, x)$. \square

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

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

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

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

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

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

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

We need to prove that the function *half* can be obtained from the basic functions (1), (2) and (3), using primitive recursion and generalized composition. One can proceed as follows.

First we define the function $\overline{sg} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\overline{sg}(x) = 1$ if $x = 0$ and $\overline{sg}(x) = 0$ otherwise:

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

Then the function $rm_2 : \mathbb{N} \rightarrow \mathbb{N}$ which returns the remainder of the division of x by 2:

$$\begin{cases} rm_2(0) &= 0 \\ rm_2(x+1) &= \overline{sg}(rm_2(x)) \end{cases}$$

Finally the function *half* : $\mathbb{N} \rightarrow \mathbb{N}$ can be defined as:

$$\begin{cases} half(0) &= 0 \\ half(x+1) &= half(x) + rm_2(x) \end{cases}$$

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

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

1. **0** : $\mathbb{N} \rightarrow \mathbb{N}$ defined by $0(x) = 0$ for each $x \in \mathbb{N}$;
2. **s** : $\mathbb{N} \rightarrow \mathbb{N}$ defined by $s(x) = x + 1$ for each $x \in \mathbb{N}$;
3. **U_j^k** : $\mathbb{N}^k \rightarrow \mathbb{N}$ defined by $U_j^k(x_1, \dots, x_k) = x_j$ for each $(x_1, \dots, x_k) \in \mathbb{N}^k$.

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

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

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

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

To define the function recursively:

$$\begin{cases} p_1(0) = 1 \\ p_1(y+2) = (y+2-1) = (y+1+(1-1)) = |y| = y \end{cases}$$

Therefore p_1 can be defined as:

$$\begin{cases} p_2(0) = 2 \\ p_2(y+1) = (y+1-2) = |y-1| = p_1(y) \end{cases}$$

Given all basic operations are defined, this is in PR .

Exercise 2

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

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

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

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

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

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

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

To show it's primitive recursive

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

Which can also be defined as:

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

Exercise

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

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

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

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

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

Example of use of bounded minimalisation to write functions primitive-recursive:

EXERCISE : show that $\text{gcd} : \mathbb{N}^2 \rightarrow \mathbb{N}$

$\text{gcd}(x, y) = \text{greatest common divisor of } x \text{ and } y$

is computable (primitive recursive)

$$\text{gcd}(x, y) = \max_{\substack{\exists z \\ \text{divisor of } x \text{ and } z \text{ divisor of } y}} z$$

$$\text{erm}(x, z) = 0 \quad \text{erm}(z, y) = 0$$

$$= \max z \leq \min(x, y) \cdot (\text{erm}(z, x) + \text{erm}(z, y) = 0)$$



$$z = \min(x, y) - \omega$$

↑ ↑
largest least

$$= \min(x, y) - \left(\mu \omega \leq \min(x, y) . (z = \min(x, y) - \omega \wedge \text{erm}(z, x) + \text{erm}(z, y) = 0) \right)$$

Exercise (2019-02-08)

Give the definition of \mathcal{PR} set of primitive recursive function and, using only the definition, show that for every $k \geq 2$ is primitive recursive the function $\text{sum}_k : \mathbb{N}^k \rightarrow \mathbb{N}$ defined by $\text{sum}_k(x_1 \dots x_k) = \sum_{i=1}^k x_i$

Solution

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

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

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

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

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

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

We then define the function by cases as follows:

$$\begin{cases} \text{sum}(x, 0) = x_1 + x_2 = 0 + 0 = 0 \\ \text{sum}(x, k + 1) = \text{sum}(x) + x_{k+2} \end{cases}$$

Exercise (2015-04-20)

Give the definition of \mathcal{PR} class of primitive recursive functions and show that the function $cpr: \mathbb{N}^2 \rightarrow \mathbb{N}$ is primitive recursive defined as:

$$cpr(x, y) = |\{p \mid x \leq p < y \wedge p \text{ prime}\}|$$

so, $cpr(x, y)$ is the number of primes in the interval $[x, y]$ (it can be assumed that sum and difference between natural numbers as well as the characteristic function of the set of prime numbers χ_{Pr} are recursive primitives, without proving it.) [Hint: It may be convenient to initially consider the function $cpr'(x, k) = |\{p \mid x \leq p < x + k \wedge p \text{ prime}\}|$]

Solution

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

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

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

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

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

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

We define the function for primitive recursion as:

$$\begin{cases} cpr(x, 0) = 0 \\ cpr(x, k + 1) = cpr(x, k) + \chi_{Pr}(x + k) \end{cases}$$

observing $cpr(x, y) = cpr'(x, y - x)$, composition of primitive recursive functions, so it is primitive recursive.

Exercise (2022-06-17-solved)

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

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

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

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

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

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

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

We can then define the function f as follows by primitive recursion. The base case is covered, 0 is an even number, no doubt. For the recursive case, consider it can either be 1 or 0, according to the specific underlying number. In this case, one could simply use the negated sign function:

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

The negated sign can be defined itself by primitive recursion as follows:

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

Exercise (2023-02-01)

Give the definition of the class \mathcal{PR} of primitive recursive functions. Show that the following functions are in \mathcal{PR}

1. $isqrt : \mathbb{N} \rightarrow \mathbb{N}$ such that $isqrt(x) = \lfloor \sqrt{x} \rfloor$;
2. $lp : \mathbb{N} \rightarrow \mathbb{N}$ such that $lp(x)$ is the largest prime divisor of x (Conventionally, $lp(0) = lp(1) = 1$.)

You can assume primitive recursiveness of the basic arithmetic functions seen in the course.

Solution

1. The basic observation is that $isqrt(x)$ is largest y such that $y^2 \leq x$ and in turn this is the smallest y such that $y^2 > x$. In addition, it is immediate to realise that such a y is bounded by x , hence we get

$$isqrt(x) = \mu y < x + 1. ((y + 1)^2 > x) = \mu y < x + 1. \overline{sg}(((y + 1)^2 - x))$$

2. Observe that, for $x > 1$, $lp(x)$ is surely smaller or equal to p_x . Hence one can count the prime divisors of x , restricting the search to p_1, \dots, p_x :

$$count(x) = \sum_{i=1}^x div(p_i, x)$$

and then $lp(x) = p_{count(x)}$. The function needs to be adjusted for $x = 0$ and $x = 1$, where $count(x) = 0$ and thus $p_{count(x)} = 0$ while $lp(x) = 1$. This is easily done as follows:

$$lp(x) = p_{count(x)} + \overline{sg}(x - 1).$$

Since we use only known primitive recursive functions, bounded sum and composition we conclude that lp is primitive recursive.

Alternatively, the idea can be to check explicitly the prime divisors of x , starting from p_x , then p_{x-1} and so on, stopping at the first. In detail, look for the smaller y , call it $i(x)$, such that p_{x-y} is a divisor of x .

$$i(x) = \mu y \leq x. \overline{sg}(div(p_{x-y}, x))$$

Then, whenever $x > 1$, $lp(x) = p_{x-i(x)}$ and the cases $x \leq 1$ must be treated separately as before:

$$lp(x) = p_{x-i(x)} \cdot sg(x - 1) + \overline{sg}(x - 1).$$

22.3 SMN-THEOREM

Note: this section requires knowing the definition then the exercises consider that you get exactly what domain and codomain are. Basically, domain is the “if” on which the input is conditioned, and the codomain is the output you get. E.g. $f(x) = y \text{ if } y < 2x, 0 \text{ otherwise}$. $2x$ is the domain and y the codomain. Then, apply the theorem to prove exactly you fixed x in order to get to that.

Exercise 3.1(p). State the smn theorem and prove it (it is sufficient to provide the informal argument using encode/decode functions).

The basic idea of the smn-theorem is creating a computable function $s_{m,n}$ such that for any computable function ϕ_e of arity m , there is a computable function $\phi_{s_{m,n}}(e)$ that behaves similarly to ϕ_e for the first m arguments.

Recalling the whole proof given [here](#):

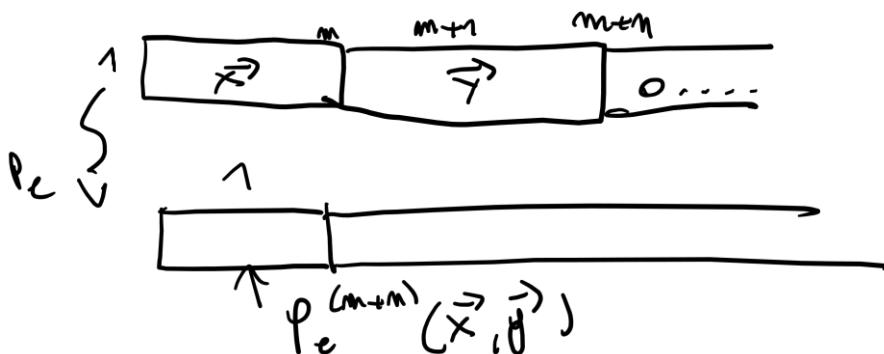
Given $m, n \geq 1$ there is a total computable function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ such that $\forall \vec{x} \in \mathbb{N}^m, \forall \vec{y} \in \mathbb{N}^n, \forall e \in \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

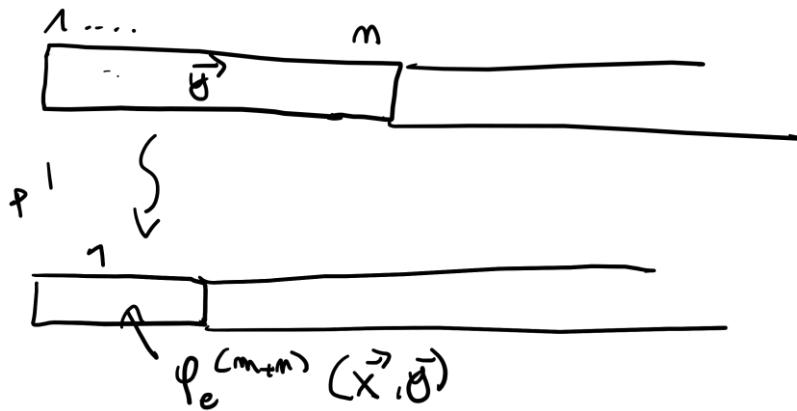
Proof

Intuitively, given $e \in \mathbb{N}, \vec{x} \in \mathbb{N}$

- We get the program $P_e = \gamma^{-1}(e)$ in standard form that computes $\phi_e^{(m+n)}$, so starting from the first drawing (this below), in which we compute the $\phi_e^{(m+n)}$ over all inputs (\vec{x}, \vec{y}) (so $\phi_e^{(m+n)}(\vec{x}, \vec{y})$)



You want, for each $\vec{x} \in \mathbb{N}^m$ fixed, a program $P' \Leftarrow$ depending on e, \vec{x} (mapping back its inputs effectively and composing function parametrizing its values, this you can see below).



P' has to

- Move \vec{y} to $m+1, \dots, m+n$ (so, move forward computation of m registers)
- Write \vec{x} in $1 \dots m$ (loading the value in the free m registers)
- Execute P_e (so, execute the computation)

The program P' can be;

$T(m, m+n) \quad // move y_n to R_{m+n}$

....

$T(1, m+1) \quad // move y_1 to R_{m+1}$

$Z(1) \quad // write x_1 to R_1$

$S(1)$

...

$S(1)$

$Z(m)$
 $S(m)$
...
 $S(m)$



$// write x_m to R_m$

Concatenation will update all the jump instructions, hence moving and writing values *for all function parametrized inside*, mapping back effectively with $P_e = \gamma^{-1}(e)$.

Once the program P has been built, we have $S(e, \vec{x}) = \gamma(P')$. Given each function is effective, existence, totality and computability of s are informally proven.

In the context of the smn-theorem, ϕ_e^k is the e^{th} partial computable function of k variables. The theorem establishes that there exists a total computable function, denoted as $s_{(m,n)}$, which can effectively "translate" or encode the computation of $\phi_e^{(m+n)}(x, y)$ into the computation of $\phi_s^n(s_{(m,n)}(e, x), n(y))$.

Exercise 3.2(p). State the theorem s-m-n and use it to prove that it exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $|W_{s(x)}| = 2x$ and $|E_{s(x)}| = x$.

This one is also present inside 2019-09-17 exam.

Given $m, n \geq 1$ there is a total computable function $s_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ such that $\forall \vec{x} \in \mathbb{N}^m, \forall \vec{y} \in \mathbb{N}^n, \forall e \in \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^n(\vec{y})$$

Given the domain should be $2x$, we find a function in which we can parametrize a value $< 2x$; given the range is x , it's simply a function which allows us to be defined computably over x . Let's give

$$g(x, y) = \begin{cases} qt(x, y), & y < 2x \\ \uparrow, & \text{otherwise} \end{cases}$$

$g(x, y)$ is computable and $sg(y) * qt(x, y) + \mu z. (y + 1 - 2x)$ is computable itself, hence giving as range x .

By the smn-theorem, there is a computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y) \forall x, y \in \mathbb{N}$. Therefore, for each function:

- $W_x = \{y \mid (g(x, y) \downarrow\} = \{y \mid y < 2x\}$
- $E_{k(n)} = \{g(x, y) \mid x \in W_{s(x)}\} = \{qt(2, y) \mid y < 2x\} = \{y + 1 - 2x \mid y + 1 < 2x\} = [0, 2x)$

as desired.

Exercise 3.3. State the smn theorem and use it to prove that there exists a total computable function $s: \mathbb{N}^2 \rightarrow \mathbb{N}$ such that $W_{s(x, y)} = \{z: x * z = y\}$

Given $m, n \geq 1$ there is a total computable function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ such that $\forall \vec{x} \in \mathbb{N}^m, \forall \vec{y} \in \mathbb{N}^n, \forall e \in \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}(\vec{y})$$

We start by defining a computable function of two arguments $f(n, x)$ which meets the conditions when viewed as a function of x , with n taken as a parameter, e.g.

$$f(n, x) = \begin{cases} \frac{x}{z}, & \text{if } y \text{ multiple of } x \\ \uparrow, & \text{otherwise} \end{cases} = qt(x, z) + \mu z. rm(x, z)$$

By the smn-theorem, there is a computable total function $k: \mathbb{N} \rightarrow \mathbb{N}$ such that $\phi_{K(n)}(x) = f(n, x) \forall n, x \in \mathbb{N}$. Therefore:

- $W_{s(k,y)} = \{x \mid f(n, x) \downarrow\} = \left\{y: \frac{x}{z}\right\} = \{z: x * z = y\}$

as desired.

Exercise 3.4(p). Prove that there is a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ such that for each $n \in \mathbb{N}$ it holds that $W_{k(n)} = \mathbb{P} = \{x \in \mathbb{N} \mid x \text{ even}\}$ and $E_{k(n)} = \{x \in \mathbb{N} \mid x \geq n\}$.

We start by a computable function $f(n, x)$ which meets the conditions over the parameter, which uses two functions in order to accomplish the thing. Such function can be, considering the even (division of 2 different from zero, which will happen if $x \geq z$, we can characterize):

$$f(n, x) = \begin{cases} \frac{x}{2} + n, & x \text{ is even} \\ \uparrow, & \text{otherwise} \end{cases} = qt(2, x) + n + \mu z. rm(2, z)$$

By the smn-theorem, there is a computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{g(n)}(x) = f(n, x) \forall n, x \in \mathbb{N}$.

Therefore, for each function:

- $W_{k(n)} = \{x \mid (f(n, x) \downarrow\} = \{x \mid x \text{ even}\} = \{x \mid x \text{ is even}\}$
- $E_{k(n)} = \{f(n, x) \mid x \in \mathbb{N}\} = \{\frac{x}{2} + n \mid x \text{ even}\} = \{x + n \geq 0\} = \{y \mid y \geq \mathbb{N}\}$

as desired.

Exercise 3.5. State the smn theorem. Use it to prove it exists a total computable function $k : \mathbb{N} \rightarrow \mathbb{N}$ such that $W_{k(n)} = \{x \in \mathbb{N} \mid x \geq n\}$ e $E_{k(n)} = \{y \in \mathbb{N} \mid y \text{ even}\}$ for all $n \in \mathbb{N}$.

Given $m, n \geq 1$ there is a total computable function $s_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ such that $\forall \vec{x} \in \mathbb{N}^m, \forall \vec{y} \in \mathbb{N}^n, \forall e \in \mathbb{N}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

Solution: We start by defining a computable function of two arguments $f(n, x)$ which enjoys the property when viewed as a function of x , with n seen as a parameter, e.g.

$$f(n, x) = \begin{cases} 2 * (x - n) & \text{if } x \geq n \\ \uparrow & \text{otherwise} \end{cases} = 2 * (x - n) + \mu z. (n - x)$$

By the smn theorem, there is a computable total function $k : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi_{k(n)}(x) = f(n, x)$ for each $n, x \in \mathbb{N}$. Therefore, as desired

- $W_{k(n)} = \{x \mid f(n, x) \downarrow\} = \{x \mid x \geq n\};$
- $E_{k(n)} = \{f(n, x) \mid x \in \mathbb{N}\} = \{2(x - n) \mid x \geq n\} = \{2(n + z - n) \mid z \geq 0\} = \{2z \mid z \in \mathbb{N}\}.$

Exercise (2023-02-20)

State the smn-theorem. Show that there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that for all $x \in \mathbb{N}, x > 0$ we have $W_{s(x)} = \mathbb{P}$ and $|E_{s(x)}| = 2x$.

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

We define a computable function g able to compute $2x$ effectively:

$$g(x, y) = \begin{cases} qt(y, 2x) + 2, & x \in \mathbb{P}, x > 0, y \leq 2x \\ \uparrow, & \text{otherwise} \end{cases}$$

The function is computable, given:

$$g(x, y) = qt(2x, y) + 2 + \mu z. (qt(2x, y) - 2x)$$

By the smn-theorem, there exists a function $h : \mathbb{N} \rightarrow \mathbb{N}$ such that $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$

Hence, we can conclude:

$$W_{s(x)} = \{y \mid g(x, y) \downarrow\} = \{y \mid y \leq 2x\} = \{x \in \mathbb{P} \mid y \leq 2x \in \mathbb{P}\} = \mathbb{P}$$

$$|E_{s(x)}| = \{|g(x, y) \mid x \in W_{s(x)}\} = \{|y \mid y \leq 2x\}| = \{|qt(2x, y) - 2x \mid y \leq 2x\} = \{|y \mid y = 2x\}$$

as desired.

Exercise

State the smn-theorem and use it to show there exists a total computable function $s: \mathbb{N}^2 \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}$ it holds $|W_{s(x,y)}| = x * y$

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

We define a function $g: \mathbb{N} \rightarrow \mathbb{N}$ able to get as range $x * y$, so able to minimize it.

$$g(x, y, z) = \begin{cases} 0, & z < x * y \\ \uparrow, & \text{otherwise} \end{cases}$$

The function g is computable and given the minimum w :

$$g(x, y, z) = \mu w. z + 1 - (x * y)$$

and we can show by the smn-theorem, there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y, z) = \phi_{s(x,y)}(z)$. So

- $|W_{s(x)}(y)| = \{y \mid g(x, y) \downarrow\} = |z \mid z < x * y|$ and so $x * y$ as desired.

Exercise (2019-11-18-solved)

State the smn-theorem and use it to show there exists a total computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall n \in \mathbb{N}$, $\phi_{k(n)}$ is total and $E_{k(n)}$ is the set of integer divisors of n .

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

We define a function $g: \mathbb{N} \rightarrow \mathbb{N}$ in which we can define:

$$g(n, x) = \begin{cases} x * n, & x \text{ is a divisor of } n \\ 1, & \text{otherwise} \end{cases}$$

This is computable, given:

$$g(n, x) = (x * n) * \overline{sg}(rm(x, n)) + sg(rm(x, n))$$

For the smn-theorem, there exists a function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{k(n)}(x) = f(n, x) \forall n, x \in \mathbb{N}$. So, as desired:

- $W_{k(n)} = \mathbb{N}$ (total)
- $E_{k(n)} = \{x \mid rm(x, n) = 0\} \cup \{1\}$, set of divisors and 1 which is always a divisor for n

Exercise 2

State the s-m-n theorem and use it to prove that there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $|W_{s(x)} \cap E_{s(x)}| = 2x$.

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

We're basically using a function which we use as index for the domain and codomain. Define a function of two arguments like:

$$g(x, y) = \begin{cases} z, & \text{there exists } a z \text{ s.t. } y = 2x \\ \uparrow, & \text{otherwise} \end{cases} = \mu z. |(y - 2x)|$$

For the smn-theorem, there exists a function $k : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y) \forall x, y \in \mathbb{N}$. So, as desired:

- $W_{k(n)} = \{x \mid g(x, y) \downarrow\} = \{2x\}$
- $E_{k(n)} = \{g(x, y) \mid x \in \mathbb{N}\} = \{z \mid z \text{ s.t. } y = 2x\} = \{z \mid z \in \mathbb{N}\}$

Exercise (2015-04-20.partial)

State the smn-theorem and use it to show there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$, $W_{s(x)} = \{(k + 2)^2 \mid k \in \mathbb{N}\}$

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

To prove it, we define a function of two arguments such that:

$$g(x, y) = \begin{cases} k, & \text{if there exists some } k \text{ s.t. } y = (x + k)^2 \\ \uparrow, & \text{otherwise} \end{cases}$$

so we set a minimization to look for that value, like $g(x, y) = \mu k. |(x + k)^2 - y|$. Such function is total and computable, and for the smn-theorem, there exists a function $k : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y) \forall x, y \in \mathbb{N}$. So, as desired:

- $W_{s(x)} = \{x \mid g(x, y) \downarrow\} = \{\exists k \in \mathbb{N} \mid y = (x + k)^2\} = \{x \mid (x + k)^2 \in \mathbb{N}\}$

Exercise (2018-11-20-parziale)

State the smn-theorem. Use it for proving there exists $k : \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall n \in \mathbb{N}$ we have $W_{k(n)} = \{z^n \mid z \in \mathbb{N}\}$ and $E_{k(n)}$ is the set of odd numbers

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n} : \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

Define a two-arguments total-computable function $f(n, x)$ respecting the conditions:

$$f(n, x) = \begin{cases} 2z + 1, & \text{if } x = z^n \text{ for some } z \\ \uparrow, & \text{otherwise} \end{cases} = 2 * (\mu z. |x - z^n|) + 1$$

By the smn-theorem, there exists a total and computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{k(n)}(x) = f(n, x) \forall n, x \in \mathbb{N}$. So, as desired:

- $W_{k(n)} = \{x \mid f(n, x) \downarrow\} = \{x \mid \exists z \in \mathbb{N}. x = z^n\} = \{z^n \mid z \in \mathbb{N}\}$
- $E_{k(n)} = \{f(n, x) \mid x \in W_{k(n)}\} = \{2^{\sqrt[n]{z}} + 1 \mid z \in \mathbb{N}\} = \{2z + 1 \mid z \in \mathbb{N}\}$

Exercise (2017-11-20)

State the smn-theorem. Use it for proving there exists $k: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall n \in \mathbb{N}$ we have $W_{k(n)} = \{x \in \mathbb{N} \mid x \geq n\}$ and $E_{k(n)} = \{y \in \mathbb{N} \mid y \text{ even}\}$ for all $n \in \mathbb{N}$.

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

Define:

$$f(n, x) = \begin{cases} 2 * (x \dot{-} n) & \text{se } x \geq n \\ \uparrow & \text{altrimenti} \end{cases} = 2 * (x \dot{-} n) + \mu z. (n \dot{-} x)$$

By the smn-theorem, there exists a total and computable function $k: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{k(n)}(x) = f(n, x) \forall n, x \in \mathbb{N}$. So, as desired:

- $W_{k(n)} = \{x \mid f(n, x) \downarrow\} = \{x \mid x \geq n\};$
- $E_{k(n)} = \{f(n, x) \mid x \in \mathbb{N}\} = \{2(x \dot{-} n) \mid x \geq n\} = \{2(n + z) \dot{-} n) \mid z \geq 0\} = \{2z \mid z \in \mathbb{N}\}.$

Exercise (2020-11-23)

State the smn-theorem. Use it for proving there exists $k: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall n \in \mathbb{N}$ we have $|W_x| = 2^x$ and $|E_x| = x + 1$.

Solution

The smn-theorem states that, given $m, n \geq 1$ there is a computable total function $s_{m,n}: \mathbb{N}^{m+1} \rightarrow \mathbb{N}$ s.t. $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n$

$$\phi_e^{m+n}(\vec{x}, \vec{y}) = \phi_{s_{m,n}(e, \vec{x})}^{(n)}(\vec{y})$$

Define:

$$g(x, y) = \begin{cases} \lfloor \log_2(y + 1) \rfloor & \text{se } y < 2^x \\ \uparrow & \text{altrimenti} \end{cases}$$

which is computable.

In fact, $g(x, y)$ when defined, is the greatest z s.t. $2^z \leq y + 1$ and the minimum s.t. $2^{z+1} > y + 1$, so:

$$g(x, y) = \mu z. \overline{s g}(2^{z+1} \dot{-} (y + 1)) + \mu w. (y + 1 \dot{-} 2^x)$$

So, by the smn-theorem, there exists a function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}$ we have $g(x, y) = \phi_{s(x)}(y)$ and so s is the desired function. Infact:

- $W_x = \{y \mid g(x, y) \downarrow\} = [0, 2^x - 1]$ e quindi $|W_x| = |[0, 2^x - 1]| = 2^x$
- $E_x = \{g(x, y) \mid 0 \leq y < 2^x\} = \{\lfloor \log_2(y + 1) \rfloor \mid 0 \leq y < 2^x\} = [0, x]$ e quindi $|E_x| = |[0, x]| = x$.

22.4 DECIDABILITY AND SEMIDEcidability

Note: this section requires knowing or remembering at least structure/projection theorem and the definition of semidecidable/decidable/knowing the implications of quantification.

Exercise 4.1. Prove the “structure theorem” of semidecidable predicates, i.e., show that a predicate $P(\vec{x})$ is semidecidable if and only if there exists a decidable predicate $Q(\vec{x}, y)$ such that $P(\vec{x}) \equiv \exists y. Q(\vec{x}, y)$.

(\Rightarrow) Let $P(\vec{x}) \subseteq \mathbb{N}^k$ be semi-decidable

$$sc_P(\vec{x}) = \begin{cases} 1, & \text{if } P(\vec{x}) \\ \uparrow, & \text{otherwise} \end{cases} \text{ is computable}$$

i.e. there is $e \in \mathbb{N}$ s.t. $sc_P = \phi_e^{(k)}$

Observe $P(\vec{x})$ iff $sc_P(\vec{x}) = 1$

iff $sc_P(\vec{x}) \downarrow$

iff $P_e(\vec{x}) \downarrow$

iff $\exists t. H^{(k)}(e, \vec{x}, t)$

If we let $Q(t, \vec{x}) = H^{(k)}(e, \vec{x}, t)$ decidable and $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$

(\Leftarrow) We assume $P(\vec{x}) \equiv \exists t. Q(t, \vec{x})$ with $Q(t, \vec{x})$ decidable

$$\begin{aligned} sc_P(\vec{x}) &= \begin{cases} 1, & \text{if } P(\vec{x}) \Leftrightarrow \exists t. Q(t, \vec{x}) \Leftrightarrow \exists t. X_Q(t, \vec{x}) = 1 \\ \uparrow, & \text{otherwise} \end{cases} \\ &= \mathbf{1}(\mu t. |X_Q(t, \vec{x}) - 1|) \\ &\quad \text{As } \Delta \vdash Q(t, \vec{x}) \text{ if it exists} \\ &\quad \uparrow \text{obviously} \end{aligned}$$

Exercise 4.2. Prove the “projection theorem”, i.e., show that if the predicate $P(x, \vec{y})$ is semidecidable then also $\exists x. P(x, \vec{y})$ is semi-decidable. Does the converse implication hold? Is it the case that if $P(x, \vec{y})$ is decidable then also $\exists x. P(x, \vec{y})$ is decidable? Give a proof or a counterexample.

Proof (Exercise present inside 2017-01-24 exam)

Let $P(x, \vec{y}) \subseteq \mathbb{N}^{k+1}$ semi-decidable. Hence, by structure theorem, there is $Q(t, x, \vec{y}) \subseteq \mathbb{N}^{k+2}$ decidable s.t.

$$P(x, \vec{y}) \equiv \exists t. Q(t, x, \vec{y})$$

Now

$$\begin{aligned} R(\vec{y}) &\equiv \exists x. P(x, \vec{y}) \equiv \exists x. \exists t. Q(t, x, \vec{y}) \\ &\equiv \exists w. Q(\underbrace{(w)_1, (w)_2, \vec{y}}_{\text{decidable}}) \end{aligned}$$

Hence R is the existential quantification of a decidable predicate \Rightarrow by structure theorem, it is semi-decidable.

Solution: No, the converse is false. Consider, for instance, $P(x, y) = (y = 2x) \wedge (y \notin W_x)$ (or, simply, $P(x, y) = x \notin W_x$), which is not semi-decidable. The existentially quantified version is constant, hence decidable.

Also the second claim is false. Take for instance $P(x, y) = H(y, y, x)$ which is decidable, while $\exists x. P(x, y) \equiv y \in K$ is only semi-decidable, but not decidable. \square

Exercise (2015-07-16-solved)

Show that a predicate $P(x, \vec{y})$ is semidecidable, then $\exists x. P(x, \vec{y})$ is semidecidable. Does the converse hold? Show it or write a counterexample.

Solution

The first one refers to the projection theorem, defined also [here](#). Observe instead that the converse implication is false. Consider, for example, the predicate $P(x, y) = x \in W_x$, which is not semi-decidable.

The predicate obtained through existential quantification $Q(y) = \exists x. P(x, y)$ is consistently true or false (although not relevant to the proof, note that since \bar{K} is nonempty, the predicate $Q(y)$ is consistently true), thus decidable.

As a less "degenerate" example, one may consider $P(x, y) = (y > x) \wedge (y \notin W_x)$ and the quantification $Q(y) = \exists x. (y > x) \wedge (y \notin W_x)$. In this case, note that with $e_0 \in N$, an index for the always indefinite function, we have $Q(y)$ is true for every $y > e_0$, thanks to which $Q(y)$ is decidable.

Exercise (2022-06-17)

- c. Show that if predicate $Q(\vec{x}, y) \subseteq \mathbb{N}^{k+1}$ is semi-decidable then also $P(\vec{x}) = \exists y. Q(\vec{x}, y)$ is semi-decidable (do not assume structure and projection theorems). Does the converse hold, i.e., is it the case that if $P(\vec{x}) = \exists y. Q(\vec{x}, y)$ is semi-decidable then $Q(\vec{x}, y)$ is semi-decidable? Provide a proof or a counterexample.

Solution

3. Let $Q(\vec{x}, y) \subseteq \mathbb{N}^{k+1}$ be semi-decidable. Then the semi-characteristic function $sc_Q : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ is computable. Let $e \in \mathbb{N}$ be such that $sc_Q = \varphi_e^{(k+1)}$.

Then $Q(\vec{x}, y)$ holds iff $\varphi_e^{(k+1)}(\vec{x}, y) = 1$ iff $\varphi_e^{(k+1)}(\vec{x}, y) \downarrow$ iff $\exists t. H^{(k+1)}(e, (\vec{x}, y), t)$.

Therefore $Q(\vec{x}, y) \equiv \exists t. H^{(k+1)}(e, (\vec{x}, y), t)$ and thus

$$P(\vec{x}) \equiv \exists y. Q(\vec{x}, y) \equiv \exists y. \exists t. H^{(k+1)}(e, (\vec{x}, y), t) \equiv \exists w. H^{(k+1)}(e, (\vec{x}, (w)_1), (w)_2)$$

Therefore $sc_P(\vec{x}) = 1(\mu w. |\chi_{H^{(k+1)}}(e, (\vec{x}, (w)_1)), (w)_2) - 1|)$ is computable, and thus $P(\vec{x})$ is semi-decidable.

The converse implication does not hold. For instance, consider the predicate $Q(x, y) \equiv \text{"}\phi_y(x) \uparrow\text{"}$. Then $P(x) \equiv \exists y. Q(x, y) \equiv \exists y. \phi_y(x) \uparrow$ is always true, hence decidable. In fact, if e_0 is an index for the always undefined function, for $y = e_0$ clearly $Q(x, y)$ for every $x \in \mathbb{N}$. Instead $Q(x, y) = \phi_y(x) \uparrow$ is not semi-decidable (it is negation of the halting predicate, which is semi-decidable but not decidable).

Exercise (30-06-2020)

Given two functions $f, g: \mathbb{N} \rightarrow \mathbb{N}$ with f total, define predicate $Q_{f,g}(x) = "f(x) = g(x)"$. Show that if f and g are computable, then $Q_{f,g}$ is semidecidable. Does the converse hold, so if $Q_{f,g}$ is semidecidable, can we deduce f and g are computable?

Solution

Let f, g be computable functions. Let $e_1, e_2 \in \mathbb{N}$ s.t. $f = \phi_{e_1}$ and $g = \phi_{e_2}$.

Then $sc_{f,g} = \mathbf{1}(\mu w. |f(x) - g(x)|)$ is computable, hence $Q_{f,g}$ is semidecidable.

If $Q_{f,g}$ is semidecidable and let e be an index of semicharacteristic function of Q , namely $\phi_e = sc_{Q_{f,g}}$

We have $f(x) = (\mu w. H(e, x, (w)_1, (w)_2) \vee H(e, y, (w)_1, (w)_3))$ which shows f and g should be computable by composition; given the halting problem is not decidable, it means we could easily use by composition the semi-characteristic function of K (sc_K) and make it work – so the converse does not hold.

Exercise (2023-02-01)

- c. Show that if predicate $Q(\vec{x}, y) \subseteq \mathbb{N}^{k+1}$ is semi-decidable then also $P(\vec{x}) = \exists y. Q(\vec{x}, y)$ is semi-decidable (do not assume structure and projection theorems). Does the converse hold, i.e., is it the case that if $P(\vec{x}) = \exists y. Q(\vec{x}, y)$ is semi-decidable then $Q(\vec{x}, y)$ is semi-decidable? Provide a proof or a counterexample.

Solution

Let $Q(\vec{x}, y) \subseteq \mathbb{N}^{k+1}$ be semi-decidable. Then the semi-characteristic function $sc_Q : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ is computable. Let $e \in \mathbb{N}$ be such that $sc_Q = \varphi_e^{(k+1)}$.

Then $Q(\vec{x}, y)$ holds iff $\varphi_e^{(k+1)}(\vec{x}, y) = 1$ iff $\varphi_e^{(k+1)}(\vec{x}, y) \downarrow$ iff $\exists t. H^{(k+1)}(e, (\vec{x}, y), t)$.

