Chapter 1

What is Computability?

1.1 Preliminaries

- This course is about the **inherent limitations** of computing: **The problems we cannot solve by *writing a program*!**

- At the **intuitive level**, any mathematician or computer scientist and any student of these two fields of study **will have no difficulty in recognising** an *algorithm* as soon as they see one

- But how about:
- – "is there an algorithm which can determine whether or not a given computer program (the latter written in, say, the C-language) is **correct**?"*

  and

  – "is there an algorithm that will determine whether or not any given Boolean formula is a tautology, doing so via computations that take no more *steps* than some (fixed) polynomial function of the input length?"

*A "correct" program produces, *for every input*, precisely the output that is expected by an *a priori* specification.

- **What do we mean** by

  "**there is *NO algorithm* that solves a given problem"—with or without restrictions on the algorithm's efficiency**— and how do you validate such a statement?

  This **appears** to be **a much harder statement to validate** than "there **IS** an algorithm that solves such and such a problem"

  ▶ for the latter, all we have to do is *to **produce** a correct such algorithm* (and a proof that it works as claimed).

  By contrast, the former statement implies, mathematically speaking, a ***provably failed search* over the entire set of *all algorithms***, while we were looking for one that solves our **problem**.

- "*Provably failed*" is a funny way to say "*we have a PROOF that the search will fail*".

To have a PROOF that NO Algorithm A Solves a Problem P we MUST have a Theory where Algorithms and Problems are the Objects of Study!

Computability is Precisely That!

Compare: To *prove* properties of *numbers* you need a Theory *about* numbers: *Number Theory*.

Analogously, to prove properties of *Algorithms* and *Problems* you need a *Theory* about such objects: *Computability*.

- How do you define the concept "*Computable Function*"?

By defining a Programming Language where we can PROGRAM such functions!

As a side-effect this approach also makes Mathematically Precise the concepts "Algorithm" and, *synonymously*, "Mechanical Procedure".

- Now, PEDAGOGICALLY, it is prudent to go from the Concrete towards the Abstract.

  We recognise that in the context of "Programming" "the concrete experience" of most CS students is Programming in *High-Level languages*.[†]

  Thus we adopt the programming language known as Shepherdson-Sturgis *Unbounded Register "Machines"* (**URM**) —which is a straightforward **abstraction**[‡] of modern high level programming languages like **C**.

  ▶ Discuss and Contrast with TMs.

[†] "High-Level" meaning "away from the machine level".
[‡] **Abstraction**: A version devoid of unnecessary DETAILS.

Within the chapter on URMs *we will also explore a restriction of the URM programming language, that of the* **loop programs** *of A. Meyer and D. Ritchie.*

▶ We will learn that while these loop programs can only compute *a very small subset* of "all the computable functions", nevertheless they are *significantly more than adequate* for programming solutions to any "practical", computationally solvable, problem.

For example, *even restricting the nesting of loop instructions to* as low as two, we can compute —*in principle*— **enormously large functions**, which with input x can **produce astronomical outputs** such as

$$\left.\begin{matrix} 2^{\cdot^{\cdot^{\cdot^{2^x}}}} \end{matrix}\right\} 10^{350000} \text{ 2's} \tag{1}$$

The qualification above, "in principle", stems from the enormity of the output displayed in (1) —even for the input $x = 0$— that renders the above function way beyond "**practical**".

- The COURSE PLAN

1. Found *Computability Theory* using URMs as the programming language.
2. Will look at functions like (1) above, and many simpler ones, collectively known as "*Primitive Recursive*". These functions provide valuable **tools** —such as "CODING"— to the theory.
3. Will show that these functions are exactly the functions that are computed by the loop programs we mentioned above.

   Then

4. We will get to the three pillars of computability:
   - The universal function theorem
   - S-m-n theorem
   - Kleene Normal Form theorem

5. We will begin our study of algorithmically **unsolvable problems**.
   - First using the technique of *diagonalisation*

 I'll explain the technique when we get there. 

   - Then using the more sophisticated technique of reducing one problem A to another B: "**If I can solve (via programming) B, then here is how to also solve A from that knowledge**"

     We depict this by the symbol $A \leq B$. Clearly, if $A \leq B$ and if it turns out that A *has no programming solution*, then neither does B.
   - We note and study the distinction between "*decidable*" and "*verifiable*" problems.
   - We prove the startling theorem of Rice.

6. We will demonstrate the intimate connection between the *algorithmic unsolvability phenomenon* of *computing* on one hand, and the *unprovability phenomenon* of applied logic on the other. The latter phenomenon was discovered by Gödel and is known as his *first incompleteness theorem.*

- We conclude the course with aspects of the **Complexity of computation**. That is,

Question: Among the functions we *can* compute, why do some take enormous amount of resources to compute?

Question: Can we **predict** (*approximately*) how **complex** a computation is going to be (**upper bound** on computation steps!) by looking at the **structure** of the **program**?

Chapter 2

Algorithms, Computable Functions and Computations

Sept. 13, 2021

2.1 A Theory of Computability

Computability is the part of logic and theoretical computer science that gives

a mathematically precise formulation

to the concepts *algorithm*, *mechanical procedure*, and *calculable/computable function*.

2.2 The URM

We now embark on defining our high level programming language *URM*.

The **alphabet** of the language is

$$\leftarrow, +, \dot{-}, :, X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \textbf{if}, \textbf{else}, \textbf{goto}, \textbf{stop} \tag{1}$$

Just like any other high level programming language, URM manipulates the *contents* of *variables*.

The "inventors", [SS63] called the variables "registers".

1) These variables are restricted to be ONLY of *natural number type*.

2) Since this programming language is **for theoretical analysis only** —rather than practical implementation— every variable is allowed to hold *any* natural number whatsoever, without limitations to its size, hence the "UR" in the language name ("unbounded register").

3) The **syntax** of the variables is simple: A variable (*name*) is a string that starts with X and continues with one or more 1:

$$\text{URM variable set:} \quad X1, X11, X111, X1111, \ldots \quad (2)$$

4) Nevertheless, as is customary for the sake of convenience, we will utilize the bold face lower case letters $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}$, with or without subscripts or primes as *metavariables* in most of our discussions of the URM, and in examples of specific programs (where yet more, convenient metanotations for variables may be employed).

2.2.1 Definition. (URM Programs) A **URM program** is a finite (ordered) sequence of instructions (or commands) of the following five types:

$$\begin{aligned} &L: \ \mathbf{x} \leftarrow a \\ &L: \ \mathbf{x} \leftarrow \mathbf{x}+1 \\ &L: \ \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1 \\ &L: \ \textbf{stop} \\ &L: \ \textbf{if } \mathbf{x}=0 \textbf{ goto } M \textbf{ else goto } R \end{aligned} \tag{3}$$

where L, M, R, a, *written in decimal notation*, are in $\mathbb{N}$, and $\mathbf{x}$ is some variable.

We call instructions of the last type if-statements.

▶ Each instruction in a URM program **must be numbered** by its *position number*, L, in the program, where ":" separates the position number from the instruction.

▶ We call these numbers *labels*. *Thus, the label of the first instruction* MUST BE *"1"*.

▶ The instruction **stop** must **occur only once** in a program, as the **last instruction**.

▶ An **if-statmemnt** is *syntactically illegal* (meaningless) if any of M or R exceed the **label** of the program's **stop** instruction. □

The *semantics* of **each instruction/command** and of a URM-**computation** are given below.

2.2.2 Definition. (Instructions and computations; semantics)

A URM **computation** is a **sequence of actions** caused by the execution of the instructions of the URM as detailed below.

Every computation **begins** with the instruction labeled "1" as the *current* instruction.

An instruction ACTS **if and only if it is THE CURRENT INSTRUCTION**.

The ACTION is as follows:

(i) $L : \mathbf{x} \leftarrow a$. Action: The value of $\mathbf{x}$ becomes the (natural) number a. Instruction $L+1$ will be the next current instruction.

(ii) $L : \mathbf{x} \leftarrow \mathbf{x}+1$. Action: This causes the value of $\mathbf{x}$ to increase by 1. The instruction labeled $L+1$ will be the next current instruction.

(iii) $L : \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$. Action: This causes the value

of **x** to decrease by 1, *if* it was originally non zero. Otherwise it remains 0. The instruction labeled $L+1$ will be the next current instruction.

(iv) L : **stop**. Action: No variable (referenced in the program) changes value. The next current instruction is still the one labeled L.

(v) L : **if** $\mathbf{x} = 0$ **goto** M **else goto** R. Action: No variable (referenced in the program) changes value. The next current instruction is numbered M if $\mathbf{x} = 0$; otherwise it is numbered R.

□

What is missing? Read/Write statements! We will come to that!

We say that a computation *terminates*, or *halts*, iff it ever *makes current* (as we say "**reaches**") the instruction **stop**.

Note that the semantics of "L : **stop**" *appear* to require the computation to continue *for ever*...

... but it does so in a trivial manner where *no variable changes value, and the current instruction remains the same*: **Practically, the computation is over**.

When discussing URM programs (or as we just say, "URMs") one usually gives them names like

$$M, N, P, Q, R, F, H, G$$

.

2.3 The Classes $\mathcal{P}$ and $\mathcal{R}$ of Partial and Total Computable Functions

NOTATION: We write $\vec{\mathbf{x}}_n$ for the sequence of variables $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$. We write $\vec{a}_n$ for the sequence of values $a_1, a_2, \ldots, a_n$.

▶ It is normal to omit the n (vector length) from $\vec{\mathbf{x}}_n$ and $\vec{a}_n$ if it is understood from the context, or **we don't care**, in which case we write $\vec{\mathbf{x}}$ and $\vec{a}$.

2.3.1 Definition. (URM As a Function) A *computation* by the URM M **computes a function** that we denote by

$$M_{\mathbf{y}}^{\vec{\mathbf{x}}_n}$$

in *this precise sense*:

The notation means that **we chose** and *designated* as **input variables** of M the following: $\mathbf{x_1}, \ldots, \mathbf{x_n}$. Also indicates that **we chose** and *designated* **one** variable $\mathbf{y}$ as *the output variable*.

We call this "*the I/O convention for M*".

We now conclude the definition of the function $M_{\mathbf{y}}^{\vec{\mathbf{x}}_n}$: For *every choice we make* for *input values* $\vec{a}_n$ *from* $\mathbb{N}^n$,

"$\mathbb{N}^n$" is borrowed from set theory. It is the *cartesian product* of n copies of $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, that is, $\mathbb{N}^n$ is the set of all length-n sequences $a_1, a_2, \ldots, a_n$ where each a_i is in $\mathbb{N}$; a natural number.

(1) We *initialise the computation* of URM M, **by doing TWO things**:

(a) We *initialise* the input variables $\mathbf{x_1}, \ldots, \mathbf{x_n}$ with the input values

$$a_1, \ldots, a_n$$

We also *initialise* **all other (not input) variables** of M to be 0.

This is an **implicit READ action**.

(b) We next make the instruction labeled "1" **current**, and thus *start* the computation.

So, the initialisation is NOT part of the computation!

(2) **If** the computation *terminates*, that is, **if** at some point the instruction **stop** becomes *current*, then the value of **y** at that point (and hence at any future point, by (iv) above), is the *value* of the function $M_{\mathbf{y}}^{\vec{\mathbf{x}}_n}$ for *input* $\vec{a}_n$.

This is an implicit **WRITE action**. □

2.3.2 Definition. (Computable Functions) A function $f : \mathbb{N}^n \to \mathbb{N}$ of n variables $x_1, \ldots, x_n$ is called *partial computable* iff for some URM, M, we have $f = M_{\mathbf{y}}^{\mathbf{x_1},\ldots,\mathbf{x_n}}$.

The *set of all partial computable functions is denoted by* $\mathcal{P}$.

The *set of all the* total *functions in* $\mathcal{P}$ —that is, those that are **defined** on *all inputs* from $\mathbb{N}$— is the set of *computable* functions and is denoted by $\mathcal{R}$. The term *recursive* is used in the literature synonymously with the term *computable*. □

Saying COMPUTABLE or RECURSIVE without qualification *implies* the *qualifier* TOTAL.

It is OK to add TOTAL on occasion for EMPHASIS!!

"PARTIAL" means "might be *total* or *nontotal*"; we do not care, or we do not *know*.

BTW, you recall from MATH1019 that the symbol

$$f : \underset{\text{left field}}{\mathbb{N}^n} \rightarrow \underset{\text{right field}}{\mathbb{N}} \tag{1}$$

simply states that f takes input values from $\mathbb{N}$ in *each* of its input variables and outputs —if it outputs anything for the given input!— a number from $\mathbb{N}$. Note also the terminology in red type in the display (1) above!

Probably your 1019 text called $\mathbb{N}^n$ and $\mathbb{N}$ above "domain" and "range". *FORGET THAT!*

What *IS* the domain of f **really**? (in symbols $\text{dom}(f)$)

$$\text{dom}(f) \overset{Def}{=} \{\vec{a}_n : (\exists y) f(\vec{a}_n) = y\}$$

that is, the set of *all* inputs that *actually* cause *an output*.

The range is the set of *all* possible outputs:

$$\text{ran}(f) \overset{Def}{=} \{y : (\exists \vec{a}_n) f(\vec{a}_n) = y\}$$

A function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is *total* iff $\text{dom}(f) = \mathbb{N}^n$.

Nontotal iff $\text{dom}(f) \subsetneq \mathbb{N}^n$.

If $\vec{a}_n \in \text{dom}(f)$ we write simply $f(\vec{a}_n)\downarrow$. Either way, we say "f is *defined* at $\vec{a}_n$".

The opposite situation is denoted by $f(\vec{a}_n)\uparrow$ and we say that "f is *undefined* at $\vec{a}_n$". We can also say "f is *divergent* at $\vec{a}_n$".

- Example of a *total* function: the "$x+y$" function on the natural numbers.
- Example of a *nontotal* function: the "$\lfloor x/y \rfloor$" function on the natural numbers. All input pairs of the form "$a, 0$" fail to produce an output: $\lfloor a/0 \rfloor$ is undefined. All the other inputs work.

Sept. 15, 2021

2.4 URM "Programming Examples"

2.4.1 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow \mathbf{x} + 1$$
$$2 : \mathbf{stop}$$

Then $M^{\mathbf{x}}_{\mathbf{x}}$ is the function f given, for all $x \in \mathbb{N}$, by $f(x) = x + 1$, the *successor* function. □

2.4.2 Remark. (λ Notation) To avoid saying verbose things such as "$M^{\mathbf{x}}_{\mathbf{x}}$ is the function f given, for all $x \in \mathbb{N}$, by $f(x) = x + 1$", we will often use Church's *λ-notation* and write instead "$M^{\mathbf{x}}_{\mathbf{x}} = \lambda x.x + 1$".

In general, the notation "$\lambda \cdots .$" marks the beginning of a sequence of input variables "$\cdots$" by the symbol "λ", and the end of the sequence by the symbol "." What comes after the period "." is the "rule" that indicates how the output relates to the input.

The template for λ-notation thus is

$$\lambda\text{"input"}.\text{"output-rule"}$$

Relating to the above example, we note that $f = \lambda x.x + 1 = \lambda y.y + 1$ is correct and we are saying that *the two functions viewed as tables are the same.*

Note that x, y, are "apparent" variables ("dummy" or bound) and are not free (for substitution).

2.4.3 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$$
$$2 : \mathbf{stop}$$

Then $M^{\mathbf{x}}_{\mathbf{x}}$ is the function $\lambda x.x \dot{-} 1$, the *predecessor* function.

The operation $\dot{-}$ is called "proper subtraction" —some people pronounce it "*monus*"— and is in general defined by

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

It ensures that subtraction (as modified) does not take us out of the set of the so-called *number-theoretic functions*, which are those with inputs from $\mathbb{N}$ and outputs in $\mathbb{N}$.

□

Pause. *Why are we restricting computability theory to number-theoretic functions*? Surely, in *practice* we can compute with

- *negative numbers*,
- *rational numbers*, and
- with *nonnumerical* entities, such as graphs, trees, etc.

Theory ought to reflect, and explain, our practices; no?◀

It does. Negative numbers and rational numbers can be coded by natural number pairs.

Computability of number-theoretic functions can handle such *pairing* (and *unpairing* or *decoding*).

Moreover, finite objects such as graphs, trees, and the like that we manipulate via computers can be also coded (and decoded) by natural numbers.

▶ After all, the internal representation *of all data in computers* is, at the lowest level, via natural numbers represented in binary notation.

Computers cannot handle infinite objects such as (irrational) real numbers.

But there is an extensive "higher type" computability theory (which originated with the work of [Kle43]) that *can* handle such numbers as inputs and also compute with them.

However, this theory is way beyond our scope.

2.4.4 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow 0$$
$$2 : \textbf{stop}$$

Then $M^{\mathbf{x}}_{\mathbf{x}}$ is the function $\lambda x.0$, the *zero function*, usually denoted by Z. □

In Definition 2.3.2 we spoke of partial computable and total computable functions.

We retain the qualifiers partial *and* total *for all* number-theoretic *functions, that is, functions*

$$f : \mathbb{N}^n \to \mathbb{N}$$

even for those that may not be computable.

Total vs. *nontotal* (no hyphen) has been defined with respect to a chosen and **fixed** *left field* for *any* function in computability.

The set union of all total *and* nontotal number-theoretic functions is the set of all *partial (number-theoretic) functions*. Thus *partial* is *not* synonymous with *nontotal*.

2.4.5 Example. The *unconditional* **goto** instruction, namely, "L : **goto** L'", can be simulated by

$$L: \ \textbf{if } \mathbf{x} = 0 \textbf{ goto } L' \textbf{ else goto } L'$$

□

2.4.6 Example. Let M be the program

$$1 : \mathbf{x} \leftarrow \mathbf{x} + 1$$
$$2 : \textbf{goto } 1$$
$$3 : \textbf{stop}$$

Then $M^{\mathbf{x}}_{\mathbf{x}}$ is the empty function $\emptyset$, sometimes written as $\lambda x.\uparrow$.

Thus the empty function is partial computable but nontotal. We have just established $\emptyset \in \mathcal{P} - \mathcal{R}$. □

2.4.7 Example. Let M be the program segment

$$
\begin{aligned}
&k-1 : \mathbf{x} \leftarrow 0\\
&k \quad\;\; : \textbf{ if } \mathbf{z} = 0 \textbf{ goto } k+4 \textbf{ else goto } k+1\\
&k+1 : \mathbf{z} \leftarrow \mathbf{z} \dot{-} 1\\
&k+2 : \mathbf{x} \leftarrow \mathbf{x} + 1\\
&k+3 : \textbf{ goto } k\\
&k+4 : \ldots
\end{aligned}
$$

What it does:

By the time the computation reaches instruction $k+4$, the program segment has set the value of $\mathbf{z}$ to 0, and has made the value of $\mathbf{x}$ equal to the value that $\mathbf{z}$ had when instruction $k-1$ was current.

In short, the above sequence of instructions simulates the following sequence

$$
\begin{aligned}
&L : \quad\;\; \mathbf{x} \leftarrow \mathbf{z}\\
&L+1 : \mathbf{z} \leftarrow 0\\
&L+2 : \ldots
\end{aligned}
$$

where the semantics of $L : \mathbf{x} \leftarrow \mathbf{z}$ *are standard in programming*:

It requires that, upon execution of the instruction, the value of $\mathbf{z}$ is copied into $\mathbf{x}$ but the value of $\mathbf{z}$ remains unchanged. □

2.4.8 Exercise. Write a program segment that simulates precisely L : $\mathbf{x} \leftarrow \mathbf{z}$; that is, copy the value of $\mathbf{z}$ into $\mathbf{x}$ without causing $\mathbf{z}$ to change as a side-effect. □

We say that the "normal" assignment $\mathbf{x} \leftarrow \mathbf{z}$ is *non destructive*.

Because of Exercise 2.4.8 above, *without loss of generality*, one may assume that any input variable, $\mathbf{x}$, of a URM M is *read-only*.

This means that its value is retained throughout any computation of the program.

Why "*without loss of generality*"? Because if $\mathbf{x}$ is not such, we can *make* it be!

Indeed, let's add a new variable as an input variable, $\mathbf{x}'$ instead of $\mathbf{x}$.

Then, in detail, do this to make $\mathbf{x}'$ read-only:

- Add at the very beginning of M the instruction $1: \mathbf{x} \leftarrow \mathbf{x}'$ of Exercise 2.4.8.
- Adjust all the following labels consistently, including, of course, the ones referenced by *if-statements*—a tedious but straightforward task.
- Call M' the so-obtained URM.

2.4.9 Example. If M is

$$M: \begin{cases} 1: & \textbf{if } \mathbf{y} = 0 \textbf{ goto } L \textbf{ else goto } R \\ 2: & \ldots \end{cases}$$

then M' is

$$M': \begin{cases} 1: & \mathbf{x} \leftarrow \mathbf{x}' \\ 2: & \textbf{if } \mathbf{y} = 0 \textbf{ goto } L+1 \textbf{ else goto } R+1 \\ 3: & \ldots \end{cases}$$

□

Clearly, $M'^{\mathbf{x}',\mathbf{y}_1,\ldots,\mathbf{y}_n}_{\mathbf{z}} = M^{\mathbf{x},\mathbf{y}_1,\ldots,\mathbf{y}_n}_{\mathbf{z}}$, and M' does not change $\mathbf{x}'$.

2.4.10 Example. (Composing Computable Functions)

Suppose that $\lambda x\vec{y}.f(x,\vec{y})$ and $\lambda\vec{z}.g(\vec{z})$ are partial computable, and say

$$f = F^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{u}}$$

while

$$g = G^{\vec{\mathbf{z}}}_{\mathbf{x}}$$

We assume without loss of generality that $\mathbf{x}$ is the only variable common to F and G. Thus, if we concatenate the programs G and F in that order, *and*

1. remove the last instruction of G (k : **stop**, for some k) —call the program segment that results from this G', and

2. renumber the instructions of F as $k, k+1, \ldots$ (and, as a result, also the references that if-statements of F make) in order to give $\left(G'F\right)$ the correct program structure,

then, $\lambda\vec{y}\vec{z}.f(g(\vec{z}),\vec{y}) = (G'F)^{\vec{\mathbf{y}},\vec{\mathbf{z}}}_{\mathbf{u}}$.

Note that all non-input variables of F will still hold 0 as soon as the execution of $(G'F)$ makes the first instruction of F current *for the first time.*

This is because none of these can be changed by G' under our assumption, **thus ensuring that F works as designed.** □

By repeating the above a finite number of times:

2.4.11 Proposition. *If* $\lambda\vec{y}_n.f(\vec{y}_n)$ *and* $\lambda\vec{z}.g_i(\vec{z})$*, for* $i = 1,\ldots,n$*, are partial computable, then so is* $\lambda\vec{z}.f(g_1(\vec{z}),\ldots,g_n(\vec{z}))$.

Note that

$$f(g_1(\vec{a}),\ldots,g_n(\vec{a}))\uparrow$$

if any $g_i(\vec{a})\uparrow$

Else $f(g_1(\vec{a}),\ldots,g_n(\vec{a}))\downarrow$ provided f is defined on all $g_i(\vec{a}_n)$.

For the record, we will define *composition* to mean the *somewhat rigidly defined operation* used in 2.4.11, that is:

2.4.12 Definition. (Composition) Given any partial functions (computable or not) $\lambda\vec{y}_n.f(\vec{y}_n)$ and $\lambda\vec{z}.g_i(\vec{z})$, for $i = 1,\ldots,n$, we say that $\lambda\vec{z}.f(g_1(\vec{z}),\ldots,g_n(\vec{z}))$ is the result of their *composition*. □

2.4.13 Example.

$$f(r(y),y) \text{ fails } 2.4.12$$

$$f(g(y),h(z)) \text{ fails } 2.4.12$$

$$f(g(y),g'(y,w)) \text{ fails } 2.4.12$$

$$f(g(y,z),g''(y,z)) \text{ obeys } 2.4.12$$

□

We observe that Definition 2.4.12 is "*rigid*".

Indeed, note that it requires *all* the arguments of f to be substituted by some $g_i(\vec{z})$—unlike Example 2.4.10, where we *substituted* a function invocation (cf. terminology in 2.4.2) *only in* one *variable of* f there, and did nothing with the variables $\vec{y}$.

Also, for each call $g_i(\ldots)$ the argument list, "...", *must be the same*; in 2.4.12 it was $\vec{z}$.

As we will show in examples later, this rigidity is only *apparent*.

Sept. 20, 2021

We can rephrase 2.4.11, saying simply that

2.4.14 Theorem. $\mathcal{P}$ is *closed under* composition.

2.4.15 Corollary. $\mathcal{R}$ is closed under composition.

Proof. Let f, g_i be in $\mathcal{R}$.

Then they are in $\mathcal{P}$, hence so is $h = \lambda\vec{y}.f\Big(g_1(\vec{y}), \ldots, g_m(\vec{y})\Big)$ by 2.4.14.

By assumption, the f, g_i are total. So, for any $\vec{y}$, we have $g_i(\vec{y}) \downarrow$—a number. Hence also $f\Big(g_1(\vec{y}), \ldots, g_m(\vec{y})\Big) \downarrow$.

That is, *h is total*, hence, *being in $\mathcal{P}$*, it *is also in $\mathcal{R}$*.

□

A very simple recursive definition of a function.

2.4.16 Definition. (Primitive Recursion) A number-theoretic function f is defined by *primitive recursion* from given functions $\lambda\vec{y}.h(\vec{y})$ and $\lambda x\vec{y}z.g(x,\vec{y},z)$ provided, *for all* $x,\vec{y}$, its values are given by the two equations below:

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

h is the *basis function*, while g is the *iterator*.

We can take for granted a fundamental (but difficult) result (see EECS 1028, W20, course notes), that *a unique f that satisfies the above schema exists.*

Moreover, if both h and g are total, then so is f as it can easily be shown by induction on x (*Later*).

NOTATION. It will be useful to use the notation $f = prim(h, g)$ to indicate in shorthand that f is defined as above from h and g (note the order). □

Note that

$$\begin{aligned} f(1,\vec{y}) &= g(0,\vec{y},\underbrace{h(\vec{y})}_{f(0,\vec{y})}) \\ f(2,\vec{y}) &= g(1,\vec{y},\underbrace{g(0,\vec{y},h(\vec{y}))}_{f(1,\vec{y})}) \\ f(3,\vec{y}) &= g(2,\vec{y},\underbrace{g(1,\vec{y},g(0,\vec{y},h(\vec{y})))}_{f(2,\vec{y})}), \text{ etc.} \end{aligned} \qquad (\dagger)$$

▶ Thus the "x-value", 0, 1, 2, 3, etc., *equals the number of times we compose g with itself* (i.e., *the number of times we iterate g*).

▶ That is why g is called the *Iterator* of the definition and h is called the *Basis*.

With (a very) little programming experience, it is easy to see that $f(x, \vec{y})$ of 2.4.16 is computed by the pseudo code below:

$$\begin{aligned} &1 : z \leftarrow h(\vec{y}) \\ &2 : \textbf{for } i = 0 \textbf{ to } x - 1 \\ &3 : z \leftarrow g(i, \vec{y}, z) \end{aligned} \qquad (\ddagger)$$

At the end of the loop, z holds $f(x, \vec{y})$.

Here is how to *implement the above* as a URM:

2.4.17 Example. (Iterating Computable Functions)

Suppose that $\lambda x\vec{y}z.g(x, \vec{y}, z)$ and $\lambda\vec{y}.h(\vec{z})$ are partial computable, and, say, $g = G_{\mathbf{z}}^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}$ while $h = H_{\mathbf{z}}^{\vec{\mathbf{y}}}$.

By earlier remarks we may assume:

(i) The only variables that H and G have in common are $\mathbf{z}, \vec{\mathbf{y}}$.

(ii) The variables $\vec{\mathbf{y}}$ are *read-only* in both H and G.

(iii) $\mathbf{i}$ is *read-only* in G.

(iv) $\mathbf{x}$ does not occur in any of H or G. *It is an "agent" that ensures we iterate x times (x is the content of $\mathbf{x}$).*

We can now see that the following URM program, let us call it F, computes f defined as in 2.4.16 from h and g, where $\boxed{H'}$ is program H with the **stop** instruction removed, $\boxed{G'}$ is program G that has the **stop** instruction removed, and instructions renumbered (and if-statements adjusted) as needed:

Comment. x counts iterations going downwards;
i goes upwards; see (†) p.40 and (‡) p.41

$\boxed{H'}{}^{\vec{\mathbf{y}}}_{\mathbf{z}}$ this is "$z \leftarrow h(\vec{\mathbf{y}})$"

r : $\mathbf{i} \leftarrow 0$
$r+1$: **if** $\mathbf{x} = 0$ **goto** $k+m+2$ **else goto** $r+2$
$r+2$: $\mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$
$\boxed{G'}{}^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}_{\mathbf{z}}$ this is "$z \leftarrow g(\mathbf{i}, \vec{\mathbf{y}}, \vec{\mathbf{z}})$"
k : $\mathbf{i} \leftarrow \mathbf{i} + 1$
$k+1$: $\mathbf{w}_1 \leftarrow 0$
⋮
$k+m$: $\mathbf{w}_m \leftarrow 0$
$k+m+1$: **goto** $r+1$
$k+m+2$: **stop**

The instructions $\mathbf{w}_i \leftarrow 0$ set explicitly to zero all the variables of G' other than $\mathbf{i}, \mathbf{z}, \vec{\mathbf{y}}$ to ensure *correct behavior of G'*. Note that the $\mathbf{w}_i$ are *implicitly* initialised to zero *only* the first time G' is executed. Clearly, the URM F simulates the pseudo program above, thus $f = F^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$. □

We have at once:

2.4.18 Proposition. *If f, g, h relate as in Definition 2.4.16 and h and g are in $\mathcal{P}$, then so is f. We say that $\mathcal{P}$ is* closed under primitive recursion.

Why?

2.4.19 Corollary. *If f, g, h relate as in Definition 2.4.16 and h and g are in $\mathcal{R}$, then so is f. We say that $\mathcal{R}$ is* closed under primitive recursion.

Proof. As $\mathcal{R} \subseteq \mathcal{P}$, we have $f \in \mathcal{P}$.

But if h and g are total, then so is f. WHY? See ‡ on p.41.

So, $f \in \mathcal{R}$. □

What does the following pseudo program do, if $g = G_{\mathbf{z}}^{\mathbf{x},\vec{\mathbf{y}}}$ for some URM G?

$$\begin{aligned} &1 : \mathbf{x} \leftarrow 0 \\ &2 : \textbf{while } g(\mathbf{x}, \vec{\mathbf{y}}) \neq 0 \textbf{ do} \\ &3 : \mathbf{x} \leftarrow \mathbf{x} + 1 \end{aligned} \tag{1}$$

▶ We are out here (exited the **while**-loop) precisely *because*

- Testing for $g(\mathbf{x}, \vec{\mathbf{y}}) \neq 0$ *never* got stuck due to calling g with some $\mathbf{x} = m$ that makes $g(m, \vec{\mathbf{y}}) \uparrow$.
- The loop kicked us out *exactly* when $g(k, \vec{\mathbf{y}}) = 0$ was detected, for some k, *for the first time*, in the **while**-test.

In short, that k satisfies

$$k = \underline{\textit{smallest}} \text{ such that } g(k, \vec{\mathbf{y}}) = 0 \wedge (\forall z < k) g(z, \vec{\mathbf{y}}) \downarrow$$

Now, this k *depends* on $\vec{\mathbf{y}}$ so we may define it as a function f, for all INPUTS $\vec{a}$ in $\vec{\mathbf{y}}$, by:

$$k = f(\vec{a}) \overset{Def}{=} \min \left\{ x : g(x, \vec{a}) = 0 \wedge (\forall y) \big(y < x \rightarrow g(y, \vec{a}) \downarrow \big) \right\}$$

□

Kleene has suggested the symbol "μ" to denote the "find the minimum" operation above, thus the above is rephrased as

$$f(\vec{a}) = (\mu y)g(y,\vec{a}) \overset{Def}{=} \begin{cases} \min\left\{y : g(y,\vec{a}) = 0 \wedge \right. \\ \left. (\forall w)_{w<y} g(w,\vec{a}) \downarrow \right\} \\ \uparrow \text{ if there is no min} \end{cases} \qquad (2)$$

where $(\forall y)_{y<x} R(y,\ldots)$ is short for $(\forall y)(y < x \to R(y,\ldots))$.

We call the operation $(\mu y)g(y,\vec{a})$ —equivalently, the **program segment** "**while** $g(\mathbf{x},\vec{a}) \neq 0$ **do**"— *unbounded search*.

Why "unbounded" search? Because we do not know *a priori* how many times we have to go around the loop. This depends on the behavior of g.

We saw how the minimum can fail to exist in one of two ways:

- Either $g(x,\vec{a}) \downarrow$ for all x but we *never* get $g(x,\vec{a}) = 0$; that is, we stay in the loop *going round and round forever*

  or

- $g(b,\vec{a}) \uparrow$ for a value b of x *before* we reach *any* c such that $g(c,\vec{a}) = 0$, thus we are *stuck forever processing the call* $g(b,\vec{a})$ *in the* ***while*** *instruction.*

Can we implement the pseudo-program (1) as a URM F? YES!

2.4.20 Example. (Unbounded Search on a URM) So suppose again that $\lambda x\vec{y}.g(x,\vec{y})$ is partial computable, and, say, $g = G^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$.

By earlier remarks we may assume that $\vec{\mathbf{y}}$ and $\mathbf{x}$ are read-only in G and that $\mathbf{z}$ is *not* one of them.

Consider the following program $F^{\vec{\mathbf{y}}}_{\mathbf{x}}$, where G' is the program G with the **stop** instruction removed, and the instructions of G being renumbered (and if-statements adjusted) as needed so that its first instruction has label 2.

$1:$ $\quad \mathbf{x} \leftarrow 0$

$\quad\quad\;\; G'^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$

$k:$ $\quad$ **if** $\mathbf{z} = 0$ **goto** $k+l+3$ **else goto** $k+1$

$k+1:$ $\quad \mathbf{w}_1 \leftarrow 0$

$\vdots$

$k+l:$ $\quad \mathbf{w}_l \leftarrow 0$

$k+l+1:$ $\mathbf{x} \leftarrow \mathbf{x}+1$

$k+l+2:$ **goto** 2

$k+l+3:$ **stop** {**Comment.** Read answer off $\mathbf{x}$.
This is the last $\mathbf{x}$-value used by G'}

□

The result of Example 2.4.20 yields at once:

2.4.21 Proposition. $\mathcal{P}$ is closed under unbounded search; that is, if $\lambda x\vec{y}.g(x,\vec{y})$ is in $\mathcal{P}$, then so is $\lambda\vec{y}.(\mu x)g(x,\vec{y})$.

2.4.22 Example. Is the function $U_i^n = \lambda\vec{x}_n.x_i$, where $1 \leq i \leq n$, in $\mathcal{P}$? Yes, and here is a program, M, for it:

$1:$ $\quad \mathbf{w}_1 \leftarrow 0$
$\vdots$
$i:$ $\quad \mathbf{z} \leftarrow \mathbf{w}_i$ {**Comment.** Cf. Exercise 2.4.8}
$\vdots$
$n:$ $\quad \mathbf{w}_n \leftarrow 0$
$n+1:$ **stop**

$\lambda\vec{x}_n.x_i = M_{\mathbf{z}}^{\vec{\mathbf{w}}_n}$. To ensure that M indeed *has all* the $\mathbf{w}_i$ as variables we reference them in instructions at least once, in any manner whatsoever. □

Chapter 3

Primitive Recursive Functions

Sept. 22, 2021

3.1 $\mathcal{PR}$-Derivations and $\mathcal{PR}$-Functions

Before we get more immersed into *partial functions* let us **redefine equality for function calls**.

