Contents

Chapter 1

Preliminaries

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

> So it is not a "How To" course —i.e., "How can I *program* a solution to this or that problem?"— but it is rather a "Why **can't I do** this by writing a program"?
>
> We develop a "theory of programs" which enables us to *demonstrate* that there is *NO WAY* to solve certain *Problems* by *Programming* and we learn to *investigate and understand* why this happens.

But what *IS* "Programming"?

What will it look like in, say, 10 years, 50 years? Read on!

The above asks, but in modern jargon, the old question "what IS a mechanical procedure?" that the Pioneers of Computability (1930s) *asked and answered*.

- At the **intuitive level**, any practicing mathematician or computer scientist—indeed any student of these two fields of study—**will have no difficulty in recognizing** a *computation* or an *algorithm* ("program") when they see one.

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

    NO!

    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?"

    Maybe YES maybe NO! At this point we simply do not know!

[*]This "is there" is not time-dependent just like Mathematics is not; it means "will there ever be?"
[†]A "correct" program produces, *for every input*, precisely the output that is expected by an *a priori* specification.

- But **what do we mean** by

  “there is *no algorithm* that solves a given problem”—with or without restrictions on the algorithm’s efficiency?

  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** such an algorithm* and a proof that it works as claimed.

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

- One evidently needs a **mathematically precise definition of the concept of algorithm** that is *neither experiential*[‡] *nor technology-dependent*[§] in order to assert that we encountered such a failed "search".

  This directly calls for a *mathematical theory* whose objects of study include *algorithms* (and, correspondingly, *computations*) **in order to construct such sets of (all) algorithms within the theory and to be able to *reason about the membership problem of such sets*.**

[‡]E.g., a FORTRAN, or C, or JAVA program.
[§]Such program might fail for trivial technological reasons. For example memory size.

- The "*theory* of computation" vs. the *meta*theory of computing.

  Within the theory of computing one **computes** to solve some Problem.

  In the (meta)theory of computing one **tackles the fundamental questions** of the *limitations of computing*,

  These limitations may rule out outright the existence of algorithmic solutions for some problems, while for others they rule out *efficient* solutions.

- Our approach is anchored on the **concrete practical knowledge** about general computer programming attained by the reader in a first year programming course.

- Our chapter on computability is **the most "general"** *meta*theory of computing.

The above line does not brag. By "general" I mean that we don't do "metatheory of JAVA" or "metatheory of FORTRAN"

Our metatheory is based on a *fictitious* programming language so that

1. We will not worry about technology-dependent issues such as *memory limitations* —our "mechanical processes" have *none*!
2. We will have control over the *choice* of instructions. Chosen to be *trivial* in terms of understanding and using them. This is essential for achieving the attribute "mechanical" for our procedures.[¶]

So we want to develop a "metatheory of programs" or "metatheory of programming" and that is not about FORTRAN or C or JAVA.

[¶]Imagine if someone had to prove a theorem in order to understand and execute an instruction!

What are the approximate features of our fictitious programs? They will

1. be able to receive input and (in principle[‖]) generate output.
2. have variables that are *not* limited as to the size of data they can hold.

   It would trivialise "unsolvability results" if a computation failed just because the result was too large!

3. only be able to perform instructions that
   - do *trivial* arithmetic —(essentially only $+1$ and -1)
     OR
   - ones that cause the computation to "jump" to this or that instruction, a decision made by the program based on the value of some variable (if-statement)
4. *That's IT!*

1, 3 and 4 address the concept of *instruction*.

[‖]It is known AND all right that some computations do NOT terminate!

So our programming language will be along the lines 1-4 above.

Its programs obviously describe "mechanical procedures" as follows from what we said about *instructions*.

Two questions are important before we start implementing all these ideas:

(1) Are results that we prove about our fictitious language valid for FORTRAN? C, etc.?

   Answer: **Yes**. It is a theorem (proved, *essentially*, in the 1930s) that the simple fictitious programming language can do anything a commercially available (now) language can do, and do so without restriction to data size.

   We also have the converse, *trivially*, since all such commercial languages can do $+1, -1$ and if-statements.

(2) What about future languages? What can we say about the future? We postpone this question until the chapter on *Church's thesis*.

In CS/MATH curricula there are two main contenders for a "fictitious programming language". The older one is the Turing Machine, the newer one is the URM.

For our part we will develop this metatheory via the programming formalism known as Shepherdson-Sturgis *Unbounded Register "Machines"* (URM)—which is a straightforward abstraction of modern *high level* programming languages.

► Contrast with TMs. ("TM" is the acronym for Turing Machine invented by Allan Turing)

These TMs imitate Assembly programming and they are very cumbersome.

Moreover, the principle of going from the "concrete" to the "abstract" speaks against using a mathematical model that looks almost exactly like Assembly language (actually even *more* cumbersome than that**):

According to the prerequisite structure of EECS2001 we are only guaranteed that students did JAVA (and Discrete MATH) before this course.

**In Assembly language you *can* manipulate Integers. In TM language you *cannot*; you manipulate, essentially, one digit at a time of the number stored in a variable.