Therefore $Q(\vec{x}, y) \equiv \exists t. H^{(k+1)}(e, (\vec{x}, y), t)$ and thus

$$P(\vec{x}) \equiv \exists y. Q(\vec{x}, y) \equiv \exists y. \exists t. H^{(k+1)}(e, (\vec{x}, y), t) \equiv \exists w. H^{(k+1)}(e, (\vec{x}, (w)_1), (w)_2)$$

Therefore $sc_P(\vec{x}) = \mathbf{1}(\mu w. |\chi_{H^{(k+1)}}(e, (\vec{x}, (w)_1)), (w)_2) - 1|)$ is computable, and thus $P(\vec{x})$ is semi-decidable.

The converse implication does not hold. For instance, consider the predicate $Q(x, y) \equiv "\phi_y(x) \uparrow"$. Then $P(x) \equiv \exists y. Q(x, y) \equiv \exists y. \phi_y(x) \uparrow$ is always true, hence decidable. In fact, if e_0 is an index for the always undefined function, for $y = e_0$ clearly $Q(x, y)$ for every $x \in \mathbb{N}$. Instead $Q(x, y) = \phi_y(x) \uparrow$ is not semi-decidable (it is negation of the halting predicate, which is semi-decidable but not decidable).

22.5 NUMERABILITY AND DIAGONALIZATION

Note: I understood overtime this category refers to “Is this set countable?” – Basically, a category we never once considered. Here for legacy reasons. See for yourself and you will prove me right.

Exercise 5.1(p). Consider the set F_0 of functions $f : \mathbb{N} \rightarrow \mathbb{N}$, possibly partial, such that $\text{cod}(f) \subseteq \{0\}$. Is the set F_0 countable? Justify your answer.

Note *cod* stays for codomain there. The intuitive answer would be no, we can't possibly have a mapping one-to-one to each element, given the set over \mathbb{N} itself is uncountable.

Let's suppose the set is countable for the sake of contradiction and make an argument about it.

- Suppose the set of all functions from $\mathbb{N} \rightarrow \mathbb{N}$ is countable. There sure would exist a bijection f between the set \mathbb{N} and all functions from $\mathbb{N} \rightarrow \mathbb{N}$
- We can construct a function $g(n)$ which maps over the image of F_0 , a subset only mapping over the number of 0. We define such function to be the output over the n^{th} computation of the image, which is over the set of natural numbers

Even restricting the values to this section, we can't possibly count them all and the set F_0 itself is uncountable.

Exercise 5.2(p). A function $f : \mathbb{N} \rightarrow \mathbb{N}$ 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.

Suppose by contradiction the set of total increasing functions is not countable. We first would have to define something total, and this can be an enumeration of such programs, such that:

$$f(y) = 1 + \sum_{n=0}^x f_n(x)$$

This here is an enumeration which is total by definition, given they are already inside the natural numbers. This also increases respecting the said order, given $f(y+1) = f(x) + f_{x+1}(x+1) + 1 \geq f(y) \geq f(x) \geq 0$ (it increases naturally. It also differs $f(y) = 1 + \sum_{n=0}^x f_n(x) \geq 1 + f_x(x+1) > f_x(x)$.

No enumeration can be defined countable over the natural numbers, given we can't possibly count and enumerate how such recursion will work in the long run, and so it can't possibly contain all total increasing functions.

Exercise 5.3(p). A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is called *total increasing* when it is total and for each $x, y \in \mathbb{N}$, if $x \leq y$ then $f(x) \leq f(y)$. It is called binary if $\text{cod}(f) \subseteq \{0, 1\}$. Is the set of binary total increasing functions countable? Justify your answer.

The key here would be defining a function which is bounded, hence showing the enumeration works for all codomain values, or something like that, intuitively. Since \mathbb{N} is countably infinite, the possible combination of choices for \mathbb{N}_0 (in 0,1), can be $2^{\mathbb{N}^0}$ which itself is uncountably infinite. We define as S the set of increasing total functions.

Let's define instead a total increasing binary function, in which we can assert the \leq property, which will be defined over all values (mapping each binary sequence to a unique natural number). We define a function in which if $f(x) = 1$, it means that $f(1) = f(2)$, hence obtaining 0 when this ordering property is defined, 1 otherwise (when they are equal, it would be). So, we define:

$$f_i(x) = \begin{cases} 0, & f_i(x_i) \leq f_{i+1}(x_i), \text{ when } x < i \\ 1, & \text{otherwise} \end{cases}$$

This represents the enumeration of total binary increasing functions, which represents a countable subset. Given the injection maps a binary sequence effectively respecting the property, it remains countable.

Exercise (2019-01-24)

Show that set $F = \{\theta \mid \theta: \mathbb{N} \rightarrow \mathbb{N} \wedge \text{dom}(\theta) \text{ finite}\}$ of unary functions with finite domain is countable.

Solution

Given any enumeration of the finite subfunctions inside of F , let's define a function $g(x) = \prod_{i=1}^n \theta_i$

Such a function is:

- total by definition
- computable, given it is the composition of computable functions

Define $\theta_i(x) = \begin{cases} \theta_i(x) + 1, & x \in W_x \\ 0, & \text{otherwise} \end{cases}$

Since $\theta_i(x)$ has a finite domain, g is finite and a unary function. g itself is in the enumeration, considering $g(x) = \theta(x) + 1$, hence F is countable.

An alternative solution (Italian one):

dobbiamo mostrare che esiste una funzione totale suriettiva da $F \rightarrow \mathbb{N}$
 ovvero $f: F \rightarrow \mathbb{N}$.
 ogni funzione $g \in F$ è del tipo $g = \{(x_i, y_i) \mid 0 \leq i \leq n, n \in \mathbb{N}\}$.
 quindi possiamo definire f come:

$$f(g) = \prod_{i=0}^n p_i^{y_i}$$

 f è definito per composizione a partire da funzioni calcolabili e
 quindi è calcolabile e per definizione totale.

Exercise (2019-02-08)

Given a function $f: \mathbb{N} \rightarrow \mathbb{N}$ define $Z(f) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \forall x \in \mathbb{N}. g(x) = f(x) \vee g(x) = 0\}$. Show that set $Z(id)$ is not countable. It is true that for all f , we have $Z(f)$ is not countable?

Solution

Define

$$f(x) = \begin{cases} x, & \text{if } \phi_x(x) \downarrow \\ 0, & \text{otherwise} \end{cases}$$

The function is:

- total by definition
- $\forall x \in \mathbb{N}, \phi_x(x) \neq f$, since $\phi_x(x+1) \neq f(x+1) \neq f(x) \neq g(x)$

Hence, this is not countable.

22.6 FUNCTIONS AND COMPUTABILITY

Note: this requires proving functions are not-computable most of the time. Use diagonalization or some functions which is written as computable but actually uses χ_K which is not computable by definition.

Exercise 6.1(p). Define a function $f : \mathbb{N} \rightarrow \mathbb{N}$ total and not computable such that $f(x) = x$ for infinite arguments $x \in \mathbb{N}$ or prove that such a function cannot exist.

In essence, there should be a case in which we compute until the x^{th} computation and then will not be defined, something like the halting problem. Given the function is not computable, we have a function not defined for all inputs, hence not decidable for all input where $f(x) = x$.

Let's define a function $g : \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$$g(x) = \begin{cases} \varphi_x(x) + 1, & x \in W_x \text{ and } f(x) \text{ total} \\ x, & \text{otherwise} \end{cases}$$

This will ensure when $\forall x \in \mathbb{N}, g(n+1) = f(n) + 1 = n + 1 \neq f(n)$ since this is not computable and it will hold as x infinite many times, giving each time a different output over the single input. Hence, everything is satisfied.

Official one:

Solution: We can define

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in W_x \\ x & \text{if } x \notin W_x \end{cases}$$

Clearly, for all $x \in \mathbb{N}$ we have $\varphi_x(x) \neq f(x)$, hence f is not computable. Moreover $x \notin W_x$ holds true infinitely many times since the empty function has infinitely many indices. Therefore also the last condition is satisfied. \square

Exercise 6.2(p). Say that a *function* $: \mathbb{N} \rightarrow \mathbb{N}$ is *increasing* if it is total and for each $x, y \in \mathbb{N}$, if $x \leq y$ then $f(x) \leq f(y)$. Is there an increasing function which is not computable? Justify your answer.

Use a basic diagonalization argument; consider an increasing function like $f(x) = \sum_{y \leq x} g(y)$ and then put:

$$g(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in W_x \\ 0 & \text{otherwise} \end{cases}$$

Exercise 6.3(p). Are there two functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ with g not computable such that the composition $f \circ g$ (defined by $(f \circ g)(x) = f(g(x))$) is computable? And requiring that f is also not computable, can the composition $f \circ g$ be computable? Justify your answer, giving examples or proving non-existence.

Solution: Yes, in both cases. In fact, let $g = \chi_k$, not computable, and f defined by

$$f(x) = \begin{cases} 0 & \text{if } x \leq 1 \\ \chi_k(x) & \text{otherwise} \end{cases}$$

not computable too, otherwise χ_K would be computable. It is easy to see that $f \circ g$ is the constant 0, which is computable. \square

Exercise 6.4(p). Is there a function $f : \mathbb{N} \rightarrow \mathbb{N}$ with finite range, total and increasing (i.e. $f(x) \leq f(y)$ for $x \leq y$) and not computable? Justify your answer with an example or a proof of non-existence. What if we relax the requirement of totality?

Solution: With the totality requirement, function f cannot exist. Indeed, we can prove that each function $f : \mathbb{N} \rightarrow \mathbb{N}$ with all the required properties is computable. The proof proceeds for induction on $M = \max\{f(x) \mid x \in \mathbb{N}\}$.

($M = 0$) Observe that in this case $f(x) = 0$ for all $x \in \mathbb{N}$, i.e. f is the constant 0 and therefore it is computable.

($M > 0$) In this case, let $x_0 = \min\{x \mid f(x) = M\}$. If $x_0 = 0$, the function f is the constant M , and therefore it is computable.

If, on the other hand, $x_0 > 0$, let $M' = f(x_0 - 1)$, i.e., the value assumed by f before M . We can then write $f(x)$ as the sum of two functions

$$f(x) = f'(x) + g(x)$$

where $f' : \mathbb{N} \rightarrow \mathbb{N}$ is:

$$f'(x) = \begin{cases} f(x) & \text{if } x < x_0 \\ M' & \text{otherwise} \end{cases}$$

and $g : \mathbb{N} \rightarrow \mathbb{N}$ is:

$$g(x) = \begin{cases} 0 & \text{if } x < x_0 \\ M - M' & \text{otherwise} \end{cases} = (M - M') \operatorname{sg}(x + 1 - x_0)$$

The function f' is total, with range included in that of f , whence finite; moreover it is increasing and $\max\{f'(x) \mid x \in \mathbb{N}\} = M' < M$. Hence it is computable by inductive hypothesis. Also g is computable as it can be expressed as a composition of computable functions. Thus f is also computable.

If we relax totality, the good idea is to use diagonalization, defining:

$$f(x) = \begin{cases} \varphi_x(x) + 1, & x \neq W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

Given $\varphi_x \neq f(x)$ is not computable, by diagonalization, this will be different from all computable functions.

In any case, one could adapt the following proof reversing the signs for this one.

Exercise 6.5(p). Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is *decreasing* if it is total and for each $x, y \in \mathbb{N}$, if $x \leq y$ then $f(x) \geq f(y)$. Is there a decreasing function which is not computable? Justify your answer.

Let's start defining a decreasing function, in which we take $k = \min\{f(x) \mid x \in \mathbb{N}\}$ and let $x_0 \in \mathbb{N}$ s.t. $f(x_0) = k$. Therefore, given f is decreasing, $f(x) = k, \forall x \geq x_0$. Define then

$$\theta(x) = \begin{cases} f(x), & \text{if } x < x_0 \\ \uparrow, & \text{otherwise} \end{cases}$$

then write f as

$$f(x) = \begin{cases} \theta(x), & \text{if } x < x_0 \\ k, & \text{otherwise} \end{cases}$$

Given θ is finite, it is computable and if we use the smn-theorem, we can write $f(x)$ as follows (combining both possibilities of the function for whichever case it is decreasing):

$$f(x) = (\mu w. ((x < x_0 \wedge S(e, x, (w)_1, (w)_2) \vee (x \geq x_0 \wedge (w)_1 = k)))_1$$

The alternative proof uses induction on $x_0 = \min\{x \mid f(x) = M\}$, then defining:

- $M = 0$, the decreasing function $f(x) = 0$ is primitive recursive for all x
- $M > 0$, given x_0
 - o if $x_0 = 0$, then f is a constant function which is primitive recursive
 - o if $x_0 > 0$, decompose f into:

$$f_1(x) = \begin{cases} M, & x < x_0 \\ f(x), & x \geq x_0 \end{cases}$$

The function f_1 is total and inductive hypothesis and primitive recursion, we can define:

$$g(x) = \begin{cases} 0, & x < x_0 \\ M - f(x), & x \geq x_0 \end{cases}$$

To avoid the function to assume the same value as before, we define $f(x) = f_1(x) + g(x)$. Given this is defined by primitive recursion and it is total, it's computable.

Exercise 6.6(p). Say if there can be a non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for any other non-computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ 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).

We are essentially saying that for every single non-computable function the quantification $f + g$ is not computable and it can't be. However, we know that the sum of two non-computable functions is not necessarily computable. This is because the composition and combination of non-computable functions can still result in a non-computable function.

If we have a case in which $g(x) = f(x)$, the sum itself would be essentially $f(x) + g(x) = f(x) + f(x) = 2 * f(x)$. However, this does not imply that $f(x)$ is computable; rather, it suggests that the sum in this specific case is proportional to $f(x)$. The key point remains that the sum of two non-computable functions is not guaranteed to be computable, and the example $f(x) + g(x)$ being proportional to $f(x)$ doesn't change this general conclusion.

Exercise 6.7. Say if there can be a non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that there exists a non-computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ for which 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).

We know the sum of non-computable functions is not expected to be computable if the two underlying functions are themselves not computable. We can have the mixed chance: one is computable, the other is not and again the sum would not be computable. There can exist a case in which we consider a codomain $K = \{0,1\}$ in which we sum the two functions $\chi_K + \chi_{\bar{K}} = 1$, where this is computable.

Exercise 6.8(p). Say if there can be a non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{dom}(f) \cap \text{img}(f)$ is finite. Justify your answer (providing an example of such f , if it exists, or proving that cannot exist).

All elements of image are also the domain. Because we like to understand things, the following is an example of function with these qualities:

$$f(x) = x^2$$

$$\text{dom}(f) = \mathbb{R}$$

$$\text{img}(f) = \mathbb{R} \text{ (all real non negative numbers)}$$

Literally, we give a function “already undefined” since it does not stop on all values – the characteristic function of the halting set, which we know it’s not computable. Define χ_K as all values which are not inside $f(x)$ and:

$$f(x) = \begin{cases} \uparrow, & x \leq 1 \\ \chi_K(x), & x > 1 \end{cases}$$

Exercise 6.9. Is there non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{dom}(f) \cap \text{img}(f)$ is empty? Justify your answer (providing an example of such f , if it exists, or proving that cannot exist).

Here, we mean the same thing as before, but know we have a domain of values which are in the image but themselves are not computable; for example, if we define again a predicate χ_K which contains the set of values which are not present for $f(x)$ and we define a function like:

$$f(x) = \begin{cases} \chi_K\left(\left\lfloor \frac{x}{2} * 2 \right\rfloor\right), & \text{if } x \text{ odd} \\ \uparrow, & \text{otherwise} \end{cases}$$

The domain $\text{dom}(f)$ is the set of odd numbers, while the codomain $\text{cod}(f) = \{0, 2, 4, \dots\}$ is the combination of even numbers, so the intersection is empty. Given we defined χ_K , the $\frac{x+1}{2} * 2$ would be computable, but given it’s not, it’s not even recursive.

Exercise 6.10. Is there a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$, such that its image $\text{cod}(f) = \{y \mid \exists x \in \mathbb{N}. f(x) = y\}$ is finite? Provide an example or show that such a function does not exists.

Here we can use diagonalization to prove it is not computable by construction, since it’s different from all values. We can give a function f as follows:

$$f(x) = \begin{cases} sg(\phi_x(x)), & x \in W_x \\ 0, & x \notin W_x \end{cases}$$

The function is total by construction, $\text{cod}(f) \subseteq \{0, 1\}$, not computable since $f(x) \neq \phi_x(x)$ given when $\phi_x(x) \downarrow$ then $f(x) = sg(\phi_x(x)) = \phi_x(x)$ and when $\phi_x(x) \uparrow$ then $f(x) = 0 \neq \phi_x(x)$

Exercise 6.11(p). Prove that the function $f : \mathbb{N} \rightarrow \mathbb{N}$, defined as

$$f(x) = \begin{cases} \varphi_x(x) & \text{if } x \in W_x \\ x & \text{otherwise} \end{cases}$$

is not computable.

In this case, we have the diagonalization function, and we observe we can simply define:

$$g(x) = \begin{cases} \phi_x\left(\frac{x}{2}\right), & x < 0 \\ \frac{x}{2}, & x \geq 0 \end{cases}$$

This is not computable, given $g(x) \neq \phi_x\left(\frac{x}{2}\right)$ and if f were computable, g would have been so.

Exercise 6.12(p). Say if there is a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that, for infinite $x \in \mathbb{N}$ it holds

$$f(x) = \varphi_x(x)$$

If the answer is negative, provide a proof, if the answer is positive, provide an example of such a function.

We can define a function as follows:

$$f(x) = \begin{cases} \phi_x(x), & x \notin W_x \\ 0, & x \in W_x \end{cases}$$

By diagonalization, we get instead a function which is recursively defined, and computable, but different from previous input, like $h : \mathbb{N} \rightarrow \mathbb{N}$:

$$h(x) = f(x) + 1 = \begin{cases} \varphi_x(x) + 1, & x \notin W_x \\ 1, & x \in W_x \end{cases}$$

In this case, $f(x) + 1 \neq \varphi_x(x), \forall x \in \mathbb{N}$.

Exercise 6.13. Say if there is a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that

$$f(x) \neq \varphi_x(x)$$

only on a single argument $x \in \mathbb{N}$. If the answer is negative provide a proof, if the answer is positive give an example of such a function.

We define a function total, which can be

$$f(x) = \begin{cases} \varphi_x(x), & x \notin W_x \\ 0, & x \in W_x \end{cases}$$

In this case, the functions remains total in both cases, while also being uncomputable given $f(x) \neq \varphi_x(x)$ for the single argument $x \in \mathbb{N}$.

Exercise 6.14. Is there non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that

$$f(x) \neq \varphi_x(x)$$

only on a single $x \in \mathbb{N}$? If the answer is negative provide a proof of non-existence, otherwise give an example of such a function.

Define by diagonalization:

$$f(x) = \begin{cases} \varphi_x(x), & x \in W_x \\ k, & \text{otherwise} \end{cases}$$

- total by construction
- non-computable, because $\varphi_x(x) \neq f(x)$
 - o if $\varphi_x(x) \downarrow$ then $\varphi_x(x) \neq f(x)$
 - o if $\varphi_x(x) \uparrow$ then $\varphi_x(x) \neq k = f(x)$
- it differs from a single argument $k \notin W_x, k \in \mathbb{N}$, as desired

Exercise

Define a total non-computable $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\text{dom}(f) \subseteq \{0,1\}$. Can the function $\overline{sg} \circ f$ be computable?

Solution

We use diagonalization in order to prove this. We will get a non-computable function anyway, given if a function is computable, the other would be too.

$$f(x) = \begin{cases} \overline{sg}(\varphi_x(x)), & x \in W_x \\ x, & x \geq 0 \end{cases}$$

By definition, we find $f(x) \neq \varphi_x(x) \forall x \in \mathbb{N}$. But again, given we used diagonalization, the function cannot be computable.

Exercise 6.15. Is there a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{cod}(f)$ is the set \mathbb{P} of even numbers? Justify your answer response (providing an example of such f , if it exists, or proving that it does not exist).

We can define a function respecting the asked requirements as follows:

$$f(x) = \begin{cases} 2\varphi_x\left(\frac{x}{2}\right) + 2, & x \in W_x \\ 0, & x \notin W_x \end{cases}$$

This allows us to get access to all even numbers, since this covers all cases for which the codomain (outputs of function) are even and thanks to diagonalization $\varphi_x\left(\frac{x}{2}\right) \neq f(x)$, effectively making it non-computable.

Exercise 6.16. Say if there is a non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that the set $D = \{x \in \mathbb{N} \mid f(x) \neq \varphi_x(x)\}$ is finite. Justify your answer.

This exists and to prove it we define a function as follows:

$$f(x) = \begin{cases} 0, & x \in W_x \\ \varphi_x(x) + 1, & \text{otherwise} \end{cases}$$

Set D must be finite in order to accompany this proof and we show it is: it's total by construction, not computable since if $\varphi_x(x) \downarrow, f(x) = 0 \neq \varphi_x(x)$, when $\varphi_x(x) \uparrow, f(x) \neq \varphi_x(x) + 1$.

Exercise 6.17. Say if there are total computable functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(x) \neq \varphi_x(x)$ for each $x \in K$ and $g(x) \neq \varphi_x(x)$ for each $x \notin K$. Justify your answer by providing a example or by proving non-existence.

We're essentially arguing if there are total computable functions which are both total and computable different from φ_x . But this means this is different from all computable functions, actually, because

$f(x) \neq \varphi_x(x), \forall x \in K$. For g we can take a constant function, which reveals computable $\forall n$, say $g(x) = k$, where $k = 1$. This exists and makes us conclude $g(x) = 1 \neq \varphi_x(x) = \uparrow$

Exercise 6.18. Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(x) = \begin{cases} 2x + 1 & \text{if } \varphi_x(x) \downarrow \\ 2x - 1 & \text{otherwise} \end{cases}$$

Is it computable? Justify your answer.

It was solved by Baldan in an old 2016 exam.

It can be written as $\chi_K = sg(f(x) - 2x)$ which would be computable if and only if χ_K is, but we know K is not recursive, so χ_K is not computable.

Exercise 6.19(p). Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(x) = \begin{cases} x & \text{sg } \forall y \leq x. \varphi_y \text{ total} \\ 0 & \text{otherwise} \end{cases}$$

Is it computable? Justify your answer.

This is computable, intuitively, because it is bounded. We need to define the function with a lower bound, considering for example $y_0 = \min\{y \mid \varphi_y \text{ is not total}\}$. Then, we consider:

$$f(x) = \begin{cases} x, & \text{if } x < y_0 \\ 0, & \text{otherwise} \end{cases} = x * sg(y_0 - x)$$

which is computable since it is defined by cases.

Exercise 6.20. Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(x) = \begin{cases} x+2 & \text{if } \varphi_x(x) \downarrow \\ x-1 & \text{otherwise} \end{cases}$$

Is it computable? Justify your answer.

Solved by the old tutor, this basically involves using the characteristic function of the halting problem, so the idea behind χ_K .

$$\begin{aligned} \chi_K(x) &= \begin{cases} 1 & \varphi_x(x) \downarrow \\ 0 & \text{otherwise} \end{cases} & K &= \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\} \\ &\downarrow & \downarrow & x \in W_x \\ \text{NOT COMPUTABLE} & & \text{NOT RECURSIVE} & \\ \uparrow & \swarrow \text{NOT COMPUTABLE} & \uparrow & \text{R.E.} \\ \chi_K(x) &= g(f(x)) & \begin{cases} 1 & \varphi_x(x) \downarrow \\ 0 & \text{otherwise} \end{cases} & \\ \uparrow \text{NOT COMP} & \downarrow \text{COMPUTABLE} & & \end{aligned}$$

$$\begin{aligned} f(x) &= \begin{cases} x+2 & \varphi_x(x) \downarrow \\ x-1 & \text{otherwise} \end{cases} & | \\ \Xi sg(|f(x) - (x+2)|) &= \begin{cases} 0 & \varphi_x(x) \downarrow \\ |(x-1) - (x+2)| & \text{otherwise} \end{cases} & \nearrow 1 \\ \chi_K(x) &= \sim |f(x) - (x+2)| & \nearrow 0 & \nearrow 0 > 0 \\ & \sim |f(x) - (x-1)| & \text{sg}(x) &= \begin{cases} 1 & x \neq 0 \\ 0 & x = 0 \end{cases} \end{aligned}$$

Exercise 6.21. Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(x) = \begin{cases} \varphi_x(x+1) + 1 & \text{if } \varphi_x(x+1) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Is it computable? Justify your answer.

Suppose $f(x)$ is computable, we define by contradiction $\chi_K = \overline{sg}(|f(x) - \phi_x(x+1) + 1|)$ which would be computable, but χ_K is not.

Exercise 6.22. Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } \varphi_y(y) \downarrow \text{ for each } y \leq x \\ 0 & \text{otherwise} \end{cases}$$

Is it computable? Justify your answer.

The function is not computable. Given $\varphi_y(y) \forall y \leq x$, define a function $y_0 = \min\{y \mid \varphi_y(y) \downarrow\}$. We then define a function

$$f(x) = \begin{cases} x, & x < y_0 \\ 0, & \text{otherwise} \end{cases} = x * sg(y_0 - (\varphi_x(x) + 1))$$

Given the function depends on the behavior of φ_x , given $f(x) \neq \varphi_x(x) + 1$, we argue the function is not computable – as always by diagonalization.

Exercise 6.23. Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$f(x) = \begin{cases} x^2 & \text{if } \varphi_x(x) \downarrow \\ x + 1 & \text{otherwise} \end{cases}$$

Is it computable? Justify your answer.

The function is not computable, given $\phi_x(x) \neq f(x) + 1 \neq x + 1 \forall x \in \mathbb{N}$. We can consider a function $g(x) = sg(f(x) - (x + 1))$, we get a function in the halting set $g(x) = X_K(x)$, given it is different from x^2 the $x + 1$ part thanks to the partial recursion.

Exercise 6.24. A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is called *almost total* if it is undefined on a finite set of points. Is there an almost total and computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $f \subseteq \chi_K$? Justify your answer by giving an example of such a function in case it exists or a proof of non-existence, otherwise.

Consider the function f , almost total and $\subseteq \chi_K$. Given this representation, it definitely has to be undefined, given the function χ_K would be computable itself in that case.

Consider simply:

$$\chi_K(x) = \begin{cases} 1, & x \in K \\ 0, & \text{otherwise} \end{cases}$$

Since f is computable, it should also be total for every possible input, anyway this does not happen for a finite set of points and therefore it cannot exist. This can equivalently be shown like the “official” solution, using a restriction (the pipe | sign for $\chi_K|_{\overline{D}}$):

Solution: Let f be almost total and assume that $f \subseteq \chi_K$. Note that, if we let $D = \text{dom}(f)$, one has that \overline{D} is finite and therefore recursive. Thus also D is recursive. Define $\theta = \chi_K|_{\overline{D}}$, which is a finite function, therefore computable.

Now, we observe that

$$\chi_K(x) = \begin{cases} f(x) & x \in D \\ \theta(x) & \text{otherwise} \end{cases}$$

and conclude that f cannot be computable, otherwise also χ_K would be computable. \square

Exercise 6.25. Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is *almost constant* if there is a value $k \in \mathbb{N}$ such that the set $\{x \mid f(x) \neq k\}$ is finite. Is there an almost constant function which is not computable? Adequately motivate your answer.

We will prove there is a function with this requirements but computable, constructing a finite subfunction:

$$\theta(x) = \begin{cases} f(x), & \text{if } f(x) \neq k \\ \uparrow, & \text{otherwise} \end{cases}$$

Therefore, we write $f(x)$ as follows:

$$f(x) = \begin{cases} \theta(x), & \text{if } f(x) \neq k \\ k, & \text{otherwise} \end{cases}$$

Since the subfunction is finite, it is computable and let $\theta = \phi_e$ (so, the finite index of program).

Thanks to this, we can describe the function as:

$$f(x) = \mu w. (f(x) \in K \wedge S(e, x, (w)_1, (w)_2) \vee (f(x) \notin K \wedge (w)_1 = k))_1$$

This way, we are considering all cases for x and also proving the function is computable, since it is defined by cases.

Exercise 6.26. Is there a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ with the property that $f(x) = x^2$ for all $x \in \mathbb{N}$ such that $\varphi_x(x) \downarrow$? Justify your answer by providing an example of such function, if it exists, or by proving that it does not exist, otherwise.

Yes, we can define such a function, which would essentially compute x^2 if $\phi_x(x) \downarrow$, for example like:

$$f(x) = \begin{cases} x^2, & \text{if } \phi_x(x) \downarrow \\ x^2 + 1, & \text{otherwise} \end{cases}$$

This holds since $f(x) \neq \varphi_x(x) + 1$, so $x^2 + 1 \neq x^2$ and it is not computable since $\chi_K(x) = \overline{sg}(f(x) - x^2)$

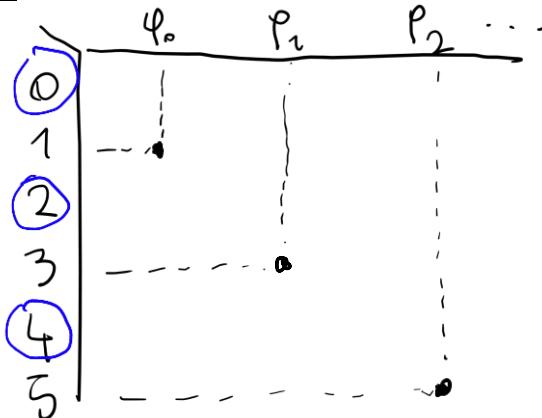
Exercise 6.27(p). Is there a non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for any non-computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ the function $f * g$ (defined as $(f * g)(x) = f(x) \cdot g(x)$) is computable? Justify your answer (providing an example of such f , if it exists, or proving that it does not exist).

Not necessarily the product of non-computable functions leads to something computable or not computable, it depends on the case. Assume that this function exists; we are not arguing about the nature of the functions themselves but consider $f = g$ and $f * f$ is computable, so we have $(f * f)(x) = f(x) * f(x) = f(x^2)$.

In this case, consider $f(x) = \mu y. |f(x^2) - y^2|$ which is computable, leading to a contradiction, considering the function was non-computable before.

Exercise 6.28(p). Define a function $f : \mathbb{N} \rightarrow \mathbb{N}$ total and not computable such that $f(x) = x/2$ for each even $x \in \mathbb{N}$ or prove that such a function does not exist.

1st idea



$$f(x) = \begin{cases} x, & \text{if } x \text{ is even} \\ \phi_{\frac{x-1}{2}}(x) + 1, & \text{if } x \text{ is odd and } \phi_{\frac{x-1}{2}}(x) \downarrow \\ 0, & \text{if } x \text{ is odd and } \phi_{\frac{x-1}{2}}(x) \uparrow \end{cases}$$

- f total
- $f(x) = x \quad \forall x$ even (infinite set)
- f not computable (total and \neq from all total computable functions) ($\forall x$ if φ_x is total, $f(2x+1) = \varphi_x(2x+1) + 1 \neq \varphi_x(2x+1)$)

Exercise 6.29. Is there a total non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that the function $g : \mathbb{N} \rightarrow \mathbb{N}$ defined, for each $x \in \mathbb{N}$, by $g(x) = f(x) - x$ is computable? Provide an example or prove that such a function does not exist.

Essentially, the problem considers a non-computable function which when subtracted always gives a valuable result, hence “stopping/halting” for each input. So, solution found, the indicator function of the halting problem, which is $\chi_K(x)$. This way, $f(x) - x$ will give a constant value as 0, otherwise, something that always keeps being inside $x \in \mathbb{N}$, being therefore computable.

Exercise 6.30(p). Is there may be a non-computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for each non-computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ 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).

If the sum is not computable, there is not a non-computable function which can make the sum computable. Let's argue it more formally; consider the case where g_f is the function that is always equal to f , i.e. $g_f(x) = f(x) \quad \forall x \in \mathbb{N}$. Since f is non-computable, g_f also is not.

The sum would essentially result in $2f$ which is again not computable; if it was, then f must be computable as well, which is not the case.

Exercise 6.31. Is there a computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{dom}(f) = K$ and $\text{cod}(f) = \mathbb{N}$? Justify your answer.

To have a function which has the domain inside the halting set, such function should always halt for all inputs, or at least, be recursively defined and having a constant value in the codomain, which is the case of the constant function over the halting set.

So, we define $f(x) = \varphi_x(x)$ and $\text{dom}(f) = K$; we will consider an index of the constant function k s.t. $f(e) = \varphi_e(e) = k$ and the codomain $\text{cod}(f) = \mathbb{N}$

Alternatively, one can use the minimization over the t steps of the halting function, simply saying that “the function will halt on input x , with output x in t steps being sure to stop, hence the -1 and the program will be computable for sure, using γ which is the encoding.

$$f(x) = (\mu t. H(x, x, t)) - 1$$

Clearly $\text{dom}(f) = K$ since $f(x) \downarrow$ if there exists some t such that $H(x, x, t)$, i.e., if $x \in K$. Furthermore, for each $x \in \mathbb{N}$ just take the program Z_k which consists of $Z(1)$ repeated x times. For the corresponding index $y = \gamma(Z_k)$ we will have $f(y) = k - 1$, which shows that $\text{cod}(f) = \mathbb{N}$. \square

Exercise 6.32. Let A be a recursive set and let $f_1, f_2 : \mathbb{N} \rightarrow \mathbb{N}$ be computable functions. Prove that the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined below is computable:

$$f(x) = \begin{cases} f_1(x) & \text{if } x \in A \\ f_2(x) & \text{if } x \notin A \end{cases}$$

Does the result hold if we weaken the hypotheses and assume A only r.e.? Explain how the proof can be adapted, if the answer is positive, or provide a counterexample, otherwise.

Solution: Let $e_1, e_2 \in \mathbb{N}$ be indexes for f_1, f_2 , respectively, namely $\varphi_{e_1} = f_1$ and $\varphi_{e_2} = f_2$. Observe that we can define f as

$$f(x) = (\mu w. ((S(e_1, x, (w)_1, (w)_2) \wedge \chi_A(x) = 1) \vee (S(e_2, x, (w)_1, (w)_2) \wedge \chi_A(x) = 0)))_1$$

showing that f is computable. Relaxing the hypotheses to recursive enumerability of A , the result is no longer true. Consider for instance $f_1(x) = 1$, $f_2(x) = 0$ and $A = K$, which is r.e. Then f defined as above would be the characteristic function of K which is not computable. \square

Exercise 6.33(p). Is there a total, non-computable function such that $\text{img}(f) = \{f(x) \mid x \in \mathbb{N}\}$ is the set Pr of Prime numbers? Justify your answer.

Yes, there is. Consider a function which allows us to obtain a prime number while also being different from all computed values, for instance, $f(x) \neq \varphi_x(x)$. Consider the set of prime number is not computable, so we would need a diagonalization argument. We can either consider a function in which we give a max or a min of all values, considering “we would never reach that”, so:

$$f(x) = \begin{cases} p, & x \in W_x \\ 0, & \text{otherwise} \end{cases}$$

In which $p = \max \{p' \in \text{Pr} \mid p' < \varphi_x(x)\}$.