3.1.1 Definition. Suppose that $\lambda\vec{x}.f(\vec{x}_n)$ and $\lambda\vec{y}.g(\vec{y}_m)$ are number-theoretic.

We extend the notion of equality $f(\vec{a}_n) = g(\vec{b}_m)$ to include the case of *undefined calls*:

For any $\vec{a}_n$ and $\vec{b}_m$, $f(\vec{a}_n) = g(\vec{b}_m)$ means *precisely one of*

- For some $k \in \mathbb{N}$, $f(\vec{a}_n) = k$ and $g(\vec{b}_m) = k$

- $f(\vec{a}_n)\uparrow$ and $g(\vec{b}_m)\uparrow$

In short,

$$f(\vec{a}_n) = g(\vec{b}_m) \equiv (\exists z)\Big(f(\vec{a}_n) = z \wedge g(\vec{b}_m) = z \vee f(\vec{a}_n)\uparrow \wedge g(\vec{b}_m)\uparrow\Big)$$

□

The definition is due to Kleene and he preferred, as I do in the text, but not in these Notes, to use a new symbol for the extended equality, namely $\simeq$.

3.1.2 Lemma. *If $f = prim(h, g)$ and h and g are **total**, then so is f.*

Proof. ▶ *We did this*, but do read this alternative proof!
Let f be given by:

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

We do induction on x to prove

$$\text{“For all } x, \vec{y},\ f(x, \vec{y}) \downarrow\text{”} \qquad (*)$$

Basis. $x = 0$: Well, $f(0, \vec{y}) = h(\vec{y})$, but $h(\vec{y}) \downarrow$ for all $\vec{y}$, so

$$f(0, \vec{y}) \downarrow \text{ for all } \vec{y} \qquad (**)$$

As I.H. (Induction *Hypothesis*) take that

$$f(x, \vec{y}) \downarrow \text{ for all } \vec{y} \text{ and } \textit{fixed } x \qquad (\dagger)$$

Do the Induction *Step* (I.S.) to **show**

$$f(x+1, \vec{y}) \downarrow \text{ for all } \vec{y} \text{ and the fixed } x \text{ of } (\dagger) \qquad (\ddagger)$$

Well, by (†) and the assumption on g,

$$g\big(x, \vec{y}, f(x, \vec{y})\big) \downarrow, \text{ for all } \vec{y} \text{ and the fixed } x \text{ of } (\dagger)$$

which says the same thing as (‡). □

3.1.3 Corollary. *$\mathcal{R}$ is closed under primitive recursion.*

Proof. Let h and g be in $\mathcal{R}$. Then they are in $\mathcal{P}$. But then $prim(h, g) \in \mathcal{P}$ as we showed in class/text and Notes.

By 3.1.2, $prim(h, g)$ is total.

By definition of $\mathcal{R}$, as **the subset of $\mathcal{P}$ that contains all total functions of $\mathcal{P}$**, we have $prim(h, g) \in \mathcal{R}$. □

Why all this dance **in colour** above? Because to prove $f \in \mathcal{R}$ you need **TWO** things: That

1. $f \in \mathcal{P}$

   AND

2. f is total

But aren't all the *total* functions in $\mathcal{R}$ anyway?

NO! They *need to be computable too!*

We will see in this course soon that NOT all total functions are computable!

3.2 The Class of Primitive Recursive Functions Defined (*at last!*)

We saw that

1. The successor —S

2. zero —Z

3. and the *generalised identity* functions —$U_i^n = \lambda \vec{x}_n . x_i$

   are all in $\mathcal{P}$

BECAUSE Each is computable by a URM.

▶ We have also shown that "*computability*" of functions is **preserved** by the operations of **composition**, **primitive recursion**, and **unbounded search**.

In this section we will explore the properties of the important **SUB**set of $\mathcal{P}$ known as the **primitive recursive functions**.

▶ Most people introduce them via **derivations** just *as one introduces the theorems of logic via proofs*, as in the definition below.

3.2.1 Definition. (**$\mathcal{PR}$-derivations; $\mathcal{PR}$-functions**)
The set

$$\mathcal{I} = \left\{ S, Z, \left(U_i^n\right)_{n \geq i > 0} \right\}$$

is the set of **Initial** $\mathcal{PR}$ functions.

A *$\mathcal{PR}$-derivation* is a *finite* (ordered!) *sequence* of *number-theoretic functions*

$$f_1, f_2, f_3, \ldots, f_i, \ldots, f_n \tag{1}$$

such that, for **each** i, *one* of the following holds

1. $f_i \in \mathcal{I}$.
2. $f_i = prim(f_j, f_k)$ and $j < i$ and $k < i$ —that is, f_j, f_k appear **to the left of** f_i in (1).
3. $f_i = \lambda \vec{y}.g\big(r_1(\vec{y}), r_2(\vec{y}), \ldots, r_m(\vec{y})\big)$, and **all** of the $\lambda \vec{y}.r_q(\vec{y})$ and $\lambda \vec{x}_m.g(\vec{x}_m)$ appear **to the left of** f_i in (1).

Any f_i in a derivation is called a **derived** function.*

*The set of primitive recursive functions, $\mathcal{PR}$, is **all those that are derived***:

$$\mathcal{PR} \overset{Def}{=} \{f : f \text{ is derived}\} \qquad \square$$

The above definition defines essentially what Dedekind ([Ded88]) called "*recursive*" functions.

*Strictly speaking, *primitive recursively derived*, but we will not consider other sets of derived functions in this section, so we omit the qualification.

Subsequently they were renamed to *primitive recursive* (by Kleene) allowing the unqualified term *recursive* to be synonymous with (total) *computable* and apply to the functions of $\mathcal{R}$.

Facts about derivations

3.2.2 Lemma. *The concatenation of two derivations is a derivation.*

Proof. Let

$$f_1, f_2, f_3, \ldots, f_i, \ldots, f_n \tag{1}$$

and

$$g_1, g_2, g_3, \ldots, g_j, \ldots, g_m \tag{2}$$

be two derivations. Then so is

$$f_1, f_2, f_3, \ldots, f_i, \ldots, f_n, g_1, g_2, g_3, \ldots, g_j, \ldots, g_m$$

because of the fact that each of the f_i and g_j satisfies the three cases of Definition 3.2.1 in the standalone derivations (1) and (2). But this property of the f_i and g_j *is preserved* after concatenation. □

3.2.3 Corollary. *The concatenation of any finite number of derivations is a derivation.*

Proof. True if we have two derivations, by 3.2.2. A third derivation can be appended at the end of this result and we still have a derivation by 3.2.2. Ditto with a fourth derivation.

At this stage of the argument you either say "ETC." or decide to do **induction** on the number of derivations, n, that we are talking about (the Basis is 3.2.2, for $n = 2$). □

3.2.4 Lemma. *If*

$$f_1, f_2, f_3, \ldots, f_k, f_{k+1}, \ldots, f_n$$

is a derivation, then so is $f_1, f_2, f_3, \ldots, f_k$.

Proof. In $f_1, f_2, f_3, \ldots, f_k$ every f_m, for $1 \leq m \leq k$, satisfies 1.–3. of Definition 3.2.1 since all conditions are in terms of what f_m is, or what lies **to the left of** f_m. Chopping the "tail" $f_{k+1}, \ldots, f_n$ in no way affects what lies to the left of f_m, for $1 \leq m \leq k$. □

3.2.5 Corollary. *$f \in \mathcal{PR}$ iff f appears at the **end** of some derivation.*

Proof.

(a) The *If*. Say $g_1, \ldots, g_n, \boxed{f}$ is a derivation. Since f occurs in it, $f \in \mathcal{PR}$ by 3.2.1.

(b) The *Only If*. Say $f \in \mathcal{PR}$. Then, by 3.2.1,

$$g_1, \ldots, g_m, \boxed{f}, g_{m+2}, \ldots, g_r \tag{1}$$

for some derivation like the (1) above.

By 3.2.4, $g_1, \ldots, g_m, \boxed{f}$ is also a derivation. □

3.2.6 Theorem. *$\mathcal{PR}$ is closed under composition and primitive recursion.*

Proof.

- Closure under **primitive recursion**. So let $\lambda\vec{y}.h(\vec{y})$ and $\lambda x\vec{y}z.g(x,\vec{y},z)$ be in $\mathcal{PR}$ and $f = prim(h,g)$. Thus we have derivations

$$h_1, h_2, h_3, \ldots, h_n, \boxed{h} \tag{1}$$

and

$$g_1, g_2, g_3, \ldots, g_m, \boxed{g} \tag{2}$$

Then the following is a derivation by 3.2.2.

$$h_1, h_2, h_3, \ldots, h_n, \boxed{h}, g_1, g_2, g_3, \ldots, g_m, \boxed{g}$$

Therefore so is

$$h_1, h_2, h_3, \ldots, h_n, \boxed{h}, g_1, g_2, g_3, \ldots, g_m, \boxed{g}, prim(h,g)$$

by applying step 2 of Definition 3.2.1.

This implies $prim(h,g) \in \mathcal{PR}$ by 3.2.1.

- **Similarly**, closure under **composition**. So let $\lambda\vec{y}.h(\vec{x}_n)$ and $\lambda\vec{y}.g_i(\vec{y})$, for $1 \leq i \leq n$, be in $\mathcal{PR}$. By 3.2.1 we have derivations

$$\boxed{\ldots, \boxed{h}} \tag{3}$$

and

$$\boxed{\ldots, \boxed{g_i}}, \text{ for } 1 \leq i \leq n \tag{4}$$

By 3.2.2,

$$\boxed{\ldots, \boxed{h}}, \boxed{\ldots, \boxed{g_1}}, \ldots, \boxed{\ldots, \boxed{g_n}}$$

is a derivation, and by 3.2.1, case 3, so is

$$\boxed{\ldots, \boxed{h}}, \boxed{\ldots, \boxed{g_1}}, \ldots, \boxed{\ldots, \boxed{g_n}}, \lambda\vec{y}.h(g_1(\vec{y}), \ldots, g_n(\vec{y}))$$

This implies $\lambda\vec{y}.h(g_1(\vec{y}), \ldots, g_n(\vec{y})) \in \mathcal{PR}$ *by 3.2.1.* □

3.2.7 Remark. *How do you prove that some $f \in \mathcal{PR}$?*

Answer. By building a derivation

$$g_1, \ldots, g_m, \boxed{f}$$

After a while this becomes easier because

▶ you might **know** an h and g in $\mathcal{PR}$ such that $f = prim(h, g)$,

▶ or you might know some $g, h_1, \ldots, h_m$ in $\mathcal{PR}$, such that $f = \lambda\vec{y}.g\big(h_1(\vec{y}), \ldots, h_m(\vec{y})\big)$.

If so, just apply 3.2.6.

How do you prove that *ALL* $f \in \mathcal{PR}$ have a property Q —that is, for all f, $Q(f)$ is true?

Answer. *By doing* ***induction on the derivation length*** *of f.* □

Here are two examples demonstarting the above questions and their answers.

3.2.8 Example. (1) To demonstrate the first Answer above (3.2.7), show (prove) that $\lambda xy.x + y \in \mathcal{PR}$. Well, observe that

$$0 + y = y$$
$$(x + 1) + y = (x + y) + 1$$

Does the above look like a primitive recursion?

Well, almost.

However, the *first equation* should have a *function call* "$H(y)$" on the rhs but instead has just a *variable* y —an input!

Also the *second equation* should have a rhs like "$G(x, y, x + y)$".

We can do that!

Take $H = U_1^1$ *and* $G = SU_3^3$ —**NOTE** the "SU_3^3" with no brackets around U_3^3; this is normal practise!

Be sure to agree that we now have

- $$0 + y = H(y)$$
$$(x + 1) + y = G\Big(x, y, x + y\Big)$$

- The functions $H = U_1^1$ (*initial*) and $G = SU_3^3$ (*composition*) are in $\mathcal{PR}$. By 3.2.6 so is $\lambda xy.x+y$.

  In terms of derivations, we have produced the derivation:

$$U_1^1, S, U_3^3, SU_3^3, \underbrace{prim\left(U_1^1, SU_3^3\right)}_{\lambda xy.x+y}$$

(2) To demonstrate the second Answer above (3.2.7), show (prove) that every $f \in \mathcal{PR}$ is **total**. Induction on derivation length, n, where f occurs.

Basis. $n = 1$. Then f is the only function in the derivation. Thus it must be one of S, Z, or U_i^m. But all these are total.

I.H. (Induction Hypothesis) *Fix an l.* Assume that the claim is true for all f that occur *at the end of derivations of lengths* $n \leq l$. That is, *we assume that all such f are total.*

I.S. (Induction Step) Prove that the claim is true for all f that occur at the *end of a derivation* —see 3.2.5— of length $n = l + 1$.

$$g_1, \ldots, g_l, \boxed{f} \tag{1}$$

We have three subcases:

- $f \in \mathcal{I}$. But we argued this under *Basis*.

- $f = prim(h, g)$, where h and g are among the $g_1, \ldots, g_l$. By the I.H. h and g are total. Elaboration: Any such g_i is at the end of a derivation of length $\leq l$. So I.H. kicks in.

  But then so is f by Lemma 3.1.2.

- **Similarly, let** $f = \lambda\vec{y}.h\Big(q_1(\vec{y}), \ldots, q_t(\vec{y})\Big)$, where the functions h and $q_1, \ldots, q_t$ are among the $g_1, \ldots, g_l$.

By the I.H. h and $q_1, \ldots, q_t$ are total. But then so is f by a Lemma in an earlier Note, where we proved that $\mathcal{R}$ is closed under composition. □

3.3 A few examples of what we can do with $\mathcal{PR}$ functions

3.3.1 Example. If $\lambda xyw.f(x,y,w)$ and $\lambda z.g(z)$ are in $\mathcal{PR}$,

how about $\lambda xzw.f(x,g(z),w)$*?*

It is in $\mathcal{PR}$ since, by *COMPOSITION*,

$$f(x,g(z),w) = f(U_1^3(x,z,w), \underline{g(U_2^3(x,z,w))}, U_3^3(x,z,w))$$

and the U_i^n are all primitive recursive.

The reader will see at once that to the right of "=" we have correctly formed compositions as expected by the "rigid" definition of composition given in these Notes.

Similarly, for the same functions above,

(1) $\lambda yw.f(2,y,w)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$f(2,y,w) = f\Big(SSZw, y, w\Big)$$

where I wrote "$SSZ(\ldots)$" as short for $S(S(Z(\ldots)))$ *for visual clarity*.

Clearly, using $SSZ\big(U_2^2(y,w)\big)$ above works as well.

(2) $\lambda xyw.\underbrace{f(y,x,w)}_{\text{the rule part}}$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$f(y,x,w) = f\Big(U_2^3(x,y,w), U_1^3(x,y,w), U_3^3(x,y,w)\Big)$$

In this connection, note that while $\lambda xy.g(x,y) = \lambda yx.g(y,x)$, yet $\lambda xy.g(x,y) \neq \lambda xy.g(y,x)$ in general.

▶ For example, the "rule" in $\lambda xy.x \dot{-} y$ asks that we subtract the second input (y) from the first (x), but in $\lambda xy.y \dot{-} x$ it asks that we subtract the first input (x) from the second (y).

(3) $\lambda xy.f(x,y,x)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$f(x,y,x) = f\big(U_1^2(x,y), U_2^2(x,y), U_1^2(x,y)\big)$$

(4) $\lambda xyzwu.f(x,y,w)$ is in $\mathcal{PR}$. Indeed, this function can be obtained by composition, since

$$\lambda xyzwu.f(x,y,w) = \lambda xyzwu.f(U_1^5(x,y,z,w,u), U_2^5(x,y,z,w,u), U_4^5(x,y,z,w,u))$$

□

The above four examples are summarised, named, and generalised in the following straightforward exercise:

3.3.2 Exercise. (The [Grz53] Substitution Operations) $\mathcal{PR}$ is closed under the following operations:

(i) *Substitution of a function invocation for a variable*:
From $\lambda\vec{x}y\vec{z}.f(\vec{x},y,\vec{z})$ and $\lambda\vec{w}.g(\vec{w})$ obtain $\lambda\vec{x}\vec{w}\vec{z}.f(\vec{x},g(\vec{w}),\vec{z})$.

(ii) *Substitution of a constant for a variable*:
From $\lambda\vec{x}y\vec{z}.f(\vec{x},y,\vec{z})$ obtain $\lambda\vec{x}\vec{z}.f(\vec{x},k,\vec{z})$.

(iii) *Interchange of two variables*:
From $\lambda\vec{x}y\vec{z}w\vec{u}.f(\vec{x},y,\vec{z},w,\vec{u})$ obtain $\lambda\vec{x}y\vec{z}w\vec{u}.f(\vec{x},w,\vec{z},y,\vec{u})$.

(iv) *Identification of two variables*:
From $\lambda\vec{x}y\vec{z}w\vec{u}.f(\vec{x},y,\vec{z},w,\vec{u})$ obtain $\lambda\vec{x}y\vec{z}\vec{u}.f(\vec{x},y,\vec{z},y,\vec{u})$.

(v) *Introduction of "don't care" variables*:
From $\lambda\vec{x}.f(\vec{x})$ obtain $\lambda\vec{x}\vec{z}.f(\vec{x})$. □

By 3.3.2 composition can simulate the Grzegorczyk operations if the initial functions $\mathcal{I}$ are present.

Of course, (i) alone can in turn simulate composition. With these comments out of the way, we see that the "rigidity" of the definition of composition is gone.

3.3.1 Nothing is really "rigid"

3.3.3 Example. The *definition* of primitive recursion is also "rigid". *However this is also an illusion* since the practice is very flexible.

Take $p(0) = 0$ and $p(x+1) = x$ —this one defining $p = \lambda x.x \dot{-} 1$ —does not fit the schema.

The schema requires *the defined function* to have *one more variable than the basis*, so no one-variable function can be directly defined!

We can get around this.

Define first $\widetilde{p} = \lambda xy.x \dot{-} 1$ as follows: $\widetilde{p}(0, y) = 0$ and $\widetilde{p}(x+1, y) = x$.

Now this can be dressed up according to the syntax of the schema,

$$\begin{aligned} \widetilde{p}(0, y) &= Z(y) \\ \widetilde{p}(x+1, y) &= U_1^3(x, y, \widetilde{p}(x, y)) \end{aligned}$$

that is, $\widetilde{p} = prim(Z, U_1^3)$.

Then we can get p *by (Grzegorczyk) substitution:* $p = \lambda x.\widetilde{p}(x, 0)$.

Incidentally, this shows that both p and $\widetilde{p}$ are in $\mathcal{PR}$:

- $\widetilde{p} = prim(Z, U_1^3)$ is in $\mathcal{PR}$ since Z and U_1^3 are, then invoking 3.2.6.

- $p = \lambda x.\widetilde{p}(x, 0)$ is in $\mathcal{PR}$ since $\widetilde{p}$ is, then invoking 3.3.2.

Another rigidity in the definition of primitive recursion is that, apparently, one can use only the first variable as the iterating variable.

Not so. This is also an illusion.

▶ Consider, for example, $sub = \lambda xy.x \dot{-} y$, hence $x \dot{-} 0 = x$ and $x \dot{-} (y+1) = p(x \dot{-} y)$

Clearly, $sub(x,0) = x$ *and* $sub(x, y+1) = p(sub(x,y))$ *is correct semantically,* but the **format** is wrong:

We are *not supposed to iterate along the second variable*! Well, we CAN!!!

Define instead $\widetilde{sub} = \lambda xy.y \dot{-} x$*:*

So

$$\begin{aligned} y \dot{-} 0 &= y \\ y \dot{-} (x+1) &= p\Big(y \dot{-} x\Big) \end{aligned}$$

That is,

$$\begin{aligned} \widetilde{sub}(0,y) &= U_1^1(y) \\ \widetilde{sub}(x+1,y) &= p\big(U_3^3(x,y,\widetilde{sub}(x,y))\big) \end{aligned}$$

Then, using variable swapping [Grzegorczyk operation (iii)], ***Similarly****, we can get sub*:

$$sub = \lambda xy.\widetilde{sub}(y, x).$$

Clearly, both $\widetilde{sub}$ and sub are in $\mathcal{PR}$. □

3.3.4 Exercise. Prove that $\lambda xy.x \times y$ is primitive recursive. Of course, we will usually write multiplication $x \times y$ in "implied notation", xy. □

3.3.5 Example. *The very important* "switch" *(or "if-then-else") function*

$sw = \lambda xyz.$if $x = 0$ then y else z

is primitive recursive.

It is directly obtained by primitive recursion on initial functions: $sw(0, y, z) = y$ and $sw(x + 1, y, z) = z$.

Sept. 27, 2021

Dressed up this is:

$sw(0, y, z) = U_1^2(y, z)$
$sw(x + 1, y, z) = U_3^4(x, y, z, sw(x, y, z))$ □

3.3.6 Exercise. $\mathcal{PR} \subseteq \mathcal{R}$.

Hint. Use induction on $\mathcal{PR}$-derivation lengths to show that "**If $f \in \mathcal{PR}$, then $f \in \mathcal{R}$.**" □

 Indeed, the above inclusion is proper, as we will see later. 

3.3.7 Example. Consider the exponential function x^y given by

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= x^y x \end{aligned}$$

Thus,

if $x = 0$ and $y = 0$, then $x^y = 1$, but $x^y = 0$ for all $y > 0$ if $x = 0$.

BUT note that x^y is "mathematically" undefined when $x = y = 0$.[†]

Thus, by Exercise 3.3.6 *the exponential cannot be a primitive recursive function*!

This is rather *preposterous*, since the *computational process for the exponential is trivial*; thus it is *ridiculous* to allow this function be non-$\mathcal{PR}$.

After all, we know *exactly where and how it is undefined* and we can remove this undefinability by *redefining "x^y" so that "$0^0 = 1$"*.

[†]In first-year university calculus we learn that "0^0" is an "indeterminate form".

In computability we do this kind of redefinition a lot in order to remove *easily recognisable points of "non definition" of calculable functions*.

We will see further examples, such as the remainder, quotient, and logarithm functions.

BUT also examples where we CANNOT do this; and WHY. □ 

For those who would rather not "change mathematics" they can take this point of view if it restores their peace of mind:

The function *we want and talk about here* is *NOT* the exponential of algebra and calculus, but a *DIFFERENT ONE* that at input $(x, y) = (0, 0)$ yields 1 but for all other inputs yields the "standard" "x^y". You can give it a different name if you want, as I do in the text.

3.4 Primitive Recursive Relations

3.4.1 Definition. A relation $R(\vec{x})$ is (*primitive*) *recursive* iff its *characteristic function*,

$$c_R = \lambda \vec{x}. \begin{cases} 0 & \text{if } R(\vec{x}) \\ 1 & \text{if } \neg R(\vec{x}) \end{cases}$$

is (primitive) recursive. *The set of all primitive recursive (respectively, recursive)* relations *is denoted by* $\mathcal{PR}_*$ *(respectively,* $\mathcal{R}_*$*).* □

Computability theory practitioners often call relations *predicates*.

It is clear that one can go from relation to characteristic function and back in a unique way,

▶ Thus, *we may think of relations as "0-1 valued" functions*.

The concept of relation is not theoretically necessary but it greatly *simplifies* the further development of the theory of primitive recursive functions.

The following is useful:

3.4.2 Proposition. *$R(\vec{x}) \in \mathcal{PR}_*$ iff some $f \in \mathcal{PR}$ exists such that, for all $\vec{x}$, $R(\vec{x}) \equiv f(\vec{x}) = 0$.*

Proof. *For the* if-*part*. Here I know that $f \in \mathcal{PR}$ and $R(\vec{x}) \equiv f(\vec{x}) = 0$.

I want $c_R \in \mathcal{PR}$.

This is so since $c_R = \lambda\vec{x}.1 \dot{-} (1 \dot{-} f(\vec{x}))$ (using Grzegorczyk substitution and $\lambda xy.x \dot{-} y \in \mathcal{PR}$; cf. 3.3.3).

For the only if-*part*, $f = c_R$ will do: $R(\vec{x}) \equiv c_R(\vec{x}) = 0$. □

3.4.3 Corollary. *Same for $\mathcal{R}_*$: $R(\vec{x}) \in \mathcal{R}_*$ iff some $f \in \mathcal{R}$ exists such that, for all $\vec{x}$, $R(\vec{x}) \equiv f(\vec{x}) = 0$.*

Proof. By the above proof, and 3.3.6 which puts monus in $\mathcal{R}$ as well (and $\mathcal{R}$ is closed under Grzegorczyk Ops). □

3.4.4 Corollary. $\mathcal{PR}_* \subseteq \mathcal{R}_*$.

Proof. By the above corollary and 3.3.6. □

3.4.5 Theorem. *$\mathcal{PR}_*$ is closed under the Boolean operations.*

Proof. It suffices to look at the cases of $\neg$ and $\vee$, since $R \to Q \equiv \neg R \vee Q$, $R \wedge Q \equiv \neg(\neg R \vee \neg Q)$ and $R \equiv Q$ is short for $(R \to Q) \wedge (Q \to R)$.

($\neg$) Say, $R(\vec{x}) \in \mathcal{PR}_*$. Thus (3.4.1), $c_R \in \mathcal{PR}$. But then $c_{\neg R} \in \mathcal{PR}$, since $c_{\neg R} = \lambda\vec{x}.1 \dot{-} c_R(\vec{x})$, by Grzegorczyk substitution and $\lambda xy.x \dot{-} y \in \mathcal{PR}$.

($\vee$) Let $R(\vec{x}) \in \mathcal{PR}_*$ and $Q(\vec{y}) \in \mathcal{PR}_*$. Then $\lambda\vec{x}\vec{y}.c_{R\vee Q}(\vec{x},\vec{y})$ is given by

$$c_{R\vee Q}(\vec{x},\vec{y}) = \text{if } R(\vec{x}) \text{ then } 0 \text{ else } c_Q(\vec{y})$$

Thus

$$c_{R\vee Q}(\vec{x},\vec{y}) = \text{if } c_R(\vec{x}) = 0 \text{ then } 0 \text{ else } c_Q(\vec{y})$$

and therefore is in $\mathcal{PR}$. □

"if $R(\vec{x})$" above means "if $c_R(\vec{x}) = 0$"

3.4.6 Remark. *Alternatively, for the $\vee$ case above, note that $c_{R\vee Q}(\vec{x},\vec{y}) = c_R(\vec{x}) \times c_Q(\vec{y})$ and invoke 3.3.4.* □

3.4.7 Corollary. $\mathcal{R}_*$ is closed under the Boolean operations.

Proof. As above, mindful of 3.3.6. □

3.4.8 Example. The relations $x \leq y$, $x < y$, $x = y$ are in $\mathcal{PR}_*$.

An addendum to λ notation: Absence of λ is allowed ONLY for relations! Then it means (the absence, that is) that ALL variables are active input!

Note that $x \leq y \equiv x \dot{-} y = 0$ and invoke 3.4.2. Finally invoke Boolean closure and note that $x < y \equiv \neg y \leq x$ while $x = y$ is equivalent to $x \leq y \wedge y \leq x$. □

Chapter 4

Thor's Hammer (Bounded Search and Friends)

4.1 Bounded Quantification and Search

4.1.1 Proposition. *If $R(\vec{x}, y, \vec{z}) \in \mathcal{PR}_*$ and $\lambda\vec{w}.f(\vec{w}) \in \mathcal{PR}$, then $R(\vec{x}, f(\vec{w}), \vec{z})$ is in $\mathcal{PR}_*$.*

Proof. Let $g \in \mathcal{PR}$ such that

$$R(\vec{x}, y, \vec{z}) \equiv g(\vec{x}, y, \vec{z}) = 0, \text{ for all } \vec{x}, y, \vec{z}$$

Then

$$R(\vec{x}, f(\vec{w}), \vec{z}) \equiv g(\vec{x}, f(\vec{w}), \vec{z}) = 0, \text{ for all } \vec{x}, \vec{w}, \vec{z}$$

Since $\lambda\vec{x}\vec{w}\vec{z}.g(\vec{x}.f(\vec{w}), \vec{z}) \in \mathcal{PR}$ by Grzegorczyk Ops, we have that $R(\vec{x}, f(\vec{w}), \vec{z}) \in \mathcal{PR}_*$. □

4.1.2 Proposition. *If* $R(\vec{x}, y, \vec{z}) \in \mathcal{R}_*$ *and* $\lambda\vec{w}.f(\vec{w}) \in \mathcal{R}$*, then* $R(\vec{x}, f(\vec{w}), \vec{z})$ *is in* $\mathcal{R}_*$*.*

Proof. Similar to that of 4.1.1. □

4.1.3 Corollary. *If $f \in \mathcal{PR}$ (respectively, in $\mathcal{R}$), then its* graph, *$z = f(\vec{x})$ is in $\mathcal{PR}_*$ (respectively, in $\mathcal{R}_*$).*

Proof. Using the relation $z = y$ and 4.1.1. □

4.1.4 Exercise. Using unbounded search, prove that if $z = f(\vec{x})$ is in $\mathcal{R}_*$ and f is total, then $f \in \mathcal{R}$. □

4.1.5 Definition. (Bounded Quantifiers) The abbreviations

$(\forall y)_{<z} R(y, \vec{x})$

$(\forall y)_{y<z} R(y, \vec{x})$

$(\forall y < z) R(y, \vec{x})$

all stand for

$$(\forall y)\big(y < z \to R(y, \vec{x})\big)$$

while correspondingly,

$(\exists y)_{<z} R(y, \vec{x})$

$(\exists y)_{y<z} R(y, \vec{x})$

$(\exists y < z) R(y, \vec{x})$

all stand for

$$(\exists y)\big(y < z \wedge R(y, \vec{x})\big)$$

Similarly for the non strict inequality "$\leq$". □

4.1.6 Theorem. *$\mathcal{PR}_*$ is closed under bounded quantification.*

Proof. *By logic it suffices to look at the case of $(\exists y)_{<z}$* since $(\forall y)_{<z}R(y,\vec{x}) \equiv \neg(\exists y)_{<z}\neg R(y,\vec{x})$.

Let then $R(y,\vec{x}) \in \mathcal{PR}_*$ and *let us give the name $Q(z,\vec{x})$ to*

$(\exists y)_{<z}R(y,\vec{x})$ for convenience.

We note that $Q(0,\vec{x})$ is false (why?) and logic says:

$$\overbrace{Q(z+1,\vec{x})}^{R(0,\vec{x})\vee R(1,\vec{x})\vee\ldots R(z-1,\vec{x})\vee R(z,\vec{x})} \equiv \overbrace{Q(z,\vec{x})}^{R(0,\vec{x})\vee R(1,\vec{x})\vee\ldots R(z-1,\vec{x})} \vee R(z,\vec{x}).$$

Thus, as the following prim. rec. shows, $c_Q \in \mathcal{PR}$.

$$\begin{aligned} c_Q(0,\vec{x}) &= 1 \\ c_Q(z+1,\vec{x}) &= c_Q(z,\vec{x})c_R(z,\vec{x}) \end{aligned}$$ □

4.1.7 Corollary. *$\mathcal{R}_*$ is closed under bounded quantification.*

Sept. 29, 2021

4.1.8 Definition. (Bounded Search) Let f be a **total** number-theoretic function of $n+1$ variables.

The symbol $(\mu y)_{<z} f(y, \vec{x})$, for all $z, \vec{x}$, stands for

$$\begin{cases} \min\{y : y < z \wedge f(y, \vec{x}) = 0\} & \text{if } (\exists y)_{<z} f(y, \vec{x}) = 0 \\ z & \text{otherwise} \end{cases}$$

So,

unsuccessful search returns the first number to the right of the search-range.

We define "$(\mu y)_{\leq z}$" to mean "$(\mu y)_{<z+1}$". □

4.1.9 Theorem. *$\mathcal{PR}$ is closed under the bounded search operation $(\mu y)_{<z}$. That is, if $\lambda y\vec{x}.f(y,\vec{x}) \in \mathcal{PR}$, then $\lambda z\vec{x}.(\mu y)_{<z} f(y,\vec{x}) \in \mathcal{PR}$.*

Proof. Set $g = \lambda z\vec{x}.(\mu y)_{<z} f(y,\vec{x})$ for convenience.

Then the following primitive recursion settles it:

*Recall that "***if** $R(\vec{z})$ **then** y **else** w*" means "***if** $c_R(\vec{z}) = 0$ **then** y **else** w*".*

Note

$$0, 1, 2, \ldots, z-1, z = \overbrace{0, 1, 2, \ldots, z-1}^{<z}, z$$

So

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

Why 0 above?

$$g(z+1,\vec{x}) = \text{if } \overbrace{(\exists y)_{<z}\Big(f(y,\vec{x}) = 0\Big)}^{F(z,\vec{x})} \text{ then } g(z,\vec{x})$$
$$\text{else if } f(z,\vec{x}) = 0 \text{ then } z$$
$$\text{else } z+1$$

□

Since the first line depicts a $\mathcal{PR}_*$ predicate (graph of f), so do the remaining lines from what we learnt before:

$$f(y, \vec{x}) = w$$

$$f(y, \vec{x}) = 0$$

$$(\exists y)_{<z} f(y, \vec{x}) = 0$$

The last one is "$F(z, \vec{x})$"

4.1.10 Corollary. *$\mathcal{PR}$ is closed under the bounded search operation $(\mu y)_{\leq z}$.*

4.1.11 Exercise. *Prove the corollary.* □

4.1.12 Corollary. *$\mathcal{R}$ is closed under the bounded search operations $(\mu y)_{<z}$ and $(\mu y)_{\leq z}$.*

Consider now a set of *mutually exclusive* relations $R_i(\vec{x})$, $i = 1, \ldots, n$, that is, $R_i(\vec{x}) \wedge R_j(\vec{x})$ *is* false, *for each* $\vec{x}$ *as long as* $i \neq j$.

Then we can define a function f *by cases* R_i from given functions f_j by the requirement (for all $\vec{x}$) given below:

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } R_1(\vec{x}) \\ f_2(\vec{x}) & \text{if } R_2(\vec{x}) \\ \ldots & \ldots \\ f_n(\vec{x}) & \text{if } R_n(\vec{x}) \\ f_{n+1}(\vec{x}) & \text{otherwise} \end{cases}$$

where, as is usual in mathematics, "if $R_j(\vec{x})$*" is short for "if* $R_j(\vec{x})$ *is true"*

and the *"otherwise" is the condition* $\neg(R_1(\vec{x}) \vee \cdots \vee R_n(\vec{x}))$.

We have the following result:

4.1.13 Theorem. (Definition by Cases) *If the functions f_i, $i = 1, \ldots, n+1$ and the relations $R_i(\vec{x})$, $i = 1, \ldots, n$ are in $\mathcal{PR}$ and $\mathcal{PR}_*$, respectively, then so is f above.*

Proof. By repeated use (Grz Ops) of if-then-else. So,

$$
\begin{aligned}
f(\vec{x}) = \text{if} \quad & R_1(\vec{x}) \text{ then } f_1(\vec{x}) \\
\text{else if } & R_2(\vec{x}) \text{ then } f_2(\vec{x}) \\
\vdots & \\
\text{else if } & R_n(\vec{x}) \text{ then } f_n(\vec{x}) \\
\text{else} & \qquad\qquad f_{n+1}(\vec{x})
\end{aligned}
$$

□

4.1.14 Corollary. Same statement as above, replacing $\mathcal{PR}$ and $\mathcal{PR}_*$ by $\mathcal{R}$ and $\mathcal{R}_*$, respectively.

The tools we now have at our disposal allow easy certification of the primitive recursiveness of some very useful functions and relations. But first a definition:

4.1.15 Definition. $(\mu y)_{<z}R(y,\vec{x})$ means $(\mu y)_{<z}c_R(y,\vec{x})$. □

► Thus, if $R(y,\vec{x}) \in \mathcal{PR}_*$ (resp. $\in \mathcal{R}_*$), then $\lambda z\vec{x}.(\mu y)_{<z}R(y,\vec{x}) \in \mathcal{PR}$ (resp. $\in \mathcal{R}$), since $c_R \in \mathcal{PR}$ (resp. $\in \mathcal{R}$).

4.1.16 Example. *The following are in $\mathcal{PR}$ or $\mathcal{PR}_*$ as appropriate*:

(1) $\lambda xy.\lfloor x/y \rfloor^*$ *(the quotient of the division x/y).*

This is another example of a nontotal function with an "obvious" way to remove the points where it is undefined (recall the case of $\lambda xy.x^y$).

Thus the symbol "$\lfloor x/y \rfloor$"

is *extended* to *mean*

$$(\mu z)_{\leq x}(z+1)y > x \qquad (*)$$

for all x, y.

$$u > v$$

▶ Pause. **Why** is the above expression correct for $\lfloor x/y \rfloor$ —$y \neq 0$?