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 of 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 outputs such as

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

The qualification above, "in principle", is to remind us that while our fictitious mathematical model CAN compute (1) for ANY x-value, a "real" computer running, say, C *cannot fit in its memory* the answer of (1) even for $x = 0$.

The number is astronomical.

- The chapter on Computability—after spending due care in developing the technique of *reductions*—concludes by demonstrating the intimate connection between the *unsolvability phenomenon* of computing on one hand, and the *unprovability phenomenon* of proving within first-order logic (cf. [Göd31]) on the other, when the latter is called upon to reason about "rich" theories such as (Peano's) arithmetic.

- **Restricted Models**. FA and NFA and their Languages.

Chapter 2

Sep. 12, 2022

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*.

► Such a mathematical formulation provides tools to prove that infinitely many Problems cannot have solutions via mechanical procedures.

The advent of computability was strongly motivated, in the 1930s, by

Hilbert's *undertaking* —or "Hilbert's *program*" as one often calls it— to found mathematics on a (metamathematically *provably*) consistent (i.e., free from contradiction) axiomatic basis ...

► ... in particular by his belief that the *Entscheidungsproblem*,

or *decision problem*, for axiomatic theories,

that is, the problem "**is this formula a theorem of that theory?**" *was solvable by a mechanical procedure that was yet to be discovered*.

*What **IS** a "mechanical procedure"?* led to the advent of computability.

Now, since antiquity, mathematicians have invented "*mechanical procedures*", e.g., Euclid's algorithm for the "greatest common divisor",* and *had no problem recognizing such procedures when they encountered them*.

But how do you mathematically *prove* the *nonexistence* of such a mechanical procedure for a particular problem?

You need a *mathematical formulation* of what *is* a "mechanical procedure" in order to do that!

*That is, the largest positive integer that is a common divisor of two given integers.

2.2. A Programming Framework for Computable Functions

So, what *is* a **computable function**, *mathematically speaking*?

There are *two main ways* to approach this question.

1. One is to *define a programming formalism*—that is, **a programming language**—and say: "a **function is computable precisely if** it can be '*programmed*' in **this** programming language".

   Examples of such *programming languages* are

   - the *Turing Machines* (or TMs) of Turing
   - and the *unbounded register machines* (or URMs) of Shepherdson and Sturgis◀ **Our choice!**

   Key in these "programming languages" is

   (a) Do not make them dependent on technology!

   (b) Be sure that individual instructions are so simple as to *require no intelligence* to execute.

Note that the term *machine* in each case is a misnomer, as both the TM and the URM formulations are really *programming languages*,

A TM being very much like the *assembly language* of "real" computers,

A URM reminding us more of (subsets of) *Algol* (or *Pascal*).

2. *The other main way* is to define a set of *computable functions* **directly**—without using a programming language as the agent of definition:

   How? By a devise that resembles **a mathematical proof**, called a **derivation**.

   ▶ In this approach we say a " **function is computable precisely if** it has a *derivation* —is derivable".

   ▶ **Analogy**: A *theorem* is a formula that has a *proof* (*proof* and *derivation* are amazingly similar concepts!)

Either way, a computable function is generated by a **finite devise** (whether a program or derivation).

In the *by-derivation approach* we start by accepting some set of **initial functions** $\mathcal{I}$ that are immediately recognizable as "intuitively computable", and choose a set $\mathcal{O}$ of *function-building operations* that preserve the "computable" property.

Compare: In the *by-proof approach* to discovering mathematical truth we start by accepting some set of **"initial truths"—the *axioms*** $\mathcal{I}$ that are immediately recognizable as "true", and choose a set $\mathcal{O}$ of *formula-building operations* that preserve truth.

2.3. The URM

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

The **alphabet** of the language is

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

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

[SS63] called the variables "registers".

1) These variables are restricted to be 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:

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

Nothing else names a variable of a URM except the names in (2) above.

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 such as X, A, B'', X_{13}.

2.3.1 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.

An if-statement is *syntactically illegal* (meaningless) if any of M or R *exceed* the label of the program's **stop** instruction. Also, zero is NOT a valid label.

► 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.

In particular, then, labels are *positive* integers.

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

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

The *semantics* of each command is given below.

2.3.2 Definition. (URM Instruction and Computation 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.

The semantic action of instructions of each type *is defined if and only if they are current*, and 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 $\mathbf{x}$ 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.

$\square$

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

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

.

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 (length) from $\vec{\mathbf{x}}_n$ and $\vec{a}_n$ if it is understood or we don't care, in which case we just write $\vec{\mathbf{x}}$ and $\vec{a}$.

2.3.3 Definition. (URM As an Input/Output Agent) 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*.

Aside. You have learnt in discrete MATH (a prerequisite of EECS2001) that A^n for any set A means

$$\overbrace{A \times \cdots \times A}^{n \textit{ copies of } A}$$

for $n > 0$, while $A^0 = \emptyset$ by definition.

Analogously, if A is the natural numbers set, $\mathbb{N}$, $\mathbb{N}^n$ is the set of length-n vectors with *components*[†] in $\mathbb{N}$ aka the set of length-n arrays with contents from $\mathbb{N}$.

[†]If $\vec{a}$ is a vector over the natural numbers, i.e., $\vec{a} \in \mathbb{N}$ for some $n > 0$, then an a_i is a *component* of said vector.