The functions:

- is total, given it's defined for all natural numbers in any case of definition
- it's not computable, since we have $\forall x \in \mathbb{N}, f(x) \neq \varphi_x(x)$, we would get a prime number smaller than the recursion (which never happens, given the prime set is not computable), otherwise we get 0
- the image is included, because given a definite prime number, using the constant function, we might get back the original value, considering is total and for each valuable index n , the ordering property of natural numbers holds.
 - o this is like saying $f(n) = \max \{p' \in \text{Pr} \mid p' > \varphi_n(n)\} > \max \{p' \in \text{Pr} \mid p' > \varphi_n(n)\}$, thus having $p' \in \text{img}(F)$ and so p

Exercise (2022-01-19)

Is there a non-computable total function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(x) = f(x+1)$ on infinitely many inputs x , i.e., such that the set $\{x \in \mathbb{N} \mid f(x) = f(x+1)\}$ is infinite? Provide an example or show that such a function cannot exist.

Solution

Such a function can exist, for instance let's define a function $f : \mathbb{N} \rightarrow \mathbb{N}$ s.t.

$$f(x) = \begin{cases} 0, & x \notin W_x \text{ or } x \text{ is not a multiple of 2} \\ \phi_{\frac{x}{2}}(x), & x \in W_x \text{ or } x \text{ is a multiple of 2} \end{cases}$$

Such function:

- is total by construction (defined by cases)
- $f(x) = f(x+1) = 0$ when x and $x+1$ are defined, given they are not multiples of 2
- it is not computable, considering $\forall x \in \mathbb{N}, f \neq \phi_x$, specifically $\phi_{\frac{x}{2}}(x) \neq \phi_{\frac{x}{2}}(x+1)$ when $\phi_{\frac{x}{2}}(x) \downarrow$.
When $\phi_{\frac{x}{2}}(x) \uparrow$, then $f(2x) = 0 \neq \phi_{\frac{x}{2}}(2x)$

An intricate yet interesting approach from prof. Baldan:

A more elegant, but less immediate solution is to take $f = \chi_K$, the characteristic function of the halting set K , which is total and not computable. It is true but not obvious that $\chi_k(x) = \chi_K(x+1)$ for infinitely many x . Assume by contradiction that, instead, $D = \{x \mid \chi_K(x) = \chi_K(x+1)\}$ is finite and let $d = \max D$. This means that for all $x > d$ it holds that $\chi_K(x) \neq \chi_K(x+1)$ and since χ_K can assume only values 0 and 1, $\chi_K(x+1) = \overline{sg}(\chi_K(x))$.

Now, let $v_x = \chi_K(x)$ for $x \in \{0, \dots, d\}$. Moreover, consider the function $g : \mathbb{N} \rightarrow \mathbb{N}$ defined by primitive recursion

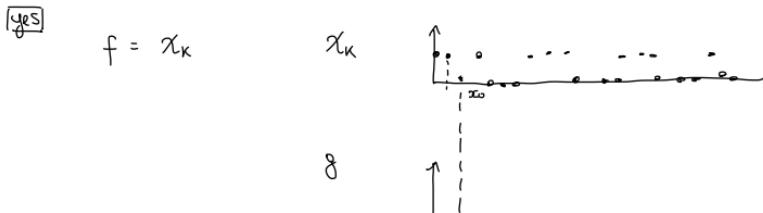
$$\begin{aligned} g(0) &= v_d \\ g(y+1) &= \overline{sg}(g(y)) \end{aligned}$$

Then we have that

$$\chi_k(x) = \begin{cases} v_x & \text{if } x \leq d \\ f(x-(d+1)) & \text{otherwise} \end{cases} = \prod_{i=1}^k (v_i \cdot sg(|x-i|)) + sg(x-d)f(x-(d+1))$$

Exercise (latest lessons 2021-2022)

Is there a total computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x) = \prod_{y < x} f(y)$ is computable?

Solution

$$x_0 = \min \{x \mid \chi_K(x) = 0\}$$

$$g(x) = \begin{cases} 1 & x < x_0 \\ 0 & \text{otherwise} \end{cases} = sg(x_0 - x)$$

So, it can't exist. Suppose $g(x) = \prod_{y < x} f(y)$ is computable then, $f(y)$ has to be computable, specifically using g as a recursive definition, something like: $f(x) = g(x - 1) * g(x)$ which should be computable considering it is the composition of computable functions, which actually is not.

Exercise (2010-03-19)

Prove if the function $f: \mathbb{N} \rightarrow \mathbb{N}$ defined as:

$$f(x) = \begin{cases} x, & \forall x \leq x, \phi_y \text{ total} \\ 0, & \text{otherwise} \end{cases}$$

is computable. Give adequate reasons for your answer.

Solution

The function is computable. Given $\phi_y(y)$ is total, define $y_0 = \min\{y \mid \phi_y(y) \downarrow\}$. We then define a function

$$f(x) = \begin{cases} x, & x < y_0 \\ 0, & \text{otherwise} \end{cases} = x * sg(y_0 - \phi_y(y))$$

which is computable.

Exercise (2019-11-18-solved)

Define a total non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $dom(f) \subseteq \{0,1\}$.

Can the function $\overline{sg} \circ f$ be computable? Motivate your answer

Solution

Define $f: \mathbb{N} \rightarrow \mathbb{N}$ by diagonalization as follows:

$$f(x) = \begin{cases} \overline{sg}(\phi_x(x)), & x \in W_x \\ 0, & \text{otherwise} \end{cases}$$

f is total but not computable, given by definition $f(x) \neq \phi_x(x) \forall x \in N$.

Observe $\overline{sg} \circ (\overline{sg} \circ f)$, so if it $\overline{sg} \circ f$ were computable, also f would be computable by composition, but it is not.

Exercise (2017-11-20)

Is there a total non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $cod(f) = \{y \mid \exists x \in \mathbb{N}. f(x) = y\}$ is finite? Show an example or show that this function cannot exist.

Solution

Yes, it exists. Consider:

$$f(x) = \begin{cases} \overline{sg}(\phi_x(x)), & x \in W_x \\ 0, & \text{otherwise} \end{cases}$$

The function f :

- is total
- is not computable since for all $x \in N$ we have $f(x) \neq \phi_x(x)$; in fact, if $\phi_x(x) \downarrow$ then $f(x) = \overline{sg}(\phi_x(x)) \neq \phi_x(x)$ and if instead $\phi_x(x) \uparrow$ then $f(x) = 0 \neq \phi_x(x)$
- clearly, $cod(f) \subseteq \{0,1\}$

Exercise (2018-11-20-parziale)

Is there an index $e \in \mathbb{N}$ and a non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$, such that denoted by $\text{dom}(f)$ and $\text{cod}(f)$ domain and codomain of f (such that $\text{dom}(f) = \{x \mid f(x) \downarrow\}$ and $\text{cod}(f) = \{y \mid \exists x. f(x) = y\}$), we have $\text{dom}(f) = W_e$ and $\text{cod}(f) = E_e$? Show an example or bring a counterexample. Can a function f such that $\text{dom}(f) = W_e$ and $\text{cod}(f) = E_e$ be found for all $e \in \mathbb{N}$?

Solution

For the first part, consider an index $e \in \mathbb{N}$ for the identity function, so $W_e = E_e = \mathbb{N}$ and define:

$$f(x) = \begin{cases} \phi_x(x) + 1, & x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

The function f is total, so $\text{dom}(f) = \mathbb{N} = W_e$. Moreover, $\text{dom}(f) = \mathbb{N} = E_e$. In fact, for each $n \in \mathbb{N}$, if $n = 0$ then, given an index x of the always undefined function, we have $f(x) = 0$. If $n > 0$, then consider whatever index x of the constant function $n - 1$ and we have $f(x) = n - 1 + 1 = n$.

For the second question, the answer is clearly no. For example, if we consider $e \in \mathbb{N}$ such that ϕ_e is the always undefined function, every f such that $\text{dom}(f) = W_e = \emptyset$ coincides with ϕ_e and so it is computable.

Exercise (2020-11-23)

Define a total non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\text{img}(f) = \{2^n \mid n \in \mathbb{N}\}$ (where $\text{img}(f) = \{f(x) \mid x \in \mathbb{N}\}$)

Solution

Proceed by diagonalization defining:

$$f(x) = \begin{cases} 2^{\phi_x(x)}, & \text{if } x \in W_x \\ 1, & \text{otherwise} \end{cases}$$

The function f is clearly total. Moreover:

- f not computable since $\forall x \in \mathbb{N}, f \neq \phi_x$, so f is different from all computable function. Infact, if $x \in W_x$ then $f(x) = 2^{\phi_x(x)} > \phi_x(x)$ and so $x \notin W_x$ we have $f(x) = 1 \neq \phi_x(x)$ given $\phi_x(x) \uparrow$.
- It holds $\text{img}(f) = \{2^n \mid n \in \mathbb{N}\}$. By definition, $\text{img}(f) \subseteq \{2^n \mid n \in \mathbb{N}\}$ given $x \in W_x$ and so $f(x) = 2^{\phi_x(x)}$ and $x \notin W_x$ then $f(x) = 1 = 2^0$. The converse implication also holds. Infact, given any $n \in \mathbb{N}$, the constant function n is clearly computable. Using x as whatever index for that function, $\phi_x(y) = n$ for all y given $x \in W_x = \mathbb{N}$ it holds $f(x) = 2^{\phi_x(x)} = 2^n$.

22.7 REDUCTION, RECURSIVENESS AND RECURSIVE ENUMERABILITY

Note: understand exactly what reduction does and read carefully what you have to prove and try to prove it the simplest way possible (knowing what recursive/semidecidable and other things are). They are “if and only if” – prove the first part and then move to the second one, keeping the conditions.

Exercise 7.1. Prove that a set A is recursive if and only if there is a total computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $x \in A$ if and only if $f(x) > x$.

If A is recursive, then its function is computable and so we want to make a function in which $f(x) > x$, so we can create for example $f(x) = x + \chi_A(x)$, where χ_A is the characteristic function. Given it holds if and only if the function is $f(x) > x$, then we can have: $\chi_A(x) = sg(f(x) - x)$ is computable and therefore A is recursive.

Exercise 7.2. Prove that a set A is recursive if and only if there are two total computable functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ such that for each $x \in \mathbb{N}$

$$x \in A \text{ if and only if } f(x) > g(x).$$

Let A be recursive, then like before the characteristic function χ_A is computable. We can give some values for the functions, say $f(x) = \frac{x}{2}$ and $g(x) = \frac{x}{3}$, respecting the condition. The functions are computable if and only if $f(x) > g(x)$, and so like before we can use $\chi_A(x) = sg(f(x) - g(x))$ and therefore A is recursive and χ_A computable.

Exercise 7.3. Prove that a set A is recursive if and only if $A \leq_m \{0\}$.

As before, let A recursive, then χ_A is computable. We try to give a value to the reducing function, which can be something which can give 0 as a result, which means it terminates. We argue the function can be $1 + \chi_A(x)$. Conversely, if reduction holds, we have $\chi_A(x) = sg(f(x))$, giving the appropriate finite value and then A is recursive and the function is computable.

Exercise 7.4. Let $A \subseteq \mathbb{N}$ be a set and let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a computable function. Prove that if A is r.e. then $f(A) = \{y \in \mathbb{N} \mid \exists x \in A. y = f(x)\}$ is r.e. Is the converse also true? That is, from $f(A)$ r.e. can we deduce that A is r.e.?

Given A is r.e., there exists a sc_A computable and so $y = f(x)$ for some $x \in A$. To prove this, we have to define how sc_A should appear, hence composing $f(A)$ and f together, obtaining this way all inputs.

Hence, we give:

$$sc_{f(A)}(y) = \mathbf{1}(\mu w. |\chi_A - 1|)$$

considering the possible function to give is

$$\chi_{f(A)}(y) = \begin{cases} 1, & \exists x \in A \text{ s.t. } f(x) = y \\ \uparrow, & \text{otherwise} \end{cases}$$

hence, if this one semidecides $f(A)$ it returns the required y if there exists some $x \in A$ s.t. $f(x) = y$. This makes $f(A)$ recursively enumerable.

The converse is not true, considering for example $f(A) = \{0\}$, which is clearly an enumerable set, but A is not r.e.

Solution: Let e, e' be such that $f = \varphi_e$ and $sc_A = \varphi_{e'}$. Then

$$sc_{f(A)}(y) = \mathbf{1}(\mu w. H(e', (w)_1, (w)_2) \wedge S(e, (w)_1, y, (w)_3))$$

hence $f(A)$ is r.e. The converse is not true. For example $\mathbf{1}(\bar{K}) = \{1\}$ is r.e., but \bar{K} is not r.e. \square

Exercise 7.5. Let A be a recursive set and $f : \mathbb{N} \rightarrow \mathbb{N}$ be a total computable function. Is it true, in general, that $f(A)$ is r.e.? Is it true that $f(A)$ is recursive? Justify your answers with a proof or counterexample.

Since A it's recursive, there exists a total computable function X_A and we consider:

$$sc_{f(A)}(x) = \chi_A(\mu w. f(x) - y)$$

The function semi-decides the predicate whether $y \in A$ but is not necessarily recursive, given the set of $f(A)$ can be infinite and so $f(A)$ is not recursive.

As a counterexample, consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined as $f(x) = 2x$ and the recursive set $A = \{x \mid x \text{ is even}\}$. The set $f(A)$ is clearly r.e., but not recursive, given we know nothing about the nature of the underlying set.

Exercise 7.6. Let $A \subseteq \mathbb{N}$ be a set and let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a computable function. Prove that if A is recursive then $f^{-1}(A) = \{x \in \mathbb{N} \mid f(x) \in A\}$ is r.e. Is the set $f^{-1}(A)$ also recursive? For the latter give a proof or provide a counterexample.

If A is recursive, there exists some function able to compute it, which is $\chi_A(f(x))$. This is infact obtained by the semicharacteristic function this way:

$$sc_{f^{-1}(A)}(x) = \chi_A(f(x))$$

The set $f^{-1}(A)$ is not recursive, given it is the halting set K (so, it holds $sc_K^{-1}(\mathbb{N}) = K$)

Exercise 7.7. Prove that a set A is r.e. if and only if $A \leq_m K$.

We can prove this in two ways:

(\Rightarrow) If A is r.e., there exists a total computable function able to compute its inputs and effectively provide outputs. Hence, the semicharacteristic function can be described by the smn-theorem as the parametrization of the underlying subinputs, specifically as $g(x, y) = sc_A(x, y)$.

(\Leftarrow) If A is reducible to K , it means there is only one $x \in A$ for which $f(x) \in K$. This means that the function $sc_A(x)$ can be defined as $sc_K(f(x))$, this way obtaining a single value as shown by the smn-theorem, like $sc_A(x, y) = 0/1$ if $f(x) \in K$ or $f(x) \notin K$

Exercise 7.8. Prove that a set A is r.e. if and only if there is a computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $A = \text{img}(f)$ (remember that $\text{img}(f) = \{y : \exists z. y = f(z)\}$).

If a set A is r.e., we consider $f(x) = x * sc_A(x)$, which means we can either obtain that value or not. Conversely, if $A = \text{img}(f)$ for the function f , we can define $sc_A(x) = \mathbf{1}(\mu w. (f(z) - y))$, which can be done by partial recursion and composition like $f(x) = \phi_e$ for a suitable $e \in N$, doing $sc_A(x) = \mathbf{1}(\mu w. S(e, (w)_1, x, (w)_2))$.

Exercise 7.9. Given a function $f : \mathbb{N} \rightarrow \mathbb{N}$, define the predicate $P_f(x, y) \equiv "f(x) = y"$, i.e., $P_f(x, y)$ is true if $x \in \text{dom}(f)$ and $f(x) = y$. Prove that f is computable if and only if the predicate $P_f(x, y)$ is semi-decidable.

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a computable function. Considering the predicate is semidecidable, there exists a semicharacteristic function able to compute $f(x)$. Specifically, if we consider f as the computation of the program on index e such that $f = \phi_e$ then we have $sc_P(x, y) = \mathbf{1}(\mu w. |f(x) - y|)$ as computable, given $P_f(x, y)$ is semidecidable.

Viceversa, if P is semidecidable, there exists a program e such that $\phi_e = sc_P$. Then, we can characterize the semicharacteristic function using the computation of index e over $f(x)$ such that:

$$f(x) = \mu w. (H(e, x, (w)_1, (w)_2) \wedge S(e, (w)_1, (w)_2))$$

which is computable thanks to f . Alternatively, for this last part:

Vice versa, let $P(x, y)$ be semidecidable and let e be an index for the semi-characteristics function of P , namely $\varphi_e^{(2)} = sc_P$. Then we have $f(x) = (\mu w. H^{(2)}(e, (x, (w)_1), (w)_2))_1$. \square

Exercise 7.10. Let $A \subseteq \mathbb{N}$. Prove that A is recursive and infinite if and only if it is the image of a function $f : \mathbb{N} \rightarrow \mathbb{N}$ computable, total and strictly increasing (i.e., such that for each $x, y \in \mathbb{N}$, if $x < y$ then $f(x) < f(y)$).

Let's start defining a function just like the exercise defined. We define a function $g : \mathbb{N} \rightarrow \mathbb{N}$ which is both recursive and infinite, like $g(x) = \prod_y \chi_A(y) + \phi_x$

This function, given the productory lists all elements according to the underlying property, assigns values in a monotone way, increasing each time as x increases. Given it is monotone, for each $x \in \mathbb{N}$, $g(x) \leq g(x + 1)$. This is also infinite, cause the result of recursion is inside the natural and so $\text{img}(\mathbb{N}) = \mathbb{N}$. We then define the function f as follows, via minimalization, hence making the computation possible (for the property $x < y$).

$$f(x) = \mu x. g(x + 1) = n + 1$$

The function is computable, given it uses minimalization, and it is total, given the image is always defined recursively over the naturals. By using this definition, it is also increasing, given $x < y$, hence constructing $g(n) < g(m) = g(f(n) + 1) < g(f(m) + 1)$ and therefore $f(n) < f(m)$

The characteristic function χ_A allows us to get x , which will get 1 if $\chi_A(x) = 1$ otherwise will allow us to get $n + 1$.

For the converse implication, we use the image of function itself as infinite, with a function total computable and increasing. Given the set A is infinite and increasing, we can see $f(x) \geq z$ s.t. $z \leq y$. This way, the characteristic function can be expressed as $X_A(x) = sg(\prod_{z=0}^y g(z) + f(x))$. This ensures $f(x) \geq z$ is satisfied and it's defined totally on all possible values, hence being computable.

Exercise 7.11. Let $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$ be the function encoding pairs of natural numbers into the natural numbers. Prove that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is computable if and only if the set $A_f = \{\pi(x, f(x)) \mid x \in \mathbb{N}\}$ is recursively enumerable.

We will prove this in two ways:

- Assuming we have the π function, which is the encoding in pairs, it holds there exists a computable function $f(x) = \phi_x = \Psi_U(x, x)$. Considering a suitable index $e \in \mathbb{N}$, we use a function $f = \phi_e$ as follows, considering both input and output will be inside the domain (using π as encoding instead of w):

$$sc_A(x) = \mathbf{1}(\mu w. S(e, (\pi)_1, x, (\pi)_2) \wedge H(e, (\pi)_1, \pi, (\pi)_3))$$

Given this definition, we consider a function $h : \mathbb{N} \rightarrow \mathbb{N}$ be such that $h(x) = \pi(g(x))$ and since g is computable, $h(x)$ also is. Hence, the semicharacteristic function semidecides the predicate and $\chi_{A(f)}$ is computable and the set is r.e.

- Assume now A_f is r.e. hence there exists a total computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $A_f = img(g)$ and it means there exists a corresponding $g(y) = \pi(x, f(x))$. By minimalization, the encoding in pairs will give us back exactly one value, the needed one, so we can express:

$$f(x) = \mathbf{1}(\mu x. (g(y) - \pi(x, f(x))))$$

Hence showing that if it is r.e., it is computable.

Exercise 7.12. Prove that a set $A \subseteq \mathbb{N}$ is recursive if and only if $A \leq_m \{0\}$.

If A is recursive, there exists its characteristic function

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases}$$

If A is recursive, by definition such function is computable, and it is total. It is immediate to see that it is a reduction function for $A \leq_m \{0\}$ since $x \in A$ iff $\chi_A(x) = 0$ iff $\chi_A(x) \in \{0\}$.

Exercise 7.13. Let $A \subseteq \mathbb{N}$ be a non-empty set. Prove that A is recursively enumerable if and only if there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $dom(f)$ is the set of prime numbers and $img(f) = A$.

We will prove this in two ways:

If A is a non-empty set, we will prove this is r.e., defining a function for example like:

$$f(x) = \begin{cases} p, & \text{if } x \in W_x \text{ and } p = \min \{p' \in \text{Pr} \mid p' < \phi_x(x)\} \\ 1, & \text{otherwise} \end{cases}$$

The function defined this way:

- is total, given it is defined for all natural numbers
- it's computable, given we characterize the following:

$$sc_A(x) = \mathbf{1}(\mu w. S(x, (w)_1, p, (w)_2))$$

also, given the condition $f(x) < \phi_x(x)$ and will halt for its values having $\phi_x(x) \downarrow$ when $f(x)$ has a prime number smaller than $\phi_x(x)$, having it defined for all cases

- we have $img(f) = A$, given from each prime number we can construct a constant function $g(x) = p - 1 \forall x \in \mathbb{N}$ and the function is computable given for a suitable index n we can construct $g = \phi_n$

For the converse implication, if A is r.e., there exists a semicharacteristic function s.t. $\text{dom}(f) = p$ and $\text{img}(f) = A$. We construct $f(x) = x * sc_A(x)$ and this will be well-defined when describing: 1 if elements are in A , \uparrow if not (in that case, the function $f(x) = 0$ for any fixed value to have it well-defined), given it will halt giving in output a definite prime number, making it computable and defined for all possible values.

Exercise 7.14. Let $\mathcal{A} \subseteq \mathcal{C}$ be a set of computable functions such that, denoted by $\mathbf{0}$ and $\mathbf{1}$ the constant functions 0 and 1, respectively, we have $\mathbf{0} \notin \mathcal{A}$ and $\mathbf{1} \in \mathcal{A}$. Define $A = \{x : \varphi_x \in \mathcal{A}\}$ and show that either A is not or \bar{A} is not r.e.

Consider in the first case A is not recursive and it is saturated, since $\{A = x : \varphi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid \varphi_x(x) \downarrow\}$. By Rice's theorem, we have that $A \neq \emptyset, A \neq \mathbb{N}$ since:

- if $e \in \mathbb{N}$, consider $\phi_e = id$ and so $e \in A$, since $e \in \mathbb{N}$ and $\phi_e = \mathbf{1} \in \mathbb{N}$, but $\neq \mathbb{N}$
- if $e' \in \mathbb{N}$ consider $\phi'_e = \emptyset$ then $e \notin A$, so $e \notin \emptyset$ and $\phi_e = \mathbf{1} \neq \emptyset$

We already see the set is not recursive. For the converse set it's literally the same proof. Therefore, given they are both not recursive, they would not be r.e.

Exercise 7.15. Establish whether an index $x \in \mathbb{N}$ can exist such that $\bar{K} = \{2^y - 1 : y \in E_x\}$. Justify your answer.

Looking at the problem definition, it cannot exist, because $2^y - 1$ to be inside E_x must be recursively defined, hence recursively enumerable inside a set. The problem gives us the complement of the halting set, which we know it is not r.e. More precisely, this can be seen as $\bar{K} = \{2^y - 1 : y \in E_x\} = \text{img}(f \circ E_x)$ which would imply the set is r.e., unlike \bar{K} and they cannot coincide.

Exercise 7.16. Given two sets $A, B \subseteq \mathbb{N}$ what $A \leq_m B$ means. Prove that given $A, B, C \subseteq \mathbb{N}$ the following hold:

- a. if $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$;
- b. if $A \neq \mathbb{N}$ then $\emptyset \leq_m A$.

For the first part, we are trying to prove transitivity, so if we consider A , there must be a computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ such that $f(x) \in B \forall x \in \mathbb{N}$ and $x \in A$.

Similarly for B , there will be a total computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall y \in \mathbb{N}, y \in B, g(y) \in C$. Using composition ensures that considering a composing function $h: \mathbb{N} \rightarrow \mathbb{N}$ as $h(x) = g(f(x))$, it will happen that $h(x) = g(f(x)) \in C$, ensuring the previous properties (given $g \circ f$ holds as computable).

For the second part, if A is not inside the naturals, we simply consider a value which is not present inside the set. This way, the always undefined function will reduce to A iff and only if it is defined on a value the original set is never defined upon, giving "empty" as a result (more formally, a fixpoint). We simply consider for example a_0 s.t. $a_0 \notin A, f(x) = a_0$. This holds for each $x \in \mathbb{N}$, hence working properly.

Exercise 7.17. Given two sets $A, B \subseteq \mathbb{N}$ define what $A \leq_m B$ means. Is it the case that $A \leq_m A \cup \{0\}$ for all sets A ? If the answer is positive, provide a proof, otherwise, a counterexample. In the second case, identify a condition (specifying whether it is only sufficient or also necessary) that make $A \leq_m A \cup \{0\}$ true.

For the first part, we recall the definition given [here](#) for reduction. The intersection does not hold, since $\{0\}$ is part of the naturals, but here can happen $x \in \mathbb{N} \setminus \{0\}$ and so the reduction cannot work.

In general, this holds, but we have to distinguish cases over the 0 value:

- if $0 \in A$ it will hold for all sets A and the function will simply map 0 into A using for example the constant function or the identity function on 0
- if $0 \notin A$, given A is finite and $\neq \mathbb{N}$ we would have $x_0 \notin A, x_0 \neq 0$ and so the reduction function can be:

$$f(x) = \begin{cases} x_0, & \text{if } x = 0 \\ x, & \text{otherwise} \end{cases}$$

Exercise 7.18. Given two sets $A, B \subseteq \mathbb{N}$ define what $A \leq_m B$ means. Prove that, given any $A \subseteq \mathbb{N}$, we have A r.e. iff $A \leq_m K$.

Given sets $A, B \subseteq \mathbb{N}$, we say that $A \leq_m B$ if there exists a total computable function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for all $x \in \mathbb{N}$, it holds $x \in A$ iff $f(x) \in B$.

Considering $A \leq_m K$, we know A is r.e., given K is r.e. too. Specifically, a semicharacteristic function can be defined as:

$$sc_A(x) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

To properly define it, given A is r.e., we can use the smn-theorem defining a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = sc_A(x) = \phi_{s(x)}(y)$ and this way, given there exists only one index on which this is parametrized, then s can be correctly considered as a reduction function for $A \leq_m K$.

Exercise 7.19. Prove that a set $A \subseteq \mathbb{N}$ is recursive if and only if A and \bar{A} are r.e.

PROOF. (\Rightarrow) If A recursive,

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & \text{otherwise} \end{cases}$$

is computable. Then $sc_A(x) = \mathbf{1}(\mu z. |\chi_A(x) - 1|)$ is computable, therefore A is r.e. Since A is recursive, then \bar{A} is recursive, thus, r.e.

(\Leftarrow) Let A, \bar{A} be r.e., then by definition sc_A and $sc_{\bar{A}}$ are computable, and we can define

$$\mathbf{1} - sc_{\bar{A}}(x) = \begin{cases} 0 & x \in \bar{A} \\ \uparrow & \text{otherwise} \end{cases}$$

that is computable. This means that $\exists e_0, e_1 \in \mathbb{N}$ such that

$$\varphi_{e_0} = sc_A \quad \varphi_{e_1} = \mathbf{1} - sc_{\bar{A}}$$

therefore we can “combine two machines” and wait until one of the two terminates. Since either $x \in A$ or $x \in \bar{A}$, then the process will terminate for sure. We can build the characteristic function of A as

$$\begin{aligned} \chi_A(x) = & (\mu \omega. |S(e_0, x, (\omega)_1, (\omega)_2 \wedge S(e_1, x, (\omega)_1, (\omega)_2)) - 1|)_1 \\ & (\mu \omega. |\chi_{S(e_0, x, (\omega)_1, (\omega)_2 \wedge S(e_1, x, (\omega)_1, (\omega)_2))} - 1|)_1 \end{aligned}$$

which is computable, therefore A is recursive.

Exercise 7.20. State and prove Rice's theorem (without using the second recursion theorem).

Theorem

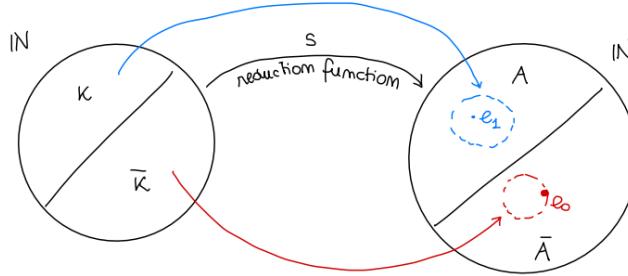
Let $A \subseteq \mathbb{N}$ be a set, $A \neq \emptyset, A \neq \mathbb{N}$. If A is saturated, then A is not recursive.

Proof

We start from the halting problem, making it reducible to A . So:

$K \leq_m A$ (since K is not recursive $\rightarrow A$ is not recursive)

To remember, this happens with reduction “behind the scenes”:



Let e_0 be an index s.t. $\phi_{e_0}(x) \uparrow \forall x$ (program for the function always undefined)

1) Assume $e_0 \notin A$

Let $e_1 \in A$ (it exists since $A \neq \emptyset$)

define

$$\begin{aligned} g(x, y) &= \begin{cases} \phi_{e_1}(y), & \text{if } x \in K \\ \phi_{e_0}(y), & \text{if } x \in \bar{K} \end{cases} \\ &= \begin{cases} \phi_{e_1}(y), & \text{if } x \in K \quad [\phi_x(x) \downarrow] \\ \uparrow, & \text{if } x \in \bar{K} \quad [\phi_x(x) \uparrow] \end{cases} \\ &= \phi_{e_1}(y) * \mathbf{1}(\phi_x(x)) \\ &\quad \uparrow \quad \begin{cases} \downarrow & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases} \\ &= \phi_{e_1}(y) * \mathbf{1}(\Psi_U(x, x)) \end{aligned}$$

computable.

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall x, y$:

$$\phi_{s(x)}(y) = g(x, y) = \begin{cases} \phi_{e_1}(y), & \text{if } x \in K \\ \phi_{e_0}(y), & \text{if } x \in \bar{K} \end{cases}$$

s is the reduction function for $K \leq_m A$

* $x \in K \Rightarrow s(x) \in A$

if $x \in K$ then $\phi_{s(x)}(y) = g(x, y) = \phi_{e_1}(y) \quad \forall y$

i.e. $\phi_{s(x)} = \phi_{e_1}$ since $e_1 \in A$ and A saturated $\rightarrow s(x) \in A$

* $x \notin K \Rightarrow s(x) \notin A$

if $x \notin K$ then $\phi_{s(x)}(y) = g(x, y) = \phi_{e_0}(y) \forall y$

i.e. $\phi_{s(x)} = \phi_{e_0}$ since $e_0 \notin A$ and A saturated $\rightarrow s(x) \notin A$

Hence, as expected by our construction, s is the reduction function and since K is not recursive, we deduce A not recursive either.

2) if instead $e_0 \in A$

$$e_0 \notin \overline{A}$$

\overline{A} saturated (since A is saturated)

$$\overline{A} \neq \emptyset \text{ (since } A \neq \emptyset\text{)}$$

$$\overline{A} \neq \mathbb{N} \text{ (since } A \neq \mathbb{N}\text{)}$$

\rightarrow by (1) applied to \overline{A} we deduce \overline{A} not recursive $\rightarrow A$ not recursive

Exercise 7.21. Define what it means for a set $A \subseteq \mathbb{N}$ to be saturated and prove that K is not saturated.

Definition

A subset $A \subseteq \mathbb{N}$ is saturated (or extensional) if $\forall m, n \in \mathbb{N}$

$$\text{if } m \in A \text{ and } \phi_m = \phi_n \text{ then } n \in A$$

(in words: given two programs, if the first program is in the set of programs satisfying the property and two programs are computing the same thing, then also the second program satisfies the property. This means that if one program with a certain property is in the set, all programs computing the same function must also be in the set.).

\Updownarrow

A saturated if $A = \{n \mid \phi_n \text{ satisfies a property of functions}\} = \{n \mid \phi_n \in A\}$

where $A \subseteq F$
 property of functions set of all functions

* $K = \{n \mid \phi_n(n) \downarrow\}$ (this is the halting problem, checking if it terminates over the program code)

Formally, I should find $m, n \in \mathbb{N}$

$$\begin{array}{ll} m \in K & \phi_m(m) \downarrow \\ m \notin K & \phi_n(n) \uparrow \end{array} \quad \text{and } \phi_m = \phi_n$$

(they have different values, but they are computing the same function).

If we were able to show there is a program $m \in \mathbb{N}$ s.t.

$$\phi_m(x) = \begin{cases} 1, & \text{if } x = m \\ \uparrow, & \text{otherwise} \end{cases}$$

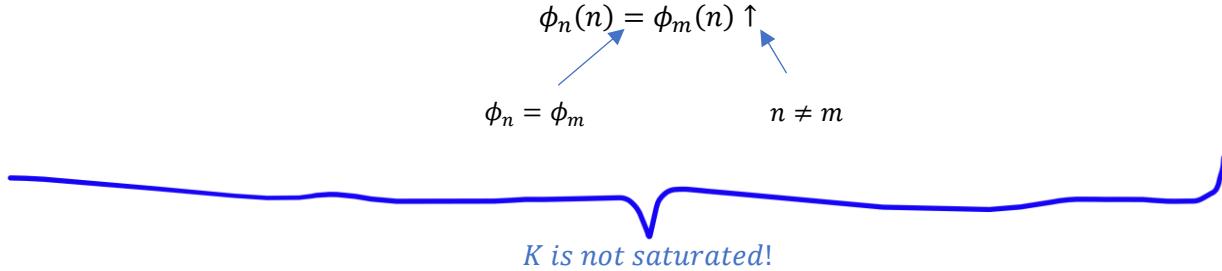


we can conclude:

1) $m \in K \quad \phi_m(m) \downarrow$

2) for a computable function there are infinitely many programs hence there is $n \neq m$ s.t. $\phi_m = \phi_n$

3) $n \notin K$



Exercise 7.22. Let $\mathcal{A} \subseteq \mathcal{C}$ be a set of functions computable and let $f \in \mathcal{A}$ such that for any function over $\theta \subseteq f$ is worth $\theta \notin \mathcal{A}$. Prove that $A = \{x \in \mathbb{N} \mid \varphi_x \in \mathcal{A}\}$ is not r.e.