*For any real number x, the symbol "$\lfloor x \rfloor$" is called the *floor* of x. It succeeds in the literature (with the same definition) the so-called "greatest integer function, $[x]$", i.e., the *integer part* of the real number x. Thus, **by definition**, $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$.

Because setting $z = \lfloor x/y \rfloor$ we have

$$z \leq \frac{x}{y} < z + 1$$

by the definition of "$\lfloor \ldots \rfloor$".

Thus, z is *smallest* such that $x/y < z + 1$, or such that $x < y(z + 1)$. ◀

It follows that, for $y > 0$, the search in $(*)$ yields the "normal math" value for $\lfloor x/y \rfloor$, while it re-defines $\lfloor x/0 \rfloor$ as $= x + 1$.

(2) $\lambda xy.rem(x,y)$ *(the remainder of the division* x/y*).*

$$rem(x,y) = x \dot{-} y\lfloor x/y \rfloor.$$

(3) *$\lambda xy.x|y$ (x divides y).*

$$x|y \equiv rem(y, x) = 0.$$

Note that if $y > 0$, we cannot have $0|y$ —*a good thing!*— since $rem(y, 0) = y > 0$.

Our redefinition of $\lfloor x/y \rfloor$ yields, however, $0|0$, but we can live with this in practice.

(4) *$Pr(x)$ (x is a prime).*

$$Pr(x) \equiv x > 1 \wedge (\forall y)_{\leq x}(y|x \rightarrow y = 1 \vee y = x).$$

More simply

$$Pr(x) \equiv x > 1 \wedge (\forall y)_{< x}(y|x \rightarrow y = 1).$$

(5) $\pi(x)$ *(the number of primes* $\leq x$*).*[†]

The following primitive recursion certifies the claim:

$$\pi(0) = 0,$$

and

$$\pi(x+1) = \text{if } Pr(x+1) \text{ then } \pi(x)+1 \text{ else } \pi(x).$$

[†]The π-function plays a central role in number theory, figuring in the so-called *prime number theorem*. See, for example, [LeV56].

Oct. 4, 2021

(6) $\lambda n.p_n$ *(the nth prime).*

First note that the graph $y = p_n$ is primitive recursive:

$y = p_n \equiv Pr(y) \wedge \pi(y) = n + 1.$

Next note that, for all n,

$p_n \leq 2^{2^n}$ (see Exercise 4.1.18 below),

thus $p_n = (\mu y)_{\leq 2^{2^n}}(y = p_n)$,

which settles the claim.

(7) $\lambda nx.\exp(n,x)$ *(*the *exponent of* p_n *in the prime factorization of* x*).*

$$\exp(n,x) = (\mu y)_{\leq x}\neg(p_n^{y+1}|x).$$

▶ Is x a good bound? **Yes!** $x = \ldots p_n^y \ldots \geq p_n^y \geq 2^y > y$.

A good bound allows you to search for ALL values that you must search for.

It does **not** stop your search prematurely.

4.1.17 Remark. *What makes* $\exp(n,x)$ *"*the *exponent of* p_n *in the prime factorization of* x*", rather than* an *exponent, is Euclid's prime number factorization theorem:* Every number $x > 1$ has a unique factorization —within permutation of factors— as a product of primes. □

(8) *Seq(x) (x's prime number factorization contains at least one prime, and no gaps).*

$Seq(x) \equiv x > 1 \wedge (\forall y)_{\leq x}(\forall z)_{\leq x}(Pr(y) \wedge Pr(z) \wedge y < z \wedge z|x \rightarrow y|x)$. □

4.1.18 Exercise. Prove by induction on n, that for all n we have $p_n \leq 2^{2^n}$.

Hint. Consider, as Euclid did,[‡] $p_0 p_1 \cdots p_n + 1$. If this number is prime, then it is greater than or equal to p_{n+1} (why?). If it is composite, then none of the primes up to p_n divide it. So any prime factor of it is greater than or equal to p_{n+1} (why?). □

[‡]In his proof that there are infinitely many primes.

4.2 CODING Sequences

4.2.1 Definition. (Coding Sequences) Any finite sequence of numbers, $a_0, \ldots, a_n$, $n \geq 0$, is *coded* by the number denoted by the symbol

$$\langle a_0, \ldots, a_n \rangle$$

which is defined as $\prod_{i \leq n} p_i^{a_i+1}$ □

Example. Code 1, 0, 3. I get $\langle 1, 0, 3 \rangle = 2^{1+1}3^{0+1}5^{3+1}$

For *coding* to be useful, we need a simple *decoding* scheme.

By Remark 4.1.17 there is no way to have $z = \langle a_0, \ldots, a_n \rangle = \langle b_0, \ldots, b_m \rangle$, *unless*

(i) $n = m$

and

(ii) For $i = 0, \ldots, n$, $a_i = b_i$.

Thus, it makes sense to correspondingly define the decoding expressions:

(*i*) *$lh(z)$ (pronounced "length of z") as shorthand for* $(\mu y)_{\leq z} \neg(p_y | z)$

▶ *A* ***comment*** *and a* ***question***:

- **The comment**: If p_y is the first prime NOT in the decomposition of z, and $Seq(z)$ holds, then since numbering of primes starts at 0, *the length of the coded sequence z is indeed y:*

$$\overbrace{p_0, p_1, \ldots, p_{y-1}}^{y \text{ primes in } z}$$

- **Question**: Is the bound z for y "good"? **Yes!**

$$z = 2^{a+1}3^{b+1}\dots p_{y\dot{-}1}^{exp(y\dot{-}1,z)} \geq \underbrace{2\cdot 2\cdots 2}_{y \text{ times}} = 2^y > y$$

(ii) $(z)_i$ *is shorthand for* $\exp(i,z) \dot{-} 1$

Note that

(a) *$\lambda iz.(z)_i$ and $\lambda z.lh(z)$ are in $\mathcal{PR}$.*

(b) If $Seq(z)$, then $z = \langle a_0, \ldots, a_n \rangle$ for some $a_0, \ldots, a_n$. In this case, $lh(z)$ equals the number of distinct primes in the decomposition of z, that is, the length $n+1$ of the coded sequence. Then $(z)_i$, for $i < lh(z)$, equals a_i. For larger i, $(z)_i = 0$. Note that if $\neg Seq(z)$ then $lh(z)$ need not equal the number of distinct primes in the decomposition of z. For example, 10 has 2 primes, but $lh(10) = 1$.

$$10 = p_0 p_2$$

The tools lh, $Seq(z)$, and $\lambda iz.(z)_i$ are sufficient to perform *decoding*, primitive recursively, once the truth of $Seq(z)$ is established. This coding/decoding scheme is essentially that of [Göd31], and will be the one we use throughout these notes.

4.2.1 Simultaneous Primitive Recursion

Start with total h_i, g_i for $i = 0, 1, \ldots, k$. Consider the new functions f_i defined by the following "*simultaneous primitive recursion schema*" for all $x, \vec{y}$.

$$\begin{cases} f_0(0, \vec{y}) & = h_1(\vec{y}) \\ \vdots & \\ f_k(0, \vec{y}) & = h_k(\vec{y}) \\ f_0(x+1, \vec{y}) & = g_0(x, \vec{y}, f_0(x, \vec{y}), \ldots, f_k(x, \vec{y})) \\ \vdots & \\ f_k(x+1, \vec{y}) & = g_k(x, \vec{y}, f_0(x, \vec{y}), \ldots, f_k(x, \vec{y})) \end{cases} \tag{2}$$

Hilbert and Bernays proved the following:

4.2.2 Theorem. If all the h_i and g_i are in $\mathcal{PR}$ (resp. $\mathcal{R}$), then so are all the f_i obtained by the schema (2) of simultaneous recursion.

Proof. IDEA: Code all the functions f_i by a single function F and convert the simultaneous recursion to a single primitive recursion for F.

Thus, define, for all $x, \vec{y}$,

$$F(x,\vec{y}) \overset{\text{Def}}{=} \langle f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})\rangle$$

$$H(\vec{y}) \overset{\text{Def}}{=} \langle h_0(\vec{y}),\ldots,h_k(\vec{y})\rangle = 2^{h_0(\vec{y})+1}3^{h_1(\vec{y})+1}\ldots p_k^{h_k(\vec{y})+1}$$

The Iterator "G" is displayed below. See it **explained** in the typeset in blue passage below that starts with "▶".

$$G(x,\vec{y},z) \overset{\text{Def}}{=} \langle g_0(x,\vec{y},(z)_0,\ldots,(z)_k),\ldots,g_k(x,\vec{y},(z)_0,\ldots,(z)_k)\rangle$$

We readily have that $H \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) and $G \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) depending on where we assumed the h_i and g_i to be. We can now rewrite schema (2) (p.111) as

$$\begin{cases} F(0,\vec{y}) & = H(\vec{y}) \\ F(x+1,\vec{y}) & = G\left(x,\vec{y},F\left(x,\vec{y}\right)\right) \end{cases} \tag{3}$$

▶ The 2nd line of (3) is obtained from

$$\begin{aligned} F(x+1,\vec{y}) &= \langle f_0(x+1,\vec{y}),\ldots,f_k(x+1,\vec{y})\rangle \\ &= \Big\langle g_0\Big(x,\vec{y},f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})\Big),\ldots,g_k\Big(\text{same as } g_0\Big)\Big\rangle \\ &= \Big\langle g_0\Big(x,\vec{y},\big(F(x,\vec{y})\big)_0,\ldots,\big(F(x,\vec{y})\big)_k\Big),\ldots,g_k\Big(\text{same as } g_0\Big)\Big\rangle \end{aligned}$$

By the above remarks, $F \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) depending on where we assumed the h_i and g_i to be. In particular, this holds for each f_i since, for all $x, \vec{y}$, $f_i(x,\vec{y}) = \big(F(x,\vec{y})\big)_i$. □

4.2.3 Example. We saw one way to justify that $\lambda x.rem(x,2) \in \mathcal{PR}$ in 4.1.16. A direct way is the following.

Setting $f(x) = rem(x,2)$, for all x, we notice that the sequence of outputs (for $x = 0, 1, 2, \ldots$) of f is

$$0, 1, 0, 1, 0, 1 \ldots$$

Thus, the following primitive recursion shows that $f \in \mathcal{PR}$:

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

Here is a way, via simultaneous recursion, to obtain a proof that $f \in \mathcal{PR}$, without using any arithmetic! Notice the infinite "matrix"

$$\begin{array}{cccccccc} 0 & 1 & 0 & 1 & 0 & 1 & \ldots \\ 1 & 0 & 1 & 0 & 1 & 0 & \ldots \end{array}$$

Let us call g the function that has as its sequence of outputs the entries of the second row —obtained by shifting the first row by one position to the left. The first row still represents our f. Now

$$\begin{cases} f(0) & = 0 \\ g(0) & = 1 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) \end{cases} \tag{1}$$

□

4.2.4 Example. We saw one way to justify that

$$\lambda x.\, \lfloor x/2 \rfloor \in \mathcal{PR}$$

in 4.1.16. A direct way is the following.

$$\begin{cases} \left\lfloor \dfrac{0}{2} \right\rfloor & = 0 \\ \left\lfloor \dfrac{x+1}{2} \right\rfloor & = \left\lfloor \dfrac{x}{2} \right\rfloor + rem(x,2) \end{cases}$$

where rem is in $\mathcal{PR}$ by 4.2.3.

Alternatively, here is a way that can do it —via simultaneous recursion— and with only the knowledge of how to add 1. Consider the matrix

$$\begin{array}{cccccccccc} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & \dots \\ 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & \dots \end{array}$$

The top row represents $\lambda x.\, \lfloor x/2 \rfloor$, let us call it "$f$". The bottom row we call g and is, again, the result of shifting row one to the left by one position. Thus, we have a simultaneous recursion

$$\begin{cases} f(0) & = 0 \\ g(0) & = 0 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) + 1 \end{cases} \tag{2}$$

□

4.3 Pairing Functions

Skip for now, or just read the definition and realise that $\lambda xy.\langle x, y\rangle$ is a pairing function in $\mathcal{PR}$ with projections —$\lambda z.(z)_0$ and $\lambda z.(z)_1$— in $\mathcal{PR}$.

Coding of sequences $a_0, a_1, \ldots, a_n$, for $n \geq 1$, has a special case; pairing functions, that is, the case of $n = 2$.

4.3.1 Definition. A **total, 1-1** function $J : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is called a *pairing function*. □

4.3.2 Remark. A decoder is a pair of **total** functions K, L from $\mathbb{N} \to \mathbb{N}$ such that, for any z that is equal to $J(x, y)$ for some x and y, they "compute" said x and y:

$$K(z) = x$$

and

$$L(z) = y$$

On non-code z, K and L give some **nonsense answer** —but answer they will; they are total!— just as the decoder $\lambda zi.(z)_i$ does when $\neg Seq(z)$ is true.

One usually encounters the (capital) letters K, L in the literature as (generic) names for projection functions of some (generic) pairing function. In turn, the generic symbol for the latter is J rather than "f". We will conform to this notational convention in what follows. □

The set of "tools" consisting of a pairing function J and its two projections K and L is a coding/decoding scheme for sequences of length two. We want to have

computable such schemes and indeed there is an abundance of *primitive recursive* pairing functions that also have primitive recursive projections.


One seeks J, K, L triples that are in $\mathcal{PR}$.


Some of those we will indicate in the examples below and others we will let the reader to discover in the exercises section.

4.3.3 Example. The function $J = \lambda xy.\, \langle x, y\rangle$ is pairing. Good decoders/projections are $K = \lambda z.(z)_0$ and $L = \lambda z.(z)_1$. All three are already known to us as members of $\mathcal{PR}$.

This J is not onto. For example, $5 \notin \mathrm{ran}(J)$. Nevertheless, K and L are total —because $\lambda iz.(z)_i$ is; indeed is in $\mathcal{PR}$. □

4.3.4 Example. The function $J = \lambda xy.2^x 3^y$ is pairing. As its projections we normally take $K = \lambda z.\exp(0, z)$ and $L = \lambda z.\exp(1, z)$ (cf. 4.1.16). All three are already known to us as members of $\mathcal{PR}$.

This J is not onto. Again, $5 \notin \mathrm{ran}(J)$. Nevertheless, K and L are total —because $\lambda iz.\exp(i, z)$ is; indeed is in $\mathcal{PR}$. □

4.3.5 Example. The function $J = \lambda xy.2^x(2y+1)$ due to Grzegorczyk is pairing. Indeed, since 2 is the only even prime, "$2^x(2y+1)$" is a forgetful depiction of a number's prime-number decomposition, where all powers of odd primes are lumped together in "$2y+1$". Clearly it is in $\mathcal{PR}$ since we have addition, multiplication, successor and $\lambda x.2^x$ in $\mathcal{PR}$.

$K = \lambda z.\exp(0,z)$ works: If $z = 2^x(2y+1)$, then $Kz = x$ as it should.

What is L? □

4.3.6 Example. In 4.3.4 and 4.3.5 we note that $J(x, y) \geq x$ and $J(x, y) \geq y$, for all x, y. Thus an alternative way to prove that the related K and L are in $\mathcal{PR}$ is to compute as follows:

$$Kz = (\mu x)_{\leq z}(\exists y)_{\leq z}(J(x, y) = z) \tag{1}$$

and

$$Lz = (\mu y)_{\leq z}(\exists x)_{\leq z}(J(x, y) = z) \tag{2}$$

Equipped with theorems 4.1.6 and 4.1.9, and Definition 4.1.15, we see that (1) and (2) establish that K and L are in $\mathcal{PR}$. □

4.3.7 Example. Here is a pairing function that does not require exponentiation. Let $J(x,y) = (x+y)^2 + x$. Clearly, $J \in \mathcal{PR}$.

So let us set $z = (x+y)^2 + x$ and solve this "equation" for x and y (uniquely, hopefully). Well, $(x+y)^2 \leq z < (x+y+1)^2$. Thus $x+y \leq \sqrt{z} < x+y+1$, hence

$$x + y = \lfloor \sqrt{z} \rfloor \tag{1}$$

Then, $z = \lfloor \sqrt{z} \rfloor^2 + x$ and therefore $Kz = z \dot{-} \lfloor \sqrt{z} \rfloor^2$. By (1), $Lz = \lfloor \sqrt{z} \rfloor \dot{-} Kz$.

As in 4.3.6, the J here satisfies $J(x,y) \geq x$ and $J(x,y) \geq y$. Their primitive recursiveness of K, L also follows from the calculations $Kz = (\mu x)_{\leq z}(\exists y)_{\leq z}(J(x,y) = z)$ and $Lz = (\mu y)_{\leq z}(\exists x)_{\leq z}(J(x,y) = z)$. □

Why bother about pairing functions when we have the coding of sequences scheme of the previous subsection? Because prime-power coding is computationally very inefficient, while quadratic schemes such as that of the previous example allow us to significantly reduce the "computational complexity" of coding/decoding. But can we code arbitrary length sequences efficiently?

Yes, because any J, K, L scheme can lead to a coding/decoding scheme for sequences $a_1, \ldots, a_n$, $n \geq 2$, for both the cases of a fixed or variable n.

4.3.8 Definition. Given a primitive recursive pairing scheme J, K, L.

For any fixed $n \geq 0$ we define by induction on n the symbol $[\![a_0, \ldots, a_n]\!]^{(n+1)}$:

For $n = 0$, we *define* $[\![x]\!]^{(1)} = x$.

For $n \geq 1$ we *define*

$$[\![y_0, y_1, \ldots, y_n]\!]^{(n+1)} = J(y_0, [\![y_1, \ldots, y_n]\!]^{(n)}). \qquad \square$$

Thus,

$$[\![x, y]\!]^{(2)} = J(x, y),\ [\![w, x, y, z]\!]^{(4)} = J\Big(w, J\Big(x, J(y, z)\Big)\Big),\ \textit{etc.}$$

How do we *decode* a z that is given as $z = [\![y_0, y_1, \ldots, y_k]\!]^{(k+1)}$? 

We need functions —called projections— Π_i^{k+1}, for $0 \leq i \leq k$, such that

$$\Pi_i^{k+1} z^{\S} = y_i, \text{ for } i = 0, \ldots, k$$

We can define them easily in terms of the K and L for the given J.

Well, clearly, $Kz = y_0$, hence

$$\Pi_0^{k+1} = K \tag{1}$$

Now note

$$Lz = [\![y_1, \ldots, y_k]\!]^{(k)} \tag{2}$$

Thus,

$$\Pi_1^{k+1} = KL \tag{1'}$$

Next note

$$LLz = [\![y_2, \ldots, y_k]\!]^{(k-1)} \tag{3}$$

Thus,

$$\Pi_2^{k+1} = KLL \tag{1''}$$

Clearly (do induction on i if you don't think it is clear), for $0 \leq i \leq k$,

$$\overbrace{LL\ldots L}^{i}{}^{\P} z = [\![y_i, \ldots, y_k]\!]^{(k-i+1)} \tag{$i+1$}$$

[§] Note the missing brackets around z!

[¶] We usually write "L^i" for "$\overbrace{LL\ldots L}^{i}$".

Thus $L^k = [\![y_k]\!]^{(1)} = y_k$ and, for $1 \leq i < k$,

$$\Pi_i^{k+1} = KL^i \qquad (1^{(i+1)})$$

and

$$\Pi_k^{k+1} = L^k \qquad (1^{(k)})$$

Another "**Clearly**": Given that k is constant, by Grzegorczyk substitution all the Π_i^k (for a given k) are in $\mathcal{PR}$.

Chapter 5

Loop Programs

Oct. 6, 2021

This is a retelling of the material in [Tou12].

5.1 Syntax and Semantics of Loop Programs

Loop programs were introduced by D. Ritchie and A. Meyer ([MR67]) as program-theoretic counterpart to the number theoretic introduction of the set of primitive recursive functions $\mathcal{PR}$.

This programming formalism among other things connected the definitional (or structural) complexity of primitive recursive functions with their (run time) computational complexity.

newpage

Loop programs are very similar to programs written in FORTRAN,

but have a number of *simplifications*,

notably they lack an unrestricted do-while instruction (equivalently, there is *NO goto instruction*), and, of course, like URMs do not have READ/WRITE instructions.

What they do have is

(1) Each program references (uses) a finite number of $\mathbb{N}$-*valued variables* that we denote *metamathematically* by single letter names (upper or lower case is all right) with or without subscripts or primes.*

(2) Instructions are of the following types (X, Y could be any variables below, including the case of two identical variables):

(i) $X \leftarrow 0$

(ii) $X \leftarrow Y$

(iii) $X \leftarrow X + 1$

(iv) **Loop** $X \ldots$ **end**,

where "..." represents a *sequence of syntactically valid instructions* (which in 5.1.1 will be called a "loop program"). The **Loop** part is matched or balanced by the **end** part as it will become evident by the inductive definition below (5.1.1).

*The precise syntax of variables will be given shortly, but even after this fact we will continue using signs such as X, A, Z', Y''_{34} for variables—i.e., we will continue using metanotation.

Loop programs do not have $X \leftarrow X \dot{-} 1$ as primitive, but can simulate it. They have $X \leftarrow Y$ which URMs do not have as a primitive (but can simulate it).

Informally, the structure of loop programs can be defined by induction:

5.1.1 Definition. (Inductive definition: Loop Programs)

- Every ONE instruction of *type* (i)–(iii) *standing by itself* is a *loop program.*

  If we already have two loop programs P and Q, then so are

- P;Q, built by *superposition* (concatenation)

  normally written vertically, without the separator ";", like this:

$$P$$
$$Q$$

and,

- for any variable X (that *may or may not* be in P),

Loop X; P; **end**, is a program,

called *the loop closure* (of P),

and normally written vertically without separators ";" like this:

Loop X
P
end

□

5.1.2 Definition. ***The set of all loop programs will be denoted by*** L. □

5.2 Loop-Computable Functions

▶ *The informal semantics of loop programs are analogous to the semantics of the URM programs.*

1. A loop program **terminates** "if it has nothing to do", that is,

   If the current instruction is EMPTY.

2. *All three assignment statements behave as in any programming language,*

   and after execution of any such instruction, *the instruction below it (possibly EMPTY) is the next CURRENT instruction.*

3. Assuming we know what the loop programs P and Q do (their semantics), then "$P;Q$" behaves as follows:
   - First it does exactly what P does. Then
   - *if this ever makes the instruction after* P *current* —this is the first instruction of Q— then the execution of "$P;Q$" continues by executing Q.

4. When the instruction

   "**Loop** X; P; **end**"

   becomes current, its *execution* DOES ONE of (a) or (b) below:

   ▶ We view the **Loop-end** construct as an "instruction" just as a **begin-end** block is in, say, Pascal.◀

   (a) *NOTHING, if $X = 0$ at that time and program execution moves to the first instruction below the loop.*

   (b) If $X = a > 0$ initially, then the instruction execution has the same effect as the program

$$a \text{ copies } \begin{cases} P \\ P \\ \vdots \\ P \end{cases}$$

▶ *So, the semantics of* **Loop-end** *is such that the number of times around the loop is NOT affected if the program CHANGES X by an assignment statement inside the loop!*

The symbol $P_Y^{\vec{X}_n}$ has exactly the same meaning as for the URMs —including that NON INPUT variables are intialised to zero— but here "P" is some loop program

It is the function computed by loop program P if we use $\vec{X}_n = X_1, X_2, \ldots, X_n$ as the input and Y as the output variables.

All $P_Y^{\vec{X}_n}$ are total.

This is trivial to prove by induction on the formation of P —that ALL loop Programs Terminate.

Basis: Let P be a one-instruction program. By 1 and 2 of page 130, such a program terminates.

I.H. Fix and Assume for programs P and Q.

I.S.

- What about the program

$$P$$
$$Q$$

By the I.H. starting at the top of program P we eventually overshoot it and make the first instruction of Q current.

By I.H. again, we eventually overshoot Q and the whole computation ends.

- What about the program

$$\textbf{Loop}X;P;\textbf{end}$$

Well, if $X = 0$ initially, then this terminates (does nothing).

So suppose X has the value $a > 0$ initially.

Then the program behaves like

$$a \text{ copies } \begin{cases} P \\ P \\ \vdots \\ P \end{cases}$$

By the I.H. for each copy of P above when started with its first instruction, the instruction pointer of the computation will eventually overshoot the copy's last instruction.

But then starting the computation with the 1st instruction of the 1st P, eventually the computation executes the 1st instruction of the 2nd P,

then, eventually, that of the 3rd P ...

and, then, eventually, that of the last (a-th) P.

We noted that each copy of P will be overshot by the computation; THUS the overall computation will be over after the LAST copy has been overshot. PROVED!

5.2.1 Definition. *We define the set of* ***loop programmable functions****,* $\mathcal{L}$*:*

The symbol $\mathcal{L}$ *stands for* $\{P_Y^{\vec{X}_n} : P \in L\}$. □

Two examples. Refer the computation of $\lambda x.rem(x,2)$ and $\lambda x.\lfloor x/2 \rfloor$ earlier.

If we let $f = \lambda x.rem(x,2)$ we saw that the following *sim. recursion* computes f.

$$\begin{cases} f(0) & = 0 \\ g(0) & = 1 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) \end{cases} \tag{1}$$

As a loop program this is implemented as the program P below —that is, $f = P_F^X$.

$G \leftarrow G + 1$
Loop X
$T \leftarrow F$
$F \leftarrow G$
$G \leftarrow T$
end

As for $\lambda x.\lfloor x/2 \rfloor$ we saw earlier that if $f = \lambda x.\lfloor x/2 \rfloor$ then we have:

$$\begin{cases} f(0) & = 0 \\ g(0) & = 0 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) + 1 \end{cases} \tag{2}$$

Loop X
$T \leftarrow F$
$F \leftarrow G$
$T \leftarrow T + 1$
$G \leftarrow T$
end

If P is the name of the above program, then $P_F^X = f$.

Subtracting by adding!
The program Q_X^X below computes $\lambda x.x \dot{-} 1$.

How?

X lags from T by one. At the end of the loop T holds the original value of X, but X is ONE behind its original value!

$T \leftarrow 0$
Loop X
$X \leftarrow T$
$T \leftarrow T + 1$
end

Addition

Program P below computes $\lambda xy.x + y$ as P_Y^{XY}.

Loop X
$Y \leftarrow Y + 1$
end

Multiplication

Program Q below computes $\lambda xy.x \times y$ as Q_Z^{XY}.

Loop X
  Loop Y
  $Z \leftarrow Z + 1$
  end
end

Why? Because we add 1 —$X \times Y$ times— to Z that starts as 0.

5.3 $\mathcal{PR} \subseteq \mathcal{L}$

5.3.1 Theorem. $\mathcal{PR} \subseteq \mathcal{L}$.

Proof. By induction over $\mathcal{PR}$ and brute-force programming we are proving THIS property of ALL $f \in \mathcal{PR}$:

"f is loop programmable".

Basis: $\lambda x.x + 1$ is P_X^X where P is $X \leftarrow X + 1$.

Similarly, $\lambda \vec{x}_n.x_i$ is $P_{X_i}^{\vec{X}_n}$ where P is

$$X_1 \leftarrow X_1; X_2 \leftarrow X_2; \ldots; X_n \leftarrow X_n$$

The case of $\lambda x.0$ is as easy.

Propagation of the property we are proving with ***Grzegorczyk substitution****.*

Just probe the function substitution case.

How does one compute $\lambda\vec{x}\vec{y}.f(g(\vec{x}),\vec{y})$ *if* $g = G_Z^{\vec{X}}$ *and* $f = F^{Z\vec{Y}}{}_W$*?*

Same as with URM programs.

One uses program concatenation and minds that Z is *the only variable common* between F and G.

$$\begin{pmatrix} G \\ F \end{pmatrix}_W^{\vec{X}\vec{Y}}$$

Propagation with primitive recursion.

So, say $h = H_Z^{\vec{Y}}$ *and* $g = G_Z^{X,\vec{Y},Z}$ *where* H *and* G *are in* L.

We indicate in pseudo-code how to compute $f = prim(h, g)$.

We have

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

The pseudo-code is

$z \leftarrow h(\vec{y}_n)$ **Computed as** $H_Z^{\vec{Y}_n}$
$i \leftarrow 0$
Loop x
$z \leftarrow g(i, \vec{y}_n, z)$ **Computed as** $G_Z^{I,\vec{Y}_n,Z}$
$i \leftarrow i + 1$
end

See the similar more complicated programming for URMs to recall precautions needed to avoid side-effects.

□

5.4 $\mathcal{L} \subseteq \mathcal{PR}$

To handle the converse of 5.3.1 we will simulate the computation of loop program P by an array of primitive recursive functions.

I want to show the every $P_Y^{\vec{X}_n}$ is in $\mathcal{PR}$.
I prove something simpler instead:

5.4.1 Definition. *For any $P \in L$ and any variable Y in P, the symbol P_Y is an abbreviation of $P_Y^{\vec{X}_n}$, where $\vec{X}_n$ are* all *the variables that occur in P.* □

5.4.2 Lemma. *For any $P \in L$ and any variable Y in P, we have that $P_Y \in \mathcal{PR}$.*

Proof.

(A) For the *Basis*, we have cases:

- P is $X \leftarrow 0$. Then $P_X = \lambda x.0 \in \mathcal{PR}$.
- P is $X \leftarrow Y$. Then $P_X = \lambda xy.y \in \mathcal{PR}$, while $P_Y = \lambda xy.y \in \mathcal{PR}$.
- P is $X \leftarrow X + 1$. Then $P_X = \lambda x.x + 1 \in \mathcal{PR}$

Let us next do the *induction step*:

(B) P is $Q; R$.

(i) **Case** where **NO variables are common** between Q and R.

Let the Q variables be $\vec{z}_k$ and the R variables be $\vec{u}_m$.

- What can we say about $\Big(Q; R\Big)_{z_i}$?

  Let $\lambda \vec{z}_k.f(\vec{z}_k) = Q_{z_i}$.

  $f \in \mathcal{PR}$ by the I.H.

  But then, so is $\lambda \vec{z}_k \vec{u}_m.f(\vec{z}_k)$ by Grzegorczyk Ops.
  But this is $\Big(Q; R\Big)_{z_i}$.

- Similarly we argue for $\Big(Q; R\Big)_{u_j}$

Oct. 18

(ii) **Case** where $\vec{y}_n$ are common between Q and R.

$\vec{z}$ and $\vec{u}$ —just as in case (i) above— are the *NON*-common variables.

▶ Thus the set of variables of $\Big(Q; R\Big)$ is $\vec{y}_n \vec{z}_k \vec{u}_m$

Now, pick an output variable w_i.

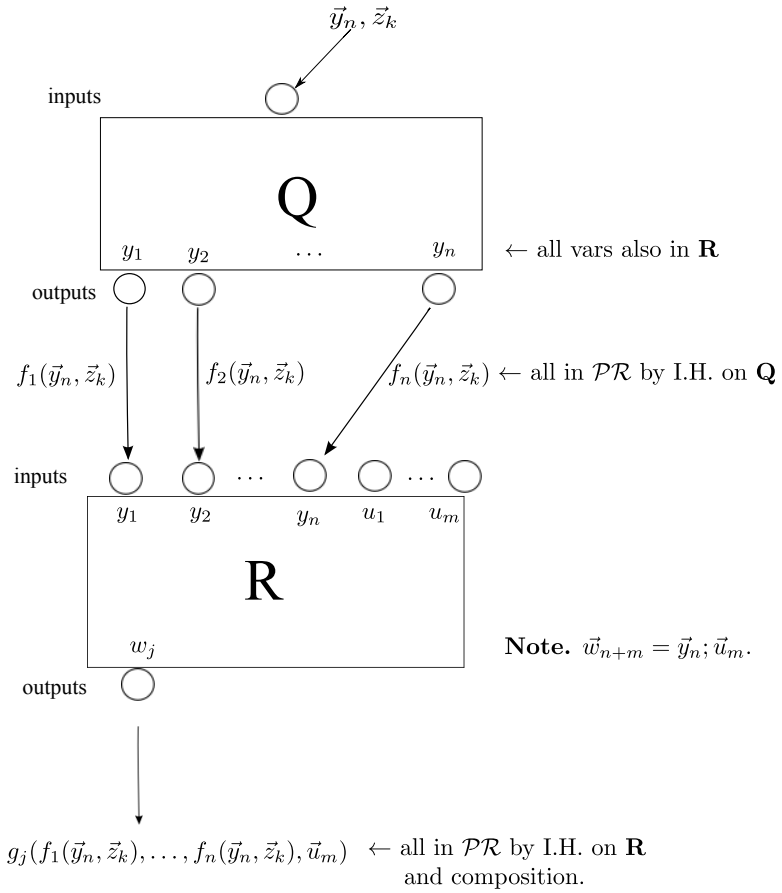
- If w_i is among the z_j, then we are back to the first bullet of case (i).

  Nothing that R does can change z_j.

- So let the w_i be a component of the vector $\vec{y}_n \vec{u}_m$ instead. This case is fully captured by the figure below.

– Define $f_i = \lambda \vec{y}\vec{z}.\Big(Q_{y_i}\Big)$. By the I.H. on Q these are in $\mathcal{PR}$. The *outputs* via the y_j are as shown: $f_j(\vec{y}_n, \vec{z}_k)$

– Define $g_j = \lambda \vec{y}\vec{u}.\Big(R_{w_j}\Big)$. By the I.H. on R these are in $\mathcal{PR}$. The *outputs* via the w_j —given what inputs go into the y_r— are as shown: $g_j(f_1(\vec{y}, \vec{z}), \ldots, f_n(\vec{y}, \vec{z}), \vec{u}_m)$. Thus $\lambda \vec{y}\vec{z}\vec{u}.g_j(f_1(\vec{y}, \vec{z}), \ldots, f_n(\vec{y}, \vec{z}), \vec{u}_m) = \Big(Q; R\Big)_{w_j} \in \mathcal{PR}$.



(C) P is **Loop** $x; Q;$ **end**.

There are two subcases: x in Q; or NOT.

(a) x not in Q:

So, let $\vec{y}_n$ be all the variables of Q; x is NOT one of them. Let

$$\boxed{\lambda x \vec{y}_n . f_0(x, \vec{y}_n)} \text{ denote } P_x \qquad (5)$$

and, for $i = 1, \ldots, n$,

$$\boxed{\lambda x \vec{y}_n . f_i(x, \vec{y}_n)} \text{ denote } P_{y_i} \qquad (6)$$

where x —being an input variable— holds the initial value we give to it before the program P starts.

In what follows we will refer to this initial value of x as "k".

Moreover, let

$$\boxed{\lambda \vec{y}_n . g_i(\vec{y}_n)} \text{ denote } Q_{y_i} \qquad (7)$$

▶ By the I.H., the g_i are in $\mathcal{PR}$ for $i = 1, 2, \ldots, n$.

We want to prove that the functions in (5) and (6) are also in $\mathcal{PR}$.

Since $f_0 = \lambda x \vec{y}_n . x$ (Why?),

we only deal with the f_i for $i > 0$.

The plan is to set up a simultaneous recursion that produces the f_i from the g_i.

Now imagine the computation of P with input $x, y_1, \ldots, y_n$.

We have two sub-subcases:

- $x = 0$.

  In this sub-subcase, the loop is skipped and no variables are changed by the program. In terms of (5) and (6), what I just said translates into

  $$f_0(0, \vec{y}_n) = 0 \tag{8}$$

  and

  $$f_i(0, \vec{y}_n) = y_i, \text{ for } i = 1, \ldots, n \tag{9}$$

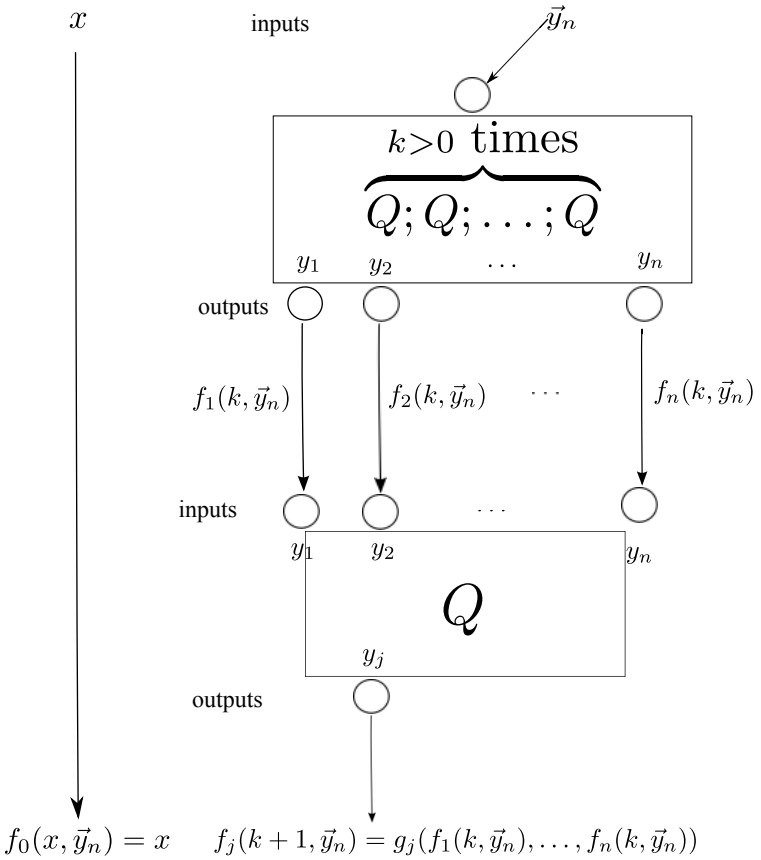
- $x = k + 1$, i.e., positive.