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 $\mathbb{N}^n$*,

(1) We —imagine we call an "I/O agent" to do it for us— *initialize the computation* of URM M, by doing two things:

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

$$a_1, \ldots, a_n$$

We also *initialize* **all other variables** of M to be 0.

This is an implicit **read action**.

(b) We next make the instruction labeled "1" *current*, and *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 $\mathbf{y}$ 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**. $\square$

2.3.4 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.* □

"Recursive" is just the invented terminology (Kleene) and it has nothing to do with procedures that call themselves.

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.

Sep. 14, 2022

BTW, you recall from MATH1019 that the symbol

$$f : \overset{\text{left field}}{\overset{\downarrow}{\mathbb{N}^n}} \rightarrow \overset{\text{right field}}{\overset{\downarrow}{\mathbb{N}}}$$

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 figure above!

Probably your 1019 text called $\mathbb{N}^n$ and $\mathbb{N}$ above "domain" and "range" (or, worse, "codomain"!). *FORGET THAT nomenclature!* What is the domain of f **really**? (in symbols $\text{dom}(f)$)

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

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

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

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

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

Nontotal iff $\mathrm{dom}(f) \subsetneqq \mathbb{N}^n$.

If $\vec{a}_n \in \mathrm{dom}(f)$ 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.

2.3.5 Example. Let M be the program

$$\begin{array}{l} 1: \mathbf{x} \leftarrow \mathbf{x}+1 \\ 2: \mathbf{stop} \end{array}$$

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. □

Given a URM M you might ask: "what is the function that M computes?"

This is an ambiguous fuzzy question. M computes as many functions as you can have choices of input and output variables. The focused question would sound like "What familiar function is $M_{\mathbf{y}}^{\vec{\mathbf{x}}}$?"

2.3.6 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).

Why do all this and not just do as in calculus (and in some sloppy discrete MATH courses) and say things like "let the function $f(x)$ be $x+1$"? Well, both *expressions* "$f(x)$" and $x+1$ are *function invocations* or *function calls*.[‡] A *function* (or *function procedure declaration*, in programming) or function definition has a *header* where the name of the function, the names of its input variables and the data type of the output are given. The header is followed by the *body* of the function that gives the algorithm that computes the output (returned value) according to the input values received.

A function *invocation* or *call* calls the defined function with appropriate inputs and returns some object —in our case a natural number. One is a number (call) the other a finite algorithm that defines a potentially infinite table of input-output pairs.

λ notation captures mathematically and abstractly the concept of a function definition (also called function *declaration* in Algol, Pascal and C).

□

[‡] $x + 1$ is a call to $\lambda z.z + 1$.

2.3.7 Example. Let M be the program

$$\begin{aligned} &1 : \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1 \\ &2 : \mathbf{stop} \end{aligned}$$

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.

2.3.8 Example. Let M be the program

$$\begin{aligned} 1 &: \mathbf{x} \leftarrow 0 \\ 2 &: \mathbf{stop} \end{aligned}$$

Then $M_{\mathbf{x}}^{\mathbf{x}}$ is the function $\lambda x.0$, the *zero function*. □

In Definition 2.3.4 we spoke of partial computable and total computable functions.

We retain the qualifiers partial *and* total *for all number-theoretic functions, 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 <u>*all*</u> functions 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*.

2.3.9 Example. The *unconditional* **goto** instruction, namely, "L : **goto** L'", can be simulated by L : **if** $\mathbf{x} = 0$ **goto** L' **else goto** L'.

□

2.3.10 Example. Let M be the program

$$\begin{aligned} &1 : \mathbf{x} \leftarrow 0 \\ &2 : \mathbf{goto}\ 1 \\ &3 : \mathbf{stop} \end{aligned}$$

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 $\emptyset \in \mathcal{P} - \mathcal{R}$.

Hence $\mathcal{R} \subsetneq \mathcal{P}$.

□

2.3.11 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 what we would have written, say, in FORTRAN as

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

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

They require that, upon execution of the instruction, the value of $\mathbf{z}$ is copied into $\mathbf{x}$ but the value of $\mathbf{z}$ remains unchanged. $\square$

2.3.12 Exercise. Write a program segment that simulates precisely the FORTRAN $L: \ \mathbf{x} \leftarrow \mathbf{z}$; 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 2.3.12 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 $\mathbf{x}'$ a read-only input variable:

- Add at the very beginning of M the instruction $1 : \mathbf{x} \leftarrow \mathbf{x}'$ of Exercise 2.3.12.
- 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.

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

2.3.13 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

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

while

$$g = G_{\mathbf{x}}^{\vec{\mathbf{z}}}$$

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, the references that if-statements of F make) in order to give $(G'F)$ the correct program structure,

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

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.* Also note that we could have called the modified "F" "F'" but we know what we mean when we write "$(G'F)_{\mathbf{u}}^{\vec{\mathbf{y}},\vec{\mathbf{z}}}$".

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

Thus, we have, by repeating the above idea a finite number of times:

2.3.14 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 2.3.14, that is:

2.3.15 Definition. 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*. □

We characterized the Definition 2.3.15 as "*rigid*".