If A is not r.e. then consider the identity function id for which it is defined for all natural numbers and consider as a finite subfunction $\theta = \mathbf{1}$ (constant 1), which might not be defined for all situations.

For example, more formally consider:

$$sc_{\bar{K}}(x) = \begin{cases} 1, & x \in \bar{K} \\ \uparrow, & \text{otherwise} \end{cases}$$

which is not r.e. and a finite subfunction like the previously defined is finite but $\notin \mathcal{A}$

Exercise

Let $A \subseteq \mathbb{N}$ be a set and let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a computable function. Is it true if A is r.e. then $f^{-1}(A) = \{x \in \mathbb{N} \mid f(x) \in A\}$ is r.e.? And if A is recursive, then A^{-1} is recursive?

Solution

Given A is a computable function, then we know $f(x) \in A$ s.t. $f(x) \downarrow$ and there is a semicharacteristic function able to compute it, for example able to compute $sc_A(f(x)) = 1$ where sc_A is the semicharacteristic function and it exists, giving A as r.e. Then, $sc_{f^{-1}(A)} = sc_A(f(x))$ computable since it is the composition of computable functions, so $f^{-1}(A)$ is r.e.

For the second part, we're arguing that if the first set is recursive, a set maintaining the same property on the image is also recursive. This does not happen however; consider the semicharacteristic function of the halting set defined as $sc_K = \{x \in K \mid f(x) \downarrow\}$, so $sc_K^{-1} = \{x \mid sc_K(x) \downarrow\} = K$ not recursive, considering N is recursive.

Exercise (16-09-2020)

Consider $A, B \subseteq \mathbb{N}$. Define the reduction notion for $A \leq_m B$. Consider set $S_4 = \{4 * n \mid n \in \mathbb{N}\}$, so the set of multiples of 4. Show that A is recursive iff $A \leq_m S_4$.

Solution

Recalling the definition for the reduction:

Given $A, B \subseteq \mathbb{N}$, we say the problem $x \in A$ reduces to " $x \in B$ " (or simply that A reduces to B) if there is a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$

$$x \in A \text{ iff } f(x) \in B$$

Set A is recursive, consider we can consider $g(x, y) = y$ if $x \in S_4$ and $g(x, y) = sc_{S_4}(x) * y$. By the smn-theorem, there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y)$ and this can be shown to be the reduction function of such problem:

- if $x \in S_4$ then $g(x, y) = \phi_{s(x)}(y) = y$, $\forall y \in \mathbb{N}$ then $E_{s(x)} = \mathbb{N}$ and $4 * \mathbb{N} \in E_{s(x)}$. So, $s(x) \in S_4$
- if $x \notin S_4$ then $g(x, y) = \phi_{s(x)}(y) \uparrow$, $\forall y \in \mathbb{N}$. In this case $E_{s(x)} = \emptyset$ and $4 * n \notin E_{s(x)}$, so $s(x) \notin S_4$

Exercise (15-05-2020)

Let $A, B \subseteq \mathbb{N}$ s.t. \overline{A} is finite and $B \neq \emptyset, \mathbb{N}$. Show $A \leq_m B$

Solution

Given $A, B \subseteq \mathbb{N}$, we say the problem $x \in A$ reduces to " $x \in B$ " (or simply that A reduces to B) if there is a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$

$$x \in A \text{ iff } f(x) \in B$$

Define:

$$g(x, y) = \begin{cases} b_1, & x \in A \\ x, & x \notin A \end{cases}$$

with $b_1 \in B$ and $b_2 \notin B$ (which exist for sure, given $B \neq \emptyset, B \neq \mathbb{N}$). Given \overline{A} is finite, it means \overline{A} is recursive, so $X_{\overline{A}}$ computable.

We can then define $g(x, y) = b_0 * \chi_{\overline{A}}(x) + b_1 * \overline{sg}(\chi_{\overline{A}}(x))$ and this is computable. By the smn-theorem, there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y)$ and this can be shown to be the reduction function of such problem:

- $x \in A \Rightarrow \phi_{s(x)}(y) = b_1 \forall y \rightarrow s(x) \in B$
- $x \notin A \Rightarrow \phi_{s(x)}(y) = b_0 \forall y \rightarrow s(x) \notin B$

Exercise (2022-01-19-solved)

- a. Provide the definition of reducibility, i.e., given sets $A, B \subseteq \mathbb{N}$ define what it means that $A \leq_m B$.
- b. Show that if A is not recursive and $A \leq_m B$ then B is not recursive.
- c. Show that if A is recursive then $A \leq_m \{1\}$.

a. Given two sets $A, B \subseteq \mathbb{N}$, we say $A \leq_m B$ if there exists a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}, x \in A \text{ iff } f(x) \in B$

b. If A is not recursive, then B is not recursive. Infact, if we consider $f: \mathbb{N} \rightarrow \mathbb{N}$ as the reduction function, we can define $\chi_A(x)$ as the characteristic function of A , which in this case is equal to $\chi_B(x)$. This is computable but consider a non-computable $f(x)$ s.t. $\chi_A = \chi_B \circ f$. Given f is not computable since B is not recursive, the whole thing is not recursive, otherwise also A would be recursive. Hence, B is not recursive.

c. To show this, if A is recursive, there exists a characteristic function χ_A s.t.

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & \text{otherwise} \end{cases}$$

Since A is recursive, the function is total and computable and the reduction function for $A \leq_m \{1\}$ holds since $x \in A$ iff $\chi_A(x) = 1$ iff $\chi_A(x) \in \{1\}$

Exercise (2019-01-24)

Given two sets $A, B \subseteq \mathbb{N}$, define the reduction $A \leq_m B$ and show $A \leq_m B$ and A is not recursive, then B is not recursive. Can a set $A \subseteq \mathbb{N}$ be such that $A \leq_m \overline{A}$? Show an example or show the non-existence of such set.

Solution

Recalling the definition for the reduction given [here](#):

Given $A, B \subseteq \mathbb{N}$, we say the problem $x \in A$ reduces to " $x \in B$ " (or simply that A reduces to B) if there is a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$

$$x \in A \text{ iff } f(x) \in B$$

If A is not recursive, B is not either. Suppose for the sake of contradiction A is recursive; then, there would be a total computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x g_A(x) = 1$ if $x \in A$, $g_A(x) = 0$ otherwise –Assuming B is recursive, the same holds, specifically $g_B(y) = g_A(y)$.

In this case, if $f(x) \in B$, then $g_B(f(x)) = 1$ and so $g_A(x) = 1$ otherwise $g_A(f(x)) = 0$ and so $g_A(x) = 0$. However, since A is not recursive, B can not be recursive either.

There is not set such that $A \leq_m \overline{A}$ because that would imply a set and its complement share basically the same elements.

In fact:

- if $x \in A$, then $f(x) \in \overline{A}$, but \overline{A} contains exactly the elements not in A , so $f(x) \notin A$ and the reduction does not hold
- if $x \notin A$, then $f(x) \in \overline{A}$, but \overline{A} contains exactly the elements not in A , so $f(x) \notin A$ and the reduction does not hold again

Exercise (2016-07-01)

Let A be a recursive set and let $f_1, f_2: \mathbb{N} \rightarrow \mathbb{N}$ be computable functions. Show that $f: \mathbb{N} \rightarrow \mathbb{N}$ defined as:

$$f(x) = \begin{cases} f_1(x), & x \in A \\ f_2(x), & x \notin A \end{cases}$$

Does the result still hold if we weaken the assumptions and assume A r.e.? Explain how the proof fits, if so, or provide a counterexample, if not.

Solution

Giving A is a recursive set, we have that $f(x)$ is defined and computable.

Considering $f_1 = \phi_{e_1}$ and $f_2 = \phi_{e_2}$ and we have:

$$f(x) = \mu w. \left(\left(S(e_1, x, w_1, w_2) \wedge \left(sg(X_A(w)) \right) \right) \vee \left(S(e_2, x, w_1, w_2) \vee \left(\overline{sg}(X_A(w)) \right) \right) \right)_1$$

which is computable. If A is r.e., then χ_A is not computable, for example considering $f_1 = 1$ and $f_2 = 0$ we have:

$$f(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$

which is not computable.

Exercise (2016-07-01)

Show that a set A is r.e. iff there exists a function $f: \mathbb{N} \rightarrow \mathbb{N}$ computable such that $A = \text{img}(f) = \{f(x): x \in \mathbb{N}\}$.

Solution

If a set is r.e., the semicharacteristic function $sc_A(x)$ is defined. We will define a simple function by composition, which is computable itself. For example, define $g(x) = sc_A(x) * x$.

Conversely, if there is a computable function with the specified properties, the set is r.e. Looking at the properties, we have the program is defined on itself, which means it uses its own index as computation (fixed point). In simpler terms, just consider e as the index of the program and $f = \phi_e$ as the computation over said program and define $sc_A(x) = \mathbf{1}(\mu w. (S(e, (w)_1, x, (w)_2)))$

Exercise (2021-09-07.solved)

Let $A, B \subseteq \mathbb{N}$. Define the reduction $A \leq_m B$. Show that $A \subseteq N$ is r.e. iff $A \leq_m K$.

Solution

Let $A, B \subseteq \mathbb{N}$. We write $A \leq_m K$ if $\exists f: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall x \in \mathbb{N}, x \in A \text{ iff } f(x) \in B$.

Set K is not recursive but is r.e. So, if $A \leq_m K$, A is r.e and we can write its semicharacteristic function as $sc_A = sc_K \circ f$ which is computable by composition.

On the converse implication, if A is r.e. there exists a total computable semicharacteristic function sc_A and consider the function of two arguments $g(x, y)$ defined as $sc_A(x)$ which is computable. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y) = sc_A(x)$.

It's easy to see s is a reduction function of A

- if $x \in A$, then $\phi_{s(x)}(y) = g(x, y) = sc_A(x) = 1 \forall y \in \mathbb{N}$, so $s(x) \in W_{s(x)} = N$ and $s(x) \in K$
- if $x \notin A$, then $\phi_{s(x)}(y) = g(x, y) = sc_A(x) = \uparrow \forall y \in \mathbb{N}$, so $s(x) \in W_{s(x)} = \emptyset$ and $s(x) \notin K$

Exercise (16-05-2020)

Let $A, B \subseteq \mathbb{N}$ and consider \bar{A} is finite and $B \neq \emptyset, \mathbb{N}$. Show $A \leq_m B$.

Solution

Recall the reduction definition; there exists a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$

$$x \in A \Leftrightarrow f(x) \in B$$

If we know \bar{A} is finite, there is a total computable function which describes it

$$\chi_{\bar{A}}(x) = \begin{cases} 1, & x \in \bar{A} \\ \uparrow, & \text{otherwise} \end{cases}$$

Given B is finite, there exists a function which is able to express the conditions of it:

$$g(x, y) = \begin{cases} 1, & x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}, g(x, y) = \phi_{s(x)}(y)$.

This is a correct reduction function:

- if $x \in B, \phi_{s(x)}(y) = 1$, so $E_{s(x)} = 1 \neq \mathbb{N} \neq \emptyset$ and so $s \in A$
- if $x \notin B, \phi_{s(x)}(y) = \uparrow$, so $E_{s(x)} = \mathbb{N}, W_{s(x)} \neq \emptyset$ and so $s \notin A$

Exercise (15-07-2020)

Let $A, B \subseteq \mathbb{N}$. Define reduction for $A \leq_m B$. Is it true that if A is recursive and B is finite, not empty then $A \leq_m B$? Show it or give a counterexample. And without finiteness hypothesis for B ? In case in general it doesn't hold with B infinite and give a condition which allows to restore the property

Solution

Recall the reduction definition; there exists a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}$

$$x \in A \Leftrightarrow f(x) \in B$$

If A is recursive, we can have χ_A computable. If we consider it as such, we have:

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

If you consider a recursive set, you can have say A be the set of all even numbers and B containing only the number 2. In this case:

$$\chi_A(x) = \begin{cases} 1, & \text{if } \frac{x}{2} \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

and

$$\chi_B(x) = \begin{cases} 1, & \text{if } x = 2 \\ \uparrow, & \text{otherwise} \end{cases}$$

Since A contains infinitely many elements and B contains only number 2, we can't map back a function satisfying all properties of A .

If B is infinite, consider say the set of prime numbers Pr , which is infinite.

$$\chi_B(x) = \begin{cases} 1, & \text{if } x \in Pr \\ \uparrow, & \text{otherwise} \end{cases}$$

and you have A recursive:

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

We can't possibly map all elements inside of the first set, given there are infinitely many non-prime numbers in the second set.

If B is r.e. then A is expressed finitely and recursive, so we can restore the property.

22.8 CHARACTERIZATION OF SETS

Note: Know exactly how to prove recursiveness, particularly Rice-Shapiro exact definition, reduction from K and \bar{K} and Rice's theorem (both definition and proof which is used practically here). First see if set is saturated then try to prove if it is r.e. and from this one move to recursive check.

Exercise 8.1. Study the recursiveness of the set $A = \{x \in \mathbb{N} : |W_x| \geq 2\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

The set is saturated (because it contains a non-trivial property), so $\{A = x | \phi_x \in \mathcal{A}\} s.t. \mathcal{A} = \{f \in \mathcal{C} | |\text{dom}(f)| \geq 2\}$. We can write here a semicharacteristic function for this set, given it would be possible for sure to find a value greater than 2 inside the domain, so we write:

$$sc_A(x) = 1(\mu w(H(x, x+2, y)) = 1(\mu w(H(x, x+2, (w)_2))$$

By Rice's theorem, this set is not recursive: considering $e_0 \in \mathbb{N}, e_1 \in \emptyset$, we see $e_0 \in A, e_1 \notin A$. Given the set is saturated, it is also not recursive by the theorem.

Given these observations, the complement of this set is also not r.e. and not recursive (otherwise both would be)

Exercise 8.2. Study the recursiveness of the set $A = \{x \in \mathbb{N} : x \in W_x \cap E_x\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

We prove that $K \leq_m A$ and define:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$.

We then define the semicharacteristic function of \bar{A} like:

$$\chi_A(x) = \mathbf{1}(\mu w.H(x, x, (w)_2) \wedge S(x, (w)_1, x, (w)_2))$$

We then conclude A is r.e. while \bar{A} is not.

As solved by an old tutor mentioned in beginning of this chapter:

Exercise 8.2. Study the recursiveness of the set $A = \{x \in \mathbb{N} : x \in W_x \cap E_x\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable. **NOT SATURATED**

$$K = \{x \in \mathbb{N} \mid \varphi_x(x) \downarrow\} \quad K \leq_m A$$

$$g(x, y) = \begin{cases} y & \boxed{x \in K} \\ \uparrow & \text{otherwise} \end{cases} = y \cdot \underbrace{sc_K(x)}_{\text{COMPUTABLE}} \quad K \text{ R.E.}$$

SMN THEOREM

$$\Rightarrow \exists s: \mathbb{N} \rightarrow \mathbb{N} \text{ TOTAL COMPUTABLE } s \uparrow \quad \varphi_{s(x)}(y) = g(x, y)$$

$$\forall x \in \mathbb{N} \quad x \in K \Leftrightarrow s(x) \in A$$

$$\textcircled{1} \quad x \in K \Rightarrow \varphi_{s(x)}(y) = y \quad \forall y \in \mathbb{N} \Rightarrow \varphi_{s(x)}(\underline{s(x)}) = \underline{s(x)}$$

\downarrow

$$s(x) \in W_{s(x)} \wedge s(x) \in E_{s(x)}$$

$$s(x) \in N_{s(x)} \cap E_-$$

$$\begin{aligned}
 \textcircled{II} \quad x \notin K &\Rightarrow s(x) \notin A \\
 &\Downarrow \phi_{s(x)}(y) \uparrow \forall y \in \mathbb{N} \Rightarrow s(x) \notin W_{s(x)} \Rightarrow s(x) \notin A
 \end{aligned}$$

so $K \leq_m A \Rightarrow A \text{ NOT RECURSIVE}$

$$\begin{aligned}
 sc_A(x) &= \mathbf{1}(\mu(y,t).H(x,y,t) \wedge S(x,y,x,t)) \\
 &= \mathbf{1}(\mu w. H(x,x,(w)_2) \wedge S(x,(w)_1,x,(w)_2)) \text{ COMPUTABLE} \\
 &\Rightarrow A \text{ R.E.} \quad B \text{ RECURSIVE} \Leftrightarrow \frac{B}{B} \text{ R.E.} \\
 \Rightarrow \bar{A} \text{ IS NOT R.E.} &
 \end{aligned}$$

Exercise 8.3. Study the recursiveness of the set

$$B = \{x \mid x \in W_x \cup E_x\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

We will try to prove the recursiveness of such set showing the reduction from the halting set K , specifically showing $K \leq_m B$.

If we can write a total computable function, by the smn-theorem, there will also be a semicharacteristic function showing B is computable and \bar{B} is not. If we look at the properties of B , we see its domain and its image have common values, hence if we write a function having a value in K , it will easily hold for the main problem condition.

We can then write a function of two parameters, considering we want to accommodate the smn-theorem structure:

$$g(x,y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x,y) = \phi_{s(x)}(y)$ and we see s is the desired reduction function. We can then write a semicharacteristic function for B as follows, using H (function halting in t steps) and the S function, which represents the “compute in t steps”, giving in this case x , after a number of steps, in this case after $(w)_2$ composed component steps:

$$sc_B(x) = \mathbf{1}(\mu w. (H(x,x,(w)_2) \vee S(x,(w)_1,x,(w)_2)))$$

We conclude this way B is r.e., while \bar{B} is not, given \bar{K} is not.

Exercise 8.4. Study the recursiveness of the set $A = \{x \in \mathbb{N} : W_x \subseteq \mathbb{P}\}$, where \mathbb{P} is the set of even numbers, i.e. establish whether A and \bar{A} are recursive/recursively enumerable.

We will try to show it is r.e., showing $K \leq_m A$ as we did until now and we consider:

$$g(x,y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

which is computable and by the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$ and we see s is the desired reduction function. This infact shows that we will get $\phi_{s(x)}(y) = g(x, y) = 1$ and the domain is over the naturals $W_{s(x)} = \mathbb{N}$.

This shows the function is computable, but we are not inside the set of even numbers, considering we halt always on an odd number, so we get $W_{s(x)} \not\subseteq P$ and so $s(x) \notin A$, so $s(x) \in \bar{A}$.

When $x \notin K$ we have that $\phi_{s(x)}(y) = g(x, y) = \uparrow, \forall y$ and so $W_{s(x)} = \emptyset, W_{s(x)} \subseteq P$ and so $(x) \in A$, so $s(x) \notin \bar{A}$. This holds, given \bar{A} is r.e., given we can write the following semicharacteristic function:

$$sc_{\bar{A}}(x) = \mathbf{1}(\mu w. (H(x, x, (w)_2,) \vee S(x, 2(w)_1 + 1, (w)_2))$$

or even:

$$sc_{\bar{A}}(x) = \mu w. H(x, 2(w)_1 + 1, (w)_2)$$

(we can use directly this one given we are already in the stopping case, and we can see only this one). Therefore, A is not r.e.

Exercise 8.5. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \exists y, z \in \mathbb{N}. z > 1 \wedge x = y^z\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

The set A is r.e. given we can write:

$$sc_A(x) = \mathbf{1}(\mu w. (S(x, y, z, t) \wedge (z > 1) \wedge (x = y^z))) = \mathbf{1}(\mu w. S(x, (w)_1, (w)_2, (w)_3) \wedge sg((w)_2 - 1) \wedge sg|x - (w)_2^{(w)_3}|)$$

The set is also not recursive, considering:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

Thanks to smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ which we show to be the correct reduction function:

- if $x \in K$, then $\phi_{s(x)}(y) = g(x, y) = 1$ for each $y \in \mathbb{N}$. So, $\phi_{s(x)}(s(x)) = s(x)^z$ for some $z \in \mathbb{N}$ and $\phi_{s(x)}(s(x)) \downarrow$ thus $s(x) \in A$
- if $x \notin K$, then $\phi_{s(x)}(y) = g(x, y) \uparrow$ for each $y \in \mathbb{N}$. Therefore, $W_{s(x)} = \emptyset$ and so $s(x) \notin A$

So, the converse of this set is also not r.e. and not recursive.

Exercise 8.6. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \phi_x(y) = y \text{ for infinitely many } y\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

As solved by an old tutor mentioned in beginning of this chapter:

Exercise 8.22. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \varphi_x(y) \stackrel{\text{def}}{\sim} y \text{ for infinitely many } y\}$, that is, say if A and \bar{A} are recursive/recursive enumerable.

$A = \{x \in \mathbb{N} : \varphi_x \in \mathcal{A}\}$

$\mathcal{A} = \{f \in \mathcal{L} : f(y) = y \text{ for infinite } y\}$

$\text{id} \in \mathcal{A} \quad \text{id}(y) = y \quad \forall y \in \mathbb{N}$

$\forall \theta \subseteq \text{id} \quad \theta \text{ FINITE} \quad \theta \notin \mathcal{A} \quad \theta(y) = y \text{ only for finite } y$

$\Rightarrow \mathcal{A} \text{ is not RE.}$

$\bar{A} \quad \text{id} \in \bar{A}$

$\underline{\phi \in \bar{A}} \quad \phi(y) \uparrow \forall y \in \mathbb{N} \quad \boxed{\phi(y) = y \text{ for infinite } y} \quad \text{NO}$

$\bar{A} \text{ is not RE.}$

Exercise 8.7. Study the recursiveness of the set $A = \{x \in \mathbb{N} : W_x \subseteq E_x\}$, i.e., establish if A and \bar{A} are recursive/recursive enumerable.

Let's understand if the set is saturated, since $A = \{x \mid \varphi_x \in \mathcal{A}\}$, where \mathcal{A} is the set of computable functions, while A is the set, that $A = \{f \mid \text{dom}(f) \subseteq \text{img}(f)\}$ and it clearly is.

Let's use Rice-Shapiro to conjecture that

- A is not r.e., since $\text{id} \notin A, \text{dom}(\text{id}) \subseteq \text{img}(\text{id})$ and there exists a finite subfunction $\theta, \theta \subseteq f \vee \theta \in \mathcal{A}$. Consider for instance $\emptyset \in \mathcal{A}, f(x) = x \forall x, f \notin B, \emptyset \subseteq f$
- \bar{A} is not r.e., since $\text{id} \notin \mathcal{A}$ but it admits \emptyset as a finite subfunction and $\emptyset \in \mathcal{A}$

From Baldan 2021/2022 ending lessons:

$$(8.7) \quad A = \{x \mid W_x \subseteq E_x\}$$

$\uparrow \quad \uparrow$

* A is saturated

$$A = \{x \mid \varphi_x \in \mathcal{A}\} \quad \mathcal{A} = \{f \mid \text{dom}(f) \subseteq \text{cod}(f)\}$$

* A is not re.

$$\underline{\text{I} \notin \mathcal{A}} \quad \text{dom}(\text{I}) = \mathbb{N} \quad \not\models \quad \{\text{I}\} = \text{cod}(\text{I})$$

$$\underline{\theta = \emptyset \subseteq \text{I}} \quad \text{dom}(\theta) = \emptyset \quad \subseteq \quad \emptyset = \text{cod}(\theta) \quad \Rightarrow \underline{\theta \in \mathcal{A}}$$

$\Rightarrow A$ is not re. (by Rice-Shapiro)

* \bar{A} is not r.e.

$$\underline{\text{pred}(x) = x - 1}$$

$$\text{dom}(\text{pred}) = \mathbb{N} \subseteq \text{cod}(\text{pred}) = \mathbb{N}$$

$$\hookrightarrow \text{pred} \in \mathcal{A} \Rightarrow \underline{\text{pred} \notin \bar{\mathcal{A}}}$$

$$\underline{\vartheta} = \begin{cases} 0 & x \leq 1 \\ \uparrow & \text{otherwise} \end{cases}$$

$$= \begin{cases} x-1 & x \leq 1 \\ \uparrow & \text{otherwise} \end{cases}$$

$$\underline{\vartheta \in \text{pred}}$$

finite

$$\text{dom}(\vartheta) = \{0, 1\} \not\models \{0\} = \text{cod}(\text{pred})$$

$$\underline{\vartheta \notin \mathcal{A}} \Rightarrow \underline{\vartheta \in \bar{\mathcal{A}}}$$

$\Rightarrow \bar{A}$ is not r.e.

Hence A, \bar{A} are neither r.e.

□

we could have considered

$$f(x) = \begin{cases} 1 & x=0 \\ 0 & x=1 \\ \uparrow & \text{otherwise} \end{cases}$$

$$\vartheta(x) = \begin{cases} 1 & x=0 \\ \uparrow & \text{otherwise} \end{cases}$$

$$f \notin \bar{\mathcal{A}} \quad \vartheta \in f \quad \vartheta \in \bar{\mathcal{A}}$$

$g(x, y)$ computable

by smm theorem

\exists total computable

$s: \mathbb{N} \rightarrow \mathbb{N}$ such that

corollary

$$\varphi_{s(x)}(y) = g(x, y)$$

Exercise 8.8. Study the recursiveness of the set $A = \{x \in \mathbb{N} : |W_x| > |E_x|\}$, i.e. establish whether A and \bar{A} are recursive/recursively enumerable.

The set A is saturated, given $A = \{x \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid \text{dom}(f) > \text{cod}(f)\}$.

By Rice-Shapiro theorem, we deduce:

- A is not r.e.

$$\exists f \notin A : f(x) = x - 1 \notin A, \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A}$$

$$\theta(x) = \begin{cases} x - 1, & x \leq 1 \\ \uparrow, & \text{otherwise} \end{cases}$$

- \bar{A} is not r.e. ($\bar{A} = \{x \in \mathbb{N} \mid |W_x| \leq |E_x|\}$)

$$\exists f \notin \bar{A} : \mathbf{1} \notin \bar{A}, \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A}$$

$$\theta(x) = \begin{cases} 1, & x = 1 \\ \uparrow, & \text{otherwise} \end{cases}$$

Exercise 8.9. Study the recursiveness of the set $A = \{x \in \mathbb{N} \mid \varphi_x(y) = x * y \text{ per some } y\}$, that is to say if A e \bar{A} are recursive/recursively enumerable.

This appears as a computable function (given it is the composition of computable functions, using the product). We can characterize its semicharacteristic function as

$$sc_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & x \notin A \end{cases} = \mathbf{1}(\mu w. S(x, (w)_1 * x, (w)_2 * y, (w)_2))$$

which is computable, so A is r.e.

We check if it's also recursive and to do so we test it with a reduction from the halting set:

$$K \leq_m A \quad (x \in K \text{ iff } s(x) \in A \text{ iff } \exists y \in W_{s(x)}, \exists k \in \mathbb{N} \text{ s.t. } y = k * s(x))$$

We consider a function of two arguments:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & x \notin K \end{cases} = \Psi_U(sc_K(x, x))$$

which is computable (given both halting set and the function with usage of smn-theorem work on the same index, that's why the universal function is used).

Thanks to smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ which we show to be the correct reduction function:

- if $x \in K$, then $\phi_{s(x)}(y) = g(x, y) = 1$ for each $y \in \mathbb{N}$. So, $\phi_{s(x)}(1) = 1 = s(x) * 1$ thus $s(x) \in A$
- if $x \notin K$, then $\phi_{s(x)}(y) = g(x, y) \uparrow$ for each $y \in \mathbb{N}$. Therefore, there is no such y such that $\phi_{s(x)}(y) = x * y$. Thus, $s(x) \notin A$

Since A is r.e. but not recursive, then \bar{A} is not recursive and also not r.e.

Exercise 8.10. Study the recursiveness of the set $A = \{x \in \mathbb{N} \mid |W_x \cap E_x| = 1\}$, i.e., establish if A e \bar{A} are recursive/recursively enumerable.

The set is saturated, given $A = \{x \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid |\text{dom}(f) \cap \text{cod}(f)| = 1\}$.

By Rice-Shapiro theorem, we deduce:

- A is not r.e.

$$\exists f \notin \mathcal{A} : f(x) = \text{id} \notin A, \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A}$$

$$\theta(x) = \begin{cases} 0, & \text{if } x = 0 \\ \uparrow, & \text{otherwise} \end{cases}$$

This way, the domain and codomain of normal set will be 0 (while the same – so, domain/codomain – will be 1 for the complement). Therefore, $\theta \in A$.

- \overline{A} is not r.e.

$$\exists f \notin \mathcal{A}: f(x) = \emptyset \notin \overline{A}, \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A}$$

Such function can be the always undefined function, for which it holds $\emptyset \in \mathcal{A}$, considering $\overline{\text{cod}(\theta)} = \overline{\text{dom}(\theta)} = \emptyset$. Therefore, $\theta \in \mathcal{A}$.

Since A is not r.e. then is not recursive, and also the same can be said for the complement.

Exercise 8.11. Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is *strictly increasing* when for each $y, z \in \text{dom}(f)$, if $y < z$ then $f(y) < f(z)$. Study the recursiveness of the set $A = \{x \mid \varphi_x \text{ sharply increasing}\}$, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Let's start reasoning from A , which considers an increasing ϕ_x , in a way such that $\forall y < z, f(y) < f(z)$. This means that for every two values we put inside our total function composed by the two ones, we get the same output, in such a way that $\forall y, \forall z, \phi_y = \phi_z$. Just a bit of formalization here, it means the set is saturated.

We might use Rice's theorem to prove this, only needing to define the function not inside the empty set and not over the naturals. So, let's use the set of computable functions A , in which we know ϕ_x is sharply increasing, in which we know $A = \{x \in \mathbb{N} \mid \phi_x \in a\}$.

By Rice-Shapiro, $\exists f, f \notin A, \exists \theta \subseteq f, \theta \text{ finite}, \theta \notin A$, we want to understand if the set is r.e. or not. A function which is always in A is the identity function, which is total.

Having $id \in A, \forall \theta \subseteq id, \theta \text{ finite}, \theta \notin A$.

We can also use the empty set function, which is $\emptyset \in A$, given it is defined and again, this is inside the identity, so $\emptyset \subseteq id \notin A$.

So, we showed A is not r.e., using the computable set A and properties alike.

If $A = \emptyset$ on some points, it means this is recursive. The complement, at this point, probably won't be. We can try to write a semicharacteristic function showing the complement is r.e.

We want the function x to stop with input z with t steps, so $S(x, z, x, t)$. The function will stop eventually, so $y \in W_x$, given x is total, so $\phi_x(y) \downarrow$, so $H(x, y, t)$. Again, w represents a tuple. This will end in t steps, giving in output a value such that the initial property is defined.

We check the y (so, the third element in tuple, given $w = (z, t, y)$, halting effectively in t steps over the search of y , given y, z elements, so its sum will be ordered the same way

$$sc_{\bar{A}}(x) = \mathbf{1}(\mu w. S(x, (w)_1, x, (w)_2) \vee ((x = f((w)_3 + (w)_4)) \wedge H(x, (w)_3, (w)_2)))$$

Hence, we will write:

$$sc_{\bar{A}}(x) = \mathbf{1}(\mu z. S(x, (w)_1, (w)_2 + (w)_3, t) \vee S(x, (w)_1 + (w)_2 + 1, (w)_3, (w)_4))$$

This is a combination of tuples, considering the sum will always be in the same order, so combining it this way will allow us to halt obtaining in the S function a value $(w)_4$ effectively.

Therefore, given on the complement Rice's theorem properties hold, \bar{A} is not recursive.

Exercise 8.12. Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is *almost total* if it is undefined on a finite set of points. Study the recursiveness of the set $A = \{x \mid \varphi_x \text{ almost total}\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

We start from understanding if the set is saturated or not and use Rice-Shapiro to guess the set is not r.e. We also need to see, for Rice's theorem, it is not empty or not the naturals. $A = \{f \in C \mid f \text{ almost total}\}$ using Rice-Shapiro and guess it is not r.e.

We show there is, using the identity function, which is total and r.e.

As solved by an old tutor mentioned in beginning of this chapter:

$$\begin{aligned}
 A &= \{x \in \mathbb{N} \mid \varphi_x \in \mathcal{A}\} \\
 \mathcal{A} &= \{f \in C \mid f \text{ almost total}\} \\
 \exists f \quad f \in \mathcal{B} \text{ and } \forall \theta \subseteq f \quad \theta \text{ FINITE } \theta \notin \mathcal{B} \quad] \Rightarrow \mathcal{B} \text{ is not r.e.} \\
 id \in \mathcal{A} \\
 \forall \theta \subseteq id \quad \theta \text{ FINITE} \Rightarrow \theta \notin \mathcal{A} \\
 \Rightarrow \exists f \quad f \notin \mathcal{B} \text{ and } \exists \theta \subseteq f \quad \theta \text{ FINITE } \theta \in \mathcal{B} \Rightarrow \mathcal{B} \\
 id \in \mathcal{A} \Rightarrow id \notin \bar{\mathcal{A}} \\
 \emptyset \in \bar{\mathcal{A}} \quad \emptyset(x) \uparrow \forall x \in \mathbb{N} \\
 \bar{\mathcal{A}} \text{ is not r.e.}
 \end{aligned}$$

Exercise 8.13. Study the recursiveness of the set $A = \{x \in \mathbb{N} : W_x \cap E_x = \emptyset\}$, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

The set is saturated, considering $A = \{x \in \mathbb{N} : \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid \text{dom}(f) \cap \text{cod}(f) = \emptyset\}$.

We now use Rice-Shapiro to argue both sets are not r.e.

- A not r.e.

In this case a subfunction clearly in this set is $\theta = \emptyset \in \mathcal{A}$, while $id \notin \mathcal{A}$ (because $\text{dom}(f) \cap \text{cod}(f) = \mathbb{N} \neq \emptyset$)

- \bar{A} not r.e.

Consider the complement of this set is $\bar{A} = \{x \in \mathbb{N} : W_x \cap E_x \neq \emptyset\}$.

In this case, simply reuse the functions of before: $id \in \mathcal{A}, \theta = \emptyset \notin \mathcal{A}$.

Given the two sets are not r.e. they are also not recursive.

Exercise 8.14. Given a set $X \subseteq \mathbb{N}$, we define $X + 1 = \{x + 1 : x \in X\}$. Study the recursiveness of the set $A = \{x \in \mathbb{N} : E_x = W_x + 1\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

The set is saturated, considering $A = \{x \in \phi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \mid \text{cod}(f) = \text{dom}(f) + 1\}$.