The effect of P is

$$k \text{ copies} \begin{cases} Q \\ Q \\ Q \\ \vdots \\ Q \\ Q \end{cases} \tag{10}$$

What is $f_i(k+1, \vec{y}_n)$, for $i > 0$?

Well, consult the picture below:



We now have a simultaneous primitive recursion that yields the f_i from the g_i. The g_i being in $\mathcal{PR}$ by the I.H. on Q, so are the f_i.

(b) x in Q:

So, let $x, \vec{y}_n$ be all the variables of Q. Let

$$\lambda x \vec{y}_n . f_0(x, \vec{y}_n) \text{ denote } P_x \tag{11}$$

and, for $i = 1, \ldots, n$,

$$\lambda x \vec{y}_n . f_i(x, \vec{y}_n) \text{ denote } P_{y_i} \tag{12}$$

Moreover, let

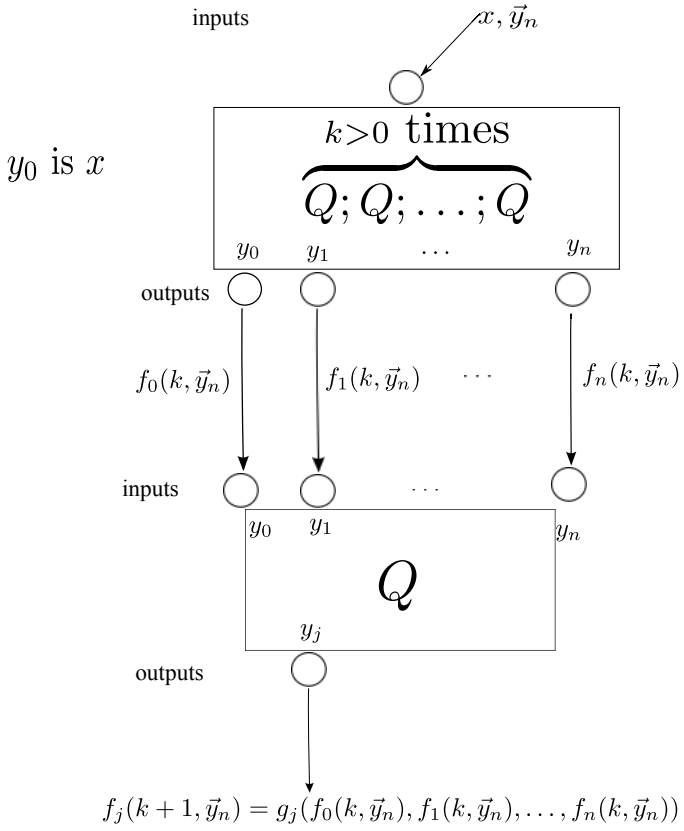
$$\lambda x \vec{y}_n . g_0(x, \vec{y}_n) \text{ denote } Q_x \tag{13}$$

$$\lambda x \vec{y}_n . g_i(x, \vec{y}_n) \text{ denote } Q_{y_i} \tag{14}$$

By the I.H., the g_i are in $\mathcal{PR}$ for $i = 1, 2, \ldots, n$.

We want to prove that the functions in (11) and (12) are also in $\mathcal{PR}$ by employing an appropriate simultaneous recursion. The basis equations are the same as (8) and (9).

For $x = k+1$ we simply consult the figure below, to yield the recurrence equations



$$f_j(k+1, \vec{y}_n) = g_j(f_0(k, \vec{y}_n), f_1(k, \vec{y}_n), \ldots, f_n(k, \vec{y}_n)), j = 0, \ldots, n$$

As the g_j are in $\mathcal{PR}$, so are the f_j.

At the end of all this we have the proof of the Lemma. □

We can now prove

5.4.3 Theorem. $\mathcal{L} \subseteq \mathcal{PR}$.

Proof. We must SHOW that if $P \in L$ then for any choice of $\vec{X}_n, Y$ in P we have

$$\boxed{P_Y^{\vec{X}_n} \in \mathcal{PR}}$$

So pick a P and also $\vec{X}_n, Y$ in it.

Let $\vec{Z}_m$ the rest of the variables (the non-input variables) of P, and let

$$f = P_Y = P_Y^{\vec{X}_n, \vec{Z}_m}$$

Define

$$g = P_Y^{\vec{X}_n}$$

By the lemma, $f \in \mathcal{PR}$.

But

$$g(\vec{X}_n) = f(\vec{X}_n, \overbrace{0, \ldots, 0}^{m \ zeros})$$

By Grzegorczyk substitution, $g = P_Y^{\vec{X}_n} \in \mathcal{PR}$. □

All in all, we have that

$$\mathcal{PR} = \mathcal{L}$$

5.5 Incompleteness of $\mathcal{PR}$

We can now see that $\mathcal{PR}$ cannot possibly contain all the *intuitively computable* total functions. We see this as follows:

(A)

It is immediately believable that we can write a program that checks if a string over the alphabet

$$\Sigma = \{X, 0, 1, +, \leftarrow, ;\ , \mathbf{Loop}, \mathbf{end}\} \qquad (1)$$

of loop programs is a correctly formed program or not.

BTW, the symbols X and 1 above generate *all* the variables,

$$X1, X11, X111, X1111, \ldots$$

We will not ever write variables down as what they really are —"$X\underbrace{1\ldots1}_{k\ 1s}$"— but we will continue using

metasymbols like

$$X, Y, Z, A, B, X'', Y'''_{23}, x, y, z'''_{15}$$

etc., for variables!

(B) We can algorithmically build the list, $List_1$, of ALL strings over Σ:

List by length; and in each length group **lexicographically**.[†]

(C) Simultaneously to building $List_1$ build $List_2$ as follows:

For every string α generated in $List_1$, copy it into $List_2$ iff $\alpha \in L$ (which we can test by (A)).

(D) Simultaneously to building $List_2$ build $List_3$:

For every P (program) copied in $List_2$ copy all the finitely many strings P_Y^X (for all choices of X and Y in P) alphabetically (think of the string P_Y^X as "$P; X; Y$").

[†]Fix the ordering of Σ as listed above.

At the end of all this we have an algorithmic list of all the functions $\lambda x.f(x)$ of $\mathcal{PR}$,

listed by their aliases, the P_Y^X programs.

Let us call this list of ALL the one-argument $\mathcal{PR}$ FUNCTIONS

$$f_0, f_1, f_2, \ldots, f_x, \ldots \tag{1}$$

Each f_i is a $\lambda x.f_i(x)$

5.6 A total "intuitively computable" function not in $\mathcal{PR}$

We can now show that

$$D \overset{Def}{=} \lambda x.1 + f_x(x) \tag{1}$$

is intuitively total computable but not in $\mathcal{PR}$.

We do not have enough tools at our disposal at this point to prove that $D \in \mathcal{R}$; but this is a fact!

5.6.1 Theorem. *$D \notin \mathcal{PR}$ but we can offer an algorithm for it, AND it s is total.*

Proof. (Informal) To compute $D(a)$ for any $a \in \mathbb{N}$ do:

- We go down $List_3$ until we find the function "f_a" in the form P_Y^X, for some $P \in L$ and X, Y in P.

- Compute $f_a(a)$ using P_Y^X.

This computation TERMINATES since $P_Y^X \in \mathcal{PR}$.

- Output $1 + f_a(a)$. This is, by (1), $D(a)$.

Suppose now that $D \in \mathcal{PR}$. If I am wrong, and since D is a one-argument function, it is a P_y^X, for some $P \in L$ and X, Y in P.

Let i be the location of this function in the $List_3$.

But then

$$D = f_i \tag{2}$$

and

$$f_i(i) \overset{(2)}{=} D(i) \overset{(1)}{=} 1 + f_i(i)$$

A contradiction since both (extreme) sides of "=" are numbers (defined). □

Chapter 6

Church's Thesis

Computability is the part of logic that gives a mathematically precise formulation to the concepts *algorithm*, *mechanical procedure*, and *calculable function* (or relation).

Intensive activity by many (Post [Pos36, Pos44], Kleene [Kle43], Church [Chu36], Turing [Tur37], Markov [Mar60]) led in the 1930s to *several formulations*, each purporting to *mathematically characterise* the concepts *algorithm*, *mechanical procedure*, and *calculable function*.

All these formulations were quickly proved *to be equivalent*; that is, the calculable functions admitted by any one of them were the same as those that were admitted by any other.

This led Alonzo Church to formulate his *conjecture*, famously known as "Church's Thesis", that

> any *intuitively and informally* calculable function is also calculable within any of these mathematical frameworks of calculability or computability. *It is in $\mathcal{P}$*.

I stress that even if this sounds like a "completeness *theorem*" in the realm of computability, it is not.

> Such a theorem would provably say "if the function f can be informally computed, then it can so be on a URM (and any other mathematically equivalent framework)". Church believes this, but it cannot be proved since the concept "informally computed" *is not mathematical*.

Oct.20, 2021

6.1 A Leap of Faith

In the task of proving that a function given mathematically —but *not by a mathematically-based programming language* such as URM— is URM-computable we are greatly helped by Church's Thesis —in short "CT"— which says, loosely, "give me a pseudo program for the computation of a function, and then I, A. Church, tell you that your function is URM computable".

We will practise this a bit in this chapter.

This does NOT say we have to believe that Church is right in his belief, but it is practically "safe" to practise CT.

For example, in the entire book of Rogers (over 500 pages) [Rog67], every argument by CT that he constructs can be converted to a precise mathematical one by a patient reader!

If I DON'T want you to rely on CT in a particular problem I will say "prove it mathematically".

CT Statement: Every *informal* algorithm (pseudo-program) that we propose for the computation of a function can be implemented (*made mathematically precise*, in other words) on each of the known models of computation —in particular on the URM.

Here is the template of **how** to use CT:

1. We **completely** present —that is, *no essential detail is missing*— an algorithm in *pseudo-code*.

   ►BTW, "pseudo-code" does not mean "sloppy-code"!◄

   ►AND "pseudo-code" CAN be incorrect! Your pseudo code must be seen or must be proved correct!◄

2. We then say: By CT, there is a URM that implements our algorithm. Hence the function that our pseudo code computes is in $\mathcal{P}$.

6.2 An Enumeration of *all* one-argument functions of $\mathcal{P}$

Recall:

6.2.1 Definition. The alphabet we use to construct the URMs is the following.

Consider the listing order of its members below **FIXED**.

$$\Gamma = \{X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \leftarrow, :, +, \dot{-}, \textbf{if}, \textbf{goto}, \textbf{else}, \textbf{stop}, ;\}$$

where we added ";" to the alphabet as a *SEPARATOR* to use when we write URMs horizontally, as STRINGS over the alphabet Γ. **Just as we did with loop programs!** □

We repeat below the construction of the effective list (computable listing) of all loop programs (and one-argument primitive recursive functions) —almost as is— but this time for URMs.

(A) It is immediately believable that we can write a program that checks if a string over the alphabet Γ for URMs

is a syntactically correct URM program or not.

BTW, the symbols X and 1 above on p.165 generate *all* the variables,

$$X1, X11, X111, X1111, \ldots$$

As in the case of Loop-Programs we do not ever write variables down as what they really are —"$X\underbrace{1\ldots1}_{k\ 1s}$"

but we will continue using *metasymbols* like

$$\mathbf{x}, \mathbf{y}''', z'_{123}, u''''_5, X, Y, Z, A, B, X'', Y'''_{23}$$

etc., for variables!

(B) We can algorithmically build the list, $List_1$, of ALL strings over Γ:

List by length; and in each length group **lexicographically** (alphabetically).

(C) Simultaneously to building $List_1$ build $List_2$ as follows:

For every string β generated in $List_1$, copy it into $List_2$ iff β checks to be a URM (which we can test by (A)).

(D) Simultaneously to building $List_2$ build $List_3$:

For every URM M (program) copied in $List_2$

copy *all the finitely many* strings M_Y^X (for all choices of X and Y in M) alphabetically (think of the string M_Y^X as "$M; X; Y$") into $List_3$.

Thus *ALL unary $\mathcal{P}$-functions* are listed by their aliases, the M_Y^X programs.

Let us call this list by its standard name in the literature ("Roger's Notation, [Rog67]"):

EFFECTIVE List of ALL one-argument $\mathcal{P}$ *FUNCTIONS*

$$\phi_0, \phi_1, \phi_2, \ldots, \phi_i, \ldots \tag{1}$$

where $\phi_i = M_Y^X$ iff M_Y^X is found in location i of $List_3$.

BTW "*EFFECTIVE List*" means "*algorithmically built List*".

6.3 A Universal function for unary $\mathcal{P}$ functions

We now have *universal* or *enumerating* function $U^{(P)}$ for *all* the unary functions in $\mathcal{P}$.

That is the function of TWO arguments

$$U^{(P)} \overset{Def}{=} \lambda ix.\phi_i(x) \tag{2}$$

So, for any i, x, $U^{(\mathcal{P})}(i, x)$ is the value (if any) of the i-th "M_Y^X" found in the $List_3$ of our earlier construction (see (1) above) run with input $X = x$.

What do I mean by "Universal" here? I mean two things:

6.3.1 Definition. (Universal function for unary $f \in \mathcal{P}$)

- For every unary $f \in \mathcal{P}$ there is an $i \in \mathbb{N}$ such that $\lambda x.U^{(\mathcal{P})}(i, x) = f$. This is because f will show up in $List_3$ as $f = \phi_i$ at some location i.

  Now look at (2): $U^{(\mathcal{P})}(i, x) = \phi_i(x) = f(x)$.
- $\lambda x.U^{(\mathcal{P})}(i, x) \in \mathcal{P}$. Again, this is because $\lambda x.U^{(\mathcal{P})}(i, x) = \phi_i = M_Y^X$, for some URM M and X, Y in M. □

We immediately obtain the Universal or *Enumeration Theorem* for *all* the unary functions in $\mathcal{P}$.

6.3.2 Theorem. (Enumeration theorem) *The Universal function of two arguments $\lambda ix.U^{(P)}(i,x)$ defined in (2) above and in 6.3.1 is partial recursive.*

Proof. To compute $U^{(P)}(i,x)$ for any i,x is to compute $\phi_i(x)$. Here is an informal algorithm to compute $\phi_i(x)$:

1. Given i,x

2. Develop the list $List_3$ of the preceding construction (part (D) on p.167) until the i-th URM M_Y^X has been listed.

3. Take this M_Y^X and compute with input x (inputed in X).

If M terminates, then Y clearly holds the value $U^{(P)}(i,x) = \phi_i(x)$ —since $M_Y^X = \phi_i$ by Rogers' definition.

Invoking CT we now declare that $\lambda ix.U^{(P)}(i,x)$ is more than informally algorithmic:

It CAN be implemented on a URM, hence is in $\mathcal{P}$. □

Hey, so *we* can *write a compiler for the* ϕ_i IN the URM Language!

This is a good point to return to something we skipped earlier. Extending equality between function calls $f(\vec{x})$ and $g(\vec{y})$ even when the calls do NOT return a value!

$$f(\vec{x}) = g(\vec{y}) \text{ iff } f(\vec{x})\!\uparrow \wedge g(\vec{y})\!\uparrow \vee (\exists z)\Big(f(\vec{x}) = z \wedge g(\vec{y}) = z\Big)$$

Another fundamental theorem in computability is the *Parametrisation* or *Iteration* or also "*S-m-n*" theorem of Kleene.

In fact, it and the universal function theorem along with a handful of initial computable functions are known to be *sufficient* to found computability axiomatically —but we will not get into this topic in this course.

▶ NEXT let us establish the fact that we can *enumerate algorithmically* —or *we can effectively list*— also the set of *ALL partial recursive functions of TWO variables*.

Just *repeat* the construction (A)–(D) on pp.166–167, but modify (D):

Here you do instead:

▶ (D') Simultaneously to building $List_2$ build $List'_3$:

For every URM M (program) copied in $List_2$ copy *all* the finitely many strings M_Z^{XY} (for all choices of X, Y and Z —keeping X, Y distinct— in M) alphabetically (think of the string M_Z^{XY} as "$M; X; Y; Z$") into $List'_3$ ◀.

The obtained effective list $List'_3$ of M_Z^{XY} is denoted ([Rog67]) by

$$\phi_0^{(2)}, \phi_1^{(2)}, \phi_2^{(2)}, \phi_3^{(2)}, \ldots, \phi_i^{(2)}, \ldots \tag{2}$$

where $\phi_i^{(2)} = M_Z^{XY}$ iff M_Z^{XY} is found in location i.

The superscript "(2)" of ϕ indicates that we are effectively listing 2-argument $\mathcal{P}$-functions, $\lambda xy.\phi_i^{(2)}(x, y)$

What if in step (D′) we find that program M has only one variable?

6.3.3 Theorem. (Parametrisation theorem) *There is a 2-argument function S^1_1 in $\mathcal{R}$ such that*

$$\phi^{(2)}_i(x,y) = \phi_{S^1_1(i,x)}(y), \text{ for all } i,x,y \qquad (1)$$

Proof. This says that given a program M that computes a function $\phi^{(2)}_i$ as $M^{\mathbf{uv}}_{\mathbf{z}}$ with **u** receiving the input value x and **v** receiving the input value y —each via an "implicit" **read** statement— we can, for **any** fixed value x, *effectively* † *construct* a **new** program **located** in position $S^1_1(i,x)$ of *the algorithmic enumeration* of *all* unary $\mathcal{P}$-functions —(1) on p.168— that has the value x "hardwired" and only "reads" the y-value; YET GIVES THE SAME ANSWER in **z**.

▶ The "*effectively construct*" above is the requirement that S^1_1 is **recursive**: Indeed this requirement says that we can *OBTAIN* the program for the rhs of (1).

Each value x for **u** is "*hardwired*" —as $\mathbf{u} \leftarrow x$— in the program for $\phi_{S^1_1(i,x)}$ rather than being inputted via an implicit "**read u**".

† Algorithmically.

The program $N_{\mathbf{z}}^{\mathbf{v}}$ for $\phi_{S_1^1(i,x)}$ —for a given i and each x— is *almost* the same as $M_{\mathbf{z}}^{\mathbf{uv}}$, namely, it is

$$N_{\mathbf{z}}^{\mathbf{v}} = \left(\overbrace{1 : \mathbf{u} \leftarrow x;}^{\text{replaces } \mathbf{read\ u}} \ M \right)_{\mathbf{z}}^{\mathbf{v}} \tag{3}$$

where *all* instruction labels of M (even *inside* **if**-statments) are shifted by adding 1 to them.

Trivially, for all i (that is, all $M_{\mathbf{z}}^{\mathbf{uv}}$) and all x, we have (2) of the theorem.

Remains to argue that we *can compute* $S_1^1(i, x)$, *for all* i, x.

- Given i, x

- Develop the list (2) of the $\phi_i^{(2)}$ on p.173 (this is "$List_3'$" of (D')) on p.173) until we can obtain its i-th member, $M_{\mathbf{z}}^{\mathbf{uv}}$

- Now, Build the URM $N_{\mathbf{z}}^{\mathbf{v}}$ from $M_{\mathbf{z}}^{\mathbf{uv}}$ as in (3) above.

- Now, Develop $List_3$ —essentially the list of the *one-argument* ϕ_j— built by (D), p.167 and go down while you keep comparing, until you find $N^{\mathbf{v}}_{\mathbf{z}}$.
- Output the location of $N^{\mathbf{v}}_{\mathbf{z}}$ that you found —this is $S^1_1(i,x)$.

> You **WILL** find said location since $List_3$ contains *ALL* unary ϕ_j (listed as URM programs M^X_Y).

- So S^1_1 is total.

By Church's thesis the above informal process can be done by URMs. Thus, $S^1_1 \in \mathcal{R}$. □

Oct. 25, 2021

$\lambda x.S_1^1(i,x)$ is strictly increasing. Indeed, $S_1^1(i,x)$ is the *location* of

$$\left(\overbrace{1 : \mathbf{u} \leftarrow x;}^{\text{replaces } \mathbf{read\ u}} \ M \right)_{\mathbf{z}}^{\mathbf{v}} \qquad (\dagger)$$

in the enumeration of the (unary) ϕ_j (p.168) while $S_1^1(i,x')$ is the *location* of

$$\left(\overbrace{1 : \mathbf{u} \leftarrow x';}^{\text{replaces } \mathbf{read\ u}} \ M \right)_{\mathbf{z}}^{\mathbf{v}} \qquad (\ddagger)$$

Now if $x < x'$ then x is earlier than x' in the *alphabetic ordering* of x and x' viewed as strings of *decimal digits*.

But then —*all other things being equal*— program $(\dagger)$ appears earlier than program $(\ddagger)$ in the lexicographic ordering of programs N_Y^X.

Thus location $S_1^1(i,x)$ is before location $S_1^1(i,x')$.

In short, $S_1^1(i,x) < S_1^1(i,x')$.

6.4 Unsolvable "Problems"; The Halting Problem

The following definition is repeated "*for the record*" *adding alternative names for "recursive relation"*.

6.4.1 Definition. (Decidable relations) "*A relation* $Q(\vec{x}_n)$ *is* ***computable****, or* ***decidable*** *or* ***solvable***" *iff it is Recursive*;

That is, *its characteristic* function

$$c_Q = \lambda \vec{x}_n . \begin{cases} 0 & \text{if } Q(\vec{x}_n) \\ 1 & \text{otherwise} \end{cases}$$

is in $\mathcal{R}$.

The collection (set) of ***all*** *computable relations we denote by* $\mathcal{R}_*$. □

Thus, "a relation $Q(\vec{x}_n)$ is *computable* or *decidable*" means that some URM computes c_Q.

But that means that some URM behaves as follows:

On input $\vec{x}_n$, it *halts* and outputs 0 iff $\vec{x}_n$ satisfies Q (i.e., iff $Q(\vec{x}_n)$),

it *halts* and outputs 1 iff $\vec{x}_n$ does **not** satisfy Q (i.e., iff $\neg Q(\vec{x}_n)$).

We say that a solvable relation has a decider, *i.e., the URM that* decides *membership of* any *tuple* $\vec{x}_n$ *in the relation.*

6.4.2 Definition. (Problems)
A "***Problem***" is —by definition— a formula of the type "$\vec{x}_n \in Q$" or, equivalently, "$Q(\vec{x}_n)$".

Thus, ***by definition***, a "problem" is a ***membership*** question.

□

6.4.3 Definition. (Unsolvable Problems) A problem "$\vec{x}_n \in Q$" is called any of the following:

Undecidable

Recursively unsolvable

or just

Unsolvable

iff $Q \notin \mathcal{R}_*$—in words, iff *Q is **not** a computable relation*. □

Here is the most famous undecidable problem:

$$\text{“}\phi_x(x)\downarrow\text{”} \tag{1}$$

I put quotes to be sure we separate the relation (a statement) from the function call $\phi_x(x)$ (a number, or undefined).

A different formulation uses the *set*

$$K \stackrel{Def}{=} \{x : \phi_x(x)\downarrow\}^{\dagger} \tag{2}$$

that is, *the set of all numbers x, such that the URM at location x on input x has a (halting!) computation.*

We shall call K the "**halting set**", and (1) we shall call the "**halting problem**".

Clearly, by (2), (1) is equivalent to

$$x \in K$$

[†]All three [Rog67, Tou84, Tou12] use K for this set, but this notation is by no means standard. It is unfortunate that this notation clashes with that for the first projection K of a pairing function J. However the context will manage to fend for itself!

6.4.4 Theorem. *The halting problem is unsolvable.*

Proof. We show, **by contradiction**, that $K \notin \mathcal{R}_*$.

Thus we start by assuming the opposite.

$$\text{Let } K \in \mathcal{R}_* \tag{3}$$

(3) says that we *can decide membership* in K via a URM, or, what is the same, we *can decide truth or falsehood* of $\phi_x(x)\downarrow$ for any x:

Consider then the infinite matrix below, *each row of which denotes a function in $\mathcal{P}$ as an array of outputs*,

the outputs being numerical, or the *special symbol* "↑" for any undefined entry $\phi_x(y)$.

By 6.3.2 and the comments following it, **EVERY** one-argument function of $\mathcal{P}$ is in some row (as an array of outputs).

$$\begin{array}{cccccc}
\phi_0(0) & \phi_0(1) & \phi_0(2) & \dots & \phi_0(i) & \dots \\
\phi_1(0) & \phi_1(1) & \phi_1(2) & \dots & \phi_1(i) & \dots \\
\phi_2(0) & \phi_2(1) & \phi_2(2) & \dots & \phi_2(i) & \dots \\
\vdots & & & & & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \dots & \phi_i(i) & \dots \\
\vdots & & & & &
\end{array}$$

We will use the assumed (3) (p.182) above AND the main diagonal (red) of the above matrix to *define a function that is a 1-argument function of $\mathcal{P}$ that is NOT a row above*.

> Cantor's famous "diagonalisation technique" dictates: Just define a function d as a row of "outputs" as
>
> $$d = \overline{\phi_0(0)}, \overline{\phi_1(1)}, \overline{\phi_2(2)}, \dots, \overline{\phi_x(x)}, \dots$$
>
> such that $\overline{\phi_x(x)} \neq \phi_x(x)$, for all x.

Thus d cannot fit row i for *any* i since $d(i)$ would have to be equal to what the matrix has on row i, position i: Namely $\phi_i(i)$. Instead $d(i) = \overline{\phi_i(i)}$, which is different!

This will contradict "*EVERY*" above.

So define, mathematically and specifically, the function d of one argument by

$$d(x) = \begin{cases} 42 & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \tag{4}$$

Here is why the function in (4) is partial recursive:

Given x, do:

- Use the *decider* for K (for "$\phi_x(x) \downarrow$", that is) — assumed to exist by (3) (p.182)— to test which condition is true in (4); top or bottom.
- If the top condition is true, then we return 42 and stop.
- If the bottom condition holds, then transfer to an infinite loop:

$$k : \textbf{goto } k$$

By CT, the 3-bullet program has a URM realisation, so d is computable.

Say now

$$d = \phi_i \tag{5}$$

Substitute ϕ_i for d in (4) to get

$$\phi_i(x) = \begin{cases} 42 & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \tag{6}$$

As *(6) is correct for all x*, it is correct for $x = i$.

So we get

$$\phi_i(i) = \begin{cases} 42 & \text{if } \phi_i(i) \uparrow \\ \uparrow & \text{if } \phi_i(i) \downarrow \end{cases}$$

Do you see the contradiction?

Cases.

- $lhs = \phi_i(i) = 42$. Then *we are in the top case*. But that implies $\phi_i(i) \uparrow$ No good!!!
- Well maybe the other case works? $lhs = \phi_i(i) = \uparrow$. *We are in the bottom case*. But this implies $\phi_i(i) \downarrow$ No good!!!

We have a contradiction no matter which case we pick.

But we knew this already from the  -remark following the Box on p.183.

So we reject (3) (p.182). Done! □

In terms of *theoretical significance*, the above is a *fundamental unsolvable problem that enables the process of finding more!* How?

As an Example we illustrate the "*program correctness problem*" (see below).

But how does "$x \in K$" help?

Through the following technique of *reduction*:

Let P be a new *problem* which we want to prove undecidable.
We proceed by contradiction like this:

We say, suppose instead that $\vec{y} \in P$ can be *solved* by a URM.

We then build a *reduction* that goes like this:

1. Let M be a URM that *decides* $\vec{y} \in P$, for any $\vec{y}$.
2. Then we show *how to use* M *as a* subroutine to also solve $x \in K$, for any x.
3. Since we know that the latter is *unsolvable* we have arrived at a contradiction that implies that *no such URM* M *exists*!

The *equivalence problem* is

> Given two programs M and N can we test to see whether they compute the same function?

Of course, "testing" for such a question *cannot be done by experiment*: We *cannot just run* M and N for *all inputs* to see if they get the same output, because, for one thing, "all inputs" *are infinitely many*, and, for another, there may be inputs that *cause one or the other program to run forever* (infinite loop).

By the way, the equivalence problem is the general case of the "*program correctness*" problem which asks

> Given a program P and a *program specification* S, does the program *fit* the specification *for all inputs*?

How so? *Well, we can view a specification as just another formalism to FINITELY express a function computation*.

Let us show now that the equivalence problem cannot be solved by any URM.

6.4.5 Theorem. (Equivalence problem) *The equivalence problem of URMs is the problem "given i and j; is $\phi_i = \phi_j$?"*

This problem is undecidable.

Proof. We will show that if we have a URM that solves the *equivalence problem*, "yes"/"no", then we have a URM that *solves the halting problem too*! (A contradiction to our assumption.)

So assume (URM) E solves the *equivalence problem*.

Let us use E to answer the question "$a \in K$"—that is, "$\phi_a(a)\downarrow$", for any a.

So, let a be given (inputted) (2)

Consider these two computable functions given by:

For all x:

$$Z(x) = 0$$

and

$$\widetilde{Z}(x) = \begin{cases} 0 & \text{if } x = 0 \wedge \phi_a(a)\downarrow \\ 0 & \text{if } x \neq 0 \end{cases}$$

Both functions are intuitively computable:

For Z we already have actually constructed a URM M that computes it.

(See (e-) Class/notes/text.)

For $\widetilde{Z}$ and input x compute as follows:

- Print 0 and stop if $x \neq 0$.
- On the other hand, if $x = 0$ then, *call* $U^{(P)}(a, a)$, which is the same as $\phi_a(a)$ (cf. 6.3.2).

  If this ever halts just print 0 and halt; *otherwise let it loop forever*.

By CT and the pseudo code above (two bullets), $\widetilde{Z}$ *has a URM program, say* $\widetilde{M}$.

We can *compute* the locations i and j of M and $\widetilde{M}$ respectively by going down the list of all $N^{\mathbf{w}}_{\mathbf{w}'}$ ($List_3$, p.167).

Thus $Z = \phi_i$ and $\widetilde{Z} = \phi_j$.

Since we ASSUMED that E solves the equivalence problem, now feed it i and j and let it crank.

By definition of a decider, E will terminate with answer one of:

- **0**. Then $Z(x) = \widetilde{Z}(x)$, for all x. *But lhs is defined at $x = 0$*, thus so is rhs. *We conclude $\phi_a(a)\downarrow$.*

- **1**. $Z(x) = \widetilde{Z}(x)$, is *NOT* true for all x. The only possibility for that is $\neg\Big(Z(0) = \widetilde{Z}(0)\Big)$ (for all other x-values, lhs=rhs).

  This can only be because rhs is undefined. We conclude $\phi_a(a)\uparrow$.

We just solved the halting problem using E as a subroutine! IMPOSSIBLE. So E does not exist. □

Chapter 7

(un)Computability via "Church's Thesis" and the S-m-n theorem; Part II

This is Part II of our Uncomputability notes.

We introduce "half-computable" relations $Q(\vec{x})$ here.

These play a central role in Computability.

The term "half-computable" is temporary here (replaced by "semi-computable" in the literature) describes them well: For each of these relations there is a URM M that *will halt precisely for the inputs* $\vec{a}$ *that make the relation true*:

i.e., $\vec{a} \in Q$ or equivalently $Q(\vec{a})$ is true.

For the inputs that make the relation false — $\vec{b} \notin Q$— M loops forever.

That is, M *verifies* membership but does not *yes/no-decide* it by halting and "printing" the appropriate 0 (yes) or 1 (no).

Can't we tweak the verifier M into a *decider* M' for such a Q?

No, not in general! For example, the *halting set* K has a verifier

Right? $x \in K \equiv \phi_x(x)\downarrow \equiv U^{(P)}(x,x)\downarrow$.

So *any program* M^X_Y for the partial recursive $\lambda x.U^{(P)}(x,x)$ is a *verifier* for $x \in K$. See also 7.1.2 below.

But we *KNOW* that $x \in K$ has *NO decider*!

Since the "yes" of a verifier M is signaled by halting but the "no" is signaled by looping forever,

the definition below does not require the verifier to print 0 for "yes". Here "yes" equals "halting".

7.1 Semi-decidable relations (or sets)

Oct. 27, 2021

7.1.1 Definition. (Semi-recursive sets)

▶ A relation $Q(\vec{x}_n)$ is *semi-decidable* or *semi-recursive* —what we called suggestively "*half-computable*" earlier—

iff

▶ There is a URM, M, which on input $\vec{x}_n$ ***has a (halting!) computation iff*** $\vec{x}_n \in Q$.

The output of M ***is unimportant!***

A *more mathematically precise* way to say the above is:

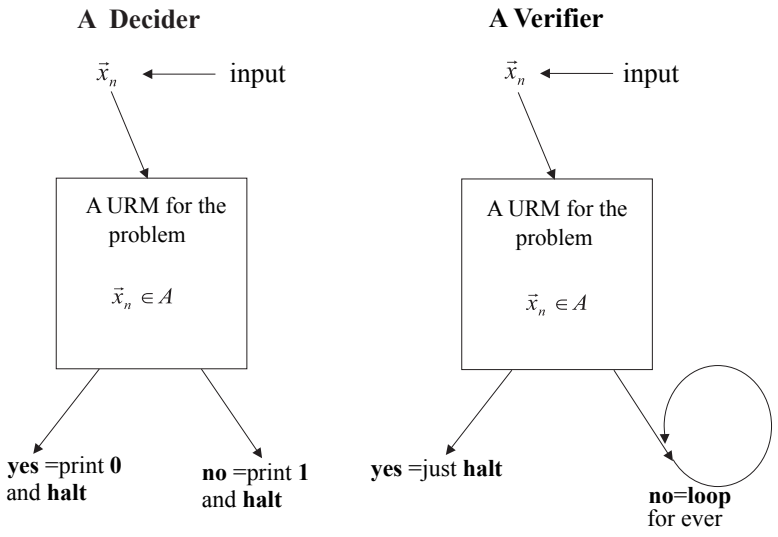
A relation $Q(\vec{x}_n)$ is *semi-decidable* or *semi-recursive* iff there is an $f \in \mathcal{P}$ such that

$$Q(\vec{x}_n) \equiv f(\vec{x}_n)\downarrow \tag{1}$$

Clearly, an $f \in \mathcal{P}$ is some $M^{\vec{\mathbf{x}}_n}_{\mathbf{y}}$. Thus, M is a verifier for Q.

The set of *all* semi-decidable relations we will denote by $\mathcal{P}_*$.[†] □

The following figure shows the two modes of handling a query, "$\vec{x}_n \in A$", by a URM.



[†]This is not a standard symbol in the literature. Most of the time the set of all semi-recursive relations has *no* symbolic name! We are using this symbol in analogy to $\mathcal{R}_*$—the latter being fairly "standard".

Here is an important semi-decidable set.

7.1.2 Example. *K is semi-decidable.* To work within the formal definition (7.1.1) we note that the function $\lambda x.\phi_x(x)$ is in $\mathcal{P}$ via the universal function theorem of Part I: $\lambda x.\phi_x(x) = \lambda x.U^{(P)}(x,x)$ and we know $U^{(P)} \in \mathcal{P}$.