Indeed, note that it requires *all* the arguments of f to be substituted by some $g_i(\vec{z})$—unlike Example 2.3.13, where we *substituted* a function invocation (cf. terminology in 2.3.6) *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, "$\ldots$", *must be the same*; in 2.3.15 it was $\vec{z}$.

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

We can rephrase 2.3.14, saying simply that

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

2.3.17 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 2.3.16.

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 $\mathcal{R}$*. $\square$

Sep. 19, 2022

Composing a number of times that *depends on the value of an input variable*—or as we may say, a *variable number of times*—is called *iteration*. The general case of iteration is called *primitive recursion*.

2.3.18 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:* 2.3.26).

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

$$f(1,\vec{y}) = g(0,\vec{y},\overbrace{h(\vec{y})}^{f(0,\vec{y})}),$$

$$f(2,\vec{y}) = g(1,\vec{y},\overbrace{g(0,\vec{y},h(\vec{y}))}^{f(1,\vec{y})}),$$

$$f(3,\vec{y}) = g(2,\vec{y},\overbrace{g(1,\vec{y},g(0,\vec{y},h(\vec{y})))}^{f(2,\vec{y})}), \text{ etc.}$$

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*).

With a little programming experience, it is easy to see that to compute $f(x, \vec{y})$ of 2.3.18 we can employ the pseudo code below:

$1 : z \leftarrow h(\vec{y})$
$2 :$ **for** $i = \quad 0$ **to** $x - 1$
$3 : z \leftarrow g(i, \vec{y}, z)$

At the end of the loop, z holds $f(x, \vec{y})$ —last value of i used in line 3 is $x - 1$.

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

2.3.19 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 and does not appear in H.

(iv) $\mathbf{x}$ does not occur in any of H or G.

We can now see that the following URM program, let us call it F, computes the f of Definition 2.3.18 for which we wrote the easy pseudo code on page 53 (we reproduce it here for convenience):

$1: z \leftarrow h(\vec{y})$
$2:$ **for** $i =$ 0 **to** $x - 1$
$3: z \leftarrow g(i, \vec{y}, z)$

In the URM below, $\boxed{H'}$ 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:

$\boxed{H'}^{\vec{\mathbf{y}}}_{\mathbf{z}}$
$r:$ $\mathbf{i} \leftarrow 0$
$r+1:$ **if** $\mathbf{x} = 0$ **goto** $k+m+2$ **else goto** $r+2$
$r+2:$ $\mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$
$r+3:$ $\boxed{G'}^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}_{\mathbf{z}}$
$k:$ $\mathbf{i} \leftarrow \mathbf{i} + 1$
$k+1:$ $\mathbf{w}_1 \leftarrow 0$
$\vdots$
$k+m:$ $\mathbf{w}_m \leftarrow 0$
$k+m+1:$ **goto** $r+1$
$k+m+2:$ **stop** /*$\mathbf{x} = 0$ and $\mathbf{i} =$ orig. $\mathbf{x}$; last $\mathbf{i}$-value *in* G' is $\mathbf{x} - 1$*/

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* initialized to zero *only* the *first* time G' is executed. Clearly, the URM F simulates the pseudo program above, thus $f = F^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$. $\square$

We just proved:

2.3.20 Proposition. *If f, g, h relate as in Definition 2.3.18 and h and g are in $\mathcal{P}$, then so is f. We say that $\mathcal{P}$ is* closed under primitive recursion.

2.3.21 Corollary. *If f, g, h relate as in Definition 2.3.18 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 we noted earlier (however proof is <u>later</u>, in 2.3.26) that if h and g is total, then so is f.

So, $f \in \mathcal{R}$. $\square$

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

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

OK. Fix an input $\vec{\mathbf{y}}$.

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

- Testing for $g(\mathbf{x}, \vec{\mathbf{y}}) \neq 0$ *never* got stuck as a result of 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, the k satisfies

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

Now, this k *depends* on $\vec{\mathbf{y}}$ so we may define it as a function f, for any INPUT $\vec{a}$ assigned into $\vec{\mathbf{y}}$, by:

$$k = f(\vec{a}) \stackrel{Def}{=} \min \Big\{ x : g(x, \vec{a}) = 0 \wedge (\forall y)\big(y < x \rightarrow g(y, \vec{a}) \downarrow\big) \Big\} \quad \square$$

Kleene has suggested the symbol "(μy)" to denote the "find the minimum y" operation above, thus the above is rephrased as

$$f(\vec{a}) = (\mu y) g(y, \vec{a}) \stackrel{Def}{=} \begin{cases} \min \Big\{ y : g(y, \vec{a}) = 0 \wedge (\forall w)_{w<y} g(w, \vec{a}) \downarrow \Big\} \\ \uparrow \text{ if the min above does not exist} \end{cases} \tag{2}$$

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

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

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

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

$\quad 2{:}\ \boxed{G'}_{\mathbf{z}}^{\mathbf{x},\vec{\mathbf{y}}}$

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

$k+1:$ $\quad\mathbf{w}_1 \leftarrow 0$ $\{$**Comment.** Setting *all non-input variables* to 0; cf. 2.3.19.$\}$

$\vdots$

$k+l:$ $\quad\mathbf{w}_l \leftarrow 0$ $\{$**Comment.** Setting *all non-input variables* to 0; cf. 2.3.19.$\}$

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

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

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

$\square$

We have at once:

2.3.23 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 $\lambda\vec{y}.(\mu x)g(x, \vec{y})$.