By Rice-Shapiro theorem, we deduce:

- A is not r.e.

$$\exists f \notin \mathcal{A} : f(x) = \text{id} \notin A, \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \mathcal{A}$$

In fact, define:

$$\theta(x) = \begin{cases} x + 1, & \text{if } x \geq 0 \\ \uparrow, & \text{otherwise} \end{cases}$$

Using the identity, we have $\text{id} \notin A$ given $\text{cod}(\text{id}) = \mathbb{N} \neq \text{dom}(\text{id}) + 1 = \mathbb{N} \setminus \{1\}$. The always undefined function respects the conditions of this case, so it's correct also using that (given it exists inside the domain and codomain)

- \bar{A} is not r.e.

$$\exists f \notin \bar{A} : f(x) = 1 \notin \bar{A}, \exists \theta \subseteq f, \theta \text{ finite s.t. } \theta \in \bar{A}$$

In this case, considering there are the complementary conditions, we have to use them to create a function using x and putting it inside its domain.

$$f(x) = \begin{cases} 1, & x \leq 1 \\ x, & \text{otherwise} \end{cases}$$

and

$$\theta(x) = \begin{cases} 1, & x = 1 \\ \uparrow, & \text{otherwise} \end{cases}$$

with $\text{cod}(f) = \text{dom}(f) + 1 = \mathbb{N} + 1$ and $\theta \subseteq f, \theta \in \mathcal{A}$ with $\text{cod}(\theta) \neq \text{dom}(\theta) + 1$.

Exercise 8.15. Let \mathbb{P} be the set of even numbers. Prove that indicated with $A = \{x \in \mathbb{N} : E_x = \mathbb{P}\}$, we have $\bar{K} \leq_m A$.

The set is saturated, because it's trivial to check if the codomain it's made of even numbers. We might argue it is not r.e. using the reduction from the complement of halting set (longer than Rice-Shapiro to do the same thing). Define:

$$g(x, y) = \begin{cases} 2y, & \text{if } \neg H(x, x, y) \\ 1, & \text{otherwise} \end{cases}$$

The function f can be defined as computable, considering we defined it is even when it does not halt, otherwise it is odd (important later).

So, f can be written as $f(x, y) = 2y * \overline{\text{sg}}(\chi_H(x, x, y)) + \chi_H(x, x, y)$. By the smn-theorem, $\phi_{s(x)}(y) \forall x, y \in \mathbb{N}$, which can be used as a reduction function.

- if $x \in \bar{K}$

In this case, the computation halts (so, $H(x, x, y) = 0$) and we get $\phi_{s(x)} = g(x, y) = 2y$ for all y and so the domain is made by even numbers and $s(x) \in A$

- if $x \notin \bar{K}$, the function halts giving 1 as number, so $\chi_H(x, x, y) = 1$ and $\phi_{s(x)}(y) = 1$, having 1 inside the domain but not as an even number. Hence, $s(x) \notin A$.

Exercise 8.16. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \varphi_x(x) \downarrow \wedge \varphi_x(x) < x + 1\}$, i.e., establish if A and \bar{A} are recursive/recursive enumerable.

In this problem, it's convenient to try to use a reduction from the halting set, so try to $K \leq_m A$. The reduction function can be shown to be:

$$g(x, y) = \begin{cases} 0, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

and is computable since it is defined by cases. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}$, this can be shown to be the correct reduction function from $K \leq_m A$.

- if $x \in K$, $g(x, y) = \phi_{s(x)}(y) = 0$. Therefore, $s(x) \in W_{s(x)} = \mathbb{N}$ and $s(x) \in A$
- if $x \notin K$, $g(x, y) = \phi_{s(x)}(y) = \uparrow$. Therefore, $s(x) \notin W_{s(x)} = \emptyset$ and $s(x) \notin A$

This set is not recursive, but it is r.e., since we can write its semicharacteristic function:

$$sc_A(x) = sg(x + 1 - \phi_x(x))$$

Since A not r.e. nor recursive, also \bar{A} is not r.e.

Exercise 8.17. Study the recursion of the set $A = \{x \in \mathbb{N} : x \in W_x \wedge \varphi_x(x) = x^2\}$, i.e., establish if A and \bar{A} are recursive/recursive enumerable.

Again, we try to use a reduction from the halting set, so try to $K \leq_m A$. The reduction function can be shown to be:

$$g(x, y) = \begin{cases} y^2, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

This is computable, since $g(x, y) = y^2 * sc_K(x)$. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$. This can be shown to be the correct reduction function, since:

- if $x \in K, \phi_{s(x)}(y) = g(x, y) = y^2 \forall y \in \mathbb{N}$. Therefore, $s(x) \in W_{s(x)} = \mathbb{N}$ and $s(x) \in A$
- if $x \notin K$ then $\phi_{s(x)}(y) = g(x, y) \uparrow \forall y \in \mathbb{N}$. Therefore, $s(x) \notin W_{s(x)} = \emptyset$ and $s(x) \notin A$

Since we can write the following semicharacteristic function, A is r.e. (we use the universal function since basically we are using as index x to find when the same function is stopping on x (ϕ_x) to retrieve x^2)

$$sc_A(x) = \mathbf{1}(\mu w. |x^2 - \phi_x(x)|) = \mathbf{1}(\mu w. |x^2 - \Psi_U(x, x)|)$$

which is computable. Since A is not recursive but r.e., \bar{A} is not r.e. and not recursive.

Exercise 8.18. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \exists k \in \mathbb{N}. \varphi_x(x+3k) \uparrow\}$, i.e., establish if A and \bar{A} are recursive/recursive enumerable.

Given the function does not terminate, one can argue a reduction from the complement of the halting set, so \bar{K} .

$(\bar{K} \leq_m A)$. The reduction function can be shown to be:

$$f(x, y) = \begin{cases} 0, & \text{if } H(x, x, y), \text{ so if } x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = f(x, y)$. This is a correct reduction function, considering:

- if $x \in \bar{K}$ then $\phi_{s(x)}(y) = g(x, y) = 0 \forall y \in \mathbb{N}$, with $s(x) \in A$
- if $x \in \bar{K}$ then $\phi_{s(x)}(y) = g(x, y) = \uparrow \forall y \in \mathbb{N}$, with $s(x) \notin A$

Considering \bar{K} is not recursive nor r.e., A is not recursive or r.e. either

$(\bar{K} \leq_m \bar{A})$. The reduction function can be shown to be:

$$f(x, y) = \begin{cases} 0, & \text{if } \neg H(x, x, y) \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$. This is a correct reduction function, considering:

- if $x \in \bar{K}$ then $\phi_{s(x)}(y) = g(x, y) = 0 \forall y \in \mathbb{N}$, with $s(x) \in \bar{A}$
- if $x \in \bar{K}$ then $\phi_{s(x)}(y) = g(x, y) = \uparrow \forall y \in \mathbb{N}$, with $s(x) \notin \bar{A}$

Considering \bar{K} is not recursive nor r.e., \bar{A} is not recursive or r.e. either

Exercise 8.19. Study the recursiveness of the set $A = \{x \in \mathbb{N} : W_x = \overline{E_x}\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

The set A is saturated, since $A = \{x : \phi_x \in \mathcal{A}\}$, since $\mathcal{A} = \{f \in C : \text{dom}(f) = \overline{\text{cod}(f)}\}$. We can use Rice-Shapiro's theorem to prove the following:

- A not r.e.

$$\exists f. C, f \in \mathcal{A} \wedge \forall \theta \subseteq f \text{ finite}, \theta \in \mathcal{A}$$

With the specified requirements, we find:

$$f(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

This way, $\theta \subseteq f, \theta \in A, \text{dom}(f) = \{1\} \neq \overline{\text{cod}(f)} = \{N \setminus 1\}$.

- \bar{A} not r.e.

$$\exists f. C, f \in \bar{A} \wedge \forall \theta \subseteq f \text{ finite}, \theta \in \bar{A}$$

If we have the previous function definition, we have $\theta \notin \mathcal{A}$. Moreover, consider the always undefined function $\theta \in \mathcal{A}$. Since $\theta \subseteq f, \theta \text{ finite}, \theta \in \bar{A}$ since $\text{dom}(\emptyset) = \overline{\text{cod}(\emptyset)} = \emptyset$

Exercise 8.20. Study the recursiveness of the set

$$B = \{\pi(x, y) \mid P_x(x) \downarrow \text{ in less than } y \text{ steps}\},$$

i.e., establish whether B and \bar{B} are recursive/recursively enumerable.

This is an interesting exercise, similar to 8.51

Set B can be defined as $B = \{x \in K \wedge H(x, x, y - 1)\}$. B is r.e. given we can express it like (using the pair encoding instead of (w) – because we use the latter only because we miss π , this is proven by theory):

$$sc_B(x) = sc_K(\pi_1(z)) * sc_H(\pi_1(z), \pi_1(z), \pi_2(z) - 1)$$

Set B is recursive, considering B can be defined as:

$$\chi_B = \begin{cases} y, & P_x(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases} = \overline{s\bar{g}}(|f(x) - y|)$$

which is computable. So, B is considered recursive.

Set \bar{B} is r.e., considering B is r.e. and also recursive, hence they are both recursive.

Exercise 8.21. Given $A = \{x \mid \varphi_x \text{ is total}\}$, show that $\bar{K} \leq_m A$.

$(\bar{K} \leq_m A)$. The reduction function can be shown to be:

$$f(x, y) = \begin{cases} y, & \text{if } \neg H(x, x, y) \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$.

This is a correct reduction function, considering:

- if $x \in \bar{K}$ then $\phi_{s(x)}(y) = g(x, y) = y \forall y \in \mathbb{N}$, with $\phi_{s(x)}(s(x)) = y \in \mathbb{N}$, so $s(x) \in A$
- if $x \in \bar{K}$ then $\phi_{s(x)}(y) = g(x, y) = \uparrow \forall y \in \mathbb{N}$, with $W_{s(x)} = \emptyset$, so $\phi_{s(x)}(s(x)) \uparrow$, so $s(x) \notin A$

Considering \bar{K} is not recursive nor r.e., \bar{A} is not recursive or r.e. either.

Exercise 8.22. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \varphi_x(y) = y \text{ for infinitely many } y\}$, that is, say if A and \bar{A} are recursive/recursive enumerable.

It's the same exercise as 8.6 and very similar to 8.49. Basically, no finite subfunction can be here, concluding easily by Rice-Shapiro.

As solved by an old tutor mentioned in beginning of this chapter:

Exercise 8.22. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \varphi_x(y) = y \text{ for infinitely many } y\}$, that is, say if A and \bar{A} are recursive/recursive enumerable.

$A = \{\varphi_x \in \mathcal{A} : \varphi_x(y) = y \text{ for infinitely many } y\}$

$\mathcal{A} = \{f \in \mathcal{C} : f(y) = y \text{ only for finite } y\}$

$\text{id} \in \mathcal{A} \quad \text{id}(y) = y \quad \forall y \in \mathbb{N}$

$\forall \theta \subseteq \text{id} \quad \theta \text{ FINITE} \quad \theta \notin \mathcal{A} \quad \theta(y) = y \text{ only for finite } y$

$\Rightarrow \mathcal{A} \text{ is not RE.}$

$\bar{A} \quad \text{id} \in \bar{A}$

$\cancel{\emptyset} \in \bar{A} \quad \cancel{\emptyset(y) \uparrow \forall y \in \mathbb{N}} \quad \boxed{\cancel{\emptyset(y) = y \text{ for infinitely many } y}} \quad \text{No}$

$\bar{A} \text{ is not RE.}$

Exercise 8.23. Given a subset $X \subseteq \mathbb{N}$ define $F(X) = \{0\} \cup \{y, y+1 \mid y \in X\}$. Studying recursiveness of the set $A = \{x \in \mathbb{N} : W_x = F(E_x)\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is saturated, since $A = \{x : \varphi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \in \mathcal{C} : \text{dom}(f) = F(\text{code}(f))\}$.

Using Rice-Shapiro's theorem we prove that both A and \bar{A} are not r.e.:

- A is not r.e.

Consider the function

$$f(x) = \begin{cases} 0 & \text{if } x = 0, 1, 2 \\ \uparrow & \text{otherwise} \end{cases}$$

We have $f \notin \mathcal{A}$, since $\text{dom}(f) = \{0, 1, 2\}$ and $\text{cod}(f) = \{0\}$. Thus $F(\text{cod}(f)) = \{0, 1\} \neq \text{dom}(f)$.

Moreover consider

$$\theta(x) = \begin{cases} 0 & \text{if } x = 0, 1 \\ \uparrow & \text{otherwise} \end{cases}$$

Clearly $\theta \subseteq f$. In addition $\text{dom}(\theta) = \{0, 1\}$ and $\text{cod}(\theta) = \{0\}$. Then $F(\text{cod}(\theta)) = \{0, 1\} = \text{dom}(\theta)$ and therefore $\theta \in \mathcal{A}$. By Rice-Shapiro's theorem, we conclude that A is not r.e.

- \bar{A} is not r.e.

Note that if θ is the function defined in the previous case, $\theta \notin \bar{A}$, but the function always undefined $\emptyset \in \bar{A}$, since $\text{dom}(\emptyset) = \text{cod}(\emptyset) = \emptyset$ and therefore $F(\text{cod}(\emptyset)) = \{0\} \neq \text{dom}(\emptyset)$. Thus, summing up $\theta \notin \bar{A}$, but it admits a finite subfunction, i.e., the function always undefined $\emptyset \in \bar{A}$. By Rice-Shapiro's theorem, we conclude that \bar{A} is not r.e.

Exercise 8.24. Study the recursiveness of the set

$$B = \{x \mid k \cdot (x + 1) \in W_x \cap E_x \text{ for each } k \in \mathbb{N}\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set A is not r.e., since $\bar{K} \leq_m A$. We prove it by considering

$$g(x, y) = \begin{cases} y & \neg H(x, x, y) \\ \uparrow & \text{otherwise} \end{cases}$$

This is computable and, by using the smn theorem, one can obtain the reduction function.

Also \bar{A} is not r.e., since $\bar{K} \leq_m \bar{A}$. The reduction function can be obtained by considering

$$g(x, y) = \begin{cases} y & x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

□

Exercise 8.25. Let \emptyset be the always undefined function. Study the recursiveness of the set $A = \{x \mid \varphi_x = \emptyset\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is non-recursive, by Rice's theorem, since it is saturated, not empty (the always undefined function is computable) and different from \mathbb{N} .

In addition \bar{A} is r.e., since

$$sc_{\bar{A}}(x) = \mathbf{1}(\mu w.H(x, (w)_1, (w)_2))$$

Thus A not r.e.

□

Exercise 8.26. Study the recursiveness of the set $A = \{x \mid \forall y. \text{ if } y + x \in W_x \text{ then } y \leq \varphi_x(y + x)\}$, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Solution: The set $\bar{A} = \{x \mid \exists y. y + x \in W_x \wedge y > \varphi_x(y + x)\}$ is not recursive, since $K \leq_m \bar{A}$. Consider the function

$$g(x, y) = \begin{cases} 0 & x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

It is computable and thus, using the smn theorem, we deduce the existence of a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$, such that $g(x, y) = \varphi_{s(x)}(y)$. The function s can be the reduction function.

In fact, if $x \in K$, we have that $\varphi_{s(x)}(y) = g(x, y) = 0$ for all y . Hence $\varphi_{s(x)}(s(x) + 1) = 0 < s(x) + 1$, and therefore $s(x) \in \bar{A}$. If, on the other hand, $x \notin K$, we have $s(x) \notin \bar{A}$.

The set \bar{A} is r.e., in fact

$$sc_{\bar{A}}(x) = \mu w.S(x, (w)_1 + x, (w)_1 + (w)_2 + 1, (w)_3)$$

where, intuitively, $(w)_1$ represents the value y we are looking for and the value of the function is required to be $(w)_1 + (w)_2 + 1 > (w)_1$.

Therefore, A is not r.e.

□

Exercise 8.27. Study the recursiveness of the set $A = \{x \mid \varphi_x(y + x) \downarrow \text{ for some } y \geq 0\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set $A = \{x \mid \varphi_x(y + x) \downarrow \text{ for some } y \geq 0\}$ is not recursive because $K \leq A$. In order to prove this fact, let us consider the function $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined, by

$$g(x, y) = \begin{cases} 1 & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

The function is computable since $g(x, y) = sc_K(x)$. Hence, by the smn-theorem, there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi_{s(x)}(y) = g(x, y)$ for all $x, y \in \mathbb{N}$. We next argue that s is a reduction function for $K \leq_m A$. In fact

- If $x \in K$ then $\varphi_{s(x)}(y) = g(x, y) = 1$ for all $y \in \mathbb{N}$. In particular, $\varphi_{s(x)}(0 + s(x)) \downarrow$. Hence $s(x) \in A$.
- If $x \notin K$ then $\varphi_{s(x)}(y) = g(x, y) \uparrow$ for all $y \in \mathbb{N}$. Hence $\varphi_{s(x)}(y + s(x)) \uparrow$ for all $y \in \mathbb{N}$. Hence $s(x) \notin A$.

The set A is r.e., since its semi-characteristic function

$$sc_A(x) = \mathbf{1}(\mu(y, t).H(x, x + y, t))$$

is computable.

Therefore, \bar{A} is not r.e. □

Exercise 8.28. Let $X \subseteq \mathbb{N}$ be finite, $X \neq \emptyset$ and define $A_X = \{x \in \mathbb{N} : W_x = E_x \cup X\}$. Study the recursiveness of A , i.e., say if A_X and \bar{A}_X are recursive/recursively enumerable.

Solution: The set A_A is saturated, since $A_X = \{x : \varphi_x \in \mathcal{A}\}$, where $\mathcal{A}_X = \{f \in \mathcal{C} : \text{dom}(f) = \text{cod}(f) \cup X\}$.

Using Rice-Shapiro's theorem we prove that A and \bar{A} are both not r.e.:

- A is not r.e.

Let $x \in X$ and $y \notin X$ and consider the function

$$f(x) = \begin{cases} x & \text{if } x \in X \cup \{y\} \\ \uparrow & \text{otherwise} \end{cases}$$

We have $f \notin \mathcal{A}$, since $\text{dom}(f) = X \cup \{y\} \neq X = X \cup \{x\} = X \cup \text{cod}(f)$. Moreover, if we consider

$$\theta(x) = \begin{cases} x & \text{if } x \in X \\ \uparrow & \text{otherwise} \end{cases}$$

clearly $\theta \leq f$. Note that $\text{dom}(\theta) = X = X \cup \{x\} = X \cup \text{cod}(\theta)$ and therefore $\theta \in \mathcal{A}$. Thus, by Rice-Shapiro's theorem we conclude that A is not r.e.

- \bar{A} is not r.e.

Note that if θ is the function defined above, $\theta \notin \bar{\mathcal{A}}$, but the function always undefined $\emptyset \in \bar{\mathcal{A}}$, since $\text{dom}(\emptyset) = \emptyset \neq X = \text{cod}(\emptyset) \cup X$. Thus, summing up $\theta \notin \bar{\mathcal{A}}$, but it admits a finite subfunction, i.e., the function always undefined $\emptyset \in \bar{\mathcal{A}}$. By Rice-Shapiro's theorem, we conclude that \bar{A} is not r.e.

Exercise 8.29. Let $A = \{x \in \mathbb{N} : W_x \cap E_x \neq \emptyset\}$. Study the recursiveness of A , i.e., say if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is saturated, since $A = \{x : \varphi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \in \mathcal{C} : \text{dom}(f) \cap \text{cod}(f) \neq \emptyset\}$. It is not empty (since $\mathbf{1} \in \mathcal{A}$) and it is not the entire \mathbb{N} (since $\emptyset \notin \mathcal{A}$), thus by Rice's theorem A is not recursive. Furthermore, A is r.e. since

$$\begin{aligned} sc_A(x) &= \mathbf{1}(\mu(y, z, t).H(x, y, t) \wedge S(x, z, y, t)) \\ &= \mathbf{1}(\mu w.H(x, (w)_1, (w)_3) \wedge S(x, (w)_2, (w)_1, (w)_3)) \end{aligned}$$

Therefore \bar{A} is not r.e. □

Exercise 8.30. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \forall k \in \mathbb{N}. x + k \in W_x\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: We prove that $\bar{K} \leq_m A$, and thus A is not r.e. In order to obtain the reduction function, consider the following computable function

$$g(x, y) = \begin{cases} y & \text{if } \neg H(x, x, y) \\ \uparrow & \text{otherwise} \end{cases}$$

and then use the smn theorem.

Also $K \leq_m A$. In order to obtain the reduction function, consider the following computable function

$$g(x, y) = \begin{cases} 1 & \text{if } x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

and again, use the smn theorem. Therefore $\bar{K} \leq \bar{A}$ and therefore \bar{A} not r.e. □

Exercise 8.31. A partial function $f : \mathbb{N} \rightarrow \mathbb{N}$ is called injective when for each $x, y \in \text{dom}(f)$, if $f(x) = f(y)$ then $x = y$. Study the recursiveness of the set $A = \{x : \varphi_x \text{ injective}\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is clearly saturated, since $A = \{x \mid \varphi_x \in \mathcal{A}\}$, where \mathcal{A} is the set of injective functions. Since $\emptyset \in \mathcal{A}$ and $\mathbf{1} \notin \mathcal{A}$, by Rice's theorem the sets A and \bar{A} are not recursive. Also \bar{A} is r.e., since

$$sc_A(x) = \mathbf{1}(\mu w. (S(x, (w)_1, (w)_3, (w)_4) \wedge S(x, (w)_2, (w)_3, (w)_4) \wedge (w)_1 \neq (w)_2)).$$

Exercise 8.32. Study the recursiveness of the set $A = \{x \in \mathbb{N} : \exists y \in E_x. \exists z \in W_x. x = y * z\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: We show that $K \leq A$, thus A is not recursive. In fact, define

$$g(x, y) = \begin{cases} 1 & \text{if } x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

The function $g(x, y)$ is computable, since

$$g(x, y) =_K (x)$$

So by the SMN theorem, there exists a total computable such function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that for each $x, y \in \mathbb{N}$

$$\varphi_{s(x)}(y) = g(x, y)$$

The function s is a reduction function of K to A . Indeed, if $x \in K$, then $\varphi_{s(x)}(y) = y$ for each y , and thus we can take $s(x) \in W_{s(x)}$ and $1 \in E_{s(x)}$ such that $s(x) = s(x) * 1$. Thus $s(x) \in A$. Otherwise, $\varphi_{s(x)} = \emptyset$ and thus it is easy to conclude $s(x) \notin A$.

Furthermore, A is r.e., since

$$sc_A(x) = \mathbf{1}(\mu w. S(x, (w)_1, (w)_2, (w)_3) \wedge (w)_1 * (w)_2 = x)$$

Therefore \bar{A} is not r.e. □

(Also present in the exam 2016-07-01)

Exercise 8.33. Study the recursiveness of the set $A = \{x \in \mathbb{N} : x \in W_x \wedge \varphi_x(x) > x\}$, i.e., establish if A and \bar{A} are recursive/recursive enumerable.

We show that $K \leq_m A$ and A is not recursive, defining a function like:

$$g(x, y) = \begin{cases} y + 1, & \text{if } x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

The function $g(x, y)$ is computable since $g(x, y) = (y + 1) * sc_K(x)$. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}$, then $\phi_{s(x)}(y) = g(x, y)$.

This can be shown the correct reduction function, since:

- if $x \in K$, we have $\phi_{s(x)} = g(x, y) = y + 1 \forall y \in \mathbb{N}$ and so $s(x) \in W_{s(x)} = \mathbb{N}$ and $\phi_{s(x)}(s(x)) = s(x) + 1 > s(x)$ and $s(x) \in A$.
- if $x \notin K$ then $\phi_{s(x)}(y) = g(x, y) \uparrow \forall y \in \mathbb{N}$ and therefore $s(x) \notin W_{s(x)} = \emptyset$ then $s(x) \notin A$.

The set A can be expressed by a semicharacteristic function like the following:

$$sc_A(x) = \mathbf{1}(\mu w. (x + 1) - \phi_x(x)) = \mathbf{1}(\mu w. (x + 1) - \Psi_U(x, x))$$

hence it is r.e. Since this holds, \bar{A} is not r.e. or recursive either.

Exercise 8.34. Let f be a total computable function such that $img(f) = \{f(x) : x \in \mathbb{N}\}$ is infinite. Study the recursiveness of the set

$$A = \{x \mid \exists y \in W_x. x < f(y)\},$$

i.e., establish if A e \bar{A} are recursive/recursively enumerable.

Set A is not recursive, given there is a reduction from the halting set, so $K \leq_m A$. Consider:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

which is computable. Therefore, by the smn theorem there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$. This is shown to be a correct reduction function:

- if $x \in K$, $\phi_{s(x)}(y) = g(x, y) = 1 \forall y \in \mathbb{N}$. Hence, $W_{s(x)} = \mathbb{N}$ and $f(W_{s(x)}) = f(\mathbb{N}) = img(f)$. So, there exists $z \in f(W_{s(x)})$ s.t. $x < z$ so $y \in W_{s(x)}$ s.t. $s(x) < f(y)$. So, $s(x) \in A$
- if $x \notin K$, $\phi_{s(x)}(y) = g(x, y) = \uparrow \forall y \in \mathbb{N}$. Hence, $W_{s(x)} = \emptyset$ and so there is no y s.t. $s(x) < f(y)$. So, $s(x) \notin A$

Exercise 8.34. Let f be a total computable function such that $\text{img}(f) = \{f(x) : x \in \mathbb{N}\}$ is infinite. Study the recursiveness of the set

$$A = \{x \mid \exists y \in W_x. x < f(y)\},$$

i.e., establish if A e \bar{A} are recursive/recursively enumerable.

Solution: The set A is not recursive since $K \leq_m A$. In fact, consider the function

$$g(x, y) = \begin{cases} 1 & x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

It is computable. Therefore for the smn theorem there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$, such that $g(x, y) = \varphi_{s(x)}(y)$. The function s is a reduction function.

In fact, if $x \in K$, we have that $\varphi_{s(x)}(y) = g(x, y) = 1$ for each y . Hence $W_{s(x)} = \mathbb{N}$, and therefore $f(W_{s(x)}) = f(\mathbb{N}) = \text{img}(f)$, which is infinite for hypothesis. Thus there certainly exists $z \in f(W_{s(x)})$ such that $x < z$, i.e., there exists $y \in W_{s(x)}$ such that $s(x) < f(y)$. Therefore $s(x) \in A$.

If, on the other hand, $x \notin K$, we have that $\varphi_{s(x)}(y) = g(x, y) = \uparrow$ for each y . Hence $W_{s(x)} = \emptyset$, and therefore, certainly there is no $y \in W_{s(x)}$ such that $s(x) < f(y)$. Thus $s(x) \notin A$.

The set A is r.e., in fact

$$sc_A(x) = \mu w. (H(x, (w)_1, (w)_2) \wedge x < f((w)_1))$$

Therefore, \bar{A} is not r.e.

□

Exercise 8.35. Study the recursiveness of the set $B = \{x \in \mathbb{N} : x \in E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Set B is not recursive, considering:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

which is computable, given $g(x, y) = f(x) * sc_K(x)$. Therefore, by the smn-theorem, there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$.

This is a correct reduction function:

- if $x \in K$, $\phi_{s(x)}(y) = g(x, y) = 1 \forall y$ and so $W_{s(x)} = \mathbb{N}$ and $s(x)_{s(x)} = \mathbb{N}$, so $x \in B$
- if $x \notin K$, $\phi_{s(x)}(y) = g(x, y) = \uparrow \forall y$ and so $W_{s(x)} = \emptyset$ and so $x \notin B$

Set B is also r.e., given:

$$sc_A(x) = \mu w. (H(x, (w)_1, (w)_2))$$

Therefore, \bar{B} is not r.e. (also not recursive, otherwise both sets would be recursive).

Exercise 8.36. Study the recursiveness of the set $V = \{x \in \mathbb{N} : W_x \text{ infinity}\}$, i.e., establish if V and \bar{V} are recursive/recursively enumerable.

As solved by an old tutor mentioned in beginning of this chapter:

Exercise 8.36. Study the recursiveness of the set $V = \{x \in \mathbb{N} : W_x \text{ infinity}\}$, i.e., establish if V and \bar{V} are recursive/recursively enumerable.

$V = \{x \in \mathbb{N} : \varphi_x \in \mathcal{U}\}$ $\not\models \emptyset$
 $\mathcal{U} = \{f \in \mathcal{L} : \text{dom}(f) \text{ INFINITE}\}$ $\not\models \mathbb{N}$

RICE-SHAPED

$\text{id} \in \mathcal{U}$ $\text{dom}(\text{id}) = \mathbb{N}$
 $\forall \theta \subseteq \text{id}$ FINITE $\theta \notin \mathcal{U}$ $\text{dom}(\theta)$ FINITE $\Rightarrow V \text{ not R.E.}$

$\text{id} \notin \bar{\mathcal{U}}$
 $\emptyset \subseteq \text{id}$ FINITE $\emptyset \in \bar{\mathcal{U}}$ $\text{dom}(\emptyset) = \emptyset$ FINITE $\Rightarrow \bar{V} \text{ not R.E.}$

Exercise 8.37. Study the recursiveness of the set $V = \{x \in \mathbb{N} : \exists y \in W_x. \exists k \in \mathbb{N}. y = k \cdot x\}$, i.e., establish if V and \bar{V} are recursive/recursively enumerable.

(Basically, it's adapting exercise 8.32 here)

We try a reduction from the halting problem, so we argue $K \leq_m V$:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

The function $g(x, y)$ is computable, given $g(x, y) = sc_K(y)$ and by the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}. \phi_{s(x)}(y) = g(x, y)$.

This is a correct reduction function, because:

- if $x \in K, \phi_{s(x)}(y) = y \forall y \in \mathbb{N}$, so we can take $s(x) \in W_{s(x)}$ and $1 \in E_{s(x)} = \mathbb{N}$ and so $s(x) \in V$
- if $x \notin K, \phi_{s(x)}(y) \uparrow \forall y \in \mathbb{N}$, so we can take $s(x) \notin W_{s(x)} = \emptyset$ and so $s(x) \notin V$

A is r.e. since we can write:

$$sc_A(x) = \mathbf{1}(\mu w. H(x, y, k, x) \wedge (y * k = x))$$

$$sc_A(x) = \mathbf{1}(\mu w. \chi_H(x, (w)_1, (w)_2, (w)_3) \wedge ((w)_2 * (w)_3 = x))$$

Given A is r.e. then it is not recursive (one could use Rice's theorem here) and also \bar{A} is not recursive (otherwise both would be recursive) or r.e.

Exercise 8.38. Study the recursiveness of the set $V = \{x \in \mathbb{N} : |W_x| > 1\}$, i.e., establish if V and \overline{V} are recursive/recursive enumerable.

We argue V is saturated because $B = \{x \mid \phi_x \in \mathcal{B}\}$ where $\mathcal{B} = \{f \in \mathcal{C} \mid \text{dom}(f) > 1\}$

- A is not r.e.

$$f(x) = \begin{cases} 1, & x \leq 1 \\ x, & \text{otherwise} \end{cases}$$

$$\theta(x) = \begin{cases} 1, & x < 1 \\ x, & \text{otherwise} \end{cases}$$

We see $f \in A$ but $\theta \notin A$ and so A is not r.e.

- \overline{A} is not r.e.

The previously defined $\theta \in \overline{A}$ but consider $\emptyset \notin \overline{A}$, given $|W_x| = \emptyset \neq \mathbb{N}$ (where $n > 1$)

Given both sets are not r.e. they are also not recursive.

Exercise 8.39. Let P be the set of even numbers and Pr the set of prime numbers. Show that $P \leq_m Pr$ and $Pr \leq_m P$.

Following the definition:

$x \in P$ reduces to $x \in Pr$ if there exists a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}, x \in P \Leftrightarrow f(x) \in Pr$

Define a function of two arguments as follows:

$$g(x, y) = \begin{cases} \frac{y}{2} + x, & x \in P \\ \uparrow, & \text{otherwise} \end{cases} = qt(2, y) + x + \mu z. rm(2, y)$$

which is computable.

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$.

This is a correct reduction function, because:

- if $x \in P, \phi_{s(x)}(y) = \frac{y}{2} + x \forall y \in \mathbb{N}$, so we can take $s(x) \in W_{s(x)}$ s.t. $p \in Pr$ and so $\frac{p}{2} + x \in E_{s(x)}$
s.t. $\frac{p}{2} + x \in \mathbb{N}$ and $1 \in E_{s(x)} = \mathbb{N}$ and so $s(x) \in Pr$
- if $x \notin P, \phi_{s(x)}(y) \uparrow \forall y \in \mathbb{N}$, so we can take $s(x) \notin W_{s(x)} = \emptyset$ and so $s(x) \notin Pr$

Following the definition:

$x \in Pr$ reduces to $x \in P$ if there exists a total computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x \in \mathbb{N}, x \in Pr \Leftrightarrow f(x) \in P$

$$g(x, y) = \begin{cases} p, & x \in Pr \\ \uparrow, & \text{otherwise} \end{cases} = \mu z. |\overline{sg}(g(x, y) - p)|$$

which is computable.

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$.

This is a correct reduction function, because:

- if $x \in Pr$, $\phi_{s(x)}(y) = p \forall y \in \mathbb{N}$, so we can take $s(x) \in W_{s(x)}$ s.t. $p \in P$ and so $p \in E_{s(x)} = \mathbb{N}$ s.t. so $s(x) \in P$
- if $x \notin Pr$, $\phi_{s(x)}(y) \uparrow \forall y \in \mathbb{N}$, so we can take $s(x) \notin W_{s(x)} = \emptyset$ and so $s(x) \notin P$

Exercise 8.40. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a fixed total computable function. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid f(x) \in E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: Observe that B is r.e., in fact we can write its semi-characteristic function as follows:

$$sc_B(x) = \mathbf{1}(\mu w. (x, (w)_1, f(x), (w)_2))$$

Moreover B is not recursive since $K \leq_m B$. In order to obtain the reduction function consider

$$g(x, y) = \begin{cases} y & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

Hence \bar{B} is not r.e. □

ù

Exercise 8.41. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a fixed total computable function. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid \text{img}(f) \cap E_x \neq \emptyset\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable. Please note that $\text{img}(f) = \{f(x) \mid x \in \mathbb{N}\}$.

B is saturated given $B = \{x \mid \phi_x(x) \in \mathcal{B}\}$ so $\mathcal{B} = \{f \in \mathcal{C} \mid \text{img}(f) \cap \text{cod}(f) \neq \emptyset\}$. Using Rice-Shapiro, we argue that:

- B is not r.e.

Consider for example $\text{id} \in \mathcal{B}$ given $\mathcal{B} = \{f \mid \text{img}(f) \cap \text{cod}(f)\} = \mathbb{N} \neq \emptyset$ and both image and codomain are finite and consider no finite subfunction can be inside B

- \bar{B} is not r.e.

Consider \emptyset (always undefined function) which conversely is definitely in this set given $\mathcal{B} = \{f \mid \text{img}(f) \cap \text{cod}(f)\} = \emptyset$, which is exactly what we want (so $\emptyset \in \mathcal{B}$)

Given both sets are not r.e. they are both not recursive.

Exercise 8.42. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid E_x \not\models W_x\}$, i.e., establish if B e \bar{B} are recursive/recursively enumerable.

As solved by Baldan by Moodle 2020-2021.