Thus $x \in K \equiv \phi_x(x)\downarrow$ settles it. By Definition 7.1.1 (statement labeled (1)) we are done. □

7.1.3 Example. Any *recursive* relation A *is also semi-recursive*.

That is,

$$\mathcal{R}_* \subseteq \mathcal{P}_*$$

Indeed, intuitively, all we need to do to *convert* a *decider* for $\vec{x}_n \in A$ into a *verifier* is to "intercept" the "print 1"-step and convert it into an "infinite loop", for example,

k : **goto** k

By CT we can certainly do the whole thing via a URM implementation.

One more way to do this: Totally mathematical this time! (no CT needed!)

OK,

$$f(\vec{x}_n) = \text{if } c_A(\vec{x}_n) = 0 \text{ then } 0 \text{ else } \emptyset(\vec{x}_n)$$

That is, using the sw function that is in $\mathcal{PR}$ and hence in $\mathcal{P}$, as in

$$f(\vec{x}_n) = \text{if } \underset{\downarrow}{c_A(\vec{x}_n)} = 0 \text{ then } \underset{\downarrow}{0} \text{ else } \underset{\downarrow}{\emptyset(\vec{x}_n)}$$
$$z \qquad\qquad u \qquad\qquad w$$

$\emptyset$ *is, of course, the empty function* which by Grz-Ops can have any number of arguments we please! For example, we may take

$$\emptyset = \lambda\vec{x}_n.(\mu y)g(y, \vec{x}_n)$$

where $g = \lambda y\vec{x}_n.SZ(y) = \lambda y\vec{x}_n.1$.

□

An important observation following from the above examples deserves theorem status:

7.1.4 Theorem. $\mathcal{R}_* \subset \mathcal{P}_*$

Proof. The $\subseteq$ part of "$\subset$" is Example 7.1.3 above.

The $\neq$ part is due to $K \in \mathcal{P}_*$ (7.1.2) and the fact that the halting problem is unsolvable ($K \notin \mathcal{R}_*$).

So, there are sets in $\mathcal{P}_*$ (e.g., K) that are not in $\mathcal{R}_*$.

□

What about $\overline{K}$, that is, the *complement*

$$\overline{K} = \mathbb{N} - K = \{x : \phi_x(x) \uparrow\}$$

of K? *Is it perhaps semi-recursive (*verifiable*)?*

The following general result helps us answer the above question *negatively*.

7.1.5 Theorem. *A relation $Q(\vec{x}_n)$ is recursive* iff ***both*** *$Q(\vec{x}_n)$ and $\neg Q(\vec{x}_n)$ are semi-recursive.*

Proof. **IF part**. We want to prove that some URM, N, **decides**

$$\vec{x}_n \in Q$$

We take two *verifiers*, *M for "$\vec{x}_n \in Q$"* and *M' for "$\vec{x}_n \in \overline{Q}$"*,[†] and run them —on input $\vec{x}_n$— as "co-routines" (i.e., we crank them simultaneously).

If M halts, then we stop everything and print "0" (i.e., "yes").

If M' halts, then we stop everything and print "1" (i.e., "no").

CT tells us that we can put the above —if we want to— into a single URM, N.

ONLY IF part. If Q is in $\mathcal{R}_*$, then so is $\overline{Q}$, by closure under $\neg$ (3.4.7), and thus each is semi-recursive by Theorem 7.1.4. □

[†]We can do that, i.e., M and M' exist, since both Q and $\overline{Q}$ are semi-recursive.

7.1.6 Example. $\overline{K} \notin \mathcal{P}_*$.

Now, **this** ($\overline{K}$) is a horrendously unsolvable problem! This problem is so hard it is not even ***semi***-decidable!

Why? Well, if instead it were $\overline{K} \in \mathcal{P}_*$, then combining this with Example 7.1.2 and Theorem 7.1.5 we would get $K \in \mathcal{R}_*$, which we know is not true. □

7.1.7 Theorem. (Restricted Boolean closure of $\mathcal{P}_*$) *$\mathcal{P}_*$ is closed under the "positive" Boolean operations $\vee, \wedge$ but is not closed under $\neg$.*

Proof. That closure under $\neg$ fails is the content of Example 7.1.6.

Now let $P(\vec{x})$ and $Q(\vec{y})$ be two semi-recursive relations and M and N are verifiers for the two respectively.

- *Here is a verifier for $P(\vec{x}) \vee Q(\vec{y})$*:

  1. Input $\vec{x}$ to M and $\vec{y}$ to N.
  2. Run M and N simultaneously (in parallel, as co-routines).
  3. If ANY OF THE TWO URMs halts, then stop everything.

  By CT there is a URM $R_\vee$ that does exactly 1+2+3 above. This URM with input $(\vec{x}, \vec{y})$ will halt iff $P(\vec{x}) \vee Q(\vec{y})$ is true, thus is a verifier of this predicate and therefore the predicate is semi-recursive.

- *Here is a verifier for $P(\vec{x}) \wedge Q(\vec{y})$*:

  1. Input $\vec{x}$ to M and $\vec{y}$ to N.
  2. Run M and N simultaneously (in parallel, as co-routines).
  3. If BOTH OF THE TWO URMs halts, then stop everything.

  By CT there is a URM $R_\wedge$ that does exactly 1+2+3 above. This URM with input $(\vec{x}, \vec{y})$ will halt iff $P(\vec{x}) \wedge Q(\vec{y})$ is true, thus is a verifier of this predicate and therefore the predicate is semi-recursive. □

7.2 Unsolvability via Reducibility

We turn our attention now to a **methodology** towards discovering new undecidable problems, and also new non semi-recursive problems, beyond the ones we learnt about so far, which are just,

1. $x \in K$,

2. $\phi_i = \phi_j$ (equivalence problem)

3. and $x \in \overline{K}$.

In fact, we will learn shortly that $\phi_i = \phi_j$ is worse than undecidable; just like $\overline{K}$ it too is *not even semi-decidable*.

The tool we will use for such discoveries is the concept of *reducibility* of one set to another:

7.2.1 Definition. (Strong reducibility) For any two subsets of $\mathbb{N}$, A and B, we write

$$A \leq_m B^{\dagger}$$

or more simply

$$A \leq B \tag{1}$$

pronounced *A is strongly reducible to B*, meaning that there is a (total) *recursive* function f such that

$$x \in A \equiv f(x) \in B \tag{2}$$

We say that "*the reduction is effected by f*".

The last sentence has the notation $A \leq^f B$. □

In words, $A \leq_m B$ says that we can *algorithmically* solve the problem $x \in A$ if we know how to solve $z \in B$. The algorithm is:

1. Given x.
2. Given the known "subroutine" $z \in B$.

[†]The subscript m stands for "many one", and refers to f. We do not require it to be 1-1, that is; *many* (inputs) to *one* (output) will be fine.

3. Compute $f(x)$.
4. Give the same answer for $x \in A$ (true or false) as *you do for* $f(x) \in B$.

When $A \leq_m B$ holds, then, intuitively,

"A is easier than B to either decide or verify"

since if we know how to *decide* or (only) *verify* membership in B then we can decide or (only) verify membership in A: *"$x \in A$?"*

All we have to do is compute $f(x)$ and ask instead the question "$f(x) \in B$" which we can *decide* or *verify*.

This observation has a very precise counterpart (Theorem 7.2.3 below).

7.2.2 Lemma. *If $Q(y,\vec{x}) \in \mathcal{P}_*$ and $\lambda\vec{z}.f(\vec{z}) \in \mathcal{R}$, then $Q(f(\vec{z}),\vec{x}) \in \mathcal{P}_*$.*

Proof. By Definition 7.1.1 there is a $g \in \mathcal{P}$ such that

$$Q(y,\vec{x}) \equiv g(y,\vec{x})\downarrow \tag{1}$$

Now, for any $\vec{z}$, $f(\vec{z})$ is some number which if we plug into y in (1) we get an equivalence:

$$Q(f(\vec{z}),\vec{x}) \equiv g(f(\vec{z}),\vec{x})\downarrow \tag{2}$$

But $\lambda\vec{z}\vec{x}.g(f(\vec{z}),\vec{x}) \in \mathcal{P}$ by Grz-Ops. Thus, (2) and Definition 7.1.1 yield $Q(f(\vec{z}),\vec{x}) \in \mathcal{P}_*$. □

7.2.3 Theorem. *If $A \leq^g B$ in the sense of 7.2.1, then*

(i) if $B \in \mathcal{R}_*$, then also $A \in \mathcal{R}_*$

(ii) if $B \in \mathcal{P}_*$, then also $A \in \mathcal{P}_*$

Proof.

(*i*) The assumption says that $z \in B$ is in $\mathcal{R}_*$.

So is $g(x) \in B$ by Grz. Ops. (4.1.2).

But $x \in A \equiv g(x) \in B$, so $x \in A$ is in $\mathcal{R}_*$.

(ii) Let $z \in B$ be in $\mathcal{P}_*$.

By 7.2.2, so is $g(x) \in B$. *But this says* $x \in A$. $\square$

Taking the "contrapositive", we have at once:

7.2.4 Corollary. *If $A \leq B$ in the sense of 7.2.1, then*

(i) *if $A \notin \mathcal{R}_*$, then also $B \notin \mathcal{R}_*$*

(ii) *if $A \notin \mathcal{P}_*$, then also $B \notin \mathcal{P}_*$*

Nov. 1, 2021

7.2.5 Example. This is important! We saw that functions that are nontotal, such that x^y (fails at $x = y = 0$) and $\lfloor x/y \rfloor$ (fails infinitely often, for $y = 0$ and all x, that is) can be extended by total (for emphasis) *primitive recursive functions*.

Recall that "g extends f" —in symbols $f \subseteq g$— means that whenever $f(a) \downarrow$ then $f(a) = g(a)$. Of course there may be b where $f(b) \uparrow$ and yet $g(b) \downarrow$.

I cautioned at the time we discussed x^y and $\lfloor x/y \rfloor$ that it is *NOT true* that this extension to a total **computable function** can always happen!

Here is a counterexample: The partial recursive function $\lambda x.\phi_x(x) + 1$ —Why is it partial recursive?— *CANNOT* be extended to a total recursive function g of one argument x.

▶ Of course, MATHEMATICALLY (but *NOT* **computationally**!) it can be extended in infinitely many different and indeed total ways!

For example, $h = \lambda x.if\ \phi_x(x)+1 \uparrow\ then\ 0\ else\ \phi_x(x)+1$ is a total extension of $\lambda x.\phi_x(x)+1$. But neither it, NOR ANY OTHER total extension is computable!◀

Here it goes, à la Cantor: Say

$$\lambda x.\phi_x(x)+1 \subseteq g \in \mathcal{R}$$

Then, by the universal function theorem, for some $i \in \mathbb{N}$, it is

$$g = \phi_i \tag{1}$$

What is $g(i)$?

Well, by (1) $g(i) = \phi_i(i)$. Hence $\phi_i(i)\downarrow$. But then —by $\lambda x.\phi_x(x)+1 \subseteq g$— $g(i) = \phi_i(i)+1$.

We ended up (red and blue) with $\phi_i(i) = \phi_i(i)+1$ which is impossible since both sides of "=" are defined but different numbers! □

I asked you in Assignment #1 to show that $\lambda x.\phi_x(x)$ too does NOT have a total computable extension.

Hint. Say such an extension is h. Work with the total computable $\lambda x.h(x)+1$ and try to get a contradiction.

We return to the task of discovering undecidable and unverifiable (not verifiable) problems.

We can use K and $\overline{K}$ as a "*yardsticks*" —or *reference* "problems"— and discover *new* undecidable and also *non semi-decidable* problems.

The idea of Corollary 7.2.4 is applicable to the so-called "complete index sets".

7.2.6 Definition. (Complete Index Sets) Let $\mathcal{C} \subseteq \mathcal{P}$ and $A = \{x : \phi_x \in \mathcal{C}\}$.

A is thus the set of **ALL** programs (known by their addresses) x that compute any *unary* $f \in \mathcal{C}$:

▶ Indeed, let $\lambda x.f(x) \in \mathcal{C}$. Thus $f = \phi_i$ for some i. Then $i \in A$.

The above is not specific to a "privileged" i.
This is true of ***all*** ϕ_m *that equal* f.

That is why we call A a *complete* **index** (programs-) set: For any $f \in \mathcal{C}$, ALL its programs i * are in A. □

*That is, i such that $f = \phi_i$.

We embark on several examples, but first note the *FORM of S-m-n Theorem* that we will be using going forward:

7.2.7 Theorem. (S-m-n in practice) *If $\psi \in \mathcal{P}$ has two arguments, then there is a unary $h \in \mathcal{R}$ such that*

$$\psi(x,y) = \phi_{h(x)}(y) \tag{1}$$

for all x, y.

Proof. *Fix an i* such that $\psi(x,y) = \phi_i^{(2)}(x,y)$, for all x, y.

By S-m-n (6.3.3), we have a recursive $\lambda ix.S_1^1(i,x)$ such that

$$\phi_i^{(2)}(x,y) = \phi_{S_1^1(i,x)}(y)$$

for all i, x, y.

But i is fixed.

Thus $\lambda x.S_1^1(i,x)$ is the "h" we want. □

7.2.8 Example. The set $A = \{x : \text{ran}(\phi_x) = \emptyset\}$ is not semi-recursive.

Recall that "range" for $\lambda x.f(x)$, denoted by $\text{ran}(f)$, is defined by

$$\{x : (\exists y) f(y) = x\}$$

We will try to show that

$$\overline{K} \leq A \tag{1}$$

If we can do that much, then Corollary 7.2.4, part ii, will do the rest since $\overline{K} \notin \mathcal{P}_*$.

Well, define

$$\psi(x,y) = \begin{cases} 0 & \text{if } \phi_x(x)\downarrow \quad \text{Says } x \in K \\ \uparrow & \text{if } \phi_x(x)\uparrow \quad \text{Says } x \in \overline{K} \end{cases} \tag{2}$$

Here is how to compute ψ:

- Given x, y, ignore y.
- Call $\phi_x(x)$ —that is, $U^{(P)}(x,x)$,

- *If the call ever returns*, then print "0" and halt everything.
- *If it never returns*, then this agrees with the specified in (2) behaviour for $\psi(x, y)$ in this case.

By CT, ψ is in $\mathcal{P}$, so, by the S-m-n Theorem, there is a recursive h such that

$$\psi(x, y) = \phi_{h(x)}(y), \text{ for all } x, y$$

You may NOT use S-m-n UNTIL after you have proved that your "$\lambda xy.\psi(x, y)$" is in $\mathcal{P}$.

We can rewrite this as,

$$\phi_{h(x)}(y) = \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \tag{3}$$

or, rewriting (3) *without arguments* (as *equality of functions*, not equality of function calls)

$$\phi_{h(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \downarrow \\ \emptyset & \text{if } \phi_x(x) \uparrow \end{cases} \quad \leftarrow \text{says } x \in \overline{K} \tag{3'}$$

In (3′), $\emptyset$ stands for $\lambda y.\uparrow$, the empty function.

Thus,

$$h(x) \in A \text{ iff } \text{ran}(\phi_{h(x)}) = \emptyset \overbrace{\text{iff}}^{\text{bottom case in } 3'} x \in \overline{K}$$

The above says $x \in \overline{K} \equiv h(x) \in A$, hence $\overline{K} \leq^h A$, and thus $A \notin \mathcal{P}_*$ by Corollary 7.2.4, part ii. □

Recall

$$K \stackrel{Def}{=} \{x : \phi_x(x) \downarrow\}$$
$$\overline{K} \stackrel{Def}{=} \{x : \phi_x(x) \uparrow\}$$

7.2.9 Example. The set $B = \{x : \phi_x \text{ has finite domain}\}$ is not semi-recursive.

This is really easy (once we have done the previous example)! All we have to do is "talk about" our findings, above, differently!

We use the same ψ as in the previous example, as well as the same h as above, obtained by S-m-n.

Looking at $(3')$ above we see that the top case has infinite domain, while the bottom one has finite domain (indeed, empty). Thus,

$$h(x) \in B \text{ iff } \phi_{h(x)} \text{ has finite domain } \overbrace{\text{iff}}^{\text{bottom case in } 3'} \phi_x(x)\uparrow$$

The above says $x \in \overline{K} \equiv h(x) \in B$, hence $\overline{K} \leq B$, hence $B \notin \mathcal{P}_*$ by Corollary 7.2.4, part ii. □

7.2.10 Example. Let us "mine" $(3')$ twice more to obtain two more important undecidability results.

1. Show that $G = \{x : \phi_x \text{ is a constant function}\}$ is undecidable.

   We (re-)use $(3')$ of 7.2.8. Note that in $(3')$ the top case defines a constant function, but the bottom case defines a non-constant. Thus

   $$h(x) \in G \equiv \phi_{h(x)} = \lambda y.0 \equiv \text{top case in } (3') \equiv x \in K$$

   Hence $K \leq G$, therefore $G \notin \mathcal{R}_*$.

2. Show that $I = \{x : \phi_x \in \mathcal{R}\}$ is undecidable. Again, we retell what we can read from $(3')$ in words that are relevant to the set I:

   $$h(x) \in I \overset{\emptyset \notin \mathcal{R}!}{\equiv} \phi_{h(x)} = \lambda y.0 \equiv x \in K$$

   Thus $K \leq I$, therefore $I \notin \mathcal{R}_*$. □

In a future section we will sharpen the result 2 of the previous example.

7.2.11 Example. (The Equivalence Problem, again) We now revisit the equivalence problem and show **it is worse than unsolvable** (cf. 6.4.5):

The relation $\phi_x = \phi_y$ is not semi-decidable.

By 7.2.2, if the 2-variable predicate above is in $\mathcal{P}_*$ then so is $\lambda x.\phi_x = \phi_y$, i.e., *taking a constant for y, that is, fixing the program y*.

Choose then for y a ϕ-index for the *empty function*.

In short,

If the equivalence problem is VERIFIABLE, then so is

$$\phi_x = \emptyset$$

$$Eq = \{x : \phi_x = \emptyset\} = \{x : \text{ran}(\phi_x) = \emptyset\} = A$$

which says the same thing as

$$\text{ran}(\phi_x) = \emptyset$$

We saw that this is NOT SEMI-RECURSIVE in 7.2.8.

□ 

7.2.12 Example. *The set $C = \{x : ran(\phi_x)$ is finite$\}$ is not semi-decidable.*

Here we cannot reuse (3′) above, because **both** cases in the definition by cases —top and bottom— have functions of ***finite range***.

We want *one* case to have a function of finite range, but the *other* to have infinite range.

Aha! This motivates us to choose a different "ψ" (hence a different "h"), and retrace the steps we took above.

OK, define

$$g(x,y) = \begin{cases} y & \text{if } \phi_x(x)\downarrow \\ \uparrow & \text{if } \phi_x(x)\uparrow \end{cases} \qquad (ii)$$

Here is an algorithm for g:

- Given x, y.
- Call $\phi_x(x)$ —i.e., call $U^{(P)}(x,x)$.
- *If this ever returns, then print* "y" and halt everything.

- *If it never returns* from the call, this is the correct behaviour for $g(x,y)$ as well:

  namely, we want $g(x,y)\uparrow$ if $x \in \overline{K}$.

By CT, g is partial recursive, thus by S-m-n, for some recursive unary k we have

$$g(x,y) = \phi_{k(x)}(y), \text{ for all } x, y$$

Thus, by (ii)

$$\phi_{k(x)} = \begin{cases} \lambda y.y & \text{if } x \in K \\ \emptyset & \text{othw, i.e., } x \in \overline{K} \end{cases} \qquad (iii)$$

Hence,

$$k(x) \in C \text{ iff } \phi_{k(x)} \text{ has finite range } \overbrace{\text{iff}}^{\text{bottom case in } iii} x \in \overline{K}$$

That is, $\overline{K} \leq^k C$ and we are done. □

7.2.13 Exercise. Show that $D = \{x : \text{ran}(\phi_x) \text{ is infinite}\}$ is undecidable. □

7.2.14 Exercise. Show that $F = \{x : \text{dom}(\phi_x) \text{ is infinite}\}$ is undecidable. □

Nov. 3, 2021
Enough "negativity"!

Here is an important "positive result" that helps to prove that certain relations *ARE* semi-decidable:

7.2.15 Theorem. (Projection theorem; Part I) *A relation $Q(\vec{x}_n)$ that is expressible as*

$$Q(\vec{x}_n) \equiv (\exists y)S(y, \vec{x}_n) \tag{1}$$

where $S(y, \vec{x}_n)$ is recursive is semi-recursive.

Q is obtained by "projecting" S along the y-co-ordinate, hence the name of the theorem.

Proof. Let $S \in \mathcal{R}_*$, and Q be connected as in (1) of the theorem.

Clearly (*How so?*),

$$(\exists y)S(y, \vec{x}_n) \equiv (\mu y)S(y, \vec{x}_n)\downarrow \tag{2}$$

and we know that

$$(\mu y)S(y, \vec{x}_n) \overset{Def}{=} (\mu y)c_S(y, \vec{x}_n), \text{ for all } \vec{x}_n \tag{3}$$

Hence

$$\lambda \vec{x}_n.(\mu y)c_S(y, \vec{x}_n) \in \mathcal{P}$$

by closure of $\mathcal{P}$ under *UNbounded* search. *Thus so is* $\lambda \vec{x}_n.(\mu y)S(y, \vec{x}_n)$ *by (3).*

Now (1) and (2) give

$$Q(\vec{x}_n) \equiv (\mu y)S(y, \vec{x}_n) \downarrow$$

We are done by Def. 7.1.1 (1). □

7.2.16 Example. The set $A = \{(x, y, z) : \phi_x(y) = z\}$ is semi-recursive.

Here is a verifier for the above predicate:

Given input x, y, z. **Comment**. Note that $\phi_x(y) = z$ is true iff two things happen: (1) $\phi_x(y) \downarrow$ **and** (2) the computed value is z.

1. Given x, y, z.
2. Call $\phi_x(y) = U^{(P)}(x, y)$.
3. If the call returns, then
   - If the output of $\phi_x(y)$ equals z, then halt everything (the "yes" output).
   - If the output of $\phi_x(y)$ does NOT equal z, then get into an infinite loop (the "no" case).
4. If the $\phi_x(y) \uparrow$, *then keep looping* (say "no", by looping).

By CT the above informal verifier can be formalised as a URM M. □

7.3 Projection Theorem II

This section provides a new powerful tool (that we will use again in the chapter on complexity) AND proves the converse of Projection Theorem Part I.

How can we trace a (computation of a) URM ?

Exactly in the same manner that we learnt to trace a commercially available program such as C.

7.3.1 Computation-simulating functions

Given a URM $M_{X_1}^{\vec{X}_m}$ where —without loss of generality— we selected X_1 as the output variable.

Let all its variables be

$$\overbrace{X_1, \ldots, X_m}^{inputs}, \overbrace{X_{m+1}, \ldots, X_n}^{Non\ inputs} \tag{1}$$

For *any input* $\vec{a}_m$, M's computation can be tabulated in a (*potentially infinite*) table —p.225 below— where for each $y \geq 0$, row y contains the *values* of *ALL the variables in (1)* as well the value of the **Instruction Pointer** IP —that points to the *CURRENT* instruction— *at step* y.

A "step" is the *act* of executing ONE instruction of M and reaching the next *CURRENT* instruction.

At step zero, $(y = 0)$ the computation *ponders the FIRST instruction* of M after the *"I/O Agent" initialised* the input variables and has set all non-input variables to zero.

At step 0 we have NO PREVIOUS INSTRUCTION.

The entries on the zeroth row are self-evident.

Table 7.1: M Simulation Table

y	IP	X_1	X_2	$\dots$	X_m	X_{m+1}	X_{m+2}	$\dots$	X_n
0	1	a_1	a_2	$\dots$	a_m	0	0	$\dots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$
i	L	b_1	b_2	$\dots$	b_m	b_{m+1}	b_{m+2}	$\dots$	b_n
$i+1$	L'	b'_1	b'_2	$\dots$	b'_m	b'_{m+1}	b'_{m+2}	$\dots$	b'_n
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$

The process for filling the table is algorithmic as follows:

Going *from row* i to *row* $i+1$ (Cf. 2.2.2) we have cases according to where L points.

1. L points to $X_k \leftarrow r$. Then $b'_k = r$ while $b'_j = b_j$ for $j \neq k$. Also $L' = L + 1$.

2. L points to $X_k \leftarrow X_k + 1$. Then $b'_k = b_k + 1$ while $b'_j = b_j$ for $j \neq k$. Moreover $L' = L + 1$.

3. L points to $X_k \leftarrow X_k \dot{-} 1$. Then $b'_k = b_k \dot{-} 1$ while $b'_j = b_j$ for $j \neq k$. Moreover $L' = L + 1$.

4. L points to **stop**. Then $b'_j = b_j$ *for all* $j \neq n$. Moreover $L' = L$.

5. L points to **if** $X_k = 0$ **goto** R **else goto** Q. Then $b'_j = b_j$ *for all* $j \neq n$. Moreover $L' = if\ b_k = 0\ then\ R\ else\ Q$.

Note that at "time" y each X_j *and the* IP *function* hold a value that depends on the initial $\vec{a}_m$ —and on the "time" y.

▶ Thus we associate with each X_j and with the IP a *TOTAL function* —$\lambda y\vec{a}_m.X_j(y,\vec{a}_m)$ and $\lambda y\vec{a}_m.IP(y,\vec{a}_m)$.

Since I can produce each such function-value, for the X_j and IP —for example, *by hand*— in the *mechanical way* indicated,

by CT, each such function X_j and IP is *RECURSIVE*.

The above is important and we record it as a theorem.

7.3.1 Theorem. *The functions* $\lambda y\vec{a}_m.X_j(\vec{a}_m,y)$ —*for* $j=1,2,\ldots n$— *and* $\lambda y\vec{a}_m.IP(y,\vec{a}_m)$ *return, for any* input $\vec{a}_m$ *(into* $\vec{X}_m$*) and* step-count y*, the value stored in the variable* X_j *and* the instruction pointer IP that points at **the current instruction**, *all this at step* y.

All these functions are in $\mathcal{R}$.

Important!

7.3.2 Theorem. *With reference to the URM* M *that we "traced" above, we have that* $g=M^{\vec{X}_m}_{X_1}$ *halts for input* $\vec{a}_m$ *iff there is some time-step value* y *where* M *makes its* ***stop*** *instruction current.*

That is

$g(\vec{a}_m)\downarrow\equiv(\exists y)IP(y,\vec{a}_m)=k$, *where* k *is the label of* **stop**

7.3.3 Theorem. (Projection Theorem Part II) *IF* $Q(\vec{x}_m)$ *is semi-recursive,* THEN *there is a recursive* $P(y, \vec{x}_m)$ *such that*

$$Q(\vec{x}_m) \equiv (\exists y) P(y, \vec{x}_m)$$

Proof. By Definition 7.1.1,

$$Q(\vec{a}_m) \equiv g(\vec{a}_m)\downarrow$$

where $g \in \mathcal{P}$.

Let then $g = M_{X_1}^{\vec{X}_m}$.

By 7.3.2,

$g(\vec{a}_m)\downarrow \equiv (\exists y) IP(y, \vec{a}_m) = k$, where k labels **stop** in M

But $IP(y, \vec{a}_m) = k$ is recursive so we may *take it as the "*$P(y, \vec{x}_m)$*" we want.* □

A user-friendly Introduction to (un)Computability and Unprovability via "Church's Thesis" Part III

7.4 Recursively Enumerable Sets

In this section we explore the rationale behind the alternative name "*recursively enumerable*" —r.e.— or "*computably enumerable*" —c.e.— that is used in the literature for *the semi-recursive or semi-computable* sets/predicates.

To avoid cumbersome codings (of n-tuples, by single numbers) *we restrict attention to the one variable case* in this section.

That is, our predicates are subsets of $\mathbb{N}$.

First we define:

7.4.1 Definition. A set $A \subseteq \mathbb{N}$ is called *computably enumerable* (c.e.) or *recursively enumerable* (r.e.) precisely if one of the following cases holds:

- $A = \emptyset$
- $A = \text{ran}(f)$, where $f \in \mathcal{R}$.

□

Thus, the c.e. or r.e. relations are exactly those we can ***algorithmically enumerate*** as **the set of outputs** of a (*total*) *recursive function*:

$$A = \{f(0), f(1), f(2), \ldots, f(x), \ldots\}$$

Hence the use of the term "c.e." replaces the non technical term "algorithmically" (in "algorithmically" enumerable) by the technical term "computably".

Note that we had to hedge and ask that $A \neq \emptyset$ *for any enumeration to take place*, because no recursive function (remember: these are total) can have an empty range.

Next we prove:

7.4.2 Theorem. ("c.e." or "r.e." vs. semi-recursive) *Any non empty* semi-recursive *relation* A $(A \subseteq \mathbb{N})$ *is the range of some (emphasis:* **total***)* recursive function *of one variable.*

Conversely, *every set* A *such that* $A = \text{ran}(f)$ —*where* $\lambda x.f(x)$ *is recursive*— *is* semi-recursive *(and, trivially, nonempty).*

Before we prove the theorem, here is an example:

7.4.3 Example. The set $\{0\}$ is c.e. Indeed, $f = \lambda x.0$, our familiar function Z, effects the enumeration *with repetitions (lots of them!)*

x	$=0$	1	2	3	4	$\ldots$
$f(x)$	$=0$	0	0	0	0	$\ldots$

□

Proof. of Theorem 7.4.2.

(I) **We prove the first sentence of the theorem**. So, let $A \neq \emptyset$ be *semi-recursive*.

By the projection theorem (see 7.2.15 and 7.3.3) there is a **recursive** relation $Q(y, x)$ such that

$$x \in A \equiv (\exists y)Q(y, x), \text{ for all } x \qquad (1)$$

Thus, the totality of the x in A are *the 2nd coordinates of ALL pairs (y, x) that satisfy $Q(y, x)$*.

So, to enumerate all $x \in A$ *just enumerate all pairs* (y, x), and OUTPUT x just in case $(y, x) \in Q$.

We enumerate *all POSSIBLE PAIRS* y, x by enumerating $z = \langle y, x\rangle = 2^{y+1}3^{x+1}$, that is,

$$(y = (z)_0, \quad x = (z)_1)$$

for each $z = 0, 1, 2, 3, \ldots$.

Recall that $A \neq \emptyset$. So fix an $a \in A$. f below does the enumeration!

$$f(z) = \begin{cases} (z)_1 & \text{if } Q((z)_0, (z)_1) \\ a & \text{othw} \end{cases}$$

The above is a definition by recursive cases hence f is a recursive function, and the values $x = (z)_1$ that it outputs for each $z = 0, 1, 2, 3, \ldots$ *enumerate* A.

(II) **Proof of the second sentence of the theorem**.

So, let $A = \text{ran}(f)$ —where f is recursive.

Thus,

$$x \in A \equiv (\exists y) f(y) = x \tag{1}$$

By Grz-Ops, plus the facts that $z = x$ *is in* $\mathcal{R}_*$ *and* the assumption $f \in \mathcal{R}$,

the relation $f(y) = x$ is *recursive*.

By (1) we are done by the Projection Theorem.

□

7.4.4 Corollary. *An $A \subseteq \mathbb{N}$ is semi-recursive iff it is r.e. (c.e.)*

Proof. For nonempty A this is Theorem 7.4.2. For empty A we note that this is r.e. by Definition 7.4.1 but is also semi-recursive by $\emptyset \in \mathcal{PR}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$. □

Corollary 7.4.4 allows us to prove some non-semi-recursiveness results by good old-fashioned Cantor diagonalisation.

See below.

Nov. 8, 2021

7.4.5 Theorem. *The complete index set $A = \{x : \phi_x \in \mathcal{R}\}$ is not semi-recursive.*

This sharpens the earlier undecidability result for A where we "only" proved $A \notin \mathcal{R}_*$.

Proof. Since *c.e. = semi-recursive*, we will prove instead that A is *not* c.e.

If not, note first that $A \neq \emptyset$ —e.g., $S \in \mathcal{R}$ and thus all ϕ-indices of S are in A.

Thus, theorem 7.4.2 applies and **there is an $f \in \mathcal{R}$ that enumerates A**:

$$A = \{f(0), f(1), f(2), f(3), \ldots\}$$

The above says:

ALL programs for unary $\mathcal{R}$-functions are $f(i)$'s.

Define now

$$d = \lambda x.1 + \phi_{f(x)}(x) \tag{1}$$

Seeing that $\phi_{f(x)}(x) = U^{(P)}(f(x), x)$ —*you remember $U^{(P)}$?*— we obtain $d \in \mathcal{P}$.

But $\phi_{f(x)}$ is total since *all the $f(x)$ are ϕ-indices of total functions* by the underlined *blue* comment above.

By the same comment,

$$d = \phi_{f(i)}, \text{ for some } i \tag{2}$$

Let us compute $d(i)$: $d(i) = 1 + \phi_{f(i)}(i)$ *by (1)*.

Also, $d(i) = \phi_{f(i)}(i)$ *by (2)*,

thus

$$1 + \phi_{f(i)}(i) = \phi_{f(i)}(i)$$

which is a contradiction *since both sides of "=" are defined*. □

One can take as d different functions, for example, either of $d = \lambda x.42 + \phi_{f(x)}(x)$ or $d = \lambda x.1 \div \phi_{f(x)}(x)$ works. And infinitely many other choices do!

7.5 Some closure properties of decidable and semi-decidable relations

We already *know* that

7.5.1 Theorem. *$\mathcal{R}_*$ is closed under all Boolean operations, $\neg, \wedge, \vee, \rightarrow, \equiv$, as well as under $(\exists y)_{<z}$ and $(\forall y)_{<z}$.*

How about closure properties of $\mathcal{P}_*$?

7.5.2 Theorem. *$\mathcal{P}_*$ is closed under $\wedge$ and $\vee$. It is also closed under $(\exists y)$, or, as we say, "under projection".*

Moreover it is closed under $(\exists y)_{<z}$ and $(\forall y)_{<z}$.
It is **not** *closed under negation (complement),* **nor** *under $(\forall y)$.*

Proof. (1. and 2. and 6. we have proved already in 7.1.7).

1. Let $Q(\vec{x}_n)$ be verified by a URM M, and $S(\vec{y}_m)$ be verified by a URM N.

   Here is how to semi-decide $Q(\vec{x}_n) \vee S(\vec{y}_m)$:

   Given input $\vec{x}_n, \vec{y}_m$, we call machine M with input $\vec{x}_n$, and machine N with input $\vec{y}_m$ and let them crank simultaneously (as "co-routines").

   If **either one** halts, then halt everything! This is the case of "yes" (input verified).

2. For $\wedge$ it is almost the same, but our halting criterion is different:

   Here is how to semi-decide $Q(\vec{x}_n) \wedge S(\vec{y}_m)$:

   Given input $\vec{x}_n, \vec{y}_m$, we call machine M with input $\vec{x}_n$, and machine N with input $\vec{y}_m$ and let them crank simultaneously ("co-routines").

   If **both** halt, then halt everything!

   By CT, each of the processes in 1. and 2. can be implemented by some URM.

3. **The $(\exists y)$ is very interesting as it relies on the Projection Theorem:**

   Let $Q(y, \vec{x}_n)$ be **semi**-decidable. Then, by Projection Theorem, a **decidable** $P(z, y, \vec{x}_n)$ exists such that

   $$Q(y, \vec{x}_n) \equiv (\exists z) P(z, y, \vec{x}_n) \tag{1}$$

   It follows that

   $$(\exists y) Q(y, \vec{x}_n) \equiv (\exists y)(\exists z) P(z, y, \vec{x}_n) \tag{2}$$

   This does *not* settle the story, as *I cannot readily conclude* that $(\exists y)(\exists z) P(z, y, \vec{x}_n)$ is semi-decidable ▶because the Projection Theorem requires a *single* $(\exists y)$ in front of a decidable predicate!

   Well, instead of saying that there are **two** values z and y that verify (along with $\vec{x}_n$) the predicate $P(z, y, \vec{x}_n)$, *I can say there is a PAIR of values* (z, y).

   In fact I can CODE the pair as $w = \langle z, y \rangle$ *and say* there is ONE value, w:

   $$(\exists w) P(\overbrace{(w)_0}^{z}, \overbrace{(w)_1}^{y}, \vec{x}_n)$$

   and thus I have —by (2) and the above—

   $$(\exists y) Q(y, \vec{x}_n) \equiv (\exists w) P((w)_0, (w)_1, \vec{x}_n) \tag{3}$$

But since $P((w)_0, (w)_1, \vec{x}_n)$ is **recursive** by the decidability of P *and* Grz-Ops, we end up in (3) quantifying the decidable $P((w)_0, (w)_1, \vec{x}_n)$ with just **one** $(\exists w)$. The Projection Theorem now applies!