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

$$
\begin{array}{ll}
1: & \mathbf{w}_1 \leftarrow \mathbf{w}_1 + 1 \\
\vdots & \\
i: & \mathbf{z} \leftarrow \mathbf{w}_i \ \{\textbf{Comment.}\ \text{Cf. Exercise 2.3.12}\} \\
\vdots & \\
n: & \mathbf{w}_n \leftarrow 0 \\
n+1: & \mathbf{stop}
\end{array}
$$

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

□

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

2.3.25 Definition. Given $\lambda\vec{x}.f(\vec{x}_n)$ and $\lambda\vec{y}.g(\vec{y}_m)$.

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, to use a new symbol for the extended equality, namely $\simeq$.

Regardless, by way of this note we agree to use the same symbol for equality for **both** total and nontotal calls, namely, "=" (this convention is common in the literature, e.g., [Rog67]).

Let's do this for posterity:

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

Proof. Do $(\forall x)(\forall \vec{y}) f(x, \vec{y}) \downarrow$ by induction on x.

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

$$\text{"For all } x, \vec{y},\ f(x, \vec{y}) \downarrow\text{"} \tag{$*$}$$

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

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

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

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

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

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

Well, by $(\dagger)$ 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 $(\ddagger)$. $\square$

2.3.27 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 2.3.26, $prim(h, g)$ 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!

Chapter 3

Primitive Recursive Functions

Sep. 21, 2022

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 $\mathcal{P}$

Thus, not only are they "*intuitively computable*", but they are so **in a precise mathematical sense**:

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**.

3.1. Primitive recursive functions —the beginning

In this section we will explore the properties of the important set of functions known as **primitive recursive**.

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

3.1.1 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.
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 the sequence.

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**.*

That is,

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

[*]**Recall**: That is, *left field* is $\mathbb{N}^n$ for some $n > 0$, and *right field* is $\mathbb{N}$.

[†]Strictly speaking, *primitive recursively derived*, but we will not considered other sets of derived functions, so we omit the qualification.

The above (3.1.1) defines essentially what Dedekind ([Ded88]) called "*recursive*" functions. In plain English: "Each such function is obtained from the initial functions by a finite number of applications of primitive recursion and composition".

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

3.1.2 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 3.1.1 in the standalone derivations (1) and (2). But this property of the f_i and g_j *is preserved* after concatenation. □

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

3.1.4 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 $1 \le m \le k$, satisfies 1.–3. of Definition 3.1.1 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 $1 \le m \le k$. □

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

Proof.

(a) The *If* (appears). Say $g_1, \ldots, g_n, \boxed{f}$ is a derivation. Since f occurs in it, $f \in \mathcal{PR}$ by 3.1.1.

(b) The *Only If* (appears). Say $f \in \mathcal{PR}$. Then, by 3.1.1,

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

for some derivation like the (1) above.

By 3.1.4, $g_1, \ldots, g_m, \boxed{f}$ is also a derivation. □

3.1.6 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 $\mathcal{PR}$. 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 3.1.2.

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

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 3.1.1.

This implies $prim(h,g) \in \mathcal{PR}$ by 3.1.1.

- Closure under **composition**. So let $\lambda\vec{y}.h(\vec{x}_n)$ and $\lambda\vec{y}.g_i(\vec{y})$, for $1 \le i \le n$, be in $\mathcal{PR}$. By 3.1.1 we have derivations

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

and

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

By 3.1.2,

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

is a derivation, and by 3.1.1, 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 3.1.1. $\square$

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

Answer. By building a *derivation*

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

(**Analogy**: Just like showing a *formula* is a *theorem*: You build a *proof!*)

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 3.1.6.

How do you prove that $\underline{ALL\ f \in \mathcal{PR} \text{ have a property } Q}$ —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.

3.1.8 Example. **(1)** To demonstrate the first Answer above (3.1.7), show (prove) that $\lambda xy.x + y \in \mathcal{PR}$. Well, observe that

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

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

$$G(x, y, \overbrace{x+y}^{\text{“recursive” call}})$$

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

- $f(x,y) = x + y$ and $f(0,y) = y = U_1^1(y)$ and $f(x+1,y) = G(x,y,f(x,y))$

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

- The functions $H = U_1^1$ (*initial*) and $G = SU_3^3$ (*composition*) are in $\mathcal{PR}$. By 3.1.6 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 (3.1.7), show (prove) that every $f \in \mathcal{PR}$ is **total**. Induction on the length n of a derivation 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 3.1.5— 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 2.3.26.

- $f = \lambda \vec{y}.h\Big(q_1(\vec{y}), \ldots, q_t(\vec{y})\Big)$, 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 proof of 2.3.17. □

[‡]By the chop the tail theorem.

Sep. 26, 2022

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

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

Simulate it with COMPOSITION!

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 class.

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 $2 = SSZ(y)$. Now use the above.