If A is saturated, $A = \{x \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{g \mid \text{img}(f) \cap \text{img}(g) = \emptyset\}$

- A not r.e.

$f \notin \mathcal{A} \quad \text{img}(g) \cap \text{img}(f) | = \text{img}(f) \neq \emptyset$ because f is total

$\emptyset \subseteq f \quad \text{img}(f) \cap \text{img}(\emptyset) \Rightarrow \emptyset \in \mathcal{A}$

So, A is not recursive $\Rightarrow \bar{A}$ not recursive

- \bar{A} not r.e.

Let e s.t. $\phi_e = f$

$sc_A(x) = \text{look for } y \text{ common output of "x" and "e"}$

$\Rightarrow \text{look for } y, z_1 \text{ input for } x, z_2 \text{ input for } e \text{ s.t. } x \text{ on } z_1 \text{ produces } y \text{ and } e \text{ on } z_2 \text{ produces } y$

$$= \mu w. S(x, (w)_2, (w)_1, (w)_4) \wedge S(e, (w)_3, (w)_1, (w)_4)$$

where $(w)_1 = y, (w)_2 = z_1, (w)_3 = z_2, (w)_4 = t$

Exercise 8.43. Let $B = \{x \mid \forall m \in \mathbb{N}. m \cdot x \in W_x\}$. Study the recursiveness of the B set, that is to say if B and \bar{B} are recursive/recursively enumerable.

As solved by an old tutor:

Exercise 8.43. Let $B = \{x \mid \forall m \in \mathbb{N}. m \cdot x \in W_x\}$. Study the recursiveness of the B set, that is to say if B and \bar{B} are recursive/recursively enumerable.

$$\begin{aligned} \bar{K} \leq_m B & \quad P_x(x) \not\rightarrow \text{in } y \text{ or diverges} \\ \forall x \in \mathbb{N} \quad x \in \bar{K} \Leftrightarrow sc(x) \in B & \quad x \in \bar{K} \text{ is stronger than } \neg H(x, x, y) \\ f(x, y) = \begin{cases} 0 & x \in \bar{K} \rightarrow \neg H(x, x, y) \\ \uparrow & \text{otherwise} \end{cases} & \quad = \mu w. \neg H(x, x, y) \\ & \quad = \mu w. \chi_{\neg H(x, x, y)} \text{ COMPUABLE} \\ \text{SIN} & \Rightarrow \exists s: \mathbb{N} \rightarrow \mathbb{N} \text{ st } f(x, y) = \varphi_{sc(x)}(y) \\ \text{TOTAL} & \\ \text{COMP} & \end{aligned}$$

$$\begin{aligned} \bar{K} \leq_m \bar{B} & \quad B = \{x \mid \exists m \in \mathbb{N} \quad m \cdot x \notin W_x\} \subset \\ f(x, y) = \begin{cases} 0 & x \in \bar{K} \\ \uparrow & \text{otherwise} \end{cases} & \quad = \overline{\exists g}(sc_K(x)) \text{ COMP} \\ \Rightarrow \varphi_{sc(x)}(y) = f(x, y) & \quad (\exists x \in \mathbb{N}) \quad \stackrel{1}{\exists} \stackrel{2}{\exists} \stackrel{3}{\exists} \stackrel{4}{\exists} sc(x) \end{aligned}$$

$\textcircled{I} \quad x \in \bar{K} \Rightarrow \varphi_{sc(x)}(y) \uparrow \quad \forall y \in \mathbb{N} \Rightarrow \varphi_{sc(x)}(\exists s(x)) \uparrow \Rightarrow \exists s(x) \notin W_{sc(x)}$
 $\Rightarrow s(x) \in \bar{B}$

$\textcircled{II} \quad x \notin \bar{K} \Rightarrow x \in K \Rightarrow \varphi_{sc(x)}(y) = 0 \quad \forall y \in \mathbb{N} \Rightarrow W_{sc(x)} = \mathbb{N} \Rightarrow \nexists m \in \mathbb{N} \quad m \cdot s(x) \notin W_{sc(x)}$
 $\Rightarrow s(x) \notin \bar{B}$

$\Rightarrow \bar{K} \leq_m \bar{B} \Rightarrow \bar{B} \text{ NOT R.E.}$

$$\begin{aligned} \textcircled{I} \quad x \in \bar{K} \Rightarrow \varphi_{sc(x)}(y) = 0 \quad \forall y \in \mathbb{N} \Rightarrow \varphi_{sc(x)}(m \cdot s(x)) = 0 \quad \forall m \in \mathbb{N} \\ \downarrow \quad m \cdot s(x) \in W_{sc(x)} \quad \forall m \in \mathbb{N} \\ \Rightarrow s(x) \in B \end{aligned}$$

$\textcircled{II} \quad x \notin \bar{K} \Rightarrow \exists y_0 \in \mathbb{N} \text{ st } \varphi_{sc(x)}(y) \uparrow \quad \forall y \geq y_0 \Rightarrow \exists m \in \mathbb{N} \text{ st } \varphi_{sc(x)}(m \cdot s(x)) \uparrow$

$\Rightarrow \exists m \in \mathbb{N} \quad m \cdot s(x) \notin W_{sc(x)} \Rightarrow s(x) \notin B$

$\bar{K} \leq_m B \Rightarrow B \text{ IS NOT R.E.}$

Exercise 8.44. Given $A = \{x \mid \varphi_x \text{ is total}\}$, show that $\bar{K} \leq_m A$.

This is present inside 2012-06-20.

Define:

$$g(x, y) = \begin{cases} y, & \text{if } \neg H(x, x, y) \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$ and this is shown to be the correct reduction function:

- if $x \in \overline{K}$, $g(x, y) = \phi_{s(x)}(y) = y$ and $H(x, x, y)$ does not terminate. So, we have $s(x) \in W_{s(x)}$, $\phi_{s(x)}(s(x)) = s(x)$. Therefore, $s(x) \in A$
 - if $x \notin \overline{K}$, $g(x, y) = \phi_{s(x)}(y) = g(x, y) \uparrow \forall y \in N$. Therefore, $s(x) \notin W_{s(x)} = \emptyset$. So, $s(x) \notin A$.

Exercise 8.45. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid \exists y > x. y \in E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

As solved by an old tutor mentioned in beginning of this chapter:

Exercise 8.45. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid \exists y > x, y \in E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

$K \leq_m B$

$$g(x, y) = \begin{cases} y & \text{if } x \in K \\ \uparrow & \text{otherwise} \end{cases} = y \cdot s_{\bar{c}_K}(x) \quad \text{COMPUTABLE}$$

$\stackrel{\text{SMN}}{\Rightarrow} \exists s: \mathbb{N} \rightarrow \mathbb{N} \text{ TOTAL COMPUTABLE st } \varphi_{s(x)}(y) = g(x, y)$

$x \in K \Leftrightarrow s(x) \in B \quad \forall x \in \mathbb{N}$

(I) $x \in K \Rightarrow \varphi_{s(x)}(y) = y \quad \forall y \in \mathbb{N}$ $\boxed{\exists \hat{y} > s(x), y \in E_{s(x)}} ?$

 $\Rightarrow \varphi_{s(x)}(s(x)+1) = s(x)+1 \Rightarrow \underbrace{s(x)+1 \in E_{s(x)}}_{\text{!}} \Rightarrow s(x) \in B$

(II) $x \notin K \Rightarrow \varphi_{s(x)}(y) \uparrow \quad \forall y \in \mathbb{N} \Rightarrow \nexists y > s(x), y \in E_{s(x)} \Rightarrow s(x) \notin B$

$K \leq_m B \Rightarrow B \text{ irr not recursive}$

$$\begin{aligned}
 S_{C_B}(x) &= \perp\!\!\!\perp_{(y,z,t)} (y > x \wedge S(x, z, y, t))) \\
 &= \perp\!\!\!\perp_{(w)} ((w)_1 > x \wedge S(x, (w)_2, (w)_1, (w)_3))) \quad \text{COMPUTABLE} \\
 &\quad \underbrace{\hspace{10em}}_{\Rightarrow B \text{ R.E.}} \\
 &\quad \Rightarrow \overline{B} \text{ is not R.E.}
 \end{aligned}$$

Given the following ones are not different and are already solved (I solved them myself as exercise, but nothing new to note, I'll paste them here for the sake of completeness)

Exercise 8.46. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid \forall y > x. \exists y \in W_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: Observe that B is not r.e. since $\bar{K} \leq_m B$. In order to get the reduction function consider

$$g(x, y) = \begin{cases} y & \text{if } \neg H(x, x, y) \\ \uparrow & \text{otherwise} \end{cases}$$

Also $\bar{B} = \{x \mid \exists y > x. \exists y \notin W_x\}$ is not r.e. In order to reduce $\bar{K} \leq_m \bar{B}$, the reduction function can be constructed from:

$$g(x, y) = \begin{cases} \uparrow & \text{if } x \notin K \\ 1 & \text{otherwise} \end{cases} = sc_K(x)$$

□

Exercise 8.47. Study the recursiveness of the set $B = \{x \in N : 1 \leq |E_x| \leq 2\}$, i.e., establish if B e \bar{B} are recursive/recursively enumerable.

Solution: The set B is saturated, since it can be expressed as $B = \{x : \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \in \mathcal{C} : 1 \leq |\text{cod}(f)| \leq 2\}$.

Using Rice-Shapiro's theorem, we prove that B and \bar{B} are both not r.e.:

- B is not r.e.

Note that $\text{id} \notin \mathcal{B}$ but there is a finite function

$$\theta(x) = \begin{cases} 0 & \text{if } x = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

such that $\theta \subseteq \text{id}$ and $\theta \in \mathcal{B}$. Hence by Rice-Shapiro's theorem we conclude that B is not r.e.

- \bar{B} is not r.e.

Note that if θ is the function defined in the previous case, $\theta \notin \bar{\mathcal{B}}$, but the function always undefined $\emptyset \in \bar{\mathcal{B}}$. By Rice-Shapiro's theorem we conclude that \bar{B} is not r.e.

□

Exercise 8.48. Study the recursiveness of the set $A = \{x \in \mathbb{N} \mid \mathbb{P} \subseteq W_x\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is saturated since $A = \{x \mid \varphi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \mid \mathbb{P} \subseteq \text{dom}(f)\}$. We can use Rice-Shapiro's theorem to show that

- A is not r.e.

In fact $\text{id} \in \mathcal{A}$ since $\mathbb{P} \subseteq \text{dom}(\text{id}) = \mathbb{N}$ and no finite $\theta \subseteq \text{id}$ can be in \mathcal{A} , since functions in \mathcal{A} necessarily have an infinite domain.

- \bar{A} not r.e.

In fact, $\text{id} \notin \bar{\mathcal{A}}$, and $\emptyset \subseteq \text{id}, \emptyset \in \bar{\mathcal{A}}$.

□

Exercise 8.49. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid \varphi_x(y) = y^2 \text{ for infinite } y\}$, i.e., establish if B and \bar{B} are recursive/recursive enumerable.

Solution: We observe that B is saturated, since $B = \{x \mid \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \mid f(y) = y^2 \text{ for infinite } y\}$. Rice-Shapiro's theorem is used to deduce that both sets are not r.e.

- B is not r.e. because \mathcal{B} contains y^2 and none of its sub-functions finite (it does not contain any finite functions).
- \bar{B} is not r.e. since $\emptyset \in \bar{\mathcal{B}}$ and \emptyset admits as an extension $y^2 \notin \bar{\mathcal{B}}$.

□

Exercise 8.50. Given $X \subseteq \mathbb{N}$, indicate by $2X$ the set $2X = \{2x : x \in X\}$. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid 2W_x \subseteq E_x\}$, i.e., establish if B and \bar{B} are recursive/recursive enumerable.

Solution: Rice-Shapiro's theorem is used to prove that both sets are not r.e.:

- B is not r.e. because it contains \emptyset , but not all functions (e.g. it does not contain $\theta = \{(1, 4)\}$).
- \bar{B} is not r.e. since it contains θ , as defined above, but not $\theta' = \{(1, 4), (2, 2)\}$.

Exercise 8.51. Study the recursiveness of the set $B = \{x \in \mathbb{N} \mid W_x \supseteq Pr\}$, where $Pr \subseteq \mathbb{N}$ is the set of the prime numbers, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: We use Rice-Shapiro's theorem for proving that both sets are not r.e.:

- B is not r.e. because it does not contain any finite functions and it is not empty (e.g. $id \in \mathcal{B}$, but no finite subfunction of id can be in \mathcal{B}).
- \bar{B} is not r.e. since it contains \emptyset , but it does not include all functions (e.g. it does not contain id , of which \emptyset is a finite subfunction).

□

Exercise 8.52. Classify the following set from the point of view of recursiveness

$$B = \{\pi(x, y) \mid P_x \text{ stops on input } x \text{ in more than } y \text{ steps}\},$$

where $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$ is the pair encoding function, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is r.e., but not recursive. In fact

$$B = \{x : x \in K \wedge \neg H(x, x, y)\}$$

For proving that it is not recursive, note that $K \leq_m B$. In fact, $x \in K$ iff $\pi(x, 0) \in B$. Furthermore, B is r.e. since its semi-characteristic function is computable:

$$sc_B(z) = sc_K(\pi_1(z)) \cdot sc_{\neg H}(\pi_1(z), \pi_1(z), \pi_2(z))$$

Thus \bar{B} non-recursive. \square

Exercise 8.53. Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is symmetric in the interval $[0, 2k]$ if $dom(f) \supseteq [0, 2k]$ and for each $y \in [0, k]$ we have $f(y) = f(2k - y)$. Study the recursiveness of the set

$$A = \{x \in \mathbb{N} : \exists k > 0. \varphi_x \text{ symmetric in } [0, 2k]\},$$

i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is r.e. In fact:

$$sc_A(x) = \mathbf{1}(\mu h. \forall y \leq h+1 \varphi_x(y) = \varphi_x(2(h+1) - y))$$

It is not recursive by Rice's theorem. In fact, A is saturated. Moreover, if e_0 ed e_1 are indices for the functions \emptyset and $\mathbf{1}$, respectively, we have that $e_0 \notin A$ and $e_1 \in A$. Hence $A \neq \emptyset, \mathbb{N}$. \square

Exercise 8.54. Given $X \subseteq \mathbb{N}$ define $inc(X) = X \cup \{x + 1 : x \in X\}$. Classify the following set from the point of view of recursiveness $B = \{x \in \mathbb{N} : inc(W_x) = E_x\}$, i.e. say if B and \bar{B} are recursive/recursively enumerable.

Solution: We have that $B = \{f \mid inc(dom(f)) = cod(f)\}$, thus the set is saturated. Furthermore $\emptyset \in B$, but $\emptyset \subseteq \mathbf{1}$ and $\mathbf{1} \notin B$ since $\mathbb{N} = inc(dom(\mathbf{1})) \neq cod(\mathbf{1}) = \{1\}$. Hence, by Rice-Shapiro's theorem, the set B is not r.e.

The function $\theta = \{(0, 0)\} \in \bar{B}$, but $\theta \subseteq id \notin \bar{B}$, therefore, again by Rice-Shapiro's theorem, also \bar{B} is not r.e. \square

Exercise 8.55. Classify the following set from the point of view of recursiveness

$$B = \{x \mid \varphi_x(0) \uparrow \vee \varphi_x(0) = 0\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: Observe that B is saturated, the corresponding set of functions can be defined as $\mathcal{B} = \{f : f(0) \uparrow \vee f(0) = 0\}$. We have that $\mathbf{1} \notin \mathcal{B}$, while the finite subfunction $\emptyset \in \mathcal{B}$. Thus, by Rice-Shapiro's theorem, B is not r.e. Instead $\bar{B} = \{x : \varphi_x(0) \downarrow \wedge \varphi_x(0) \neq 0\}$ is r.e., since $sc_B(x) = \overline{sg}(\varphi_x(0))$ is computable. \square

Exercise 8.56. A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is said *increasing* when for each $x, y \in \text{dom}(f)$, if $x < y$ then $f(x) < f(y)$. Define $B = \{x \in \mathbb{N} : \varphi_x \text{ increasing}\}$ and show that $\bar{K} \leq_m B$.

Solution: One can mimic the proof of Rice-Shapiro's theorem and define

$$g(x, y) = \begin{cases} y & \text{if } \neg H(x, x, y) \\ 0 & \text{otherwise} \end{cases}$$

Thus, if $x \in \bar{K}$ then g , seen as a function of y , will be the identity, which is increasing. Otherwise there exists a number of steps y such that $H(x, x, y)$ and therefore from that point onward $g(x, y)$ is constantly equal to 0 and thus not increasing.

More precisely, observe that the function $g(x, y)$ is computable, since

$$g(x, y) = y \cdot \chi_{\neg H}(x, x, y)$$

Thus, by the SMN theorem, there exists a function $s : \mathbb{N} \rightarrow \mathbb{N}$ total and computable such that for each $x, y \in \mathbb{N}$

$$\varphi_{s(x)}(y) = g(x, y)$$

The function s is a reduction function of \bar{K} into B . In fact

- If $x \in \bar{K}$ then for every $y \in \mathbb{N}$ the predicate $H(x, x, y)$ is false. Therefore $\varphi_{s(x)}(y) = g(x, y) = y$ for all $y \in \mathbb{N}$. Hence $\varphi_{s(x)}$ is increasing and therefore $s(x) \in B$.
- If $x \notin \bar{K}$ then there exists a $y \in \mathbb{N}$ such that $H(x, x, y)$ holds true, and therefore also $H(x, x, y+1)$ holds. Thus $\varphi_{s(x)}(y) = 0 = \varphi_{s(x)}(y+1)$. Then $\varphi_{s(x)}$ is not increasing and therefore $s(x) \notin B$.

Alternatively, more simply, we can observe that the function always undefined is increasing and the constant 0 is not. So just define $g(x, y) = sc_K(x)$ (semi-characteristic function of the set K , which is known to be computable since K is r.e.) and then proceed as above. \square

Exercise 8.57. Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is k -bounded if $\forall x \in \text{dom}(f)$ we have $f(x) < k$. For each $k \in \mathbb{N}$, study the recursiveness of the set

$$A_k = \{x \in \mathbb{N} : \varphi_x \text{ } k\text{- bounded}\},$$

i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set \bar{A}_k is r.e. In fact:

$$sc_{\bar{A}_k}(x) = \mathbf{1}(\mu w.S(x, (w)_1, (w)_2 + k, (w)_3))$$

It is not recursive by Rice's theorem. In fact, \bar{A}_k is saturated. Moreover, if e_0 and e_1 are indices for the functions \emptyset and id , we have that $e_0 \notin \bar{A}_k$ and $e_1 \in \bar{A}_k$. Thus $\bar{A}_k \neq \emptyset, \mathbb{N}$ and we conclude that \bar{A}_k is not recursive. Therefore A_k is not r.e. \square

Exercise 8.58. Classify the following set from the point of view of recursiveness $B = \{x + y : x, y \in \mathbb{N} \wedge \varphi_x(y) \uparrow\}$, i.e., establish whether B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is recursive. In fact, let z_0 be the minimum index for the function always undefined. Then, for every $z \geq z_0$ we can express z as $z_0 + (z - z_0)$ and we have $\varphi_{z_0}(z - z_0) \uparrow$. Hence $z \in B$. Therefore, if we denote by $\theta = \chi_{B \cap [0, z_0 - 1]}$, the finite subfunction of the characteristic function restricted to the interval $[0, z_0 - 1]$, we have

$$\chi_B(z) = \begin{cases} \theta(z) & \text{if } z < z_0 \\ 1 & \text{otherwise} \end{cases}$$

Since θ and the constant 1 are computable, and the predicate $z < z_0$ is decidable, the characteristic function is computable. \square

Exercise 8.59. Let f be a total computable function. Classify the following set from the point of view of recursiveness $B_f = \{x \in \mathbb{N} : \varphi_x(y) = f(y) \text{ for infinitives } y\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: Rice-Shapiro's theorem is used for both sets

- B is not r.e. because it contains f and none of its finite subfunctions (since f is total, B does not contain any finite function)
- \bar{B} is not r.e. since $\emptyset \in \bar{B}$ and \emptyset admits $f \notin \bar{B}$ as an extension.

□

Exercise 8.60. Let f be a total computable function, different from the identity. Classify the following set from the point of view of recursiveness $B_f = \{x \in \mathbb{N} : \varphi_x = f \circ \varphi_x\}$, i.e., establish if B_f and \bar{B}_f are recursive/recursively enumerable.

Solution: Observe that B_f is saturated since it can be expressed as $B_f = \{x \mid \varphi_x \in \mathcal{B}_f\}$ where $\mathcal{B}_f = \{g \mid g = f \circ g\}$.

We can use Rice-Shapiro's theorem to show that B_f is not r.e. In fact the identity $id \notin \mathcal{B}_f$ since $id \neq f = f \circ id$. Moreover the function always undefined $\emptyset \in \mathcal{B}_f$ since $\emptyset = f \circ \emptyset$ and clearly $\emptyset \subseteq id$.

Moreover, the complement \bar{B}_f is r.e. In fact, let e be an index for f , i.e., such that $\varphi_e = f$. Then we have that $x \in \bar{B}_f$ iff there is an input z where $v = \varphi_x(z) \downarrow$ and $\varphi_e(v) \neq v$. Hence the semi-characteristic function of \bar{B}_f can be expressed as follows:

$$sc_{\bar{B}_f}(x) = \mu w. (S(x, (w)_1, (w)_2, (w)_3) \wedge S(e, (w)_2, (w)_4, (w)_3) \wedge (w)_2 \neq (w)_4)$$

□

Exercise 8.61. Study the recursiveness of the set $B = \{x \in \mathbb{N} : \exists k \in \mathbb{N}. k \cdot x \in W_x\}$, i.e. establish whether B and \bar{B} are recursive/recursively enumerable.

Solution: We show that $K \leq B$ and therefore B is not recursive. In fact, define

$$g(x, y) = \begin{cases} \varphi_x(y) & \text{if } x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

The function $g(x, y)$ is computable, since

$$g(x, y) = \psi_U(x, x)$$

Hence, by the SMN theorem, we have that there exists a function $s : \mathbb{N} \rightarrow \mathbb{N}$ total computable such that for each $x, y \in \mathbb{N}$

$$\varphi_{s(x)}(y) = g(x, y)$$

The function s is a reduction function of K to B .

Furthermore, B is r.e., since

$$sc_B(x) = \mathbf{1}(\mu w. H(x, (w)_1 \cdot x, (w)_2))$$

Therefore \bar{B} not r.e. □

Exercise 8.62. Classify from the point of view of recursiveness the set $B = \{x \in \mathbb{N} : \forall k \in \mathbb{N}. k+x \in W_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: We show that $\bar{K} \leq B$ and therefore B is not r.e. In fact, define

$$g(x, y) = \begin{cases} 0 & \text{if } \neg H(x, x, y) \\ \uparrow & \text{otherwise} \end{cases}$$

The function $g(x, y)$ is computable, since

$$g(x, y) = \mu z. \chi_H(x, x, y)$$

So by the SMN theorem, we have that there exists a function $s : \mathbb{N} \rightarrow \mathbb{N}$ total computable such that for each $x, y \in \mathbb{N}$

$$\varphi_{s(x)}(y) = g(x, y)$$

The function s reduces K to B .

Furthermore, \bar{B} not r.e., since $\bar{K} \leq \bar{B}$. In fact, define

$$g(x, y) = \begin{cases} 0 & x \in K \\ \uparrow & \text{otherwise} \end{cases}$$

and proceed as before. \square

Exercise 8.63. Classify from the point of view of recursiveness the set $V = \{x \in \mathbb{N} : E_x \text{ infinite}\}$, i.e., establish if V and \bar{V} are recursive/recursively enumerable.

Solution: The set V is saturated since $V = \{x \mid \varphi_x \in \mathcal{A}\}$, dove $\mathcal{A} = \{f \mid \text{cod}(f) \text{ infinite}\}$. Then we can use Rice-Shapiro's theorem:

- $\text{id} \in \mathcal{A}$, but no finite subfunction of id is in \mathcal{A} , hence A is not r.e.;
- $\emptyset \in \bar{\mathcal{A}}$, $\emptyset \subseteq \text{id}$, but $\text{id} \notin \bar{\mathcal{A}}$, hence \bar{A} is not r.e.

Exercise 8.64. Classify the following set from the point of view of recursiveness $B = \{x \in \mathbb{N} : x \in W_x \setminus \{0\}\}$, i.e. establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is r.e., since

$$sc_A(x) = \mathbf{1}(\mu w. (H(x, (w)_1, (w)_2) \wedge x \neq 0)).$$

and not recursive. In fact, $K \leq_m B$. In order to prove this fact consider

$$g(x, y) = \begin{cases} \varphi_x(x) & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

By the smn theorem, since the function is computable, we obtain $s : \mathbb{N} \rightarrow \mathbb{N}$, computable and total such that $\varphi_{s(x)}(y) = g(x, y)$. This is almost the reduction function, except for the fact that it might have value 0 for some input. However, it is sufficient to take an index $k \neq 0$ for the function φ_0 and consider:

$$s'(x) = \begin{cases} s(x) & \text{if } s(x) \neq 0 \\ k & \text{otherwise} \end{cases}$$

and we are done. \square

Exercise 8.65. Classify the following set from the point of view of recursiveness

$$A = \{x \mid W_x \setminus E_x \text{ infinite}\},$$

i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is saturated since $A = \{x \mid \varphi_x \in \mathcal{A}\}$ with $\mathcal{A} = \{f \mid \text{dom}(f) \setminus \text{cod}(f) \text{ infinite}\}$.

By Rice-Shapiro's theorem:

- A is not r.e., since $\mathbf{1} \in \mathcal{A}$, but no finite subfunction $\theta \subseteq \mathbf{1}$ can belong to \mathcal{A} . In fact $\text{dom}(\theta)$ is finite and therefore also $\text{dom}(\theta) \setminus \text{cod}(\theta)$ is finite. Therefore $\theta \notin \mathcal{A}$.
- \bar{A} is not r.e., since $\emptyset \in \bar{\mathcal{A}}$, $\mathbf{1} \notin \bar{\mathcal{A}}$ and $\emptyset \subseteq \mathbf{1}$.

□

Exercise 8.66. Classify the following set from the point of view of recursiveness $B = \{x \in N : |W_x \setminus E_x| \geq 2\}$, i.e., establish if B e \bar{B} are recursive/recursively enumerable.

Solution: The set B is saturated, since $B = \{x : \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \in \mathcal{C} : |\text{dom}(f) \setminus \text{cod}(f)| \geq 2\}$.

Using Rice-Shapiro's theorem we prove that B and \bar{B} are not r.e.:

- B not r.e.

Observe that $f(x) = x - 2 \notin \mathcal{B}$ ($\text{dom}(f) = \text{cod}(f) = \mathbb{N}$, thus $\text{dom}(f) \setminus \text{cod}(f) = \emptyset$) but there is a finite subfunction

$$\theta(x) = \begin{cases} 0 & \text{if } x \leq 2 \\ \uparrow & \text{otherwise} \end{cases}$$

such that $\theta \subseteq f$ and $\theta \in \mathcal{B}$. By Rice-Shapiro's theorem therefore we conclude that B is not r.e.

- \bar{B} not r.e.

Note that if θ is the function defined above, then $\theta \notin \bar{\mathcal{B}}$, but the function always undefined $\emptyset \in \bar{\mathcal{B}}$. By Rice-Shapiro's theorem therefore we conclude that \bar{B} is not r.e.

□

Exercise 8.67. Classify the following set from the point of view of recursiveness $B = \{x \in \mathbb{N} : \exists k \in \mathbb{N}. \forall y \geq k. \varphi_x(y) \downarrow\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is clearly saturated since it is the set of indexes of functions in $\mathcal{B} = \{f \in \mathcal{C} \mid \exists k \in \mathbb{N}. \forall y \geq k. f(y) \downarrow\}$.

We can conclude that B and \bar{B} are non-r.e. using Rice-Shapiro's theorem. In fact:

- B is not r.e., since $\text{id} \in \mathcal{B}$ but obviously no finite subfunction $\theta \subseteq \text{id}$ can belong to \mathcal{B} (which does not contain any finite function).
- \bar{B} is not r.e., since $\text{id} \notin \bar{\mathcal{B}}$, but there is a finite subfunction $\emptyset \subseteq \text{id}$ with $\emptyset \in \mathcal{B}$.

□

Exercise 8.68. Classify the following set from the point of view of recursiveness $B = \{x \in \mathbb{N} : x > 0 \wedge x/2 \notin E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: Observe that \bar{B} is r.e., in fact we can write its semi-characteristic function as follows:

$$sc_{\bar{B}}(x) = \mathbf{1}(\mu w. x = 0 \vee S(x, (w)_1, x/2, (w)_2))$$

Moreover \bar{B} is not recursive since $K \leq_m \bar{B}$. In order to get the reduction function consider

$$g(x, y) = \begin{cases} y & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

Then, by s.m.n theorem, we have that $g(x, y) = \varphi_{s(x)}(y)$ for some total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$. This is almost the reduction function. We need to be sure that when $x \notin K$ then $s(x)$, which is an index of the empty function, is not 0. This can be done by “modifying” function s . More precisely take any index $n_0 > 0$ such that $\varphi_{n_0} = \emptyset$ (there is such n_0 since \emptyset has infinitely many indices). Then define $s'(x) = s(x)$ if $s(x) \neq 0$ and $s'(x) = n_0$, otherwise. Then s' is still total and computable, and works as a reduction function.

Hence \bar{B} is not r.e. □

Exercise 8.69. Classify the following set from the point of view of recursiveness

$$B = \{x \in \mathbb{N} : \forall y \in W_x. \exists z \in W_x. (y < z) \wedge (\varphi_x(y) > \varphi_x(z))\},$$

i.e., establish if B and \bar{B} are recursive/recursively enumerable.

This is also present inside exam 2016-07-01.

Comments aside, set B is saturated, considering $B = \{x : y_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \in \mathcal{C} : \forall y \in \text{dom}(f). \exists z \in \text{dom}(f). (y < z) \wedge (f(y) > f(z))\}$. Using Rice-Shapiro we show that:

- B is not r.e.

Consider $id \notin B$ considering all values must be ordered and different from each other, but there exists a finite subfunction θ for which it holds $\theta \subseteq id, \theta \notin \mathcal{B}$. This subfunction is the always undefined one, so \emptyset . So, this set is not r.e.

- \bar{B} is not r.e.

In this case, there is a finite subfunction which is not inside the complement, which again is the always undefined function, so $\bar{B} = \mathcal{C} \setminus \{\emptyset\}$. Consider the modified definition of this set, which is:

$$\bar{B} = \{f \mid \exists y \in \text{dom}(f). \forall z > y. (z \notin \text{dom}(f)) \vee (f(y) \leq f(z))\}$$

In this case, it is possible to write a semicharacteristic function, considering we are interested in making the set halt and it holds that $\chi_{\bar{B}} = sc_{\bar{B}} * x$, where $sc_{\bar{B}} = \mathbf{1}(\mu w. H(x, (w)_1, (w)_2))$.

Given it is r.e, to understand if this is also recursive, we have to use Rice's theorem.

We showed before already the set $\neq \emptyset$. There also holds $\bar{B} \neq N$ and we know B is saturated and it's easy to see (literally copying what was present before) that \bar{B} is saturated. So, this is not recursive.

Exercise 8.70. Classify the following set from the point of view of recursiveness

$$B = \{x \in \mathbb{N} : \forall y \in W_x. \exists z \in W_x. (y < z) \wedge (\varphi_x(y) < \varphi_x(z))\},$$

i.e., establish if B and \bar{B} are recursive/recursive enumerable.

Solution: The set B is saturated, given that $B = \{x : \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \in \mathcal{C} : \forall y \in \text{dom}(f). \exists z \in \text{dom}(f). (y < z) \wedge (f(y) < f(z))\}$.

The set B is not r.e. by Rice-Shapiro's theorem. In fact, observe that $\mathbf{1} \notin \mathcal{B}$, but $\emptyset \subseteq \mathbf{1}$ and $\emptyset \in \mathcal{B}$.

For the complement $\bar{B} = \{f \mid \exists y \in \text{dom}(f). \forall z \in \text{dom}(f). y < z \rightarrow (f(y) \geq f(z))\}$, observe that if θ is any finite function, $\theta \neq \emptyset$, $y = \max(\text{dom}(\theta))$ clearly satisfies the condition definition of \bar{B} . Hence, it is enough to observe that $\text{id} \notin \bar{B}$ and consider $\theta \subseteq \text{id}, \theta \neq \emptyset$ noting that $\theta \in \bar{B}$. \square

Exercise 8.71. Classify the following set from the point of view of recursiveness

$$A = \{x \mid W_x \cup E_x = \mathbb{N}\},$$

i.e., establish if A and \bar{A} are recursive/recursive enumerable.

Solution: The set A is saturated since $A = \{x \mid \varphi_x \in \mathcal{A}\}$ with $\mathcal{A} = \{f \mid \text{dom}(f) \cup \text{cod}(f) = \mathbb{N}\}$.

By Rice-Shapiro's theorem:

- A is not r.e., since $\text{id} \in \mathcal{A}$, but no finite subfunction $\theta \subseteq \text{id}$ can belong to \mathcal{A} . In fact $\text{dom}(\theta)$ is finite and therefore also $\text{cod}(\theta)$ is finite. Hence their union $\text{dom}(\theta) \cup \text{cod}(\theta)$ is again finite, which implies that $\text{dom}(\theta) \cup \text{cod}(\theta) \neq \mathbb{N}$. Therefore $\theta \notin \mathcal{A}$.
- \bar{A} is not r.e., since $\emptyset \in \bar{\mathcal{A}}$, $\text{id} \notin \bar{\mathcal{A}}$ and $\emptyset \subseteq \text{id}$.

Exercise 8.72. Classify the following set from the point of view of recursiveness

$$B = \{x \mid \exists k \in \mathbb{N}. kx \in W_x\},$$

i.e., establish if B and \bar{B} are recursive/recursive enumerable.

Solution: We observe that $K \leq_m A$. Define

$$g(x, y) = \begin{cases} 1 & x \in K \\ \uparrow & \text{otherwise} \end{cases} = sc_K(x)$$

By s.m.n theorem, we obtain a function $s : \mathbb{N} \rightarrow \mathbb{N}$ which is total and computable, such that $g(x, y) = \varphi_{s(x)}(y)$ and it is easy to see that s can be the reduction function.

Furthermore, A is r.e., in fact

$$sc_A(x) = \mathbf{1}(\mu w.H(x, x \cdot (w)_1, (w)_2))$$

We therefore conclude that \bar{A} is not r.e. \square

Exercise 8.73. Given $X, Y \subseteq \mathbb{N}$ define $X + Y = \{x + y \mid x \in X \wedge y \in Y\}$. Study the recursiveness of the set

$$B = \{x \mid x \in W_x + E_x\},$$

i.e., establish if B and \bar{B} are recursive/recursive enumerable.