4. For $(\exists y)_{<z} Q(y, \vec{x})$, where $Q(y, \vec{x})$ is semi-recursive, we first note that

$$(\exists y)_{<z} Q(y, \vec{x}) \equiv (\exists y)\Big(y < z \wedge Q(y, \vec{x})\Big) \qquad (*)$$

By $\mathcal{PR}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$, $y < z$ is semi-recursive. By closure properties established *SO FAR* in this proof, the rhs of $\equiv$ in $(*)$ is semi-recursive, thus so is the lhs.

5. For $(\forall y)_{<z}Q(y,\vec{x})$, where $Q(y,\vec{x})$ is semi-recursive, we first note that (by Strong Projection) a **decidable** P exists such that

$$Q(y,\vec{x}) \equiv (\exists w)P(w,y,\vec{x})$$

By the above equivalence, we need to prove that

$$(\forall y)_{<z}(\exists w)P(w,y,\vec{x}) \text{ is semi-recursive} \qquad (**)$$

$(**)$ says that

for **each** $y = 0, 1, 2, \ldots, z-1$ there is a w-value w_y — in general dependent on y— so that $P(w_y, y, \vec{x})$ holds

Since all those w_y are finitely many (z many!) there is a value u bigger than **all** of them (for example, take $u = \max(w_0, \ldots, w_{z-1}) + 1$). Thus $(**)$ says (i.e., **is equivalent to**)

$$(\exists u)(\forall y)_{<z}(\exists w)_{<u}P(w,y,\vec{x})$$

The blue part of the above is **decidable** (by closure properties of $\mathcal{R}_*$, since $P \in \mathcal{R}_*$ —you may peek at 7.5.1). We are done by *strong projection*.

6. Why is $\mathcal{P}_*$ *not closed under negation* (complement)?
   Because we know that $K \in \mathcal{P}_*$, but also know that $\overline{K} \notin \mathcal{P}_*$.

$$x \in \overline{K} \equiv \neg(x \in K)$$

7. Why is $\mathcal{P}_*$ not closed under $(\forall y)$?

   Well,

$$x \in K \equiv (\exists y)Q(y, x) \tag{1}$$

   for some recursive Q (Projection Theorem) and *by the known fact (quoted again above) that* $K \in \mathcal{P}_*$.

   (1) is equivalent to

$$x \in \overline{K} \equiv \neg(\exists y)Q(y, x)$$

   which in turn is equivalent to

$$x \in \overline{K} \equiv (\forall y)\neg Q(y, x) \tag{2}$$

   Now, by closure properties of $\mathcal{R}_*$ See 7.5.1), $\neg Q(y, x)$ is recursive, hence also is in $\mathcal{P}_*$ since $\mathcal{R}_* \subseteq \mathcal{P}_*$.

   Therefore, if $\mathcal{P}_*$ were closed under $(\forall y)$, then the above $(\forall y)\neg Q(y, x)$ *would be semi-recursive*.
   But that is $x \in \overline{K}$! □

7.6 Some tricky reductions

This section highlights a more sophisticated reduction scheme that *improves our ability to effect reductions of the type* $\overline{K} \leq A$.

7.6.1 Example. Prove that $A = \{x : \phi_x \text{ is a constant}\}$ is *not semi-recursive*. This is not amenable to the technique of saying "OK, if A is semi-recursive, then it is r.e. Let me show that it is not so by diagonalisation". This worked for $B = \{x : \phi_x \text{ is total}\}$ but *no obvious diagonalisation comes to mind for A.*

Nor can we simplistically say, OK, start by defining

$$g(x,y) = \begin{cases} 0 & \text{if } x \in \overline{K} \text{ (same as } \phi_x(x)\uparrow) \\ \uparrow & \text{othw} \end{cases}$$

The problem is that if we plan next to say "by CT g is partial recursive *hence by S-m-n, etc.*", *we shouldn't!*

The underlined part is wrong: $g \notin \mathcal{P}$, *provably*! (Why, INTUITIVELY, it should not be computable is obvious; right?)

▶ For if it *is* computable, then so is $\lambda x.g(x,x)$ by Grz-Ops.

But

$$g(x,x)\downarrow \text{ iff we have the top case, iff } x \in \overline{K}$$

In short,

$$x \in \overline{K} \equiv g(x,x)\downarrow$$

which proves that $\overline{K} \in \mathcal{P}_*$ using the verifier for "$g(x,x)\downarrow$". **Contradiction**. □

7.6.2 Example. (7.6.1 continued) Now, "**Plan B**" is to "**approximate**" the top condition $\phi_x(x)\uparrow$ (same as $x \in \overline{K}$).

The idea is that, "**practically**", if the computation $\phi_x(x)$ after a "huge" number of steps y has still not hit **stop**, this situation *approximates* —let me say once more, "practically"— the situation $\phi_x(x)\uparrow$. This fuzzy thinking suggests that we try next

$$f(x,y) = \begin{cases} 0 & \text{if } \phi_x(x) \text{ did not return in } \leq y \text{ steps} \\ \uparrow & \text{othw} \end{cases}$$

If the top condition is true for a given x it means that at step y the URM that we picked to compute $\phi_x(x)$ has not hit **stop** yet.

The "othw" says, of course, that the computation of the call $\phi_x(x)$ —or $U^{(P)}(x,x)$— *did return in* $\leq y$ *steps*.

Next task is to invoke an S-m-n theorem application, so we must show that f defined above is computable. Well here is an informal algorithm:

(0) **proc** $f(x,y)$
(1) **Call** $\phi_x(x)$; keep count of computation steps
(2) **Return** 0 if $\phi_x(x)$ did **not return** in $\leq y$ steps
(3) **"Loop"** if $\phi_x(x)$ **returned** in $\leq y$ steps

Of course, the "command" **Loop** means

"transfer to the subprogram" k : **goto** k.

By CT, the pseudo algorithm (0)–(3) is implementable as a URM. That is, $f \in \mathcal{P}$.

By S-m-n applied to f there is a recursive k such that

$$f(x,y) = \phi_{k(x)}(y) = \begin{cases} 0 & \text{if } \phi_x(x) \text{ did not return} \\ & \text{in} \leq y \text{ steps} \\ \uparrow & \text{othw} \end{cases} \tag{1}$$

Analysis of (1) in terms of the "key" conditions $\phi_x(x)\uparrow$ and $\phi_x(x)\downarrow$:

(A) Case where $\phi_x(x)\uparrow$.

Then, $\phi_x(x)$ did **not** halt in y steps, for any y.

Thus, by (1), we have $\phi_{k(x)}(y) = 0$, for all y, that is,

$$\phi_x(x)\uparrow \Longrightarrow \phi_{k(x)} = \lambda y.0 \tag{2}$$

(B) Case where $\phi_x(x)\downarrow$. Let $m =$ *smallest* y such that the call $\phi_x(x)$ ended in m steps. Therefore,

- for step counts $y = 0, 1, 2, \ldots, m-1$ the computation of $\phi_x(x)$ has not yet hit **stop**, so the **top** case of definition (1) holds. We get

$$\begin{array}{lllll} \text{for } y & =0, & 1, & \ldots, & m-1 \\ \phi_{k(x)}(y) & =0, & 0, & \ldots, & 0 \end{array}$$

- for step counts $y = m, m+1, m+2, \ldots$ the computation of $U^{(P)}(x,x)$ has already halted (it hit **stop**), so the **bottom** case of definition (1) holds. We get

$$\begin{array}{lllll} \text{for } y & =m, & m+1, & m+2, & \ldots \\ \phi_{k(x)}(y) & =\uparrow, & \uparrow, & \uparrow, & \ldots \end{array}$$

In short:

$$\phi_x(x)\downarrow \Longrightarrow \phi_{k(x)} = \overbrace{(0,0,\ldots,0)}^{\text{length } m} \qquad (3)$$

In

$$\phi_{k(x)} = \overbrace{(0,0,\ldots,0)}^{\text{length } m}$$

we depict the function $\phi_{k(x)}$ *as an array of its m output values*.

Thus, in Plain English, when $\phi_x(x)\downarrow$, the function $\phi_{k(x)}$ is NOT a constant! Not even total!

Our analysis yielded:

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x)\uparrow \\ \text{not a constant function} & \text{if } \phi_x(x)\downarrow \end{cases} \tag{4}$$

We conclude now as follows for $A = \{x : \phi_x$ is a constant$\}$:

$k(x) \in A$ iff $\phi_{k(x)}$ is a constant iff the top case of (4) applies
iff $\phi_x(x)\uparrow$

That is, $x \in \overline{K} \equiv k(x) \in A$, hence $\overline{K} \leq A$. □

7.6.3 Example. *Prove (again) that* $B = \{x : \phi_x \in \mathcal{R}\} = \{x : \phi_x \text{ is total}\}$ *is not semi-recursive.*

We piggy back on the previous example and the same f through which we found a $k \in \mathcal{R}$ such that

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \uparrow \\ \overbrace{(0,0,\ldots,0)}^{\text{length } m} & \text{if } \phi_x(x) \downarrow \end{cases} \tag{5}$$

The above is (4) of the previous example, but we will use different English words to describe the kind of functions we get in each of the two cases, which we displayed explicitly in (5).

Note that $\overbrace{(0,0,\ldots,0)}^{\text{length } m}$ is a non-recursive (nontotal) function listed as a finite array of outputs. On the other hand, the function in the top case is recursive. Thus we have

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \uparrow \\ \text{nonrecursive function} & \text{if } \phi_x(x) \downarrow \end{cases} \tag{6}$$

and therefore

$k(x) \in B$ iff $\phi_{k(x)} \in \mathcal{R}$ iff the top case of (6) applies iff $\phi_x(x) \uparrow$

That is, $x \in \overline{K} \equiv k(x) \in B$, hence $\overline{K} \leq B$. □

7.6.4 Example. We will prove that $D = \{x : \text{ran}(\phi_x) \text{ is infinite}\}$ is *not semi-recursive*.

We (heavily) piggy back on Example 7.6.2 above.

We want to find $j \in \mathcal{R}$ such that

$$\phi_{j(x)} = \begin{cases} \text{inf. range} & \text{if } \phi_x(x) \uparrow \\ \text{finite range} & \text{if } \phi_x(x) \downarrow \end{cases} \qquad (*)$$

OK, define ψ (almost) like f of Example 7.6.2 by

$$\psi(x,y) = \begin{cases} y & \text{if the call } \phi_x(x) \text{ did } \underline{\text{not}} \text{ return in } \leq y \text{ steps} \\ \uparrow & \text{othw} \end{cases}$$

Other than the trivial difference (function name) the important difference is that we force infinite range in the top case by outputting the input y.

The argument that $\psi \in \mathcal{P}$ goes as the one for f in Example 7.6.2. The only difference is that in the algorithm (0)–(3) we change "**Return** 0" to "**Return** y".

The question $\psi \in \mathcal{P}$ having been settled, by S-m-n there is a $j \in \mathcal{R}$ such that

$$\phi_{j(x)}(y) = \begin{cases} y & \text{if the call } \phi_x(x) \text{ did not return} \\ & \text{in } \leq y \text{ steps} \\ \uparrow & \text{othw} \end{cases} \qquad (\dagger)$$

Analysis of (†) in terms of the "key" conditions $\phi_x(x)\uparrow$ and $\phi_x(x)\downarrow$:

(I) Case where $\phi_x(x)\uparrow$.

Then, for all input values y, $\phi_x(x)$ is still not at **stop** after y steps. Thus by (†), we have $\phi_{j(x)}(y) = y$, for all y, that is,

$$\phi_x(x)\uparrow \Longrightarrow \phi_{j(x)} = \lambda y.y \qquad (1)$$

(II) Case where $\phi_x(x)\downarrow$. Let m = *smallest* y such that *the call* $\phi_x(x)$ *returned in* m *steps*.

As before we find that for $y = 0, 1, \ldots, m-1$ we have $\phi_{j(x)}(y) = y$, that is,

for y	$=0,$	$1,$	$\ldots,$	$m-1$
$\phi_{j(x)}(y)$	$=0,$	$1,$	$\ldots,$	$m-1$

and as before,

for y	$=m,$	$m+1,$	$m+2,$	$\ldots$
$\phi_{j(x)}(y)$	$=\uparrow,$	$\uparrow,$	$\uparrow,$	$\ldots$

that is,

$$\phi_x(x)\downarrow \Longrightarrow \phi_{j(x)} = (0, 1, \ldots, m-1) \text{ —finite range} \qquad (2)$$

(1) and (2) say that we got $(*)$ —p.253— above. Thus

$j(x) \in D$ **iff** $\mathrm{ran}(\phi_{j(x)})$ infinite **iff** top case holds, **iff** $\phi_x(x)\uparrow$

Thus $\overline{K} \leq D$ via j. □

Chapter 8

The Ackermann Function

Nov. 10, 2021

Overview

The "Ackermann function" was proposed, of course, by Ackermann. The version here is a simplification by Robert Ritchie.

It provides us with an example of a *recursive* function that is *not* in $\mathcal{PR}$. Unlike the example in Chapter 5, which provided an alternative such function by diagonalisation, the proof that the Ackermann function is not primitive recursive is by a *majorisation argument*. Simply, it is too big to be primitive recursive!

But this function is more than just *intuitively* computable! It *is* computable —no hedging— as we will show by proving it to be a member of $\mathcal{R}$ (in the next chapter) —mathematically, without help from CT.

8.1 A very fast growing function: Definition and properties

8.1.1 Definition. The Ackermann function, $\lambda nx.A_n(x)$, is given, for *all* $n \geq 0, x \geq 0$ by the equations

$$\begin{aligned} A_0(x) &= x+2 \\ A_{n+1}(x) &= A_n^x(2) \end{aligned}$$

where h^x is function iteration. □

For any $\lambda y.h(y)$, the function $\lambda xy.h^x(y)$ —the "iteration of h"— is given by the primitive recursion

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

It is obvious then that if $h \in \mathcal{PR}$ then so is $\lambda xy.h^x(y)$.

8.1.2 Remark. An alternative way to define the Ackermann function, extracted directly from Definition 8.1.1, is as follows:

$$\begin{aligned} A_0(x) &= x+2 \\ A_{n+1}(0) &= 2 \\ A_{n+1}(x+1) &= A_n(A_{n+1}(x)) \end{aligned}$$ □

8.1.3 Lemma. *For each* $n \geq 0$, $\lambda x.A_n(x) \in \mathcal{PR}$.

Proof. Induction on n: For the basis, clearly $A_0 = \lambda x.x+2 \in \mathcal{PR}$. Assume now the case for (arbitrary, fixed) n —i.e., $A_n \in \mathcal{PR}$— and go to that for $n+1$. Immediate from Definition 8.1.2, last two equations. □

It turns out that the function blows up in size far too fast with respect to the argument n. We now quantify this remark.

The following unassuming lemma is the key to proving the growth properties of the Ackermann function. It is also the least straightforward to prove, as it requires a *double induction* —at once on n and x— as dictated by the fact that the "recursion" of Definition 8.1.2 does not leave any argument fixed.

The above shows in particular that, for all n and all x, $A_n(x)\downarrow$. That is, $\lambda nx.A_n(x)$ is total.

8.1.4 Lemma. *For each* $n \geq 0$ *and* $x \geq 0$, $A_n(x) > x+1$.

Proof. We start an induction on n:

n-Basis. $n=0$: $\quad A_0(x) = x+2 > x+1$; true.

n-I.H.[†] For all x and a fixed (but unspecified) n, *assume* $A_n(x) > x+1$.

n-I.S.[‡] For all x and the above fixed (but unspecified) n, we must *prove* $A_{n+1}(x) > x+1$.

We do the n-I.S. by induction on x:

x-Basis. $x=0$: $\quad A_{n+1}(0) = 2 > 1$; true.

x-I.H. For the above fixed n, we now fix an x (but leave it unspecified) for which we *assume* $A_{n+1}(x) > x+1$.

x-I.S. For the above fixed (but unspecified) n and

[†]To be precise, what we are proving is "$(\forall n)(\forall x)A_n(x) > x+1$". Thus, as we start on an induction on n, its I.H. is "$(\forall x)A_n(x) > x+1$" for a fixed unspecified n.

[‡]To be precise, the step is to prove —from the basis and I.H.— "$(\forall x)A_{n+1}(x) > x+1$" for the n that we fixed in the I.H. It turns out that this is best handled by induction on x.

x, *prove* $A_{n+1}(x+1) > x+2$:

$$\begin{aligned} A_{n+1}(x+1) &= A_n(A_{n+1}(x)) \quad \text{by Def. 8.1.2} \\ &> A_{n+1}(x)+1 \quad \text{by } n\text{-I.H.} \\ &> x+2 \quad \text{by } x\text{-I.H.} \end{aligned}$$ □

8.1.5 Lemma. $\lambda x.A_n(x) \nearrow$.

"$\lambda x.f(x) \nearrow$" means that the (total) function f is *strictly increasing*, that is, $x < y$ implies $f(x) < f(y)$, for any x and y. Clearly, to establish the property one just needs to check for the arbitrary x that $f(x) < f(x+1)$.

Proof. We handle two cases separately.

A_0: $\lambda x.x+2 \nearrow$; immediate.

A_{n+1}: $A_{n+1}(x+1) = A_n(A_{n+1}(x)) > A_{n+1}(x)+1$ —the ">" by Lemma 8.1.4. □

8.1.6 Lemma. $\lambda n.A_n(x+1) \nearrow$.

Proof. $A_{n+1}(x+1) = A_n(A_{n+1}(x)) > A_n(x+1)$ —the ">" by Lemmata 8.1.4 (left argument > right argument) and 8.1.5. □

The "$x+1$" in Lemma 8.1.6 is important since $A_n(0) = 2$ for all n. Thus $\lambda n.A_n(0)$ is increasing but *not* strictly (constant).

8.1.7 Lemma. $\lambda y.A_n^y(x) \nearrow$.

Proof. $A_n^{y+1}(x) = A_n(A_n^y(x)) > A_n^y(x)$ —the ">" by Lemma 8.1.4. □

8.1.8 Lemma. $\lambda x.A_n^y(x) \nearrow$.

Proof. Induction on y: For $y = 0$ we want that $\lambda x.A_n^0(x) \nearrow$, that is, $\lambda x.x \nearrow$, which is true. We next take as I.H. that

$$A_n^y(x+1) > A_n^y(x) \tag{1}$$

We want

$$A_n^{y+1}(x+1) > A_n^{y+1}(x) \tag{2}$$

But (2) follows from (1) and Lemma 8.1.5, by applying A_n to both sides of ">". □

8.1.9 Lemma. *For all n, x, y, $A_{n+1}^y(x) \geq A_n^y(x)$.*

Proof. Induction on y: For $y = 0$ we want that $A_{n+1}^0(x) \geq A_n^0(x)$, that is, $x \geq x$, which is true. We now take as I.H. that

$$A_{n+1}^y(x) \geq A_n^y(x)$$

We want

$$A_{n+1}^{y+1}(x) \geq A_n^{y+1}(x)$$

This is true because

$$\begin{aligned} A_{n+1}^{y+1}(x) &= A_{n+1}\Big(A_{n+1}^y(x)\Big) \\ &\quad \text{by Lemma 8.1.6} \\ &\geq A_n\big(A_{n+1}^y(x)\big) \\ &\quad \text{Lemma 8.1.5 and I.H.} \\ &\geq A_n^{y+1}(x) \end{aligned}$$

□

8.1.10 Definition. Given a predicate $P(\vec{x})$, we say that $P(\vec{x})$ *is true almost everywhere* —in symbols "$P(\vec{x})$ a.e."— iff the set of (vector) inputs that make the predicate *false* is *finite*. That is, the set $\{\vec{x} : \neg P(\vec{x})\}$ is finite.

A statement such as "$\lambda xy.Q(x,y,z,w)$ a.e." can also be stated, less formally, as "$Q(x,y,z,w)$ a.e. *with respect to* x *and* y". □

8.1.11 Lemma. *$A_{n+1}(x) > x + l$ a.e. with respect to x.*

Thus, in particular, $A_1(x) > x + 10^{350000}$ a.e.

Proof. In view of Lemma 8.1.6 and the note following it, it suffices to prove

$$A_1(x) > x + l \text{ a.e. with respect to } x$$

Well, since

$$A_1(x) = A_0^x(2) = \overbrace{(\cdots(((y+2)+2)+2)+\cdots+2)}^{x \text{ 2's}}\|_{\text{evaluated at } y=2} = 2 + 2x$$

we ask: Is $2 + 2x > x + l$ a.e. with respect to x? It is so *for all* $x > l - 2$ (only $x = 0, 1, \ldots, l - 2$ fail). □

8.1.12 Lemma. *$A_{n+1}(x) > A_n^l(x)$ a.e. with respect to x.*

Proof. If one (or both) of l and n is 0, then the result is trivial. For example,

$$A_0^l(x) = \overbrace{(\cdots(((x+2)+2)+2)+\cdots+2)}^{l \text{ 2's}} = x + 2l$$

We are done by Lemma 8.1.11.

Let us then assume that $l \geq 1$ and $n \geq 1$. We note that (straightforwardly, via Definition 8.1.1)

$$A_n^l(x) = A_n(A_n^{l-1}(x))$$

$$= A_{n-1}^{A_n^{l-1}(x)}(2) = A_{n-1}^{A_{n-1}^{A_n^{l-2}(x)}(2)}(2) = A_{n-1}^{A_{n-1}^{A_{n-1}^{A_n^{l-3}(x)}(2)}(2)}(2)$$

The straightforward observation that *we have a "ladder" of k A_{n-1}'s precisely when the topmost exponent is $l-k$* can be ratified by induction on k (left to the reader). Thus we state

$$A_n^l(x) = {}^{k\ A_{n-1}}\left\{\begin{matrix} \cdot^{\cdot^{A_{n-1}^{l-k}(x)}} \\ A_{n-1}^{\cdot^{\cdot}} \end{matrix}\right. (2)$$

In particular, taking $k=l$,

$$\begin{aligned} A_n^l(x) &= {}^{l\ A_{n-1}}\left\{\begin{matrix} \cdot^{\cdot^{A_{n-1}^{l-l}(x)}} \\ A_{n-1}^{\cdot^{\cdot}} \end{matrix}\right. (2) \qquad (*)\\ &= {}^{l\ A_{n-1}}\left\{\begin{matrix} \cdot^{\cdot^{A_{n-1}^{x}(2)}} \\ A_{n-1}^{\cdot^{\cdot}} \end{matrix}\right. (2) \end{aligned}$$

Let us now take $x > l$.

Thus, by $(*)$,

$$A_{n+1}(x) = A_n^x(2) = {}^{x\ A_{n-1}}\left\{\begin{matrix} \cdot^{\cdot^{A_{n-1}^{2}(2)}} \\ A_{n-1}^{\cdot^{\cdot}} \end{matrix}\right. (2) \qquad (**)$$

By comparing $(*)$ and $(**)$ we see that the first "ladder" is topped (after l A_{n-1} "steps") by x and the second is topped by

$${}^{x-l\ A_{n-1}}\left\{\begin{matrix} \cdot^{\cdot^{A_{n-1}^{2}(2)}} \\ A_{n-1}^{\cdot^{\cdot}} \end{matrix}\right. (2)$$

after l steps.

Thus —in view of the fact that $A_n^y(x)$ increases with respect to each of the arguments n, x, y— we conclude by asking ...

"Is ${}^{x-l\ A_{n-1}}\left\{\begin{matrix} \cdot^{\cdot^{A_{n-1}^{2}(2)}} \\ A_{n-1}^{\cdot^{\cdot}} \end{matrix}\right. (2) > x$ a.e. with respect to x?"

. . . and answering, "Yes", because by $(**)$ this is the same question as "is $A_{n+1}(x-l) > x$ a.e. with respect to x?", which we answered affirmatively in 8.1.11. □

8.1.13 Lemma. *For all n, x, y, $A_{n+1}(x+y) > A_n^x(y)$.*

Proof.

$$\begin{aligned} A_{n+1}(x+y) &= A_n^{x+y}(2) \\ &= A_n^x\Big(A_n^y(2)\Big) \\ &= A_n^x\Big(A_{n+1}(y)\Big) \\ &> A_n^x(y) \quad \text{by Lemmata 8.1.4 and 8.1.8} \end{aligned}$$ □

8.2 Majorisation of $\mathcal{PR}$ functions

Nov. 15, 2021

We say that a function f *majorizes* another function, g, iff $g(\vec{x}) \leq f(\vec{x})$ for all $\vec{x}$. The following theorem states precisely in what sense "*the Ackermann function majorizes all the functions of $\mathcal{PR}$*".

8.2.1 Theorem. *For every function $\lambda\vec{x}.f(\vec{x}) \in \mathcal{PR}$ there are numbers n and k, such that for all $\vec{x}$ we have $f(\vec{x}) \leq A_n^k(\max(\vec{x}))$.*

Proof. The proof is by induction with respect to $\mathcal{PR}$. Throughout I use the abbreviation $|\vec{x}|$ for $\max(\vec{x})$ as this is notationally friendlier.

For the basis, f is one of:

- *Basis.*

  Basis 1. $\lambda x.0$. Then $A_0(x)$ works $(n = 0, k = 1)$.

Basis 2. $\lambda x.x+1$. Again $A_0(x)$ works $(n=0,k=1)$.

Basis 3. $\lambda\vec{x}.x_i$. Once more $A_0(x)$ works $(n=0,k=1)$: $x_i \leq |\vec{x}| < A_0(|\vec{x}|)$.

- *Propagation with composition.* Assume as I.H. that

$$f(\vec{x}_m) \leq A_n^k(|\vec{x}_m|) \tag{1}$$

and

$$\text{for } i=1,\ldots,m,\ g_i(\vec{y}) \leq A_{n_i}^{k_i}(|\vec{y}|) \tag{2}$$

Then

$$\begin{aligned} f(g_1(\vec{y}),\ldots,g_m(\vec{y})) &\leq A_n^k(|g_1(\vec{y}),\ldots,g_m(\vec{y})|), \text{ by (1)} \\ &\leq A_n^k(|A_{n_1}^{k_1}(|\vec{y}|),\ldots,A_{n_m}^{k_m}(|\vec{y}|)|), \text{ by 8.1.8 and (2)} \\ &\leq A_n^k\Big(A_{\max n_i}^{\max k_i}(|\vec{y}|)\Big), \text{ by 8.1.8 and 8.1.9} \\ &\leq A_{\max(n,n_i)}^{k+\max k_i}(|\vec{y}|), \text{ by 8.1.9} \end{aligned}$$

- *Propagation with primitive recursion.* Assume as I.H. that

$$h(\vec{y}) \leq A_n^k(|\vec{y}|) \tag{3}$$

and

$$g(x,\vec{y},z) \leq A_m^r(|x,\vec{y},z|) \tag{4}$$

Let f be such that

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

I claim that

$$f(x,\vec{y}) \leq A_m^{rx}\Big(A_n^k(|x,\vec{y}|)\Big) \tag{5}$$

I prove (5) by induction on x:

For $x = 0$, I want $f(0, \vec{y}) = h(\vec{y}) \leq A_n^k(|0, \vec{y}|)$. This is true by (3) since $|0, \vec{y}| = |\vec{y}|$.

As an I.H. assume (5) for fixed x.

The case for $x + 1$:

$$\begin{aligned} f(x+1, \vec{y}) &= g(x, \vec{y}, f(x, \vec{y})) \\ &\leq A_m^r(|x, \vec{y}, f(x, \vec{y})|), \text{ by (4)} \\ &\leq A_m^r\Big(\Big|x, \vec{y}, A_m^{rx}\Big(A_n^k(|x, \vec{y}|)\Big)\Big|\Big) \\ &\text{by the I.H. (5), and 8.1.8} \\ &= A_m^r\Big(A_m^{rx}\Big(A_n^k(|x, \vec{y}|)\Big)\Big) \\ &\text{by 8.1.8 and } A_m^{rx}\big(A_n^k(|x, \vec{y}|)\big) \geq |x, \vec{y}| \\ &= A_m^{r(x+1)}\Big(A_n^k(|x, \vec{y}|)\Big) \\ &\leq A_m^{r(x+1)}\Big(A_n^k(|x+1, \vec{y}|)\Big) \text{ by 8.1.8} \end{aligned}$$

With (5) proved, let me set $l = \max(m, n)$. By Lemma 8.1.9 I now get

$$f(x, \vec{y}) \leq A_l^{rx+k}(|x, \vec{y}|) \underset{\text{Lemma 8.1.13}}{<} A_{l+1}(|x, \vec{y}| + rx + k) \tag{6}$$

Now, $|x, \vec{y}| + rx + k \leq (r+1)|x, \vec{y}| + k$ thus, (6) and 8.1.5 yield

$$f(x, \vec{y}) < A_{l+1}((r+1)|x, \vec{y}| + k) \tag{7}$$

To simplify (7) note that there is a number q such that

$$(r+1)x + k \leq A_1^q(x) \tag{8}$$

for all x. Indeed, this is so since (easy induction on y) $A_1^y(x) = 2^y x + 2^y + 2^{y-1} + \cdots + 2$. Thus, to satisfy (8),

just take $y = q$ large enough to satisfy $r + 1 \leq 2^q$ and $k \leq 2^q + 2^{q-1} + \cdots + 2$.

By (8), the inequality (7) yields, via 8.1.5,

$$f(x, \vec{y}) < A_{l+1}(A_1^q(|x, \vec{y}|)) \leq A_{l+1}^{1+q}(|x, \vec{y}|)$$

(by Lemma 8.1.9) which is all we want. □

8.2.2 Remark. Reading the proof carefully we note that the *subscript argument* of the *majorant*† is precisely the maximum depth of nesting of primitive recursion that occurs in a derivation of f.

Pause. In which derivation? There are infinitely many.◀

Indeed, the initial functions have a majorant with subscript 0; composition has a majorant with subscript no more than the maximum subscript of the component parts —*no increase*; primitive recursion has a majorant with a subscript that is bigger than the *maximum* subscript of the h- and g-majorants by precisely 1. □

8.2.3 Corollary. $\lambda nx.A_n(x) \notin \mathcal{PR}$.

Proof. By contradiction: If $\lambda nx.A_n(x) \in \mathcal{PR}$ then also $\lambda x.A_x(x) \in \mathcal{PR}$ (identification of variables —the so-called *diagonalisation of* $A_n(x)$). By the theorem above, for some n, k, $A_x(x) \leq A_n^k(x)$, for all x, hence, by 8.1.12

$$A_x(x) < A_{n+1}(x), \text{ a.e. with respect to } x \qquad (1)$$

On the other hand, $A_{n+1}(x) < A_x(x)$ a.e. with respect to x —indeed for all $x > n + 1$ by 8.1.6— which contradicts (1). □

†The function that does the majorising.

See the next mini-chapter for a mathematical proof that $\lambda nx.A_n(x) \in \mathcal{R}$.

Chapter 9

The Recursion Theorem and Applications

Here we present the (2nd) recursion theorem of Kleene — originally discovered by Gödel in a slightly different form towards his *Incompleteness theorem* ([Göd31])— and a few applications.

An easy extension (Exercise!!!) of the techniques leading to the effective enumerations (in Section 6.3) of unary

$$\phi_0, \phi_1, \phi_2, \dots$$

and binary partial computable functions

$$\phi_0^{(2)}, \phi_1^{(2)}, \phi_2^{(2)}, \dots$$

leads to the effective enumeration of the *n-ary partial computable functions*, for any n:

$$\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)}, \dots$$

Correspondingly one proves the following version of the Smn theorem using the analogous proof as in 6.3.3 (*EXERCISE!!!*)

9.0.1 Theorem. *There is a 2-argument function S_1^n in $\mathcal{R}$ such that*

$$\phi_i^{(n+1)}(z, \vec{x}_n) = \phi^{(n)}_{S_1^n(i,z)}(\vec{x}_n), \text{ for all } i, z, \vec{x}_n \qquad (1)$$

9.0.2 Theorem. (Kleene's 2nd recursion theorem)

If $\lambda z \vec{x}. f(z, \vec{x}_n) \in \mathcal{P}$, then for some $e \in \mathbb{N}$,

$$\phi_e^{(n)}(\vec{x}_n) = f(e, \vec{x}_n) \text{ for all } \vec{x}_n$$

Proof. By Grzegorczyk substitution

$$\lambda z \vec{x}_n. f(S_1^n(z, z), \vec{x}_n) \in \mathcal{P}$$

Thus, for some i, and all $z, \vec{x}_n$,

$$\phi_i^{(n+1)}(z, \vec{x}_n) = f(S_1^n(z, z), \vec{x}_n)$$

In particular, all $\vec{x}_n$,

$$\phi^{(n)}_{S_1^n(i,i)}(\vec{x}_n) \overset{9.0.1}{=} \phi_i^{(n+1)}(i, \vec{x}_n) = f(S_1^1(i, i), \vec{x}_n)$$

Take $e = S_1^n(i, i)$. □

9.1 Two Applications of the Recursion Theorem

9.1.1 Theorem. (Rice) *A complete index set $A = \{x : \phi_x \in \mathcal{C}\}$ is recursive iff it is trivial.*

Rice uses the term "trivial" here to mean "$A = \emptyset$ or $A = \mathbb{N}$".

Thus, "algorithmically" we can only "decide" trivial properties of "programs".

Proof. (The idea of this proof is attributed in [Rog67] to G.C. Wolpin.)

if-part. Immediate, since $c_{\emptyset} = \lambda x.1$ and $c_{\mathbb{N}} = \lambda x.0$.

only if-part. By contradiction, suppose that $A = \{x : \phi_x \in \mathcal{C}\}$ is nontrivial, yet $A \in \mathcal{R}_*$. So, let $a \in A$ and $b \notin A$. Define f by

$$f(x) = \begin{cases} b & \textbf{if } x \in A \\ a & \textbf{if } x \notin A \end{cases}$$

The assumption that A is recursive makes f recursive (def. by recursive cases).

Clearly,

$$x \in A \text{ iff } f(x) \notin A, \text{ for all } x \tag{1}$$

By the recursion theorem, there is an e such that $\phi_{f(e)} = \phi_e$ (apply 9.0.2 to $\lambda xy.\phi_{f(x)}(y) = \lambda xy.U^{(P)}(f(x), y) \in \mathcal{P}$).

Thus, $e \in A$ iff $\phi_e \in \mathcal{C}$ iff $\phi_{f(e)} \in \mathcal{C}$ iff $f(e) \in A$, contradicting (1). □

All complete index sets that we proved unsolvable, Rice's theorem establishes to be so at once.

For example $A = \{x : \phi_x \text{ is a constant}\}$:

$$\emptyset \overset{Z \text{ is a const}}{\neq} A \overset{S \text{ is not!}}{\neq} \mathbb{N}$$

New notation (Rogers)

W_x means $\text{dom}(\phi_x)$

Note that

$$y \in W_x \equiv \phi_x(y) \downarrow$$

Thus W_x sets are semi-recursive.

9.1.2 Example. By Rice's theorem both $2 \in W_x$ and $W_x = \emptyset$ are unsolvable.

For the first note that $\{x : 2 \in \text{dom}(\phi_x)\}$ is *nontrivial*: Indeed, note that

$$f(x) = \begin{cases} 0 & if\ x \neq 2 \\ \uparrow & otw \end{cases}$$

is in $\mathcal{P}$, thus $B = \{x : 2 \in \text{dom}(\phi_x)\} \neq \mathbb{N}$. On the other hand, $2 \in \text{dom}(Z)$, so $B \neq \emptyset$.

As for $C = \{x : \text{dom}(\phi_x) = \emptyset\}$, the empty function (infinitely many) contributes programs to C, while no program of Z is in C. Thus C is nontrivial. □

Nov. 17, 2021

The second application of the recursion theorem is about self-referential (recursive) definitions of functions F such as the one below —thinking of $F \overset{Def}{=} \lambda nx.A_n(x)$.

9.1.3 Example. Here is an easy proof that the Ackermann function is recursive: $\lambda nx.A_n(x) \in \mathcal{R}$.

$A_n(x)$ is given by

$$A_n(x) = \begin{cases} x+2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ A_{n\dot{-}1}\big(A_n(x \dot{-} 1)\big) & \text{otherwise} \end{cases} \tag{1}$$

How can the recursion theorem help in the proof of recursiveness?