(2) $\lambda xyw.f(y,x,w)$ 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, $\lambda xy.x \dot{-} y$ asks that we subtract the second input (y) from the first (x), but $\lambda xy.y \dot{-} x$ 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:

3.1.10 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 3.1.10 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.

3.1.11 Example. The definition of primitive recursion is also rigid. *However this is also an illusion.*

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 3.1.6.

- $p = \lambda x.\widetilde{p}(x, 0)$ is in $\mathcal{PR}$ since $\widetilde{p}$ is, then invoking 3.1.10.

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 $x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 = p(x \dot{-} y)$

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, 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)], we can get sub:

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

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

3.1.12 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. $\square$

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

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

is primitive recursive.

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

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

Hint. Do induction on derivation length to show if $f \in \mathcal{PR}$ then $f \in \mathcal{R}$. $\square$

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

3.1.15 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 $x = 0, y > 0$.

BUT x^y is "mathematically" undefined when $x = y = 0$. What do we do?.[§]

Thus, by Example 3.1.8, item 2, *the exponential cannot be a primitive recursive function*!

This is rather *silly*, since the *computational process for the exponential is extremely easy*; thus it is *ridiculous* to declare the function 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 "$0^0 = 1$"*.

We already did this redefinition in equation one setting $x^0 = 1$ for any x.

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

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

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

BUT also examples where we CANNOT do this (LATER!); and WHY. □

3.1.16 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: We just re-coded the outputs* **t** *and* **f** *to 0 and 1 respectively!*.

The concept of relation *significantly simplifies* the further development and exposition of the theory of primitive recursive functions.

The following is useful:

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

Proof. *For the* if-*part*, *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. 3.1.11).

For the only if-*part*, taking $f = c_R$ will do. □

3.1.18 Corollary. *$R(\vec{x}) \in \mathcal{R}_*$ 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 3.1.14. □

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

Proof. By the above corollary and 3.1.14. □

Sep. 28, 2022

3.1.20 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 (3.1.16), $c_R \in \mathcal{PR}$. 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

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

which is the same as

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

and therefore is in $\mathcal{PR}$. □

3.1.21 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 3.1.12.* □

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

Proof. As above, mindful of 3.1.14. □

3.1.23 Example. The relations $x \le y$, $x < y$, $x = y$ 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 $x \le y \equiv x \dot{-} y = 0$ and invoke 3.1.17. Finally invoke Boolean closure and note that $x < y \equiv \neg y \le x$ while $x = y$ is equivalent to $x \le y \wedge y \le x$.

Or, directly: $x = y \equiv |x - y| = 0$; Note that $|x - y| = x \dot{-} y + y \dot{-} x$. □

Chapter 4

$\mathcal{PR}$: Basic Properties Part II

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. By Proposition 3.1.17, 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

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

By 3.1.17, and 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 $R(\vec{x}, f(\vec{w}), \vec{z}) \in \mathcal{PR}_*$. $\square$

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. $\square$

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. $\square$

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

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

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

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?).
Moreover, logic says:

$Q(z + 1, \vec{x}) \equiv Q(z, \vec{x}) \vee R(z, \vec{x})$.

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

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

$\square$

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

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" <u>means</u> "**if** $c_R(\vec{z}) = 0$ **then** y **else** w".*

$$\text{Note that } 0,1,2,\ldots,z-1,z = \overbrace{0,1,2,\ldots,z-1}, z$$

So

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

Why 0 above?

$$\begin{aligned} g(z+1,\vec{x}) = \text{if } & \overbrace{(\exists y)_{<z}\Big(f(y,\vec{x}) = 0\Big)}^{\textit{name it } Q(z,\vec{x})} \text{ then } g(z,\vec{x}) \\ & \text{else if } f(z,\vec{x}) = 0 \text{ then } z \\ & \text{else } z+1 \end{aligned}$$

□

The *iterator* above (or "G-part") is

$$\begin{aligned} G(z,\vec{x},w) = \text{if } & \overbrace{Q(z,\vec{x})}^{\textit{same as } c_Q(z,\vec{x})=0} \text{ then } \overbrace{w}^{\textit{rec. call here!}} \\ & \text{else if } f(z,\vec{x}) = 0 \text{ then } z \\ & \text{else } z+1 \end{aligned}$$

□

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

4.1.11 Exercise. *Prove the corollary.* □

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

Oct. 3, 2022

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 $\lambda xy.x^y$).

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

is *extended* to *mean*

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

for all x, y.

▶ Pause. **Why** is the above expression correct?

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

[*]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**, $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$.

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

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

Thus, z is *smallest* such that $x/y < z+1$, or such that $x < y(z+1)$. BTW, I have <u>no division</u> in "$x < y(z+1)$" to bother me! ◀

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, that $0|0$, but we can live with this in practice.

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

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

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

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

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].

(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,

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

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

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 us to search long enough. Too small a bound might obstruct a full search. In short, if b is a good bound then if a solution exists it will be found among the numbers $0, 1, 2, \ldots, b$.

(8) *$Seq(x)$ (x's prime number factorisation contains at least one prime, but no gaps).*

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

$\square$

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

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?). $\square$

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

4.2. CODING Sequences

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

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

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

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

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* $(\mu y)_{\le z} \neg(p_y|z)$

▶ *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.*

- **Question**: Is the bound z sufficient? **Yes!**

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

(*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 $lh(10) = 1$.