Solution: We observe that $K \leq_m A$. Define

$$g(x, y) = \begin{cases} 0 & x \in K \\ \uparrow & \text{otherwise} \end{cases} = \mathbf{0}(sc_K(x))$$

By smn theorem, we obtain a function $s : \mathbb{N} \rightarrow \mathbb{N}$ total computable and such that $g(x, y) = \varphi_{s(x)}(y)$. It is easy to see that s can be the reduction function.

Furthermore, B is r.e., in fact

$$sc_B(x) = \mathbf{1}(\mu w.(S((w)_1 + (w)_2, (w)_1, (w)_2, (w)_3))$$

We therefore conclude that \bar{A} is not r.e. □

Exercise 8.74. Classify from the point of view of recursiveness the set $A = \{x \in \mathbb{N} : W_x \cap E_x = \mathbb{N}\}$, i.e., say if A and \bar{A} are recursive/recursively enumerable.

Solution: The set A is clearly saturated since $A = \{x \mid \varphi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid cod(f) \cup img(f) = \mathbb{N}\}$. We can deduce, by using Rice-Shapiro's theorem, that A is not r.e., in fact $id \in \mathcal{A}$ but clearly no finite subfunction $\theta \subseteq id$ can be in \mathcal{A} since $cod(f), img(f)$ are finite and thus $cod(f) \cup img(f) \neq \mathbb{N}$.

The complement is not r.e. again by Rice-Shapiro's theorem. E.g., $id \notin \bar{\mathcal{A}}$, but it admits \emptyset as finite subfunction and $\emptyset \in \bar{\mathcal{A}}$. □

Exercises (2022-01-19)

Exercise 3

Say that a function $f : \mathbb{N} \rightarrow \mathbb{N}$ is quasi-total if it is undefined on a finite number of inputs, i.e., $\overline{dom(f)}$ is finite. Classify the set $A = \{x \in \mathbb{N} \mid \varphi_x \text{ quasi-total}\}$ from the point of view of recursiveness, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Solution: Observe that A is saturated, since it can be expressed as $A = \{x \in \mathbb{N} \mid \varphi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \mid f \text{ quasi-total}\}$.

Hence, by Rice-Shapiro's theorem, we conclude that A and \bar{A} are not r.e., and thus they are not recursive. More in detail:

- A is not r.e.

The identity $id \in \mathcal{A}$ and for all $\theta \subseteq id$, θ finite, clearly $\theta \notin \mathcal{A}$. In fact, $\overline{dom(\theta)}$ is finite and thus $\overline{dom(\theta)}$ is infinite and thus θ is not quasi-total. Hence by Rice-Shapiro's theorem we conclude that A is not r.e.

- \bar{A} is not r.e.

In fact, $id \notin \bar{\mathcal{A}}$, but the always undefined function $\theta = \emptyset \subseteq id$ and $\theta \in \bar{\mathcal{A}}$, since $\overline{dom(\theta)} = \emptyset$ and thus $\overline{dom(\theta)} = \mathbb{N}$ is infinite. Hence by Rice-Shapiro's theorem we conclude that \bar{A} is not r.e.

Exercise 4

Classify the set $B = \{x \in \mathbb{N} \mid \exists y > 2x. y \in E_x\}$ from the point of view of recursiveness, i.e., establish whether B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is not recursive since $K \leq_m B$. In order to prove this fact, let us consider the function $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined, by

$$g(x, y) = \begin{cases} y & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

The function is computable since $g(x, y) = sc_k(x)$. Hence, by smn-theorem, there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi_{s(x)}(y) = g(x, y)$ for all $x, y \in \mathbb{N}$. We next argue that s is a reduction function for $K \leq_m B$. In fact

- If $x \in K$ then $\varphi_{s(x)}(y) = g(x, y) = y$ for all $y \in \mathbb{N}$. Hence, if we set $y = 2s(x) + 1 > 2s(x)$ we have $\varphi_{s(x)}(y) = y = 2s(x) + 1$. Hence $2s(x) + 1 \in E_{s(x)}$ and thus $s(x) \in B$.
- If $x \notin K$ then $\varphi_{s(x)}(y) = g(x, y) = \uparrow$ for all $y \in \mathbb{N}$. Hence $E_{s(x)} = \emptyset$ and therefore there cannot be $y > 2x$ such that $y \in E_{s(x)}$. Hence $s(x) \notin B$.

The set B is r.e., in fact its semi-characteristic function is

$$sc_B(x) = \mathbf{1}(\mu w.(S(x, (w)_1, x + 1 + (w)_2, (w)_3)),$$

In fact the minimalisation search for a input $(w)_1$ for the machine x , such that in some number $(w)_3$ of steps, the machine stops providing as an output $x + 1 + (w)_2$ for some $(w)_2$. When $(w)_2$ ranges over the naturals, $x + 1 + (w)_2$ ranges over all values greater than x .

The semi-characteristic function is computable, since it is the minimalisation of computable functions, hence B is r.e.

Since B is r.e. and not recursive, its complement \bar{B} is not r.e. (otherwise both B and \bar{B} would be recursive). Thus \bar{B} is not recursive.

Exercise (2022-06-17)**Exercise 3**

Study the recursiveness of the set $A = \{x \mid \varphi_x(y + x) \downarrow \text{ for some } y \geq 0\}$, i.e., establish if A and \bar{A} are recursive/recursively enumerable.

Solution: The set $A = \{x \mid \varphi_x(y + x) \downarrow \text{ for some } y \geq 0\}$ is not recursive because $K \leq A$. In order to prove this fact, let us consider the function $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined, by

$$g(x, y) = \begin{cases} 1 & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

The function is computable since $g(x, y) = sc_K(x)$. Hence, by the smn-theorem, there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi_{s(x)}(y) = g(x, y)$ for all $x, y \in \mathbb{N}$. We next argue that s is a reduction function for $K \leq_m A$. In fact

- If $x \in K$ then $\varphi_{s(x)}(y) = g(x, y) = 1$ for all $y \in \mathbb{N}$. In particular, $\varphi_{s(x)}(0 + s(x)) \downarrow$. Hence $s(x) \in A$.
- If $x \notin K$ then $\varphi_{s(x)}(y) = g(x, y) \uparrow$ for all $y \in \mathbb{N}$. Hence $\varphi_{s(x)}(y + s(x)) \uparrow$ for all $y \in \mathbb{N}$. Hence $s(x) \notin A$.

The set A is r.e., since its semi-characteristic function

$$sc_A(x) = \mathbf{1}(\mu(y, t).H(x, x + y, t))$$

is computable.

Therefore, \bar{A} is not r.e.

Exercises (2023-02-01)

Exercise 3

Classify from the point of view of recursiveness the set $A = \{x \in \mathbb{N} \mid W_x \neq \emptyset \wedge W_x \subseteq E_x\}$, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Solution: Observe that A is saturated, since it can be expressed as $A = \{x \in \mathbb{N} \mid \varphi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \mid \text{dom}(f) \neq \emptyset \wedge \text{dom}(f) \subseteq \text{dom}(f)\}$.

Hence, by Rice-Shapiro's theorem, we conclude that A and \bar{A} are not r.e., and thus they are not recursive. More in detail:

- A is not r.e.

Consider the predecessor function $\text{pred}(x) = x - 1$. Then $\text{pred} \in \mathcal{A}$ since $\text{dom}(\text{pred}) = \mathbb{N} = \text{cod}(\text{pred})$, hence $\text{dom}(\text{pred}) \neq \emptyset$ and $\text{dom}(\text{pred}) \subseteq \text{cod}(\text{pred})$. Moreover, consider a generic finite $\theta \subseteq \text{pred}$. If $\theta \neq \emptyset$, i.e., θ is not the always undefined function, then it is easy to realise that $\text{dom}(\theta) \not\subseteq \text{cod}(\theta)$. In fact, if $k = \max(\text{dom}(\theta))$ necessarily $k \notin \text{cod}(\theta)$ (since $\max(\text{cod}(\theta)) = k - 1$). Hence no finite subfunction of pred is in \mathcal{A} and therefore, by Rice-Shapiro, A is not r.e.

- \bar{A} is not r.e.

In fact, $\text{pred} \notin \bar{\mathcal{A}}$ and $\theta = \emptyset \subseteq \text{pred}$, $\theta \in \bar{\mathcal{A}}$. Hence by Rice-Shapiro's theorem we conclude that A is not r.e.

Exercise 4

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be some fixed total computable function and for $X \subseteq \mathbb{N}$ define $f(X) = \{f(x) \mid x \in X\}$. Study the recursiveness of the set $B = \{x \mid x \in f(W_x) \cup E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is not recursive because $K \leq B$. In order to prove this fact, let us consider the function $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined, by

$$g(x, y) = \begin{cases} y & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

The function is computable since $g(x, y) = y \cdot sc_K(x)$. Hence, by the smn-theorem, there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi_{s(x)}(y) = g(x, y)$ for all $x, y \in \mathbb{N}$. We next argue that s is a reduction function for $K \leq_m B$. In fact

Exercise 4

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be some fixed total computable function and for $X \subseteq \mathbb{N}$ define $f(X) = \{f(x) \mid x \in X\}$. Study the recursiveness of the set $B = \{x \mid x \in f(W_x) \cup E_x\}$, i.e., establish if B and \bar{B} are recursive/recursively enumerable.

Solution: The set B is not recursive because $K \leq B$. In order to prove this fact, let us consider the function $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined, by

$$g(x, y) = \begin{cases} y & \text{if } x \in W_x \\ \uparrow & \text{otherwise} \end{cases}$$

The function is computable since $g(x, y) = y \cdot sc_K(x)$. Hence, by the smn-theorem, there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $\varphi_{s(x)}(y) = g(x, y)$ for all $x, y \in \mathbb{N}$. We next argue that s is a reduction function for $K \leq_m B$. In fact

- If $x \in K$ then $\varphi_{s(x)}(y) = g(x, y) = y$ for all $y \in \mathbb{N}$. Hence $W_{s(x)} = E_{s(x)} = \mathbb{N}$ and thus $s(x) \in f(W_{s(x)}) \cup E_{s(x)} = f(\mathbb{N}) \cup \mathbb{N} = \mathbb{N}$.
- If $x \notin K$ then $\varphi_{s(x)}(y) = g(x, y) \uparrow$ for all $y \in \mathbb{N}$. Hence $W_{s(x)} = E_{s(x)} = \emptyset$ and thus $s(x) \notin f(W_{s(x)}) \cup E_{s(x)} = f(\emptyset) \cup \emptyset = \emptyset$.

The set B is r.e. In fact $x \in B$ if and only if one of the following conditions hold

- $x \in f(W_x)$, i.e.. there is $z \in W_x$ such that $f(z) = x$ or
- $x \in E_x$, i.e.. there is $z \in W_x$ such that $\varphi_x(z) = x$

Hence the semi-characteristic function of B can be written:

$$sc_B(x) = \mathbf{1}(\mu w.(H(x, (w)_1, (w)_2) \wedge (f((w)_1) = x) \vee (S(x, (w)_1, x, (w)_2)))$$

and this shows that it is computable.

Therefore, \bar{B} is not r.e. (hence not recursive).

Exercises (2023-02-20)

Exercise 3

Let $X \subseteq \mathbb{N}$ be a fixed non-empty finite set. Classify from the point of view of recursiveness the set

$$A = \{x \mid E_x \cap X \neq \emptyset\},$$

i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Solution: Observe that A is saturated, since it can be expressed as $A = \{x \in \mathbb{N} \mid \varphi_x \in \mathcal{A}\}$, where $\mathcal{A} = \{f \mid \text{cod}(f) \cap A \neq \emptyset\}$.

Moreover $A \neq \emptyset, \mathbb{N}$. In fact

- if $e \in \mathbb{N}$ is such $\varphi_e = id$ then $e \in A$, since $X \cap E_e = X \cap \mathbb{N} = X \neq \emptyset$;
- if $e' \in \mathbb{N}$ is such $\varphi_{e'} = \emptyset$ then $e' \notin A$, since $X \cap E_{e'} = X \cap \emptyset = \emptyset$.

Hence by Rice's theorem A is not recursive.

The set A is r.e. In fact $x \in A$ if and only if there exists an input $y \in \mathbb{N}$ such that $\varphi_x(y) \downarrow$ and $\varphi_x(y) \in X$. The latter condition can be easily checked since X is finite and thus recursive. Hence we can just search for such an input.

Formally the semi-characteristic function of A can be written as:

$$\begin{aligned} sc_A(x) &= \mathbf{1}(\mu w.(S(x, (w)_1, (w)_2, (w)_3) \wedge (w)_2 \in Y)) \\ &= \mathbf{1}(\mu w.(|\chi_S(x, (w)_1, (w)_2, (w)_3) * \chi_Y((w)_2) - 1|)) \end{aligned}$$

and, since S is decidable and X is recursive (since it is finite), this shows that sc_A is computable.

Therefore, \bar{A} is not r.e. (hence not recursive).

Exercise 4

Classify from the point of view of recursiveness the set

$$B = \{x \in \mathbb{N} \mid W_x \neq \emptyset \wedge \min(W_x) > 0\},$$

i.e., establish whether B and \bar{B} are recursive/recursively enumerable.

Solution: Observe that B is saturated, since it can be expressed as $B = \{x \in \mathbb{N} \mid \varphi_x \in \mathcal{B}\}$, where $\mathcal{B} = \{f \mid \text{dom}(f) \neq \emptyset \wedge \min(\text{dom}(f)) > 0\}$.

Hence, by Rice-Shapiro's theorem, we conclude that B and \bar{B} are not r.e., and thus they are not recursive. More in detail:

- B is not r.e.

Consider the identity function $\text{id}(x) = x$. Then $\text{id} \notin \mathcal{B}$ since $\text{dom}(\text{id}) = \mathbb{N}$, hence $\min(\text{dom}(\text{id})) = 0$. Moreover, consider the finite function $\theta : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$\theta(x) = \begin{cases} 1 & \text{if } x = 1 \\ \uparrow & \text{otherwise} \end{cases}$$

Clearly $\theta \subseteq \text{id}$ and $\min(\text{dom}(\theta)) = \min(\{1\}) = 1 > 0$ hence $\theta \in \mathcal{B}$. Therefore, by Rice-Shapiro, B is not r.e.

- \bar{B} is not r.e.

In fact, if θ is the function defined above, $\theta \notin \bar{\mathcal{B}}$. Moreover $\theta' = \emptyset \subseteq \theta$, $\theta' \in \bar{\mathcal{B}}$. Hence by Rice-Shapiro's theorem we conclude that B is not r.e.

Exercise (2021-06-30-solved)

Study from the point of view of recursiveness the set $A = \{x \in \mathbb{N} \mid P \cap W_x = \emptyset\}$, where P is the set of even numbers and determine if A and \bar{A} are recursive/r.e.

Solution

Observe A is saturated, considering $A = \{x \in \mathbb{N} \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid P \cap \text{dom}(f) = \emptyset\}$.

Using Rice-Shapiro, A is not r.e. Specifically, considering the set of computable functions, we will prove $\exists f \in C. f \notin A \wedge \exists \theta \subseteq f \text{ finite}, \theta \in A \Rightarrow A \text{ not r.e.}$

Consider $\text{id} \notin \mathcal{A}$, given $P \cap \text{dom}(\text{id}) = P \neq \emptyset$ while the always undefined function $\emptyset \in \mathcal{A}$, given $P \cap \text{dom}(\emptyset) = \emptyset \cap P = \emptyset$. Using Rice's theorem, considering $A \neq \emptyset$ and $A \neq \mathbb{N}$, also A saturated, we conclude this set is not recursive.

Let's try to show the same for \bar{A} using Rice-Shapiro in the same way. Define:

$$f(x) = \begin{cases} 2, & x \equiv 0 \pmod{2} \\ \uparrow, & \text{otherwise} \end{cases}$$

and

$$\theta(x) = \begin{cases} 2, & x \geq 2 \\ \uparrow, & \text{otherwise} \end{cases}$$

Clearly, $f \in \bar{A}$ but there isn't a function θ which is not inside \bar{A} . So, Rice-Shapiro fails.

This maybe represents \bar{A} is r.e., so let's try to write the semicharacteristic function:

$$sc_{\bar{A}} = \mu w. H(x, 2 * (w)_1, (w)_2)$$

So, \overline{A} is r.e. and not recursive, A is neither r.e. nor recursive.

Exercise (2021-06-30-solved)

Study from the point of view of recursiveness the set $B = \{x \in \mathbb{N} \mid \exists y, z \in W_x . x = y * z\}$ and determine if B and \overline{B} are recursive/r.e.

Solution

Set B is not recursive and we show it via $K \leq_m B$. To do so, consider the following function:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

Such function is computable, considering $g(x, y) = sc_K$. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y) \forall n \in \mathbb{N}$.

s can be considered the correct reduction function, infact:

- if $x \in K$, $g(x, y) = \phi_{s(x)}(y) * 1 = \phi_{s(x)}(y) \forall y \in \mathbb{N}$. Infact, $W_{s(x)} = N$ having $s(x) = 1 * s(x)$ and so $s(x) \in B$
- if $x \notin K$, $g(x, y) = \phi_{s(x)}(y) = \uparrow \forall y \in \mathbb{N}$. Infact, $W_{s(x)} = \emptyset$ and so there cannot exist indices y, z s.t. $s(x) = y * z \in W_{s(x)}$. So, $s(x) \notin B$

Is B r.e.? Yes, it is, considering a semicharacteristic function can be defined:

$$sc_B(x) = \mathbf{1}(\mu w. (H(x, (w)_1, (w)_3) \wedge H(x, (w)_2, (w)_3) \wedge x = (w)_2 * (w)_3))$$

which is computable. So, B is r.e. and not recursive, so \overline{B} is not r.e. (otherwise, they would be both recursive) so \overline{B} is not recursive.

Exercise (2021-02-25-solved.pdf)

Study the recursiveness of set $A = \{x \in N \mid W_x = \overline{E_x}\}$, i.e. establish if A and \overline{A} are recursive/r.e.

Solution

We observe first A is saturated, considering it can be defined with the set of computable functions $A = \{x \in N \mid \text{dom}(f) \setminus \text{cod}(f)\}$ for the set $A = \{x \in \mathbb{N} \mid \phi_x \in \mathcal{A}\}$

Using Rice-Shapiro's theorem, the following are not r.e and not recursive.

- A not r.e.

To show this we define:

$$f(x) = \begin{cases} \uparrow, & x = 0 \\ x, & \text{otherwise} \end{cases}$$

Note $f \in \mathcal{A}$ given $\text{dom}(f) = \{N\} \setminus \{0\}$ and $\overline{\text{cod}(f)} = \overline{\{0\}}$. Problem is, there is no finite subfunction able to compute this in the complement set, given $\text{dom}(\theta)$ and $\text{cod}(\theta)$ are finite and can't be complement one of the other, given that set would be infinite, but here is finite. For Rice-Shapiro, we conclude A is not r.e.

- \overline{A} not r.e.

In this case, we have that $\overline{A} = \{x \in \mathbb{N} \mid \phi_x \notin \mathcal{A}\}$. We use the empty function $\emptyset \in \mathcal{A}, \theta = \emptyset$ given $\text{dom}(f) \neq \overline{\text{cod}(f)} = \overline{\emptyset}$ and they compute a different value, hence there exists a finite subfunction but different from all the set values. So, for Rice-Shapiro, we conclude this is not r.e.

Exercise (2021-02-25-solved.pdf)

Study the recursiveness of set $A = \{x \in N \mid W_x \setminus E_x \text{ finite}\}$, i.e. establish if A and \overline{A} are recursive/recursively enumerable

Solution

Observe A is a saturated set given $A = \{x \in N \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{f \mid \text{dom}(f) \setminus \text{cod}(f)\}$ is finite, hence $A = \{f \mid \text{dom}(f) = \overline{\text{cod}(f)}\}$. Using Rice-Shapiro's theorem, both sets are not r.e.

- A not r.e.

We use the constant function $\mathbf{1} \in A$, given $\text{dom}(\mathbf{1}) \setminus \text{cod}(\mathbf{1}) = \mathbb{N} \setminus \{1\}$ is not finite and with the empty function $\emptyset \subseteq \mathbf{1}, \emptyset \in A$, given $\text{dom}(\emptyset) \setminus \text{cod}(\emptyset) = \emptyset$ finite and by Rice-Shapiro, A is not r.e.

- \overline{A} not r.e.

Again, the constant function $\mathbf{1} \in \overline{A}$ but no finite subfunction $\in \overline{A}$, considering $\text{dom}(\theta)$ and $\text{cod}(\theta)$ are finite and also $\text{dom}(f) \setminus \text{cod}(f)$ is finite. Hence, by Rice-Shapiro, \overline{A} is not r.e.

Exercise (2021-02-25-solved.pdf)

Study the recursiveness of set $B = \{x \in \mathbb{N} \mid \exists y. (x \leq y \leq 2x \wedge y \in W_x)\}$, i.e. establish B and \overline{B} are recursive/r.e.

Solution

Let's use a reduction for the halting set showing B is not recursive, arguing that $K \leq_m B$ and to prove this, we know its semicharacteristic function is made by two arguments and is like:

$$sc_K(x) = g(x, y) = \begin{cases} 1, & x \in W_K \\ \uparrow, & \text{otherwise} \end{cases}$$

This function is computable and by the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y) \forall x, y \in \mathbb{N}$ and this is the reduction function. To argue this last one:

- if $x \in K$, then $g(x, y) = \phi_{s(x)}(y) = 1 \forall x \in \mathbb{N}$ and for sure $y \in W_x$ s.t. $s(x) \leq y \leq 2s(x)$ for a function $y = s(x)$, so $s(x) \in B$
- if $x \notin K$, then $g(x, y) = \phi_{s(x)} \uparrow \forall y \in \mathbb{N}$ so $W_{s(x)} = \emptyset$ and there is no such function $y = 2x$ s.t. $s(x) \leq y \leq 2s(x)$ and so $x \notin B$.

Given these assumptions, the set B has a semicharacteristic function, which can be expressed by the halting oracle H and the function S terminating in t steps. In this case, the rule is to respect the definition of function inside $x \leq y \leq 2x$, s.t.:

$$sc_B(x) = \mathbf{1}(\mu w. (H(x, (w)_1, (w)_2) \wedge (x \leq (w)_1 \leq 2x)))$$

is computable. By the same assumption, there is no function able to express the complement and \overline{B} is not recursive.

Exercise (2021-02-09-solved.pdf)

Study the recursiveness of set $B = \{x \in N \mid \exists x \in E_x\}$, i.e. establish if B and \bar{B} are recursive/r.e.

Solution

The set B is not recursive since we can show by a reduction for the halting problem $K \leq_m B$. In this case, we consider $g(x, y) = y$ if $x \in K$ and $g(x, y) = sc_K(x) * y$. By the smn-theorem, there is a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)}(y) = g(x, y)$ and this can be shown to be the reduction function of such problem:

- if $x \in K$ then $g(x, y) = \phi_{s(x)}(y) = y$, $\forall y \in \mathbb{N}$ then $E_{s(x)} = \mathbb{N}$ and $3 * s(x) \in E_{s(x)}$. So, $s(x) \in B$
- if $x \notin K$ then $g(x, y) = \phi_{s(x)}(y) \uparrow$, $\forall y \in \mathbb{N}$. In this case $E_{s(x)} = \emptyset$ and $3 * s(x) \notin E_{s(x)}$, so $s(x) \notin B$

The set B is not recursive but it's r.e., considering we can write its semicharacteristic function this way:

$$sc_B(x) = \mathbf{1}(\mu w. (S(x, (w)_1, 3x, (w)_2))$$

Given B is r.e. but not recursive, the complement by definition of its conditions is not r.e. either and also not recursive.

Exercise (2022-01-19-solved)

Say that a function $f: \mathbb{N} \rightarrow \mathbb{N}$ is quasi-total if it is undefined on a finite number of inputs, i.e., $\overline{\text{dom}(f)}$ is finite. Classify the set $A = \{x \in \mathbb{N} \mid \varphi_x \text{ quasi-total}\}$ from the point of view of recursiveness, i.e., establish whether A and \bar{A} are recursive/recursively enumerable.

Solution

Observe the set is saturated, given $A = \{x \in \mathbb{N} \mid \phi_x \in \mathcal{A}\}$ since $A = \{f \mid f \text{ quasi-total}\}$. By Rice-Shapiro's theorem, A and \bar{A} are not r.e. and thus not recursive, so let's show:

- A is r.e.

Consider the case of the identity function $id \in \mathcal{A} \forall \theta \subseteq id, \theta \text{ finite}, \theta \notin \mathcal{A}$ given $\text{dom}(\theta)$ is finite, the codomain is undefined on a set of points, being quasi-total (this one is defined on all possible values) and $\overline{\text{dom}(\theta)}$ is infinite.

- A is not r.e.

Consider the case of \emptyset function, given $\theta = \emptyset \subseteq id$ and $\forall \theta \subseteq id, \theta \text{ finite}, \theta \notin \mathcal{A}$. Since $\text{dom}(\theta) = \emptyset$ and thus $\overline{\text{dom}(\theta)} = \mathbb{N}$ is infinite (this one is never defined).

Exercise (2023-07-04.pdf)

Classify the following set from the point of view of recursiveness

$$B = \{x \in \mathbb{N} \mid 4x + 1 \in E_x\},$$

i.e., establish if A and \bar{A} are recursive/recursively enumerable.

We argue that a set B is not recursive considering $K \leq_m B$. The reduction function can be defined as:

$$g(x, y) = \begin{cases} 4x + 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases}$$

The function $g(x, y)$ can be shown to be the correct reduction function using the smn-theorem via $x: N \rightarrow N$ s.t. $\forall x, y \in N: g(x, y) = \phi_{s(x)}(y)$. Infact, if $x \in K$, then $\phi_{s(x)}(y) = g(x, y) = 4x + \forall y \in N$ and if $x \notin K$, then $g(x, y) = \uparrow$, having no such y such that $\phi_{s(x)}(y) = 4x + 1$.

Furthermore, B is r.e., considering we can write the semicharacteristic function:

$$sc_B(x) = \mathbf{1}(\mu w. H(x, (w)_1, (w)_3) \wedge S(x, (w)_1, (w)_2, (w)_3)_1$$

which is computable. Therefore \overline{A} is not r.e., considering A is r.e. and non-recursive.

Exercise (2023-07-04)

- a. Provide the definition of a recursive set.
- b. Provide the definition of a recursively enumerable (r.e.) set.
- c. Show that if $A, B \subseteq \mathbb{N}$ are recursive then also $A \setminus B = \{x \in \mathbb{N} \mid x \in A \wedge x \notin B\}$ is recursive. Does this extend to r.e. sets, i.e., is it the case that if A and B are r.e. then also $A \setminus B$ is r.e.? Provide a proof or a counterexample.

a)

DEFINITION 13.1. A set $A \subseteq \mathbb{N}$ is *recursive* if its characteristic function

$$\chi_A : \mathbb{N} \rightarrow \mathbb{N}$$

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

is computable.

b)

DEFINITION 15.1 (Recursively enumerable set). We say that $A \subseteq \mathbb{N}$ is *recursively enumerable* if the semi-characteristic function

$$sc_A(x) = \begin{cases} 1 & x \in A \\ \uparrow & \text{otherwise} \end{cases}$$

is computable.

c) Assume A and B are recursive, hence we can write two characteristic functions as follows:

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

and:

$$\chi_B(x) = \begin{cases} 1, & x \in B \\ \uparrow, & \text{otherwise} \end{cases}$$

Also, for $A \setminus B = \{x \in A \wedge x \notin B\}$ is recursive because we can define:

$$\chi_{A \setminus B}(x) = \begin{cases} 1, & x \in A \wedge x \notin B \\ \uparrow, & \text{otherwise} \end{cases}$$

If this extends to r.e. sets, it means that both for A and B we are able to create semicharacteristic functions. Infact, we can write:

$$sc_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & \text{otherwise} \end{cases}$$

and:

$$sc_B(x) = \begin{cases} 1, & x \in B \\ \uparrow, & \text{otherwise} \end{cases}$$

Still, $A \setminus B$ is not r.e. since we can consider for example A and $B = \overline{A}$ and in this case $A \setminus B$ is equivalent to $A \cap \overline{A}$. If A is r.e., also \overline{A} is r.e. but the intersection gives the empty set, which is recursive, but not r.e.

Exercise (2019-01-24)

Study recursiveness of set $B = \{x \mid \phi_x(x) \downarrow \wedge \phi_x(x) \text{ odd}\}$, determining if B and \overline{B} are r.e. or not.

Solution

Observe B is saturated, considering $B = \{x: \phi_x(x) \in B\}$ where $B = \{f \in C \mid f(x) \downarrow \wedge f(x) \text{ odd}\}$.

B is r.e. given we can write

$$sc_B(x) = \mathbf{1} \left(\mu w. \left(H \left(\frac{x}{2}, (w)_1, (w)_2 \right) \wedge S \left(\frac{x}{2} + 1, (w)_2, (w)_1, (w)_3 \right) \right) \right)$$

which is computable.

By Rice's Theorem, B is not recursive, considering e_0 and e_1 indexes for id and \emptyset so $e_0 \in B$ and $e_1 \notin B$ not recursive.

$\overline{B} = \{f \in C \mid f(x) \downarrow \vee f(x) \text{ odd}\}$. By Rice-Shapiro, this is not r.e.; consider $M = \max\{f(x) \mid x \in \mathbb{N}\}$ and considering θ is any finite function, then $\theta \neq \emptyset, y = M(dom(\theta)), \theta \in \overline{B}$. Observe also, given the definition of set, $id \notin \overline{B}$ (because, in words, the property is not satisfied for every natural number) and consider $\theta \subseteq id, \theta \neq \emptyset$ with $\theta \in \overline{B}$.

Exercise

Study recursiveness of set $A = \{x \in \mathbb{N}; \exists y \in W_x. \exists z \in E_x. x + y = z\}$, so establish if A and \overline{A} are recursive/r.e.

Solution

This is very similar to 8.32.

We show $K \leq_m A$ and so A is not recursive. Define:

$$g(x, y) = \begin{cases} z, & \text{if } y \in W_x \text{ so } H(x, x, y) \\ \uparrow, & \text{otherwise} \end{cases}$$

(we say the function terminates, given it outputs a value when whenever value + what we get from image, so $f(x)$, is z , so this allows the function to terminate).

The function is computable, since $g(x, y) = z * \chi_H(x, x, y)$.

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$.

The function s is shown to be a correct reduction function:

- if $x \in K, \forall y \in \mathbb{N}, H(x, x, y)$ holds as true and therefore $\phi_{s(x)}(y) = g(x, y) = z \forall y \in \mathbb{N}$. So $s(x) \in A$
- if $x \notin K, \neg H(x, x, y)$ and so $\phi_{s(x)}(y) = \uparrow$, so $W_{s(x)} = \emptyset$. So $s(x) \notin A$

Set A is r.e. given we can write:

$$sc_A(x) = \mu w. (H(x, (w)_1, (w)_2, (w)_3 \wedge (w)_1 + (w)_2 = x))$$

Given A is r.e. but not recursive, \overline{A} is not r.e. (otherwise, they would be both recursive) and also not recursive.

Exercise (16-09-2020)

Study recursiveness of set $B = \{x \in \mathbb{N} : W_x \cup E_x = \mathbb{N}\}$, so establish if B and \overline{B} are recursive/r.e.

Solution

This is similar to 8.74 exercise.

Observe A is saturated because $A = \{x \mid \phi_x \in \mathcal{A}\}$ and $\mathcal{A} = \{f \mid \text{dom}(f) \cup \text{cod}(f) = \mathbb{N}\}$.

By Rice-Shapiro, we show that:

- A is not r.e.

This happens because $\text{id} \in \mathcal{A}$ but no finite subfunction $\theta \subseteq \text{id}$ can be in A , since $\text{dom}(f), \text{cod}(f)$ are finite and so $\text{cod}(f) \cap \text{img}(f) \neq \mathbb{N}$.

- \overline{A} is not r.e.

This happens because $\text{id} \in \mathcal{A}$, but consider the always undefined function \emptyset as a finite subfunction θ , so we have $\theta \in \mathcal{A}$.

By Rice's theorem, both sets are effectively not recursive, neither r.e. as we showed.

Exercise (2018-11-20-parziale.pdf)

Is there an index $e \in N$ and a total non-computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\text{dom}(f)$ and $\text{cod}(f)$ (where $\text{dom}(f) = \{x \mid f(x) \downarrow\}$ and $\text{cod}(f) = \{y \mid \exists x. f(x) = y\}$) there is $\text{dom}(f) = W_e$ and $\text{cod}(f) = E_e \forall e \in \mathbb{N}$?

Solution

Consider as index e the one of the identity function, so $e = \phi_e(\text{id})$ and so $W_e = E_e = \mathbb{N}$. Define the function $f: \mathbb{N} \rightarrow \mathbb{N}$ as follows:

$$f(x) = \begin{cases} \phi_x(x) + 1, & x \in W_x \\ 0, & \text{otherwise} \end{cases}$$

The function f is total, so $\text{dom}(f) = \mathbb{N} = W_e$. Also, $\text{dom}(f) = \mathbb{N} = E_e$. Infact, $\forall n \in \mathbb{N}$ if $n = 0$ then considering an index x of the always undefined function, you have $f(x) = 0$.

If $n > 0$ consider whatever index x of the constant function $n - 1$ and you have $f(x) = (n - 1) + 1 = n$.

For the second part, the answer is clearly no. Consider $e \in \mathbb{N}$ s.t. ϕ_e is the always undefined function, every f s.t. $\text{dom}(f) = W_e = \emptyset$ coincides with ϕ_e and so it is computable.

Exercise (2017-01-24)

Study from the point of view of recursiveness the set $B = \{x \mid \exists k \in \mathbb{N}. kx \in W_x\}$ so establish if B and \overline{B} are recursive/r.e.

Solution

B is r.e., given it can be defined as $sc_B(x) = \mathbf{1}(\mu w. S(x, (w)_1 * x, (w)_2, (w)_3))$.

We consider $K \leq_m B$ to show the set is not recursive:

$$g(x, y) = \begin{cases} 1, & x \in K \\ \uparrow, & \text{otherwise} \end{cases} = sc_K(x)$$

g is computable and by the smn-theorem there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y$ we have $\phi_{s(x)}(y) = g(x, y)$. This is shown to be the correct reduction function:

- if $x \in K, \phi_{s(x)}(y) = g(x, y) = 1 \forall y \in \mathbb{N}, W_{s(x)} = \mathbb{N}$ and so $\exists k \in \mathbb{N}. k * f(x), f(x) \in W_{f(x)}$, so $s(x) \in B$
- if $x \notin K, \phi_{s(x)}(y) = g(x, y) = \uparrow \forall y \in \mathbb{N}, W_{s(x)} = \emptyset$ and so $\exists k \in \mathbb{N}. \nexists k * f(x), f(x) \in W_{f(x)}$, so $s(x) \notin B$

Exercise (16-09-2020)

Study from the point of view of recursiveness the set $A = \{x \in \mathbb{N}: \exists y \in W_x, \exists z \in E_x. x = y + z\}$ so establish if A and \overline{A} are recursive/r.e.