OK. So we want $A_n(x) = \phi_e^{(2)}(n, x)$, for some e (*to be determined!*) and all n, x.

If we succeed, then we will have $\phi_e^{(2)}$ satisfying (1), namely, we will have (2) below:

$$\phi_e^{(2)}(n,x) = \begin{cases} x+2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ \phi_e^{(2)}\big(n \dot{-} 1, \phi_e^{(2)}(n, x \dot{-} 1)\big) & \text{otherwise} \end{cases} \tag{2}$$

Hmmm. This does not look any easier! How on earth do you find such an e?

But wait!

Work with the following, and then use the recursion theorem to force $z = e$ for some $e \in \mathbb{N}$:

$$F(z,n,x) = \begin{cases} x+2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ \phi_z^{(2)}\big(n \dot{-} 1, \phi_z^{(2)}(n, x \dot{-} 1)\big) & \text{otherwise} \end{cases} \tag{3}$$

Now, by the universal function theorem (or directly by CT),

$$\lambda znx.\phi_z^{(2)}(n,x)$$

is in $\mathcal{P}$ and

so is $\lambda znx.F(z,n,x)$ since (3) is definition by (prim.) recursive cases.

By 9.0.2, there is an e such that $F(e,n,x) = \phi_e^{(2)}(n,x)$ for all n, x. But then $\phi_e^{(2)}$ satisfies (2) and as $\phi_e^{(2)} = \lambda nx.A_n(x)$, we proved the recursiveness of the Ackermann function!

WAIT! 1 Why is the Ackermann function total?

WAIT! 2 Why is $\phi_e^{(2)} = \lambda nx.A_n(x)$??

Why can't (1) have TWO (or more) solutions of (1) —that is, can't the recurrence relation defining Ackermann define TWO different functions or even more? If so, then the computable solution we found is NOT guaranteed to be the same as the Ackermann function!

That (1) indeed does have a unique, in fact total, solution is an easy (double) induction exercise that shows

$F'(n,x) = A_n(x)$ for all n, x if F' satisfies

$$F'(n,x) = \begin{cases} x+2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ F'\big(n \dot{-} 1, F'(n, x \dot{-} 1)\big) & \text{otherwise} \end{cases} \tag{1'}$$

EXERCISE!!! □

Chapter 10

Complexity of $\mathcal{PR}$ Functions

Overview

The literature refers to the complexity theory of the partial recursive functions as "*high level complexity*" due to the mostly theoretical and far removed from practice essence of it. The present chapter is about more down to earth functions. We will be looking into the complexity of $\mathcal{PR}$ functions using a *hierarchy approach*.

In the first instance, we build our hierarchies according to the *definitional* or *static* complexity of function derivations, or (loop) programs. As we develop the theme the reader will note that the choice of measure of static complexity or *complexity of the definition* is not always the same (some times it is nesting level of primitive recursion, other times it is size of function output that *is determined at definition time*). In the end of all this we will have various hierarchies —called *sub-recursive hierarchies* in the literature— to *compare*, and a powerful tool, the Ritchie-Cobham property (theorem) that al-

lows all these static hierarchies to be viewed as *complexity classes named* by time-bounding functions (of URM or Loop-program computations).

So, how much down to earth are the $\mathcal{PR}$ functions? Much more *than* the partial recursive and recursive functions; however, absolutely speaking, functions of interest to the practitioner are found only in the first couple of levels of the various hierarchies. For example, *level two* of the loop program hierarchy already contains the function we mentioned before,

$$\left.\begin{matrix} 2^{\cdot^{\cdot^{2^x}}} \end{matrix}\right\} 10^{350000} \text{ 2s}$$

which has impossibly large output even for $x = 0$. Totally impractical! The following function, where the "ladder" height is a function of the input x (ladder of 2s height is equal to the argument x)

$$\left.\begin{matrix} 2^{\cdot^{\cdot^{2^x}}} \end{matrix}\right\} x \text{ 2's}$$

cannot be computed if we only allow loop nesting depth (or level) two (that is, one loop inside the other) in one or more places in an (attempted) program; it *requires* nesting level *three* and is even more astronomical than the previous "ladder" function.

Impressive as this discussion may be, the dull truth is that, *in practice*, we compute *finite* functions because we only have *finite* (computing) resources such as CPU time and/or memory space.

Still we do *not* compute by *brute force table look-up*. Rather, we *pretend* that our programs will *run for any*

input,[†] thus we write *general algorithms* (programs) with that in mind. It is then clear that theories like the complexity of primitive recursive functions can enrich our *programming practice*.

For example we learn that if we program with loop programs that never nest loops more than two levels, then all the functions $\lambda\vec{x}.f(\vec{x})$ that we can so compute have outputs that are *majorised* (or bounded; see Section 8.2) by

$$\left.\begin{matrix} 2^{\cdot^{\cdot^{2^{\max(\vec{x})}}}} \end{matrix}\right\} \text{c 2s}$$

where c depends on f.

10.1 The URM-simulating functions again, and the Kleene predicate

We return to the discussion of Subsection 7.3.1 to sharpen Theorem 7.3.1. We will prove here

10.1.1 Theorem. *The functions $\lambda y\vec{a}_m.X_j(y,\vec{a}_m)$ —for $j=1,2,\ldots n$— and $\lambda y\vec{a}_m.IP(y,\vec{a}_m)$ return, for any* input *$\vec{a}_m$ (into $\vec{X}_m$) and* step-count *y, the value stored in the variable X_j and* the instruction pointer *IP* that points at **the current instruction** *respectively, and all this at step y.*

All these functions are in $\mathcal{PR}$.

Proof. The proof is by translating the simulation table of p.225 into a simultaneous recursion for the functions $\lambda y\vec{a}_m.X_j(y,\vec{a}_m)$ —$j=1,\ldots,n$— and $\lambda y\vec{a}_m.IP(y,\vec{a}_m)$.

[†]They will not, due to computing time and memory finiteness.

The initialisation of the recursion is trivial:

$$X_j(0, \vec{a}_m) = a_j, \text{ for } j = 1, \ldots, m,$$
$$X_j(0, \vec{a}_m) = 0, \text{ for } j = m+1, \ldots, n, \text{ and}$$
$$IP(0, \vec{a}_m) = 1$$

Refer to the simulation table and the discussion following it (on p.225) for the iteration steps below:

For $k = 1, \ldots, n$,

$$X_k(y+1, \vec{a}_m) = r \textbf{ if } IP(y, \vec{a}_m) = L \text{ and } L : X_k \leftarrow r \textit{ is in } M$$
$$X_k(y+1, \vec{a}_m) = X_k(y, \vec{a}_m) + 1 \textbf{ if } IP(y, \vec{a}_m) = L \text{ and } L : X_k \leftarrow X_k + 1 \textit{ is in } M$$
$$X_k(y+1, \vec{a}_m) = X_k(y, \vec{a}_m) \dot{-} 1 \textbf{ if } IP(y, \vec{a}_m) = L \text{ and } L : X_k \leftarrow X_k \dot{-} 1 \textit{ is in } M$$
$$X_k(y+1, \vec{a}_m) = X_k(y, \vec{a}_m) \textbf{ otw}$$

For IP the recurrence is

$$IP(y+1, \vec{a}_m) = IP(y, \vec{a}_m) \textbf{ if } IP(y, \vec{a}_m) = L \text{ and } L : \textbf{stop} \text{ is in } M$$
$$IP(y+1, \vec{a}_m) = L' \textbf{ if } IP(y, \vec{a}_m) = L \text{ and } M \text{ includes } L : \textbf{if } X_k = 0 \textbf{ goto } L' \textbf{ else goto } L'' \text{ and } X_k(y, \vec{a}_m) = 0$$
$$IP(y+1, \vec{a}_m) = L'' \textbf{ if } IP(y, \vec{a}_m) = L \text{ and } M \text{ contains } L : \textbf{if } X_k = 0 \textbf{ goto } L' \textbf{ else goto } L'' \text{ and } X_k(y, \vec{a}_m) \neq 0$$
$$IP(y+1, \vec{a}_m) = IP(y, \vec{a}_m) + 1 \textbf{ otw}$$

Note that "**if** $IP(y, \vec{a}_m) = L$ and M contains" even though it sounds "wordy" and "vague" it becomes precise once a specific URM is given (here we are working with the arbitrary URM M without precise knowledge of its exact instructions).

Also note that the recursive calls happen within trivially "easy" functions such as $\lambda z.z$, $\lambda z.z+1$ or $\lambda z.z \dot{-} 1$ or trivial predicates like $\lambda z.z = y$ and $\lambda z.z \neq y$ (y a constant).

All are primitive recursive. The theorem follows. □

10.1.2 Example. This takes away the mystery, if any, from the above word-loaded proof.

Let M be the program below

$1 : X_1 \leftarrow X_1 + 1$
$2 : X_2 \leftarrow X_2 \dot{-} 1$
$3 :$ **if** $X_2 = 0$ **goto** 4 **else goto** 1
$4 :$ **stop**

Let us assume that X_2 is the input variable and X_1 is the output variable. The simulating equations take the concrete form below, where a denotes the input value:

$$X_1(0, a) = 0$$
$$X_2(0, a) = a$$

For $y \geq 0$ we have

$$X_1(y+1, a) = \begin{cases} X_1(y, a) + 1 & \text{if } IP(y, a) = 1 \\ X_1(y, a) & \text{otherwise} \end{cases}$$

$$X_2(y+1, a) = \begin{cases} X_2(y, a) \dot{-} 1 & \text{if } IP(y, a) = 2 \\ X_2(y, a) & \text{otherwise} \end{cases}$$

$$IP(y{+}1,a) = \begin{cases} 4 & \text{if } IP(y,a) = 3 \wedge X_2(y,a) = 0 \\ 1 & \text{if } IP(y,a) = 3 \wedge X_2(y,a) \neq 0 \\ 4 & \text{if } IP(y,a) = 4 \\ IP(y,a)+1 & \text{otherwise} \end{cases}$$

□

10.1.3 Definition. (The Kleene Predicate)

The Kleene "T-predicate" is denoted by $T(z,x,y)$ and is —*by definition*— true iff the URM of one argument (looks like M_U^X) found at location z when given input x (into X) has a halting computation of y steps (or less) —that is, for said input, M reaches **stop** in $\leq y$ steps.

By a minor abuse of language we abbreviate the expression "the URM of one argument (looks like M_U^X) found at location z" by "the URM z".

□

10.1.4 Proposition. $T(z,x,y) \in \mathcal{PR}_*$.

Proof. Let M_U^X be at location z. So, $M_U^X = \phi_z$ and thus $\phi_z(x) = U^{(P)}(z,x)$, for all z, x.

Let N_W^{ZX} compute $U^{(P)}$, that is, $N_W^{ZX} = U^{(P)}$.

Thus, if the function IP is associated with N_W^{ZX}, then we have

1. IP is primitive recursive, and so is its graph $IP(y,z,x) = w$.

2. We note that

$$\begin{aligned} T(z,x,y) &\equiv N_W^{ZX} \text{ stops in } \leq y \text{ steps} \\ &\equiv IP(y,z,x) = q \text{ where } q : \textbf{stop} \text{ is in } N \end{aligned}$$

We are done by 2. above. □

10.1.5 Theorem. (Kleene's Normal Form Theorem) *Let T be the Kleene predicate of 10.1.3. Then*

(I)

$$\phi_z(x) \downarrow \equiv (\exists y)T(z,x,y) \tag{1}$$

for all z, x.

(II) *There is a primitive recursive function $\lambda yzx.out(y,z,x)$ such that, for all z, x,*

$$\phi_z(x) = out\Big((\mu y)T(z,x,y), z, x\Big) \tag{2}$$

Proof.

(I) The lhs of (1) says that the URM z with input x has a halting computation. The rhs says precisely the same.

(II) The lhs of (2), if defined is what is found in the output variable W of N_W^{ZX} of the preceding discussion.

If we rename the simulating function for the variable W —from "$\lambda yzx.W(y,z,x)$"— to

$$\text{"}\lambda yzx.out(y,z,x)\text{"}$$

and seeing that if $\phi_z(x) \downarrow$ then N halts as soon as it hits step number $(\mu y)T(z,x,y)$, we see that (2) follows at once. □

Nov. 22, 2021

10.2 The Axt, loop-program and Grzegorczyk hierarchies

10.2.1 Remark. In the proof that $\mathcal{PR}$ is majorised by the Ackermann function (8.2.1)we did induction on $\mathcal{PR}$ rather than on length of derivations.

Analogously, $\mathcal{PR}$ can be *inductively defined* as the smallest class containing the initial functions Z, S and $\lambda\vec{x}.x_i$ that is closed under composition and also the operation on functions $prim(h, g)$. That is,

An f is primitive recursive iff it is one of

1. Z, S, U_i^n, for all $n \geq i \geq 1$.
2. $f = prim(h, g)$ where h, g are primitive recursive
3. $f = \lambda\vec{y}.h(g_1(\vec{y}), \ldots g_n(\vec{y}))$ where h and the g_j are all primitive recursive.

This is the preferred (more elegant) alternative to giving $\mathcal{PR}$ as just the container of what derivations produce.

Indeed, the inclusion of the initial functions says that the derivations of length 1 produce primitive recursive functions. Then, for example, closure under $prim(h, g)$ translates into "if we know that h, g are in $\mathcal{PR}$ then so is $prim(h, g)$, or —in the jargon of derivations— if we have derived h and g via derivations $\boxed{\ldots h}$ and $\boxed{\ldots g}$, then the derivation $\boxed{\ldots h, \ldots, g, prim(h, g)}$ derives $prim(h, g)$".

Compare with 2. above.

You note the elegance and the economy on verbiage!

Notation. The symbol $\mathrm{Cl}(\mathcal{I},\mathcal{O})$ is short for any such inductive definition from initial objects $\mathcal{I}$ and operations $\mathcal{O}$.

E.g., one would write

$$\mathcal{PR} = \mathrm{Cl}\Big(\{Z, S, U_i^n, \text{ for } 1 \leq i \leq n\},\ \{\textbf{composition}, \textbf{prim}\}\Big)$$

$\mathrm{Cl}(\mathcal{I},\mathcal{O})$ is read "the *closure* of the set $\mathcal{I}$ under the operations in $\mathcal{O}$".

□

10.2.2 Definition. ([Axt65, Hei61]) We build a hierarchy $\mathcal{K} = (K_n)_{n\geq 0}$ by induction on the level n, and at each level K_n is also defined inductively as a "Cl(...)":

We set $K_0 = \mathrm{Cl}(\{\lambda x.x+1, \lambda x.x\}, \mathcal{O})$ where $\mathcal{O}$ contains only Grzegorczyk substitution.

Having defined K_n, we define K_{n+1}: First, let

$$R_{n+1} \overset{Def}{=} \{f : f = prim(h,g) \wedge \{h,g\} \subseteq K_n\}$$

Then set

$$K_{n+1} \overset{Def}{=} \mathrm{Cl}(K_n \cup R_{n+1}, \mathcal{O}) \tag{1}$$

If $f \in K_n$ then we say its *level* is $\leq n$. If $f \in K_{n+1} - K_n$ then we say its *level* is $= n+1$. □

The definition clearly assumes that *it is* primitive recursion *that adds to the (definitional) complexity*, namely, the *nesting levels* of it: K_n already has functions $f' = prim(h',g')$ with h', g' in K_{n-1}. Going to K_{n+1} we are adding another primitive recursion on top of $f' = prim(h',g')$.

10.2.3 Proposition.

1. *For* $n \geq 0$, $K_n \subseteq K_{n+1}$.

2. $\mathcal{PR} = \bigcup_{n\geq 0} K_n$

Proof. Easy exercise. Item 1 is a direct consequence of (1) on the previous page.

For item 2, $\subseteq$ direction, do induction over $\mathcal{PR}$ since it is inductively defined. □

10.2.4 Lemma. $A_n \in K_n$, *for* $n \geq 0$, *where* $\lambda nx.A_n(x)$ *is the Ackermann function.*


Of course, "A_n" means "$\lambda x.A_n(x)$", for each n.


Proof. Induction on n. For $n = 0$ we have $A_0 = \lambda x.x+2$ which is obtained from $\lambda x.x + 1$ by Grzegorczyk operations. Thus $A_0 \in K_0$. Fix now n and assume (I.H.) $A_n \in K_n$.

For K_{n+1}:

$$\begin{aligned} A_{n+1}(0) &= 2 \\ A_{n+1}(x+1) &= A_n(A_{n+1}(x)) \end{aligned}$$

By the I.H. and the definition of K_{n+1} the above primitive recursion places $A_{n+1} \in K_{n+1}$. □

10.2.5 Lemma. (Majorising lemma for $\mathcal{K}$) *For* $n \geq 0$, $\lambda\vec{x}.f(\vec{x}) \in K_n$ *implies that for some* m *depending on* f *we have* $f(\vec{x}) \leq A_n^m(|\vec{x}|)$, *for all* $\vec{x}$.


As this is related to Theorem 8.2.1 we use "$|\vec{x}|$" for "$\max(\vec{x})$".


Proof. This is a trivial corollary of the proof of 8.2.1: Grzegorczyk substitution does not raise the index n of the (bounding) Ackerman function, but *one* primitive recursion raises it *by one*.

So fix n. If $f = prim(h, g)$ is in K_{n+1} because it is in R_{n+1} of Definition 10.2.2, then the I.H. and the proof of 8.2.1 say that f is majorised by A^m_{n+1} (some m) since (I.H.) h and g are majorised by A^k_n and A^r_n for some k and r. If f got into K_{n+1} after a finite number of substitutions *after* we first obtained $prim(h, g)$ —i.e., f is in $\mathrm{Cl}(K_n \cup R_{n+1}, \mathcal{O})$— then the bounding function A^m_{n+1} above will at most change to $A^{m'}_{n+1}$ from A^m_{n+1}. □

10.2.6 Theorem. *The hierarchy* $\mathcal{K} = (K_n)_{n\geq 0}$ *is* proper *or* nontrivial*, that is,* $K_n \subsetneq K_{n+1}$*, for all* n*.*

Proof. Suffices to find for each n some f in $K_{n+1} - K_n$. Well, $A_{n+1} \in K_{n+1} - K_n$, the positive part by 10.2.4: $A_{n+1} \in K_{n+1}$; and the negative part by 10.2.5: if $A_{n+1} \in K_n$ then $A_{n+1}(x) \leq A^m_n(x)$ for some m and all x. This is not so (8.1.12.) □

If we replace *primitive recursion* by *simultaneous* (primitive) *recursion* then *we obtain an easier to work with hierarchy*.

10.2.7 Definition. We build a hierarchy $\mathcal{K}^{sim} = (K^{sim}_n)_{n\geq 0}$ by induction on the level n, while at each level K^{sim}_n is defined as a closure.

We set $K^{sim}_0 = \mathrm{Cl}(\{\lambda x.x+1, \lambda x.x\}, \mathcal{O}) = K_0$ where $\mathcal{O}$ contains only Grzegorczyk substitution.

Having defined K^{sim}_n, we define K^{sim}_{n+1}: First, let

$$R^{sim}_{n+1} \overset{Def}{=} \{f_j : j \leq r \wedge \vec{f}_r = simprim(\vec{h}_r, \vec{g}_r) \wedge \{\vec{h}_r, \vec{g}_r\} \subseteq K^{sim}_n\}$$

Then set

$$K_{n+1}^{sim} \stackrel{Def}{=} \mathrm{Cl}(K_n^{sim} \cup R_{n+1}^{sim}, \mathcal{O}) \qquad (\dagger)$$

If $f \in K_n^{sim}$ then we say its *level* is $\leq n$. If $f \in K_{n+1}^{sim} - K_n^{sim}$ then we say its *level* is $= n+1$. □

10.2.8 Lemma. *For all $n \geq 0$, $K_n \subseteq K_n^{sim}$.*

Proof. This is trivial since all other things being equal, primitive recursion is a special case of simultaneous recursion. □

10.2.9 Lemma. (Majorising lemma for $\mathcal{K}^{sim}$) *For $n \geq 0$, $\lambda\vec{x}.f(\vec{x}) \in K_n^{sim}$ implies that for some m depending on f we have $f(\vec{x}) \leq A_n^m(|\vec{x}|)$, for all $\vec{x}$.*

Proof. Exercise. □

10.2.10 Proposition.

1. *For $n \geq 0$, $K_n^{sim} \subseteq K_{n+1}^{sim}$.*
2. *$\mathcal{PR} = \bigcup_{n\geq 0} K_n^{sim}$*

Proof. 1. is immediate from †) above. For 2. the $\supseteq$ is trivial while the $\subseteq$ follows from 10.2.3(2) and 10.2.8. Exercise. □

10.2.11 Theorem. *The hierarchy $\mathcal{K}^{sim} = (K_n^{sim})_{n\geq 0}$ is* proper*, that is, $K_n^{sim} \subsetneq K_{n+1}^n$, for all n.*

Proof. By 10.2.8, $A_{n+1} \in K_{n+1}^{sim}$. By 10.2.9 $A_{n+1} \notin K_n^{sim}$. □

At this point we pause to offer some examples of familiar functions and place them in the hierarchies we have

so far, and to develop some tools that will help us do coding (needed to go from simultaneous to single primitive recursion).

10.2.12 Proposition. (Examples and Tools) *The following table lists some simple $\mathcal{PR}$ functions and their placement in the $\mathcal{K}$ hierarchy.*

Function	*Upper bound of level*
1) $\lambda xy.x + y$	1
2) $\lambda xy.x \dot{-} 1$	1
3) $\lambda xy.x(1 \dot{-} y)$ (Restricted if-then-else)	1
4) $\lambda xy.x \times y$	2
5) $\lambda x.2^x$	2
6) $\lambda xy.x \dot{-} y$	2
7) $\lambda xy.\lvert x - y\rvert$	2
8) $\lambda xy.\max(x, y)$	2
9) $\lambda x.\left\lfloor \frac{x(x+1)}{2} \right\rfloor$	2

Proof. [Sch69] attributes the table to [Hei61]. We do 9) and leave *all else* to the reader (Exercise). Since $\left\lfloor \frac{x(x+1)}{2} \right\rfloor = \sum_{j\leq x} j$ we note

$$\begin{aligned} \textstyle\sum_{j\leq 0} j &= 0 \\ \textstyle\sum_{j\leq x+1} j &= x + 1 + \textstyle\sum_{j\leq x} j \end{aligned}$$

But $\lambda x.x+1 \in K_0$ and the $+$-function is of level at most 1.

10.2.13 Definition. If $\mathcal{C}$ is any subclass of $\mathcal{PR}$ we denote by $\mathcal{C}_*$ the class of the *corresponding predicates*:

$$\mathcal{C}_* \overset{Def}{=} \{f(\vec{x}) = 0 : f \in \mathcal{C}\}$$

□

We have the following easy lemmata:

10.2.14 Lemma. *$K_{n,*}$ and $K^{sim}_{n,*}$ are closed under Boolean operations (all $n \geq 1$).*

Proof. Exercise. □

10.2.15 Lemma. *K_n and K^{sim}_n are closed under definition by cases (all $n \geq 1$).*

Proof. Because our familiar *sw* is in K_1. Exercise. □

10.2.16 Example. The reader is encouraged to revisit Example 4.2.3. From the work there follows that $\lambda x.rem(x,2) \in K^{sim}_1$. On the other hand, [Rit65, TW68] have shown that $rem \notin K_1$ establishing $K_1 \subsetneq K^{sim}_1$. □

We just saw that K_j, $j = 0, 1$, contain trivial functions and the corresponding predicates (10.2.13) are trivial too.

We will see soon that the Grzegorczyk "small classes" $\mathcal{E}^0, \mathcal{E}^1$ have enormously rich structure. Even their corresponding *relation* (predicate) *sets* are impressive: One can easily place a version of the Kleene T-predicate in $\mathcal{E}^2_*$ and with some more work (coding URMs and their computations as in the text [Tou12] will be needed) even in $\mathcal{E}^0_*$. We will postpone this matter until after we introduce the loop programmable functions hierarchy which is

essentially $\mathcal{K}^{sim}$ from a "programming" perspective, but we only have tools in these notes to prove $T \in \mathcal{E}^2_*$.

10.2.17 Definition. (A hierarchy of Loop programs) L_0 is the set of all loop programs that contain *no loop-end instruction.*

Having defined L_n, we define L_{n+1} as a closure:

$$\mathrm{Cl}\Big(L_n \cup \{\textbf{Loop } X;\ P;\ \textbf{end} \ \Big|\ P \in L_n\}, \mathcal{O}\Big)^{\dagger}$$

where $\mathcal{O}$ contains only the operation for *program concatenation* (cf. 5.1.1), namely, $(P, Q) \mapsto P; Q$. □

10.2.18 Remark. It is clear that $L_n \subsetneq L_{n+1}$ since L_{n+1} contains, but L_n does not, programs with nesting level of **Loop-end** instruction equal to $n+1$, that is, L_0 programs have zero nesting (of **Loop-end** instruction), and if P has at least one occurrence of nesting level n, but none higher, then **Loop** X; P; **end** contains a loop of nesting level $n+1$; the outermost.

Comparing with Definition 5.1.1 it is immediate that $L = \bigcup_{n\geq 0} L_n$. □

We can now separate the functions in $\mathcal{L} = \mathcal{PR}$ into a hierarchy of *functions.*

10.2.19 Definition. For $n \geq 0$,

$$\mathcal{L}_n \overset{Def}{=} \{P^{\vec{X}}_Y : P \in L_n, \text{ where } Y \text{ and the } X_i \text{ are all in } P\}$$

□

10.2.20 Theorem. ($\mathcal{K}^{sim}$ **vs.** $\mathcal{L}$) $K^{sim}_n = \mathcal{L}_n$, $n \geq 0$.

†We deviated from our norm and used here the notation $\{x| \ldots\}$ rather than $\{x : \ldots\}$ in the interest of more visual clarity.

Proof. An adaptation of 5.3.1 and 5.4.2, from the $\mathcal{PR}$ vs. $\mathcal{L}$ case to the K_n^{sim} vs. $\mathcal{L}_n$ case. Exercise. □

With 10.2.20 settled we directly get the hierarchy corollary.

10.2.21 Corollary. *$\mathcal{L} = (\mathcal{L}_n)_{n\geq 0}$ is a proper hierarchy, that is,*

1. $\mathcal{L}_n \subsetneq \mathcal{L}_{n+1}$ *(all n)*
2. $\bigcup_{n\geq 0} \mathcal{L}_n = \mathcal{L}$.

10.2.1 The Grzegorczyk hierarchy

This static hierarchy does not let primitive recursion to force functions into the next level. It does this by restricting *acceptable primitive recursions* to be those that produce functions that are *majorised* by functions *earlier derived*. Hence the concept of *bounded* or more frequently called *limited recursion*.

10.2.22 Definition. [Grz53] Given functions h, g, b, we say that f is defined by *limited recursion* or *bounded recursion from them* provided the two equations and one inequality below are satisfied for all $y, \vec{x}$.

$$\begin{aligned} f(0,\vec{x}) &= h(\vec{x}) \\ f(y+1,\vec{x}) &= g(y,\vec{x},f(y,\vec{x})) \\ f(y,\vec{x}) &\leq b(y,\vec{x}) \end{aligned}$$

□

We can now define the classes $\mathcal{E}^n$ of the Grzegorczyk hierarchy.

10.2.23 Definition. (The Grzegorczyk Hierarchy) We select a sequence of bounding functions, $(g_n)_{n\geq 0}$ (using the same letter as in [Grz53]) by

$$g_0 = \lambda x.x + 1$$
$$g_1 = \lambda xy.x + y$$
$$g_2 = \lambda xy.xy$$

and, for $n \geq 2$,

$$g_{n+1} = \lambda xy.A_n\big(\max(x, y)\big)$$

where $\lambda ny.A_n(x)$ is the Ackermann function *version* we used in Chapter 8.

The hierarchy $(\mathcal{E}^n)_{n\geq 0}$ is defined as follows: $\mathcal{E}^n$ is the *closure* of

$$\{\lambda x.x + 1, \lambda x.x, g_n\}$$

under (Grzegorczyk) *substitution* and *bounded primitive recursion.* □

10.2.24 Remark. (1) Note that closure of $\mathcal{E}^n$ under limited (primitive) recursion means that if f is produced from h, g as $f = prim(h, g)$ and is majorised by b, then it is in $\mathcal{E}^n$, *if all* h, g, b are.

(2) The version of the Ackermann function (and the few "initial" small bounding functions) are as follows still using "g" here, which this time stands for the original bounding functions, as in [Grz53]):

$g_0(x, y) = y + 1$, $g_1(x, y) = x + y$, $g_2(x, y) = (x + 1)(y + 1)$ and, for $n \geq 0$,

$$g_{n+1}(0, y) = g_n(y + 1, y + 1)$$
$$g_{n+1}(x + 1, y) = g_{n+1}(x, g_{n+1}(x, y))$$

[Tou84] contains a direct proof from first principles that g_{n+1} and A_n have the same order of magnitude. Here we will verify this indirectly by virtue of the comparison of the $\mathcal{E}^{n+1}$ with the $K_n^{sim} = \mathcal{L}_n$, for $n \geq 0$. The original $\mathcal{E}^{n+1}$ and the ones based on A_n are the same.

(3) There has been a lot of interest in $\mathcal{E}^0$ and its predicate counterpart, $\mathcal{E}^0_*$ due to its unexpected wealth of nontrivial (albeit "small") functions. □

Nov. 24, 2021

The following theorem facilitates the study of the low Grzegorczyk classes. All the listed results are from [Grz53].

10.2.25 Theorem. *Let us call $\mathcal{M}$ any class of primitive recursive functions that contains $\lambda x.x$ and is closed under substitution and limited recursion. The smallest such class (closure) we will denote, following [War71], by $\mathcal{E}^{-1}$.*

(a) *$\mathcal{M}$ contains $\lambda x.x \dot{-} 1, \lambda xy.x \dot{-} y, \lambda xy.x(1 \dot{-} y)$.*

(b) *$\mathcal{M}_*$ is closed under Boolean operations and bounded quantification (bounds $< z$ and $\leq z$).*

(c) *$\lambda xy.x \leq y, \lambda xy.x < y, \lambda xy.x = y, \lambda xy.x \neq y$ are in $\mathcal{M}_*$.*

(d) *$\mathcal{M}$ is closed under $(\mathring{\mu}y)_{<x}$ and $(\mathring{\mu}y)_{\leq x}$, where unsuccessful search is designated by returning 0.*

(e) *$\mathcal{M}$ is closed under definition by cases, provided the defined function $\lambda \vec{x}_n.f(\vec{x}_n)$ is majorised for all $\vec{x}$ by some constant, or by x_i (for some $1 \leq i \leq n$).*

Proof.

(a) • $\lambda x.x \dot{-} 1$:

$$\begin{aligned} 0 \dot{-} 1 &= 0 \\ (x+1) \dot{-} 1 &= x \\ x \dot{-} 1 &\leq x \end{aligned}$$

- $\lambda xy.x \dot{-} y$:

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

- $\lambda xy.x(1 \dot{-} y)$:

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

(b)
- $(\neg)$: Say $P(\vec{x}) \in \mathcal{M}_*$. Then (10.2.13), for some $p \in \mathcal{M}$, $P(\vec{x}) \equiv p(\vec{x}) = 0$. Then $\neg P(\vec{x}) \equiv (1 \dot{-} p(\vec{x})) = 0$. But $\lambda\vec{x}.1 \dot{-} p(\vec{x}) \in \mathcal{M}$ (use the substitution $x \leftarrow 1$ in $\lambda xy.x(1 \dot{-} y)$ to obtain $\lambda y.1 \dot{-} y$).
- $(\vee)$: Say $P(\vec{x})$ and $Q(\vec{y})$ in $\mathcal{M}_*$. Let p be as above, and similarly let q be so for Q. Then

  $$P(\vec{x}) \vee Q(\vec{y}) \Leftrightarrow p(\vec{x}) \cdot q(\vec{y}) = 0 \Leftrightarrow p(\vec{x}) \cdot (1 \dot{-} (1 \dot{-} q(\vec{y}))) = 0$$

  The last equivalence is the operative one, since the "full"multiplication is not postulated as a member of $\mathcal{M}$, while the function in the third bullet of (a) *is* in $\mathcal{M}$.

- $((\exists y)_{<z}$ and $(\exists y)_{\leq z}$): Let $R(y,\vec{x})$ be in $\mathcal{M}_*$, that is, for some $r \in \mathcal{M}$, it is, $R(y,\vec{x}) \equiv r(y,\vec{x}) = 0$. We are looking for a $g \in \mathcal{M}$ such that

$$(\exists y)_{<z} R(y,\vec{x}) \equiv g(z,\vec{x}) = 0$$

Note that $(\exists y)_{<z+1} R(y,\vec{x}) \equiv R(z,\vec{x}) \vee (\exists y)_{<z} R(y,\vec{x})$, which leads to the limited recursion

$$\begin{aligned} g(0,\vec{x}) &= 1 \quad \textbf{Comment}.\ (\exists y)_{<0} R(y,\vec{x}) \text{ is false} \\ g(z+1,\vec{x}) &= r(z,\vec{x}) \cdot (1 \dot{-} (1 \dot{-} g(z,\vec{x})) \\ g(z,\vec{x}) &\leq 1 \end{aligned}$$

Also note that $(\exists y)_{\leq z} R(y,\vec{x}) \equiv R(z,\vec{x}) \vee (\exists y)_{<z} R(y,\vec{x})$.

- $((\forall y)_{<z}$ and $(\forall y)_{\leq z})$: Previous bullet and closure under $\neg$.

Since all of $\equiv, \rightarrow, \wedge$ are defined in terms of $\neg, \vee$ we have closure of $\mathcal{M}_*$ under *all* Boolean operators (connectives).

(c)
- $(\lambda xy.x \leq y:)\ x \leq y \equiv x \dot{-} y = 0$.
- $(\lambda xy.x < y:)\ x < y \equiv \neg(y \leq x)$.

  Thus we have at once,

  $x = y \equiv x \leq y \wedge y \leq x$, hence is in $\mathcal{M}_*$ and so is $x \neq y \equiv \neg x = y$ by closure under negation.

(d) $((\mathring{\mu} y)_{<x}$ and $(\mathring{\mu} y)_{\leq x})$: Let us set $g = \lambda z \vec{y}.(\mathring{\mu} x)_{<z} f(x,\vec{y})$,

where $f \in \mathcal{M}$. Then

$$\begin{aligned} g(0,\vec{y}) &= 0 \\ g(z+1,\vec{y}) &= \text{if } \overbrace{\neg(\exists x)_{<z} f(x,\vec{y}) = 0 \wedge f(z,\vec{y}) = 0}^{\in \mathcal{M}_*} \\ &\qquad \text{then } z \text{ else } g(z,\vec{y}) \\ g(z,\vec{y}) &\leq z \end{aligned} \tag{1}$$

We have to show how to rewrite the above limited recursion without using "full" if-then-else, which is not postulated to be in $\mathcal{M}$ (in fact, it is *not* in the special cases $\mathcal{E}^{-1}$ and $\mathcal{E}^0$ of $\mathcal{M}$).

Noting that $g(z,\vec{y}) \leq z$, the *iterator* function H of the primitive recursion part of (1) is given by

$$\begin{aligned} H(z,u,\vec{y},w) \overset{Def}{=} &\text{ if } u = 0 \text{ then } z \\ &\Big[\text{ else if } w \leq z \text{ then } w \text{ else } 0 \Big] \end{aligned}$$

The variable u receives the $\mathcal{M}_*$-predicate

$$\neg(\exists x)_{<z} f(x,\vec{y}) = 0 \wedge f(z,\vec{y}) = 0$$

in the form of a function call $F(z,\vec{y})$ where $F \in \mathcal{M}$ is chosen to satisfy

$$\neg(\exists x)_{<z} f(x,\vec{y}) = 0 \wedge f(z,\vec{y}) = 0 \equiv F(z,\vec{y}) = 0$$

The variable w receives the "recursive call" in (1), which returns a value $\leq z$ always, so the "else 0" never applies! We now rewrite the definition of H

as a limited recursion:

$$\begin{aligned} H(z,0,\vec{x},w) &= z \\ H(z,u+1,\vec{x},w) &= \Big(1 \dot{-} (w \dot{-} z)\Big)w \\ H(z,u,\vec{x},w) &\leq z \end{aligned}$$

Thus $H \in \mathcal{M}$. We are done.