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

Oct. 5, 2022

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 $x, \vec{y}$.

$$\begin{cases} f_0(0,\vec{y}) & = h_0(\vec{y}) \\ \vdots & \\ f_k(0,\vec{y}) & = h_k(\vec{y}) \\ f_0(x+1,\vec{y}) & = g_0(x,\vec{y},f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})) \\ \vdots & \\ f_k(x+1,\vec{y}) & = g_k(x,\vec{y},f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})) \end{cases} \tag{2}$$

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. Define, for all $x, \vec{y}$,

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

$$H(\vec{y}) \stackrel{\text{Def}}{=} \langle h_0(\vec{y}), \ldots, h_k(\vec{y})\rangle$$

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

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.120) as

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

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

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

So, for all $x, \vec{y}, w$,

$$G(x,\vec{y},w) = \Big\langle g_0\Big(x,\vec{y},\big(w\big)_0,\ldots,\big(w\big)_k\Big), \ldots, g_k\Big(\text{same as } g_0\Big)\Big\rangle$$

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 $x, \vec{y}$, $f_i(x, \vec{y}) = \big(F(x, \vec{y})\big)_i$. $\square$

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"

$$\begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 & \ldots \\ 1 & 0 & 1 & 0 & 1 & 0 & \ldots \end{matrix}$$

Let us call g the function that has as its sequence 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

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

$\square$

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

$$\begin{array}{ccccccccc} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & \ldots \\ 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & \ldots \end{array}$$

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

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

$\square$

Chapter 5

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 <u>definitional</u> (or <u>structural</u>) <u>complexity</u> of primitive recursive functions with their (<u>run time</u>) <u>computational</u> complexity.

5.1. Preliminaries

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*).

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.

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

5.1.1 Definition.

- 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:

$$\begin{array}{c} P \\ Q \end{array}$$

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:

$$\begin{array}{l}\textbf{Loop } X\\ \quad P\\ \textbf{end}\end{array}$$

$\square$

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

The informal semantics of loop programs are precisely those given in [Tou12].

They are almost identical to the semantics of the URM programs.

5.2. Semantics of Loop Programs

5.2.1 Definition. (Semantics)

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 (if any) is the next CURRENT instruction.*

3. When the instruction

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

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

   ► We view the **Loop-end** construct as an "instruction" just as a **begin-end** block is in, say, Pascal or C. ◄

   (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

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

$\square$

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

5.3. Loop Programs as (Computable) Functions

5.3.1 Definition. *The symbol $P_Y^{\vec{X}_n}$ has* exactly the same meaning as for the URMs*, 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. As in URMs, an "agent" that *is NOT involved in the computation* initialises the input variables, reads the output from Y when the program ends (they all end) and also intialises all non-input variables to zero (0). □

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 3 of 5.2.1, such a program terminates.

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

I.S.

- What about the program

$$\begin{array}{c} P \\ Q \end{array}$$

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

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

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!

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

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

5.3.1. "Programming" Examples

Refer to the Examples 4.2.3 and 4.2.4, 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:

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

We translate the above recursion easily to

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 behind 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.

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

Oct. 17, 2022

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

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

"f is loop programmable".

Basis (Derivation length $n = 1$; Initial Functions of $\mathcal{PR}$):

$\lambda x.x + 1$ 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.

I.H. Assume claim for derivation length $n \le k$.

I.S. Prove for $n = k + 1$. So let

$$f_1, f_2, \ldots, f_k, f$$

be a derivation of f.

Cases:

1. f is initial. This has already been argued.
2. f is the result of **Grzegorczyk substitution**[†] using two of the f_i, say, f_m and f_t where $f = \lambda \vec{x}\vec{z}\vec{y}.f_t(\vec{x}, f_m(\vec{z}), \vec{y})$.

   By the I.H. f_m, f_t are *loop programmable*, say $f_m = M^{\vec{\mathbf{z}}}_{\mathbf{w}}$ and $f_t = T^{\vec{\mathbf{x}}\mathbf{w}\vec{\mathbf{y}}}_{\mathbf{u}}$ where the loop programs T and M, **wlg**, have only the variable $\mathbf{w}$ common.

   Then f is

$$\begin{pmatrix} M \\ T \end{pmatrix}^{\vec{\mathbf{x}}\vec{\mathbf{z}}\vec{\mathbf{y}}}_{\mathbf{u}}$$

[†]We have been using substitution for a while as an alternative to composition.

3. $f = prim(h, g)$ where h and g are among the f_i. So let $h = H_Z^{\vec{Y}}$ *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

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

   Program the above as follows:

   The pseudo-code is

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

   See the similar more complicated programming for URMs to recall precautions needed to avoid side-effects. For example, I must be read-only in the G-program and $\vec{Y}_n$ must be read only in both H and G. X does not occur in G or H —just ensures going round the loop a times, where a is the original value of X. The last value of i used in the blue line is $a - 1$. □

5.5. $\mathcal{L} \subseteq \mathcal{PR}$

To handle the converse of 5.4.1 we will simulate the computation of any loop program P by an array of primitive recursive functions.