Solution

The set A is clearly saturated, since we can write $A = \{x \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{x \in \mathbb{N}: \exists y \in \text{dom}(f), \exists z \in \text{cod}(f). x = y + z\}$

The set is not recursive, since $K \leq_m A$ to show the set is not recursive:

$$g(x, y) = \begin{cases} y, & x \in K \\ \uparrow, & \text{otherwise} \end{cases} = y * sc_K(x)$$

g is computable and by the smn-theorem there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y$ we have $\phi_{s(x)}(y) = g(x, y)$. This is shown to be the correct reduction function:

- if $x \in K, \phi_{s(x)}(y) = g(x, y) = y \forall y \in \mathbb{N}, W_{s(x)} = \mathbb{N}$ and so $\phi_{s(x)}(s(x)) = \phi_{s(x)} + z \in \mathbb{N}$, so $s(x) \in A$
- if $x \notin K, \phi_{s(x)}(y) = g(x, y) = \uparrow \forall y \in \mathbb{N}, W_{s(x)} = \emptyset$ and so $\phi_{s(x)}(s(x)) = \uparrow$ with $E_{s(x)} = W_{s(x)} = \emptyset$ and so $s(x) \notin A$

The set is r.e. since we can write:

$$sc_A(x) = \mathbf{1}(\mu w. (H(x, x, y) \wedge S(x, y + z, x, t))) = \mathbf{1}(\mu w. (H(x, x, (w)_1) \wedge S(x, (w)_2 + (w)_3, x, (w)_4)))$$

Therefore, given A is r.e. but not recursive, also \overline{A} is not r.e. and not recursive (otherwise both would be recursive).

Exercise

Let $\mathbb{P} = \{2k \mid k \in \mathbb{N}\}$ be the set of even numbers. Study recursiveness of set $A = \{x \in \mathbb{N}: |W_x \cap \mathbb{P}| \geq 2\}$, so establish if A and \overline{A} are recursive/r.e.

Solution

The set A is clearly saturated, since we can write $A = \{x \mid \phi_x \in \mathcal{A}\}$ where $\mathcal{A} = \{x \in \mathbb{N}: |\text{dom}(f) \cap \mathbb{P}| \geq 2\}$.

We conjecture this set cannot be r.e. since this requirement would involve looking for every possible value on the domain in case of infinite sets to consider.

Let's use Rice-Shapiro to prove this set is not r.e. A function easily respecting the requirements would be the identity, since $|W_x \cap \mathbb{P}| = \mathbb{N} \geq 2$ since the intersection with the even numbers set holds, so $\text{id} \in \mathcal{A}$. Also, there exists a θ finite, $\theta \notin \mathcal{A}$ since $\theta = \emptyset$ given $|W_\emptyset \cap \mathbb{P}| = \emptyset * P = \emptyset$ which is no ≥ 2 . So A not r.e.

Consider the complement $\overline{A} = \{x \in \mathbb{N}: |W_x \cup \mathbb{P}| \leq 2\}$

A function clearly respecting this property would be the constant function 0 which domain with even numbers can be at least in the base case ≤ 2 . Conversely, the subfunction $\theta = \emptyset \notin \mathcal{A}$.

Hence, A and \overline{A} are not r.e. and not recursive.

22.9 SECOND RECURSION THEOREM

Note: This is fairly straightforward, given the theorem is pretty simple and basically says that you use a function of two arguments which by the smn-theorem fixes x . This x is then replaced by the index used inside the theorem to prove the statement. Other times, the set is not saturated and so by the theorem, using the problem conditions, there will be two indices not computing the same thing.

Exercise 9.1. State and prove the second recursion theorem.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

There goes the proof:

Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be total computable.

Observe $x \mapsto \phi_x(x)$ computable

$$\Psi_U(x, x)$$

$x \mapsto f(\phi_x(x))$ computable

define

$$\begin{aligned} g(x, y) &= \phi_{f(\phi_x(x))}(y) && \text{convention } \phi_\uparrow = \uparrow \\ &= \Psi_U(f(\phi_x(x)), y) \\ &= \Psi_U(f(\Psi_U(x, x)), y) && \text{computable} \end{aligned}$$

By the smn-theorem, there is $s: \mathbb{N} \rightarrow \mathbb{N}$ total and computable s.t. $\forall x, y$

$$\phi_{s(x)}(y) = g(x, y) = \phi_{f(\phi_x(x))}(y) \quad (*)$$

Since s is computable, there is $m \in \mathbb{N}$ s.t. $S = \phi_m$.

Substituting in $(*)$

$$\phi_{\phi_m(x)}(y) = \phi_{f(\phi_x(x))}(y) \quad \forall x, y$$

In particular, for $x = m$

$$\phi_{\phi_m(m)}(y) = \phi_{f(\phi_m(m))}(y) \quad \forall y$$

Hence

$$\phi_{\phi_m(m)} = \phi_{f(\phi_m(m))}$$

If we let $e_0 = \phi_m(m) \downarrow$ and replace in the previous equation, we conclude

$$\phi_{e_0} = \phi_{f(e_0)}$$

(note that $\phi_m = s$ total, hence $\phi_m(m) \downarrow$)

Exercise 9.2. State the second recursion theorem and use it to prove that K is not recursive.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Consider this comes from the notes and (18.1) refers to the definition just given here.

PROOF. Let $k = \{x \mid x \in W_x\}$ recursive for the sake of the argument. and let e_0, e_1 be indexes s.t. $\varphi_{e_0} = \emptyset$ and $\varphi_{e_1} = \lambda x . x$.

Define $f: \mathbb{N} \rightarrow \mathbb{N}$

$$\begin{aligned} f(x) &= \begin{cases} e_0 & x \in K \\ e_1 & x \notin K \end{cases} \\ &= e_0 \cdot \chi_K(x) + e_1 \cdot \chi_{\bar{K}}(x) \end{aligned}$$

If K were recursive, then χ_K and $\chi_{\bar{K}}$ would be computable, thus f would be both computable and total, then by (18.1), there would be $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{f(e)}$, but

- if $e \in K$, then $f(e) = e_0$, so $\varphi_e(e) \downarrow \neq \varphi_{f(e)}(e) = \varphi_{e_0}(e) \uparrow$
- if $e \in \bar{K}$, then $f(e) = e_1$, so $\varphi_e(e) \uparrow \neq \varphi_{f(e)}(e) = \varphi_{e_1}(e) = e \downarrow$

which is absurd, so K cannot be recursive.

Exercise 9.3. State the Second Recursion Theorem and use it for proving that there exists $x \in \mathbb{N}$ such that $\varphi_x(y) = y^x$, for each $y \in \mathbb{N}$.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

As solved by an old tutor:

$$\begin{aligned} f(x, y) &= y^x \text{ COMPUTABLE} \\ \xrightarrow{\text{SMN THEOREM}} \exists s: \mathbb{N} \rightarrow \mathbb{N} \text{ TOTAL COMPUTABLE} \text{ st } f(x, y) &= \varphi_{s(x)}(y) = y^x \\ \xrightarrow{\text{II RECURSION THEOREM}} \exists x_0 \in \mathbb{N} \text{ st } \varphi_{x_0} = \varphi_{s(x_0)} &\text{ BECAUSE } s \text{ TOTAL COMPUTABLE} \\ \varphi_{x_0}(y) &= \varphi_{s(x_0)}(y) = y^x \end{aligned}$$

Exercise 9.4. State the Second Recursion Theorem and use it for proving that there exists $n \in \mathbb{N}$ such that $W_n = E_n = \{x \cdot n : x \in \mathbb{N}\}$.

(Same exercise as 9.18)

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Define a function of two arguments as follows:

$$g(n, y) = \begin{cases} y, & \text{if } y = x * n \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists $n \in \mathbb{N}$ s.t. $g(x, y) = \phi_{h(x)}(y)$ and by the second recursion theorem, there exists an index $\in \mathbb{N}$ s.t. $\phi_n = \phi_{h(e)}$ and $\phi_e(y) = \phi_{s(x)}(y) = g(e, y) = e, \forall y \in \mathbb{N}$.

As solved by an old tutor:

Exercise 9.4. State the Second Recursion Theorem and use it for proving that there exists $n \in \mathbb{N}$ such that $W_n = E_n = \{x \cdot n : x \in \mathbb{N}\}$.

φ_n

$$f(n, y) = \begin{cases} y & \exists x \in \mathbb{N} y = x \cdot n \\ \uparrow & \text{otherwise} \end{cases}$$

$$= y \cdot \mathbf{U}(\mathbf{M}x. \mathbf{I}y = x \cdot n)$$

computable

$\xrightarrow{\text{SMN}}$ $\exists s: \mathbb{N} \rightarrow \mathbb{N}$ $\xrightarrow{\text{TOTAL COMP}}$ st $f(n, y) = \varphi_{s(n)}(y)$

$\xrightarrow{\text{II REC}}$ $\exists n_0 \in \mathbb{N}$ st $\varphi_{n_0} = \varphi_{s(n_0)}$ $\text{dom}(\varphi_{s(n_0)}) = W_{s(n_0)}$

$$W_{n_0} = \{x \cdot n_0 : x \in \mathbb{N}\} = E_{n_0}$$

$$W_{s(n_0)} = \{y \in \mathbb{N} : \exists x \in \mathbb{N} y = x \cdot n\}$$

$$= \{x \cdot n \in \mathbb{N} : \exists x \in \mathbb{N}\} \text{ ok}$$

$$E_{s(n_0)} = \{z \in \mathbb{N} : \exists y \in \mathbb{N} \varphi_{s(n_0)}(y) = z\}$$

$$= \{y \in \mathbb{N} : \exists y \in \mathbb{N} \varphi_{s(n_0)}(y) = y\}$$

$$\{y \in \mathbb{N} : \exists x \in \mathbb{N} y = x \cdot n\} = W_{s(n_0)} = W_{n_0}$$

Exercise 9.5. State the Second Recursion Theorem and use it for proving that $x \in \mathbb{N}$ exists such that $\varphi_x(y) = x + y$.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Define a function of two arguments as follows:

$$g(x, y) = \begin{cases} x + y, & x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists $n \in \mathbb{N}$ s.t. $g(x, y) = \phi_{h(x)}(y)$ and by the second recursion theorem, there exists an index $e \in \mathbb{N}$ s.t. $\phi_e = \phi_{h(e)}$ and $\phi_e(y) = \phi_{s(x)}(y) = g(e, y) = e + y, \forall y \in \mathbb{N}$.

Exercise 9.6. State the Second Recursion Theorem and use it for proving that there exists $x \in \mathbb{N}$ such that $\varphi_x(y) = x - y$.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Define a function of two arguments as follows:

$$g(x, y) = \begin{cases} x - y, & x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists $n \in \mathbb{N}$ s.t. $g(x, y) = \phi_{h(x)}(y)$ and by the second recursion theorem, there exists an index $e \in \mathbb{N}$ s.t. $\phi_e = \phi_{h(e)}$ and $\phi_e(y) = \phi_{s(x)}(y) = g(e, y) = e - y \forall y \in \mathbb{N}$.

As solved by an old tutor:

$$\begin{aligned} f(x, y) &= x - y && \text{COMPUTABLE} \\ \xrightarrow{\text{SMN}} \exists s: \mathbb{N} \rightarrow \mathbb{N} & \xrightarrow{\text{TOTAL COMPUTABLE}} \text{st } f(x, y) = \varphi_{s(x)}(y) = x - y \\ \xrightarrow{\text{II RECURSION THEOREM}} \exists x_0 \in \mathbb{N} & \text{ st } \varphi_{x_0} = \varphi_{s(x_0)} \\ \varphi_{x_0}(y) &= \varphi_{s(x_0)}(y) = x_0 - y \end{aligned}$$

Exercise 9.7. State the second recursion theorem and use it for proving that there exists a $n \in \mathbb{N}$ such that φ_n is total and $|E_n| = n$.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$. Consider:

$$g(x, y) = \begin{cases} \phi_x(x), & \text{if } x \in W_n \\ 0, & x \geq 0 \end{cases}$$

This is computable since $g(x, y) = \Psi_U(x, x)$. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N} \phi_{s(x)}(y) = g(x, y)$.

By the Second Recursion Theorem, there exists an index $e \in \mathbb{N}$ s.t. $\phi_{s(e)} = \phi_e$ and so $\phi_{e(y)} = \phi_{s(e)}(y) = \phi_e(e), \forall e \in \mathbb{N}$. In both cases of $n \neq 0$, when $n = 0, e = 0 \in \mathbb{N}$, when $n \neq 0, E_n \in \mathbb{N}$ and so $|E_n| = n$

Exercise 9.8. State the second recursion theorem and use it for proving that the function $\Delta : \mathbb{N} \rightarrow \mathbb{N}$, defined by $\Delta(x) = \min\{y : \varphi_y \neq \varphi_x\}$, is not computable.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

By the theorem, such function does not have a fixed point, given the function is total and if the fixpoint existed, we would have $\phi_x = \phi_y$ and so $\phi_{\Delta(x)} = \phi_x$, hence extending the index property over all set, which is the practical definition of what the theorem says.

Here, instead, we have the exact opposite of a definition for the problem, hence $\Delta(x)$ is not computable.

Exercise 9.9. State the second recursion theorem and use it for proving that, if we indicate by e_0 an index of the function always undefined \emptyset and by e_1 an index of the identity function, the function $h : \mathbb{N} \rightarrow \mathbb{N}$, defined by

$$h(x) = \begin{cases} e_0 & \text{if } \varphi_x \text{ is total} \\ e_1 & \text{otherwise} \end{cases}$$

is not computable.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

The function is total but not computable by definition (uses diagonalization), hence $\phi_x \neq \phi_{h(x)} \forall x$, since ϕ_x is total, but $\phi_{h(x)}$ is not. And so, by the second recursion theorem, there exists an index $e \in \mathbb{N}$ s.t. $\phi_e \neq \phi_{s(e)}$ and so h cannot be computable.

Exercise 9.10. State the Second Recursion Theorem and use it for proving that there exists an index $x \in \mathbb{N}$ such that

$$\varphi_x(y) = \begin{cases} y^2 & \text{if } x \leq y \leq x+2 \\ \uparrow & \text{otherwise} \end{cases}$$

Solution: Consider the function

$$f(x, y) = \begin{cases} y^2 & \text{if } x \leq y \leq x+2 \\ \uparrow & \text{otherwise} \end{cases}$$

This is clearly computable, hence, by the smn theorem there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(x, y) = \varphi_{s(x)}(y)$. Applying the second recursion theorem to s we conclude. \square

Exercise 9.11. State the second recursion theorem and use it for proving that the set $C = \{x : 2x \in W_x \cap E_x\}$ is not saturated.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Let's define a computable function of two arguments, like:

$$g(x, y) = \begin{cases} 2x, & y = 2x \\ \uparrow, & \text{otherwise} \end{cases}$$

which is a computable function and by the smn-theorem, the exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$.

By the Second Recursion Theorem, there exists an index e s.t. $\phi_{s(e)} = \phi_e$ and so $\phi_{e(y)} = 2e$.

Therefore, we have $E_{s(x)} = y, E_e = \{e\}, e \in C$.

To show that is not saturated, take and index $e' \neq e$ and it holds that $\forall e' \neq e, e \notin E_{e'} = E_e$ and so $e \notin C$. Since for two different indices in the same set, they calculate different values (this is the textual-practical explanation), the set C is not saturated. Infact consider $2e \in W_e \cap E_e \neq W_{e'} \cap E_{e'}$.

Exercise 9.12. State the second recursion theorem. Use it for proving that the set $C = \{x \in \mathbb{N} \mid x \in E_x\}$ not saturated.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Let's define a computable function of two arguments, like:

$$g(x, y) = x$$

which is a computable function and by the smn-theorem, the exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$.

By the Second Recursion Theorem, there exists an index e s.t. $\phi_{s(e)} = \phi_e$ and so $\phi_{e(y)} = e$.

Therefore, we have $E_{s(x)} = y, E_e = \{e\}, e \in C$.

To show that is not saturated, take and index $e' \neq e$ and it holds that $\forall e' \neq e, e \notin E_{e'} = E_e$ and so $e \notin C$. Since for two different indices in the same set, they calculate different values (this is the textual-practical explanation), the set C is not saturated.

Exercise 9.13. Let e_0 and e_1 be indices for the function always undefined \emptyset and the constant 1, respectively. State the Second Recursion Theorem and use it to prove that the function $g: \mathbb{N} \rightarrow \mathbb{N}$ defined as below, is not computable:

$$g(x) = \begin{cases} e_0 & \varphi_x \text{ total} \\ e_1 & \text{otherwise} \end{cases}$$

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

The function is total, since ϕ_x total when $e_0 \in \emptyset$ and $e_1 \in \{1\}$. By definition, the function is not computable, since if it were, for the Second Recursion Theorem, there would be $e \in N$ s.t. $\phi_e = \phi_{g(e)}$. Bu definition of g , we have that ϕ_e is total when $\phi_{g(e)}$ is not since it holds $\forall x, \phi_{g(x)} \neq \phi_x$.

Exercise 9.14. State the second recursion theorem. Prove that, given a function $f : \mathbb{N} \rightarrow \mathbb{N}$ total computable injective, the set $C_f = \{x : f(x) \in W_x\}$ is not saturated.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Define

$$g(x, y) = \begin{cases} f(y) & \text{if } x = f(y) \\ \uparrow & \text{otherwise} \end{cases}$$

By the smn theorem, we obtain a function $s : \mathbb{N} \rightarrow \mathbb{N}$ total computable, such that $g(x, y) = \varphi_{s(x)}(y)$ and by the second recursion theorem there exists $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{s(e)}$. Therefore:

$$\varphi_e(y) = \varphi_{s(e)}(y) = g(e, y) = \begin{cases} f(e) & \text{if } x = f(e) \\ \uparrow & \text{otherwise} \end{cases}$$

Thus $e \in C_f$. Now, if we take a different index e' such that $\varphi_e = \varphi_{e'}$ we will have that, by injectivity of f , it holds $f(e') \neq f(e)$ and thus $f(e') \notin W_{e'} = W_e = \{f(e)\}$. Hence $e' \notin C_f$. \square

Exercise 9.15. State the second recursion theorem. Use it for proving that if C is a set such that $C \leq_m \overline{C}$, then C is not saturated.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Consider the reduction definition; if you let f be the reduction function, we consider $f : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $x \in C$ iff $f(x) \in \overline{C}$ (so, $f(x) \notin C$).

Since f is computable, by the second recursion theorem, there exists an index $e \in \mathbb{N}$ s.t. $\phi_e = \phi_{s(e)}$ so it holds $\phi_e = \phi_{f(e)}$.

Since C is saturated, we have that e' s.t. $\phi_e = \phi_{e'}$ gives two different functions and programs on their computations, so $e \notin C$ and the reduction function cannot exist.

Exercise 9.16. State the Second Recursion Theorem and use it for proving that there is an index $a \in \mathbb{N}$ such that

$$\varphi_a(y) = \begin{cases} y + e & \text{if } y \text{ multiple of } e \\ \uparrow & \text{otherwise} \end{cases}$$

Solution: Define

$$g(x, y) = \begin{cases} x + y & \text{if } y \text{ multiple of } x \\ \uparrow & \text{otherwise} \end{cases} = (x + y) \cdot \mathbf{1}(\mu z. |z * x - y|)$$

By smn theorem, $g(x, y) = \varphi_{s(x)}(y)$ with s computable total. Then the II recursion theorem can be used to conclude. \square

Exercise 9.17. State the second recursion theorem. Use it for proving that every function f which is not total, but undefined only on a single point, i.e. $\text{dom}(f) = \mathbb{N} \setminus \{k\}$ for some $k \in \mathbb{N}$, admits a fixed point, i.e., there is $x \neq k$ such that $\varphi_x = \varphi_{f(x)}$.

Solution: Let h be such that $\varphi_h \neq \varphi_k$ and define

$$f'(x) = \begin{cases} f(x) & \text{if } x \neq k \\ h & \text{if } x = k \end{cases}$$

Clearly f' is computable (since f and the constant k are computable, and the predicate $x = k$ is decidable) and total. Therefore for the second recursion theorem there exists $x \in \mathbb{N}$ such that $\varphi_{f'(x)} = \varphi_x$. And by construction $x \neq k$, thus $f'(x) = f(x)$. \square

Exercise 9.18. State the Second Recursion Theorem and use it for proving that there is $n \in \mathbb{N}$ such that $W_n = E_n = \{x \cdot n : x \in \mathbb{N}\}$.

Solution: Define

$$g(n, y) = \begin{cases} y & \text{if } y = x \cdot n \\ \uparrow & \text{otherwise} \end{cases}$$

The smn theorem and the second recursion theorem can then be used to conclude. \square

Exercise 9.19. Prove that there exists $n \in \mathbb{N}$ such that $\varphi_n = \varphi_{n+1}$ and also $m \in \mathbb{N}$ such that $\varphi_m \neq \varphi_{m+1}$.

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{e(n)}$.

We must then show the existence of such a function, which can be successfully used with the second recursion theorem.

An example could be given by using the successor function $\text{succ}: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\text{succ}(n) = n + 1 \forall n \in \mathbb{N}$. This can be extended to any computable function, considering by the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\phi_{s(x)} = g(x, y)$ with

$$g(x, y) = f(x) = \begin{cases} x + 1, & \text{if } \phi_x(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases} = x + 1 + \overline{s\bar{g}}(\mu w. H(x, y, t)) \quad \text{computable}$$

We can conclude by using Second Recursion Theorem, saying there exists an index e such that $\phi_{s(e)} = \phi_e$ and then:

$$\phi_e(x) = \begin{cases} e + 1, & \text{if } \phi_e(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

Considering $\exists n \in \mathbb{N}$ s.t. $\phi_n = \phi_{\text{succ}(n)} = \phi_{n+1}$, this hold for any $e' \neq e$ s.t. $\phi_{e(n)} = \phi_{e+1}$

For the second part, this essentially says all computable do not coincide for every possible index; so, the first case says computable functions can coincide on an index (which happens, because the theorem refers to a fixed point), this second one shows there is at least one where this does not hold.

Conversely from before, we can consider the predecessor function $\text{pred}: \mathbb{N} \rightarrow \mathbb{N}$ such that:

$$\text{pred}(n) = \begin{cases} 0, & n = 0 \\ n - 1, & \text{otherwise} \end{cases}$$

This function is total and computable, by the second recursion theorem, there exists a program e such that $\phi_e = \phi_{f(e)}$. Specifically, consider $\exists m \in \mathbb{N} s.t. \phi_m = \phi_{pred(m)} = \phi_{m-1} \neq \phi_{m+1}$, proving what was present above.

Exercise 9.20. State the second recursion theorem. Use it for proving that the set $B = \{x \in \mathbb{N} : \exists k \in \mathbb{N}. k \cdot x \in W_x\}$ is not saturated.

Solution: The Second Recursion Theorem states that given a total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\varphi_{h(e)} = \varphi_e$.

Concerning the question, we proceed similarly to the proof of the fact that K is not saturated and find an index e such that $\varphi_e = \{(e, e)\}$. Also, we can assume that $e \neq 0$. In fact, define

$$g(e, x) = \begin{cases} e & \text{if } x = e \\ \uparrow & \text{otherwise} \end{cases}$$

Note that g is computable and therefore by the SMN theorem, we derive the existence of a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that for each $e, x \in \mathbb{N}$

$$\varphi_{s(e)}(x) = g(e, x)$$

By the II recursion theorem, there exists an index e such that $\varphi_{s(e)} = \varphi_e$ and then

$$\varphi_e(x) = \begin{cases} e & \text{if } x = e \\ \uparrow & \text{otherwise} \end{cases}$$

We can assume $e \neq 0$ because if it were $e = 0$, it would be sufficient to consider s' such that $s'(0) = e_0$ (index of the function always undefined) and $s'(x) = s(x)$ otherwise, and apply the same reasoning again. The fixed point will certainly be $\neq 0$, since $\varphi_0 \neq \emptyset = \varphi_{e_0} = \varphi_{f(0)}$.

Now, we have that

- $e \in B$, since $e = 1 \cdot e \in W_e = \{e\}$;
- given any index $e' > e$ such that $\varphi_e = \varphi_{e'}$ (it certainly exists, since there are infinite indices for a computable function) we have that $e' \notin B$, since there cannot be a $k \in \mathbb{N}$ such that $k \cdot e' \in W_{e'} = W_e = \{e'\}$. In fact, for $k > 0$ we have that $k \cdot e' > e$ and for $k = 0$, we have $k \cdot e' = 0 \neq e$, by construction.

Thus B not saturated. □

Exercise 9.21. State the second recursion theorem. Use it for proving that the set $C = \{x \in \mathbb{N} : \varphi_x(x) = x^2\}$ is not saturated.

Solution: The Second Recursion Theorem states that given a total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$, there exists $e \in \mathbb{N}$ such that $\varphi_{h(e)} = \varphi_e$.

Concerning the question, as in the case of the proof for K we can find an index e such that $\varphi_e = \{(e, e^2)\}$. Then we have $e \in C$, but any other index for the same function is not in C . □

Exercise 9.22. State the second recursion theorem and use it for proving that there is an index k such that $W_k = \{k * i \mid i \in \mathbb{N}\}$.

Solution: Consider the following function

$$g(x, y) = \begin{cases} 0 & \text{if there exists } i \text{ such that } y = x * i \\ \uparrow & \text{otherwise} \end{cases} = \mu i. |x \cdot i - y|$$

It is computable, hence we can use the smn theorem and the second recursion theorem to conclude.

□

Exercise 9.23. State the second recursion theorem. Use it for proving that the set $C = \{x \in \mathbb{N} : [0, x] \subseteq W_x\}$ is not saturated.

Solution: The Second Recursion Theorem states that given a total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\varphi_{h(e)} = \varphi_e$.

Concerning the question, as in the case of the proof for K we can find an index e such that $W_e = [0, e]$ and we can assume that $e \neq 0$. In fact, let us define

$$g(e, x) = \begin{cases} e & \text{if } x \leq e \\ \uparrow & \text{otherwise} \end{cases}$$

This is computable and therefore by SMN theorem, we derive the existence of a computable total function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that for each $e, x \in \mathbb{N}$

$$\varphi_{s(e)}(x) = g(e, x)$$

By the II recursion theorem there exists an index e such that $\varphi_{s(e)} = \varphi_e$ and then

$$\varphi_e(x) = \begin{cases} e & \text{if } x \leq e \\ \uparrow & \text{otherwise} \end{cases}$$

Given any index $e' > e$ such that $\varphi_e = \varphi_{e'}$ (it certainly exists since there are infinite indices for a computable function) we have that $e' \notin C$, since $[0, e'] \not\subseteq [0, e] = W_{e'}$.

Thus C is not saturated. □

Exercise 9.24. State the second recursion theorem and use it for proving that there is an index $n \in \mathbb{N}$ such that $\varphi_{p_n} = \varphi_n$, where p_n is the n -th prime number.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

$$g(x, y) = \begin{cases} p_x, & \text{if } \phi_x(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

For the smn-theorem, there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$

By the second recursion theorem, there exists an index e s.t. $\phi_{s(e)} = \phi_e$ and so:

$$\phi_e(y) = g(e, y) = \begin{cases} p_e, & \text{if } \phi_e(e) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

Therefore, $\phi_e(y) = \phi_{s(e)}(y) = p_e$ and so $p - e$ is the e^{th} prime number $\in W_e$ and also $W_e = \mathbb{N}$, so $e \in C$.

Exercise 9.25. State the second recursion theorem. Use it for proving that there is an index x such that $W_x = \{kx \mid k \in \mathbb{N}\}$.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

$$g(x, y) = \begin{cases} 0, & \exists z \text{ s.t. } z = x * y = \mu w. |z * x - y| \\ \uparrow, & \text{otherwise} \end{cases}$$

For the smn-theorem, there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \phi_{s(x)}(y) = g(x, y)$

By the second recursion theorem, there exists an index e s.t. $\phi_{s(e)} = \phi_e$

$$\phi_e(y) = g(e, y) = \begin{cases} y, & \text{if } e * y \in W_e \\ \uparrow, & \text{otherwise} \end{cases}$$

Therefore, $\phi_e(y) = \phi_{s(e)}(y) = e$ and $W_e = \mathbb{N}$ as intended.

Exercise 9.26. State the second recursion theorem. Use it for prove that there is an index $e \in \mathbb{N}$ such that $W_e = \{e^n : n \in \mathbb{N}\}$.

Solution: The Second Recursion Theorem states that given a total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\varphi_{h(e)} = \varphi_e$.

Concerning the question, define

$$g(x, y) = \begin{cases} \log_x y & \text{if } y = x^n \text{ for some } n \\ \uparrow & \text{otherwise} \end{cases} = \mu n. |y - x^n|$$

It is a computable function and therefore by the smn theorem, we have that there is a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ such that for each $x, y \in \mathbb{N}$

$$\varphi_{s(x)}(y) = g(x, y)$$

By the II recursion theorem there exists an index e such that $\varphi_{s(e)} = \varphi_e$ and then

$$\varphi_e(y) = \begin{cases} \log_e y & \text{if } y = e^n \text{ for some } n \\ \uparrow & \text{otherwise} \end{cases}$$

Therefore $W_e = \{e^n \mid n \in \mathbb{N}\}$. \square

Exercise 9.27. Use the second recursion theorem to prove that the following set is not saturated

$$C = \{x \mid W_x = \mathbb{N} \wedge \varphi_x(0) = x\}.$$

This one is also present inside 2017-01-24 exam.

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

$$g(x, y) = \begin{cases} x, & \text{if } \phi_x(x) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

For the smn-theorem, there exists a total computable function $s : \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}, \varphi_{s(x)}(y) = g(x, y)$

By the second recursion theorem, there exists an index e s.t. $\phi_{s(e)} = \phi_e$

$$\phi_e(y) = \begin{cases} e, & \text{if } \phi_e(e) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

Therefore, $\phi_e(y) = \phi_{s(e)}(y) = e$ and so $\phi_e(0) = e$ and also $W_e = \mathbb{N}$, so $e \in C$. Consider any $e' \neq e$ s.t. $\phi_{e'} = \phi_e$ and so $W_{e'} = W_e$ with $\phi_{e'}(0) = \phi_e(0)$. Since it is not saturated, we have that this cannot hold, so $\phi_{e'} \neq \phi_e$ and so $e' \notin C$.

Exercise (15-07-2020)

State the Second Recursion Theorem and use it to show that for all $k \geq 0$ there are two indices $x, y \in \mathbb{N}$ s.t. $x - y = k$ and $\phi_x = \phi_y$

Solution

The second recursion theorem states that for each total computable function $h : \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(n)}$. Define:

$$g(x, y) = \begin{cases} x - y, & \text{if } \phi_x(y) \downarrow \\ \uparrow, & \text{otherwise} \end{cases} = (x - y) - \mathbf{1}(\mu w. |k - \phi_x(y)|) = (x - y) - \mathbf{1}(\mu w. |k - \Psi_U(x, y)|)$$

which is computable. By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$. By the second recursion theorem, there exists an index $e \in \mathbb{N}$ s.t. $\phi_{s(e)}(y) = \phi_e(y)$. So, we define:

$$g(e, y) = \begin{cases} e - y, & \text{if } \phi_e(y) \downarrow \\ \uparrow, & \text{otherwise} \end{cases}$$

Therefore, for any index $e \in E_e$, we have that $\phi_e = \phi_{e'}$ since $\phi_x = \phi_y$, given $\phi_{s(x)} = \phi_{s(y)}$, $\forall k \geq 0$.

Exercise (30-06-2020)

State the Second Recursion Theorem and use it to show that the set $B = \{x \in \mathbb{N}: |W_x| = x + 1\}$ is not saturated.

Solution

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Define:

$$g(x, y) = \begin{cases} y + 1, & x \in W_k \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$. By the second recursion theorem, there exists $e \in \mathbb{N}$ s.t. $\phi_e = \phi_{s(e)}$. Therefore:

$$\phi_e(y) = \phi_{s(e)}(y) = g(e, y) = \begin{cases} e + 1, & x \in W_e \\ \uparrow, & \text{otherwise} \end{cases}$$

$e \in C_f$, considering $W_e = \{e\}$, $e \in C$ and by the Second Recursion Theorem, $\phi_x(y) = \phi_{s(x)}(y) = h(e, y) = e + 1 \forall y \in \mathbb{N}$.

Exercise (2019-01-24)

State the Second Recursion Theorem and use it to show there exists $x \in \mathbb{N}$ such that $|W_x| = x$

Solution

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

Define:

$$g(x, y) = \begin{cases} y, & x \in W_k \\ \uparrow, & \text{otherwise} \end{cases}$$

By the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $g(x, y) = \phi_{s(x)}(y)$. By the second recursion theorem, there exists $e \in \mathbb{N}$ s.t. $\phi_e = \phi_{s(e)}$. Therefore:

$$\phi_e(y) = \phi_{s(e)}(y) = g(e, y) = \begin{cases} e, & x \in W_e \\ \uparrow, & \text{otherwise} \end{cases}$$

$e \in C_f$, considering $W_e = \{e\}$, $e \in C$ and by the Second Recursion Theorem, $\phi_x(y) = \phi_{s(x)}(y) = h(e, y) = e \forall y \in \mathbb{N}$.

Exercise (2019-02-08)

State the Second Recursion Theorem and use it to show the set $A = \{x \mid W_x \subseteq \{x\}\}$ is not saturated.

Solution

The second recursion theorem states that for each total computable function $h: \mathbb{N} \rightarrow \mathbb{N}$ there exists $e \in \mathbb{N}$ such that $\phi_e = \phi_{h(e)}$.

To answer this one, define:

$$g(x, y) = \begin{cases} x, & x \in W_x \\ \uparrow, & \text{otherwise} \end{cases}$$

which is computable and by the smn-theorem, there exists a total computable function $s: \mathbb{N} \rightarrow \mathbb{N}$ s.t. $\forall x, y \in \mathbb{N}$ we have $\phi_{s(x)}(y) = g(x, y)$.

By the second recursion theorem, there exists an index e s.t. $\phi_{s(e)} = \phi_e$ and so

$$\phi_e = \phi_{s(e)} = \phi_{e(y)} = g(e, y) = \begin{cases} e, & e \in W_e \\ \uparrow, & \text{otherwise} \end{cases}$$

Therefore, we have $W_e \subseteq e \in \mathbb{N}$. Consider any $e' \neq e$ s.t. $\phi'_e = \phi_e$ and so one would have $e \notin W'_e = W_e$ and therefore $e \notin A$, given $\phi_e(e) \downarrow \neq \phi_{e'}(e') \downarrow$. Hence, A is not saturated.