Regarding the $\leq z$ bound note that

$$(\mathring{\mu}x)_{\leq z} f(x,\vec{y}) = \text{if } \neg(\exists x)_{<z} f(x,\vec{y}) = 0 \wedge f(z,\vec{y}) = 0 \text{ then } z \text{ else } (\mathring{\mu}x)_{<z} f(x,\vec{y})$$

where we handle the if-then-else as above.

(e) Let

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } R_1(\vec{x}) \\ \vdots & \vdots \\ f_r(\vec{x}) & \text{if } R_r(\vec{x}) \end{cases}$$

First,

$$y = f(\vec{x}) \equiv y = f_1(\vec{x}) \wedge R_1(\vec{x}) \vee y = f_2(\vec{x}) \wedge R_2(\vec{x}) \vee \cdots \vee y = f_r(\vec{x}) \wedge R_r(\vec{x})$$

Thus $y = f(\vec{x}) \in \mathcal{M}_*$ by (b).

But $f(\vec{x}) = (\mathring{\mu}y) \leq x_i (y = f(\vec{x}))$.

□

There are several corollaries:

10.2.26 Corollary. *For $n \geq 0$, $\mathcal{E}^n$ is closed under $(\mathring{\mu}y)_{<z}$ and $(\mathring{\mu}y)_{\leq z}$, as well as under definition by cases. The latter is unrestricted for $n \geq 1$ but for $n = 0$ requires a restriction similar to that in (e) above: The resulting function must be bounded by $x_i + k$ for some x_i among its arguments, and some k.*

Proof. Trivially,* $\mathcal{M} \subseteq \mathcal{E}^n$, for $n \geq -1$. □

*Look at the initial functions.

Nov. 29, 2021

10.2.27 Corollary. *The graphs* $z = x + y$ *and* $z = xy$ *are in* $\mathcal{E}^0_*$.

Proof. Note that [War71] proves a general result that puts these graphs in $\mathcal{E}^{-1}_*$.

Now $z = x + y \equiv z = 0 \vee z > 0 \wedge z \dot{-} x = y$. Also

$$z = xy \equiv (x = 0 \vee y = 0) \wedge z = 0 \vee z > 0 \wedge x|z \wedge y = \lfloor z/x \rfloor$$

Why are $x|z$ in $\mathcal{E}^0_*$ and $\lambda zx.\lfloor z/x \rfloor \in \mathcal{E}^0$? Exercise! (if you give up see bullet 2 among posted notes at URL `http://www.cs.yorku.ca/~gt/courses/EECS5111F21/c5111.html`.) □

10.2.28 Corollary. *For $n \geq 2$, if $f \in \mathcal{E}^{n+1}$ then, for some m depending on f, $f(\vec{x}) \leq A_n^m(|\vec{x}|)$ for all $\vec{x}$.*

Proof. Easy *Exercise*! Only substitution requires a minimum *trace* of work,[†] namely to look back at the proof of the majorisation of primitive recursive functions by Ackermann's function. □

[†]In 8.2.1 we saw that if $h = \lambda\vec{y}.g(f_1(\vec{y}), \ldots, f_m(\vec{y}))$ and g, f_j are majorised by A_n^k for fixed n and various k depending on the majorised function among g, f_j, then h is majorised by A_n^l for some l, same n. See aslso Remark 8.2.2.

10.2.29 Corollary. $\mathcal{E}^n \subseteq \mathcal{E}^{n+1}$, $n \geq 0$.

Proof. For $n \geq 3$, note that majorisation of functions in $\mathcal{E}^n$ by A^m_{n-1} (10.2.28) imply also majorisation by A^k_n thus

all limited recursions that happen in $\mathcal{E}^n$

work in $\mathcal{E}^{n+1}$ too.

Wait! But how about the initial function A_{n-1} of $\mathcal{E}^n$? Why is it also in $\mathcal{E}^{n+1}$?

Hint. Note that $A_0 \in \mathcal{E}^{n+1}$ since S is and $A_0 = SS$. But then A_1 is as it is obtained by primitive recursion from A_0 *and is* bounded by $A^k_n \in \mathcal{E}^{n+1}$ (for some k). But then A_2 is as it is obtained by primitive recursion from A_1 *and is* bounded by $A^r_n \in \mathcal{E}^{n+1}$ (for some r). Etc., etc., all the way to: *but then A_{n-1} is as it is obtained by primitive recursion from A_{n-2}* and is *bounded by $A^t_n \in \mathcal{E}^{n+1}$ (for some t).*

Also note, for the small classes $n < 3$, for

$$\mathcal{E}^0 \subseteq \mathcal{E}^1 \subseteq \mathcal{E}^2 \subseteq \mathcal{E}^3$$

the argument is the same as for the general case, using the bounding (initial) functions $\lambda xy.y + 1$, $\lambda xy.x + y$, $\lambda xy.xy$, and A_2. □

The Grzegorczyk classes, ever since [Grz53] was published have been studied by many researchers. The low classes $\mathcal{E}^0, \mathcal{E}^1, \mathcal{E}^2$ have enormous expressive power (the reader will be asked to place the Kleene predicate in $\mathcal{E}^2$, which will immediately render the equivalence problem of the functions in this class non semi-recursive. (Why?)

[Rit63] studied $\mathcal{E}^2$ extensively (we will see an important role it plays in the simulation of URMs by simultaneous recursions) while $\mathcal{E}^3$ coincides with the well-known class of *elementary functions* directly defined by Kalmár as

10.2.30 Definition. ([Kal43]) The class of the *elementary functions* $\mathcal{E}$ is the closure of the set of initial functions $\{\lambda xy.x + y, \lambda xy.x \dot{-} y\}$ under Grzegorczyk *substitution*, *summation* $(\sum_{i\leq z})$ and product $(\prod_{i\leq z})$. □

The elementary functions contain $\lambda x.2^x$ and hence also "monster functions" like the one on p.276. Intuitively they are viewed as the boundary class between "practical" and "impractical" computing.

10.2.31 Remark. It is easy to show that $\lambda xy.x+y \in \mathcal{E}^1 - \mathcal{E}^0$, $\lambda xy.xy \in \mathcal{E}^2 - \mathcal{E}^1$, $\lambda x.2^x \in \mathcal{E}^3 - \mathcal{E}^2$, hence $\mathcal{E}^0 \subsetneq \mathcal{E}^1 \subsetneq \mathcal{E}^2 \subsetneq \mathcal{E}^3$.

The above hinge on VERY easy proofs that

1. For each function $f \in \mathcal{E}^0$ there are j, k such that $f(\vec{x}) \leq x_i + k$, for all $\vec{x}$.
2. For each function $f \in \mathcal{E}^1$ there are C, k such that $f(\vec{x}) \leq C|\vec{x}| + k$, for all $\vec{x}$.
3. For each function $f \in \mathcal{E}^2$ there are C, k, l such that $f(\vec{x}) \leq C|\vec{x}|^k + l$, for all $\vec{x}$.

   In particular, from the last majorising result, $A_2 \in \mathcal{E}^2 - \mathcal{E}^1$, since A^2 grows like 2^x and thus it cannot be bounded by $C|\vec{x}|^k + l$, for any C, k, l.

Incidentally, it is also $\mathcal{E}^{-1} \subsetneq \mathcal{E}^0$ since $\lambda x.x+1 \notin \mathcal{E}^{-1}$.[‡]

□

[‡]By induction on the formation of $\mathcal{E}^{-1}$ one shows that for every f in this class there are an i and C (depend on f) such that $f(\vec{x}) \leq x_i$ or $f(\vec{x}) \leq C$ for all $\vec{x}$.

10.2.32 Theorem. *$(\mathcal{E}^n)_{n\geq 0}$ is a proper hierarchy of $\mathcal{PR}$, that is,*

- $\mathcal{E}^n \subsetneq \mathcal{E}^{n+1}$, *for* $n \geq 0$

  and

- $\mathcal{PR} = \bigcup_{n\geq 0} \mathcal{E}^n$

Proof.

- Firstly, the non-strict inclusion has been proved in 10.2.29. For $n \leq 2$, $\mathcal{E}^n \subsetneq \mathcal{E}^{n+1}$ by Remark 10.2.31. On the the hand, for $n \geq 3$, we have $A_n \in \mathcal{E}^{n+1} - \mathcal{E}^n$.

  Note that $A_n \in \mathcal{E}^n$ would entail $A_n(x) \leq A^k_{n-1}(x)$ for some k and all x, contradicting 8.1.12.

- The $\supseteq$ is trivial.

  The $\subseteq$ is easy but nontrivial:

  Say then that $f \in \mathcal{PR}$ and let us do induction on the formation of f:

  1. Case where f is initial. Then $f \in \mathcal{E}^0$ as initial.
  2. Assume for h and g and let $f = prim(h, g)$. By the I.H. on h, g we have an n[§] such $\{h, g\} \subseteq \mathcal{E}^{n+1}$, hence $h \leq A^k_n$ and $g \leq A^r_n$. But then $f \leq A^m_{n+1}$ for some m, placing f in $\mathcal{E}^{n+2}$ since $prim(h, g)$ is a limited (by A^m_{n+1}) recursion in $\mathcal{E}^{n+2} \supseteq \mathcal{E}^{n+1} \supseteq \{h, g\}$.
  3. Finally, assume $f = comp(g, h_1, \ldots, h_l)$. By the I.H. g and the h_j are all (wlog) in $\mathcal{E}^{n+1}$. But the latter class is closed under composition thus $f \in \mathcal{E}^{n+1}$. □

[§]WLOG, ONE n.

10.3 The Ritchie-Cobham property and hierarchy comparison

The comparison of the hierarchies involves proving that the *dynamic complexity* of the functions in the various classes we studied ($K_n, K_n^{sim}, \mathcal{L}_n, \mathcal{E}^{n+1}$) goes hand in hand with their *definitional complexity* —one complexity predicts the other. This result is the "*Richie*[†]*-Cobham property*" which is the central tool in this section.

[†]D. Ritchie.

As we will simplify our "programming tasks" by using loop programs *instead of* URMs we will start by making explicit the implementation of the **Loop** X**-end** instruction that respects the stated loop-program semantics, in particular the part where *changing the loop variable in the body of the loop does not change the number of iterations*. We need this elaboration as we shall be looking into loop-program run times, thus we need to know what exactly is going on in the looping mechanism.

We imagine that we have *numbered* all instructions in our loop-programs in the style of URMs.

Now, each of several **Loop** X-**end** instruction-*pairs* —I emphasise: with the *same* variable X— is assigned a *distinct* "*hidden*" variable H_X^m assigned to the *m-th* such loop *encountered from top to bottom of the overall program*. We translate so ("macro expand") each **Loop** X-**end** pair —which may occur multiple times.

A program segment

$$L: \ \textbf{Loop}\ X$$
$$\vdots$$
$$R: \ \textbf{end}$$

is *implemented* as follows (e.g., on a URM or "real life compiler"), where for simplicity of notation we call the hidden variable assigned just "H".

Why "hidden"? This is an often used nomenclature for compiler-generated "internal" (to the compiler) variables that *are not accessible to the user* (programmer).

$$L : \textbf{Loop } X \textbf{ translates: } \begin{cases} L : \ H \leftarrow X \\ L+1 : \text{if } H = 0 \text{ goto } R+2 \text{ else goto } L+2 \end{cases}$$

$$\vdots$$

$$R : \textbf{end} \qquad \textbf{translates: } \begin{cases} R : \ H \leftarrow H \dot{-} 1 \\ R+1 : \text{goto } L+1 \end{cases}$$

With the above *clarification* of what **Loop** X-**end** does exactly, we note that thus a URM can simulate a given loop program *without any run time loss*.

> *But what about the fact that the URM does not have the instruction $X \leftarrow Y$ (as primitive) that loop programs do have, and thus the former spends as much time as $O(y)$, where y is the contents of Y ?* [b]
>
> [b] The reader recalls that the simulation is rather "expensive". Cf. 2.4.7 and 2.4.8.

The simple (and also theoretically correct) answer is "OK; let's retroactively fit the URM with such instructions. This does not change the computability theory we developed so far, since the instruction can be simulated anyway".

Dec. 1, 2021

We next refer back to the *simulation of a URM* by a *simultaneous recursion*, specifically, Theorem 10.1.1. We quote the concluding sentence in the proof of the latter theorem:

> *Also note that the recursive calls happen within trivially "easy" functions such as* $\lambda z.z$, $\lambda z.z+1$ *or* $\lambda z.z \dot{-} 1$ *or trivial predicates like* $\lambda z.z = y$ *and* $\lambda z.z \neq y$ *(*y *a constant). All in* $\mathcal{PR}$ *and* $\mathcal{PR}_*$.

As a result of our work in Section 1 of this chapter, the part in the quote above "All in $\mathcal{PR}$ and $\mathcal{PR}_*$" can be replaced by "All (subfunctions/subpredicates of the "iterator" are) in K_1^{sim} and $K_{1,*}^{sim}$" and thus the simulating functions IC and X_j are all in $K_2^{sim} = \mathcal{L}_2$".

10.3.1 Lemma. *All the simulating functions in Theorem 10.1.1, i.e., the X_i and IC are in $K_2^{sim} = \mathcal{L}_2$.*

In fact, all simulating functions of a URM are in $\mathcal{E}^2$. We prove first

10.3.2 Lemma. *All $\mathcal{E}^n$, $n \geq 2$, are closed under limited simultaneous recursion.*

Proof. Let

$$\text{for } i = 1, \dots, k \begin{cases} f_i(0, \vec{y}) & = h_i(\vec{y}) \\ f_i(x+1, \vec{y}) & = g_i(x, \vec{y}, f_1(x, \vec{y}), \dots, f_k(x, \vec{y})) \\ f_i(x, \vec{y}) & \leq b_i(x, \vec{y}) \end{cases} \tag{1}$$

where the h_i, g_i, b_i are in $\mathcal{E}^n$.

We convert the simultaneous recursion to a **simple (single) limited recursion** in the style of 4.2.2, HOWEVER using the coding of Definition 4.3.8 and the projections —Π_i^{k+1}— therein, *expressed as substitutions* using only K and L.

This is using the quadratic (non onto) pairing function from 4.3.7 $J = \lambda xy.(x+y)^2+x$ to define the "code" $[\![\vec{x}_k]\!]^{(k)}$ and the projections Π^k_i, $i = 1, \ldots, k$.

All these projections are in $\mathcal{E}^2$ since they are obtained by substitution from the projections of J

$$Kz^{\dagger} = (\mathring{\mu}x)_{\leq z}(\exists y)_{\leq z} z = J(x,y)$$

and

$$Lz = (\mathring{\mu}y)_{\leq z}(\exists x)_{\leq z} z = J(x,y)$$

Note that $\lambda \vec{x}_k.[\![\vec{x}_k]\!]^{(k)}$ itself is in $\mathcal{E}^2$, being obtained by a finite number of substitutions from $J(x,y)$ (cf. 4.3.8).

[†]It is quite common to write Kz for $K(z)$ and $\Pi^k_i z$ for $\Pi^k_i(z)$.

So, following the Hilbert and Bernays ([HB68]) process of 4.2.2, we set

$$F(x,\vec{y}) \overset{Def}{=} [\![f_1(x,\vec{y}),\ldots,f_k(x,\vec{y})]\!]^{(k)}$$

thus the simultaneous recursion becomes a simple (single) limited recursion

$$\begin{aligned} F(0,\vec{y}) &= [\![h_1(x,\vec{y}),\ldots,h_k(x,\vec{y})]\!]^{(k)} \\ F(x+1,\vec{y}) &= [\![\ldots, g_i(x,\vec{y},\Pi_1^k F(x,\vec{y}),\ldots,\Pi_k^k F(x,\vec{y})),\ldots]\!]^{(k)} \\ F(x,\vec{y}) &\leq [\![\ldots, b_i(x,\vec{y}),\ldots]\!]^{(k)} \end{aligned}$$

where the inequality is valid since the J is increasing with respect to both variables and composing it a finite number of times with itself does not change this fact.

By the presence of the $[\![\ldots]\!]^{(k)}$ and the Π_i^k (all in $\mathcal{E}^2$), it is $F \in \mathcal{E}^n$, $n \geq 2$. Hence so is $f_i = \Pi_i^k F$.

□

We apply the above to the simulation via simultaneous primitive recursion of URM computations (cf. 10.1.1 and 10.3.1).

10.3.3 Lemma. *The simulating functions for a given URM, i.e., $\lambda y\vec{x}.X_i(y,\vec{x})$ and $\lambda y\vec{x}.IC(y,\vec{x})$, are all in $\mathcal{E}^2$.*

Proof. It suffices to prove that the simultaneous recursion we introduced in 10.1.1 is *limited (bounded)*, with bounds from $\mathcal{E}^2$.

Why so? Because the part-functions and and predicates in the iterator of the *simulating simultaneous recursion* are all in K_1^{sim} and $K_{1,*}^{sim}$ hence trivially also in $\mathcal{E}^2$ and $\mathcal{E}_*^2$, respectively.

All that remains to get a bound from $\mathcal{E}^2$.

To this end, we note that, for all variables of the under simulation URM M, we have

$$X_i(y, \vec{x}) \leq \max(\vec{x}) + \max\{a : a \text{ occurs in an } X \leftarrow a \text{ instruction of } M\} + y$$

The estimate above is so since in *each* of y steps the most we can add to any variable is 1.

Also, $IC(y, \vec{x}) \leq k$, where k labels **stop** in M.

Thus the bounding functions are all in $\mathcal{E}^1 \subseteq \mathcal{E}^2$. □

This result is also needed:

10.3.4 Lemma. *For $n \geq 2$, $\mathcal{E}^n$ is closed under $\sum_{i \leq z}$.*

Proof. Let $f \in \mathcal{E}^n$, $n \geq 2$. We will show that

$$g = \lambda z \vec{y}. \sum_{i \leq z} f(i, \vec{y}) \in \mathcal{E}^n \tag{1}$$

Indeed,

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

The iterator $f(z+1, \vec{y}) + w$ being in $\mathcal{E}^n$ we next look for a *bound* for g.

- First, the $n = 2$ case:

$$g(z,\vec{y}) \leq \sum_{i\leq z} f(i,\vec{y}) \leq \sum_{i\leq z} \Big(C\Big(\max(i,\vec{y})\Big)^r + l\Big)$$
$$\leq C(z+1)\Big(\big(\max(z,\vec{y})\big)^r + l\Big)$$

The bound $\lambda z\vec{y}.C(z+1)\Big(\big(\max(z,\vec{y})\big)^r + l\Big)$ is in $\mathcal{E}^2$ by the fact that $\lambda zw.zw \in \mathcal{E}^2$.

- The $n > 2$ case: As A^k_{n-1} majorises *every* function in $\mathcal{E}^n$, for a k that depends on the function, we have

$$\begin{aligned} g(z, \vec{y}) &\leq \sum_{i \leq z} f(i, \vec{y}) \\ &\leq \sum_{i \leq z} A^k_{n-1}\Big(|i, \vec{y}|\Big) \qquad (1) \\ &\leq (z+1) A^k_{n-1}\Big(|z, \vec{y}|\Big) \end{aligned}$$

  Since $\lambda xy.xy \in \mathcal{E}^n$, for $n \geq 2$, and $A_{n-1} \in \mathcal{E}^n$, we have that the bound in (1) is in $\mathcal{E}^n$. □

10.3.5 Lemma. (Brute force direction of R-C) *For $n \geq 2$, if $\lambda \vec{x}_k.f(\vec{x}_k) \in \mathcal{E}^n$, then there is a URM M —such that $f = M_{X_1}^{\vec{X}_k}$— that* runs within time *$\lambda \vec{x}_k.t(\vec{x}_k) \in \mathcal{E}^n$, that is, for every input $\vec{x}_k$, $M_{X_1}^{\vec{X}_k}$ halts (i.e., reaches* **stop***) in $\leq t(\vec{x}_k)$ steps.*

Proof. Just program it!

As noted earlier, *it suffices to carry out our programming on the loop programs formalism*. The proof is by induction on $n \geq 2$.

For each n there is another induction on the closure that the set $\mathcal{E}^n$ is.

n-Basis: $n = 2$. Induction on $\mathcal{E}^2$ requires to verify the contention for the initial functions $\lambda x.x$, $\lambda x.x + 1$ and $\lambda xy.xy$ first. The first two need one-line (loop) programs whose run time is 1 —a constant. *But $\mathcal{E}^2$ contains all constant functions*. Good!

As for $\lambda xy.xy$, the loop program

$$P: \begin{cases} \textbf{Loop } X \\ \quad \textbf{Loop } Y \\ \quad\quad Z \leftarrow Z + 1 \\ \quad \textbf{end} \\ \textbf{end} \end{cases}$$

satisfies $P_Z^{XY} = \lambda xy.xy$. *Its run time is* —since each execution of **end** counts for two steps and of **Loop** counts for one (except *upon first entry*, where an extra one step is charged)— $\leq ((4y + 1) + 3)x + 1$.

Clearly, $\lambda xy.((4y + 1) + 3)x + 1 \in \mathcal{E}^2$!

We can also observe simply that ignoring constant overhead we can estimate the run time quickly seeing how many times we go around the loop and say at once that the run time is $O(xy)$.

We postpone the induction over $\mathcal{E}^2$ as we prefer to do all these simultaneously for $n \geq 2$ (*the reasoning is identical, for each* n).

So we next bound from above the run time when computing the *initial* functions of $\mathcal{E}^n$, $n > 2$, on appropriate loop programs (as proxies for appropriate URMs).

Only A_{n-1} is left to consider.

We will find a run time *upper bound*, T, of a well chosen loop program that computes A_{n-1}, $n > 2$, and show that $T \in \mathcal{E}^n$.

Now, $A_{n-1} \in K_{n-1}^{sim} = \mathcal{L}_{n-1}$.

Let then M be a loop program in L_{n-1} (Definition 10.2.17) such that

$$A_{n-1} = M_Y^X$$

Modify M into a new $\widetilde{M}$ that is still in L_{n-1} and still computes A_{n-1}, namely,

$$A_{n-1} = \widetilde{M}_Y^X$$

The modification is THIS: We choose a variable Z that *does not occur* in M and strategically place several instructions $Z \leftarrow Z + 1$ in it to obtain $\widetilde{M}$.

These are placed, *one copy* before *each* non-**Loop**, non-**end** instruction of M.

For instructions **Loop** or **end** we act according to our simulation protocol of loop programs by URMs, so we place *one copy* of $Z \leftarrow Z + 1$ *before* a **Loop** instruction

and *one after*,

while we put *two* copies *before* an **end** instruction.

See below.

$\vdots$

$Z \leftarrow Z+1$ **Comment**. *Entry of loop count*

Loop X

$Z \leftarrow Z+1$ **Comment**. *Loop's second count*

$\vdots$

$Z \leftarrow Z+1$ **Comment**. *Count for next M-instruction*

A non(loop/end) M**-instruction:** $U \leftarrow 0$ *or* $U \leftarrow U+1$ *or* $U \leftarrow W$

$\vdots$

$Z \leftarrow Z+1$ **Comment**. *Execution of* **end** *count one*

$Z \leftarrow Z+1$ **Comment**. *Execution of* **end** *count two*

end

$\vdots$

Clearly $\widetilde{M}^X_Y = A_{n-1}$ STILL!, and $\widetilde{M}^X_Z$ is in $\mathcal{L}_{n-1}$ (*no new loops added*) *and measures the run time* for A_{n-1} *according to* M.

Since $\widetilde{M}^X_Z \in K^{sim}_{n-1} = \mathcal{L}_{n-1}$, for some r, this function is majorised by $A^r_{n-1}(x)$, for all inputs x that are "read" in X.

A^r_{n-1} is the "T" we promised:

1. It majorises the run time of $A_{n-1} = M^X_Y$ on some URM M.
2. Since $A_{n-1} \in \mathcal{E}^n$, and $\mathcal{E}^n$ is closed under substitution, this run-time majorant, $T = A^r_{n-1}$, is also in $\mathcal{E}^n$.

Dec. 6, 2021

We finally are ready to do induction over the closure $\mathcal{E}^n$, $n \geq 2$, to conclude, that if $f \in \mathcal{E}^n$, then, for some URM (loop program will do by earlier remarks) M, $f = M_Y^{\vec{X}_k}$ and the run time is majorised by a function from $\mathcal{E}^n$.

We have already established the Basis, that all *initial* functions of $\mathcal{E}^n$ have the property.

We note

- The property propagates with substitution. So let h and g have the property, and consider $f = h(g(\vec{x}), \vec{y})$.

  We write t_f, etc., for the run *time majorising function* on some appropriate URM.

  Programming in the obvious way (superposition) we have

  $$t_f(\vec{x}, \vec{y}) = t_h(g(\vec{x}), \vec{y}) + t_g(\vec{x})$$

  As our $\mathcal{E}^n$ is closed under substitution, contains $\lambda xy.x + y$ and h and g, have the property, we have $t_f \in \mathcal{E}^n$. The other substitution cases are trivial.

- The property propagates with limited recursion. So let f be given by the following schema, from h, g, b in $\mathcal{E}^n$:

$$\begin{aligned} f(0, \vec{y}) &= h(\vec{y}) \\ f(x+1, \vec{y}) &= g(x, \vec{y}, f(x, \vec{y})) \\ f(x, \vec{y}) &\leq b(x, \vec{y}) \end{aligned}$$

The recursion is implemented on a loop program as

$$\begin{aligned} & i \leftarrow 0 \\ & z \leftarrow h(\vec{y}) \\ \textbf{Loop } & X \\ & z \leftarrow g(i, \vec{y}, z) \\ & i \leftarrow i + 1 \\ \textbf{end} & \end{aligned}$$

We next estimate —recalling how **Loop-end** works— an upper bound $t_f(x, \vec{y})$ for the run time of the above.

$$t_f(x, \vec{y}) = t_h(\vec{y}) + 2 + \sum_{i<x} (4 + t_g(i, \vec{y}, f(i, \vec{y})))$$

By the I.H. $\{t_g, t_h\} \subseteq \mathcal{E}^n$ and so is f. Thus, by Lemma 10.3.4 we have that $t_f \in \mathcal{E}^n$. □

10.3.6 Theorem. (The Ritchie-Cobham property of $\mathcal{E}^n, n \geq 2$) *$f \in \mathcal{E}^n$, for $n \geq 2$, iff f runs on some URM M —that is, $f = M_Y^{\vec{X}_k}$— within time $t_f \in \mathcal{E}^n$.*

Proof. The *only if* is Lemma 10.3.5. For the *if*, say $f = M_Y^{\vec{X}_k}$ and t_f on that M is in $\mathcal{E}^n, n \geq 2$.

By Lemma 10.3.3, $\lambda y \vec{x}_k . Y(y, \vec{x}_k) \in \mathcal{E}^2$. But $f(\vec{x}_k) = Y(t_f(\vec{x}_k), \vec{x}_k)$ for all $\vec{x}_k$, and thus $f \in \mathcal{E}^n$ via substitution. □

10.3.7 Corollary. *$\mathcal{L}_n = \mathcal{E}^{n+1}$, for $n \geq 2$.*

Proof. A function $\lambda\vec{x}.f(\vec{x})$ in $\mathcal{L}_n$ is computable by a loop program from L_n and hence by a URM M —i.e., $f = M_Y^{\vec{X}}$— within a run time that is bounded by A_n^k, for some k.

Thus $f(\vec{x}) = Y(A_n^k(|\vec{x}|), \vec{x})$, for all $\vec{x}$. But $\lambda y\vec{x}.Y(y, \vec{x}) \in \mathcal{E}^2$ and $A_n^k \in \mathcal{E}^{n+1}$.

Conversely, say $f \in \mathcal{E}^{n+1}$. Let a URM M compute f —that is $f = M_Y^{\vec{X}_k}$— in time $t_f \in \mathcal{E}^{n+1}$. Then for some r, $t_f(\max(\vec{x}_k)) \leq A_n^r(\max(\vec{x}_k))$.

Therefore, $f(\vec{x}_k) = Y\Big(A_n^r(\max(\vec{x}_k)), \vec{x}_k\Big)$, for all $\vec{x}_k$, as well. (Why?)

It follows that $f \in K_n^{sim}$, since A_n^r is and also the simulating function Y (in fact the latter is in K_2^{sim} (10.3.1)). But $K_n^{sim} = \mathcal{L}_n$. □

10.3.8 Corollary. $K_n^{sim} = \mathcal{E}^{n+1}$, *for* $n \geq 2$.

10.3.9 Corollary. $K_n \subseteq \mathcal{E}^{n+1}$, *for* $n \geq 2$.

Proof. $K_n \subseteq K_n^{sim}$, $n \geq 0$. Now apply 10.3.8. □

10.3.10 Corollary. *$K_n = \mathcal{E}^{n+1}$, for $n \geq 4$.*

Proof. We prove $\mathcal{E}^{n+1} \subseteq K_n$, for $n \geq 4$.

We did not get the URM simulating functions *placement* in the $\mathcal{K}$ hierarchy. Let's do it here:

Aside: *The simulating functions for a URM M are in K_4.*

Proof.

We work as in 10.3.2 converting the simultaneous recursion of Theorem 10.1.1 into a single recursion.

As in 10.3.2 we start with the quadratic pairing function $J = \lambda xy.(x+y)^2+x \in K_2$. Thus $I_k \overset{Def}{=} \lambda \vec{x}_k. [\![\vec{x}_k]\!]^{(k)} \in K_2$, for any fixed k (cf. 10.2.12).

In 4.3.7 we derived $Kz = z \dot{-} \lfloor \sqrt{z} \rfloor^2$ and $Lz = \lfloor \sqrt{z} \rfloor \dot{-} Kz$. Since $\lambda z. \lfloor \sqrt{z} \rfloor \in K_3$ (Exercise!), so are K and L and thus the Π_i^k as well.

Therefore, imitating the proof of 10.3.2 (*not* using a bounding function here), we set

$$F(y,\vec{x}) = [\![\, f_1(y,\vec{x}),\ldots,f_k(y,\vec{x}) \,]\!]^{(k)}$$

and thus the iteration equation for F is

$$F(y{+}1,\vec{x}) = [\![\ldots, g_i(y,\vec{x},\Pi^k_1 F(y,\vec{x}),\ldots,\Pi^k_k F(y,\vec{x})),\ldots]\!]^{(k)} \tag{1}$$

Since we iterate —in (1)— the K_3-functions Π^k_j, the result of the recursion, F and hence also the $f_j = \Pi^k_j F$, for all j, are in K_4.

Wait! What is the contribution of the "g_i"? Negligible, as these are definitions by cases of K_1 functions and $K_{1,*}$ predicates.

Let $f \in \mathcal{E}^{n+1}$, $n \geq 4$. Then $f = M^{\vec{X}}_Y$ for some URM M that computes f within time $t_f \leq A^k_n$, for some k.

The simulation function for the output variable, $\lambda\vec{x}.Y(y,\vec{x})$ is in K_4 by the **Aside** above and thus $f = \lambda\vec{x}.f(\vec{x}) = \lambda\vec{x}.Y(A^k_n(|\vec{x}|),\vec{x})$ is in K_n, $n \geq 4$. □

10.3.11 Remark. The Ritchie-Cobham property does two things:

1. Connects the definitional with the computational complexity of primitive recursive functions.
2. Is a powerful tool to compare primitive recursive hierarchies.

Note however that the correspondence between definitional and computational complexity might be overstated sometimes, inflating the computational complexity *a priori* estimate. For example the program P below

Loop X
  Loop Y
    Loop Z
      $W \leftarrow W + 1$
    end
  end
end

computes $P_W^{XYZ} = \lambda xyz.xyz$, a function in $\mathcal{L}_2$ running in $O(xyz)$ time. The predictions of our theory are (correct but) pessimistic: "P_W^{XYZ} runs in time $O(A_3^m(\max(x, y, z)))$, for some m".

The realisation that $\lambda xyz.xyz$ is computable in $O(xyz)$ time on some URM (that simulates our loop program) means that this function is in $\mathcal{E}^3 = \mathcal{L}_2$. The question of whether we can place an arbitrary function given by a program in the lowest level of the hierarchies is recursively unsolvable ([MR67]). □

Bibliography

[Axt65] P. Axt, *Iteration of Primitive Recursion*, Zeitschrift für math. Logik **11** (1965), 253–255.

[Chu36] Alonzo Church, *An unsolvable problem of elementary number theory*, Amer. Journal of Math. **58** (1936), 345–363, (Also in [Dav65, 89–107]).

[Dav65] M. Davis, *The undecidable*, Raven Press, Hewlett, NY, 1965.

[Ded88] R. Dedekind, *Was sind und was sollen die Zahlen?*, Vieweg, Braunschweig, 1888, (In English translation by W.W. Beman [Ded63]).

[Ded63] ———, *Essays on the Theory of Numbers*, Dover Publications, New York, 1963, (First English edition translated by W.W. Beman and published by Open Court Publishing, 1901).

[Göd31] K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatshefte für Math. und Physik **38** (1931), 173–198, (Also in English in [Dav65, 5–38]).

[Grz53] A. Grzegorczyk, *Some classes of recursive functions*, Rozprawy Matematyczne **4** (1953), 1–45.

[HB68] D. Hilbert and P. Bernays, *Grundlagen der Mathematik I and II*, Springer-Verlag, New York, 1968.

[Hei61] W. Heinermann, *Untersuchungungen über die Rekursionzahlen rekursiver Funktionen*, Ph.D. thesis, Münster Univers., 1961.

[Kal43] L. Kalmár, *A Simple Example of an Undecidable Arithmetical Problem (Hungarian with German abstract)*, Matematikai és Fizikai Lapok **50** (1943), 1–23.

[Kle43] S.C. Kleene, *Recursive predicates and quantifiers*, Transactions of the Amer. Math. Soc. **53** (1943), 41–73, (Also in [Dav65, 255–287]).

[LeV56] William J. LeVeque, *Topics in number theory*, vol. I and II, Addison-Wesley, Reading, MA, 1956.

[Mar60] A. A. Markov, *Theory of algorithms*, Transl. Amer. Math. Soc. **2** (1960), no. 15.

[MR67] A. R. Meyer and D. M. Ritchie, *Computational complexity and program structure*, Technical Report RC-1817, IBM, 1967.

[Mul73] H. Muller, *Characterisation of the elementary functions in terms of depth of nesting of primitive recursive functions*, Recursive Function Theory: Newsletter **5** (1973), 14–15.

[Pos36] Emil L. Post, *Finite combinatory processes*, J. Symbolic Logic **1** (1936), 103–105.

[Pos44] ———, *Recursively enumerable sets of positive integers and their decision problems*, Bull. Amer. Math. Soc. **50** (1944), 284–316.

[Rit63] R. W. Ritchie, *Classes of Predictably Computable Functions*, Transactions of the Amer. Math. Soc. **106** (1963), 139–173.

[Rit65] D.M. Ritchie, *Complexity Classification of Primitive Recursive Functions by their Machine Programs*, Term paper for Applied Mathematics 230, Harvard University, 1965.

[Rog67] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[Sch69] Helmut Schwichtenberg, *Rekursionszahlen und die Grzegorczyk-Hierarchie*, Arch. math. Logik **12** (1969), 85–97.

[SS63] J. C. Shepherdson and H. E. Sturgis, *Computability of recursive functions*, Journal of the ACM **10** (1963), 217–255.

[Tou84] G. Tourlakis, *Computability*, Reston Publishing, Reston, VA, 1984.

[Tou12] ———, *Theory of Computation*, John Wiley & Sons, Hoboken, NJ, 2012.

[Tur37] Alan M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math Soc. **2** (1936, 1937), no. 42, 43, 230–265, 544–546, (Also in [Dav65, 115–154].).

[TW68] D. Tsichritzis and P. Weiner, *Some Unsolvable Problems and Partial Solutions*, Tech. Report 69, Dept. of Electrical Eng. Comp. Sciences Lab, Princeton University, Princeton, N.J., 1968.

[War71] J. C. Warkentin, *Small Classes of Recursive Functions and Relations*, Tech. Report CSRR 2052, Dept. of Appl. Analysis and Comp. Science Research Report, University of Waterloo, 1971.