5.5.1 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* <u>all</u> *the variables that occur in P.* $\square$

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

Proof. We do induction on the way loop-programs are built:

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

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

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 $\Big(Q; R\Big)_{z_i}$?

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

  $f \in \mathcal{PR}$ 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 $\Big(Q; R\Big)_{u_j}$.

(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.

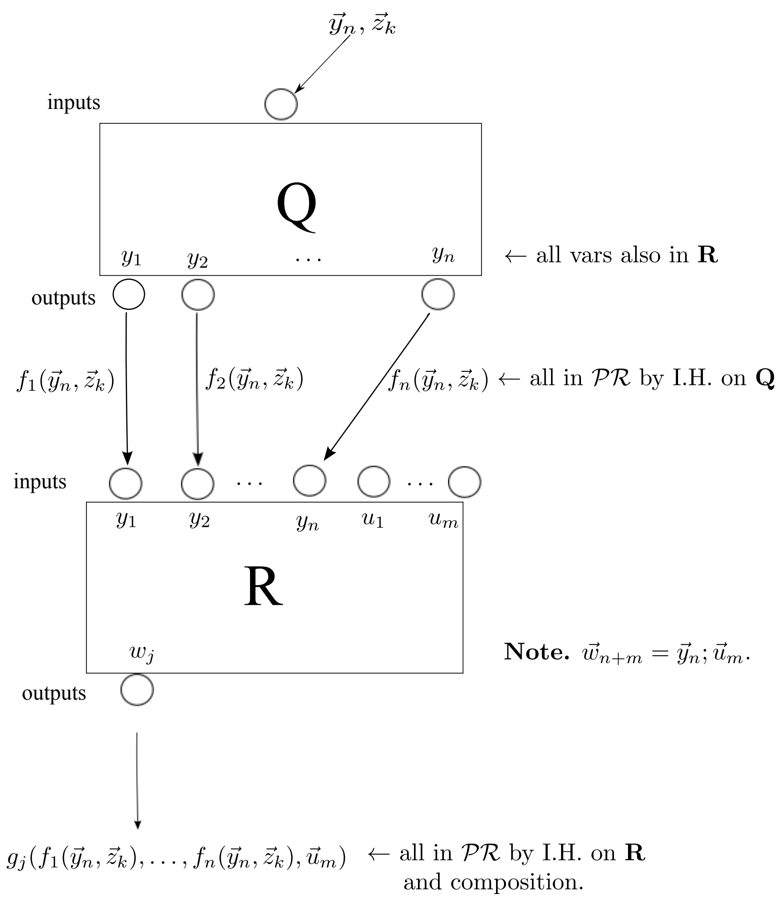
► Thus the set of variables of $\left(Q;R\right)$ is $\vec{y}_n\vec{z}_k\vec{u}_m$

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) because 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. In the figure of this page we utilise this notation:

$$f_i = Q_{y_i} = Q_{y_i}^{\vec{y}_n\vec{z}_k} \text{ and } g_j = R_{w_j} = R_{w_j}^{\vec{y}_n\vec{u}_m}$$



(C) P is

$$\begin{array}{l}\textbf{Loop } X\\ Q\\ \textbf{end}\end{array}$$

NOTATION: Let

$$g_j \stackrel{Def}{=} Q_{Y_j} = Q_{Y_j}^{\vec{Y}_n} \tag{1}$$

Thus

$$Y_j \text{ holds}^\dagger\ g_j(\vec{y}_n) \text{ at the end of the } Q\text{-computation} \tag{1'}$$

if y_m **is the input value in** Y_m**, for** $m = 1, 2, \ldots, n$**.**

Similarly, define

$$f_k \stackrel{Def}{=} P_{Y_k} = P_{Y_k}^{X\vec{Y}_n},\ k \geq 1, \text{ and } f_{\mathbf{0}} \stackrel{Def}{=} P_X \stackrel{Def}{=} P_{Y_0} = P_X^{X\vec{Y}_n} \tag{2}$$

using also the name Y_0 as an alternative to the name X.

Thus

$$X \text{ and } Y_t,\ t \geq 1, \text{ store } f_X(a, \vec{y}_n) \text{ and } f_t(a, \vec{y}_n) \text{ respectively,} \tag{2'}$$

at the end of the P-computation, **if** a **is the input value in** X**, i.e., in** Y_0**, and** y_m **is the input value in** Y_m**, for** $m = 1, 2, \ldots, n$**.**

By the I.H. all g_j are in $\mathcal{PR}$.

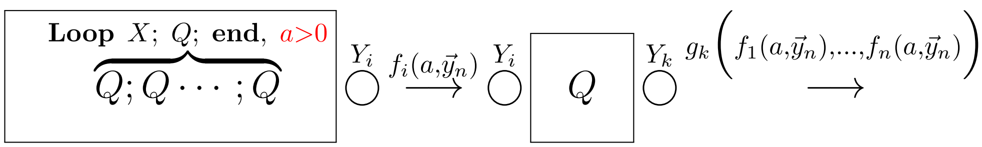
$$\boxed{\text{We will prove that } f_X\ (= f_0) \text{ and all } f_t, t \geq 1, \text{ are also in } \mathcal{PR}.}$$

[†]Meaning “stores”, “contains”.

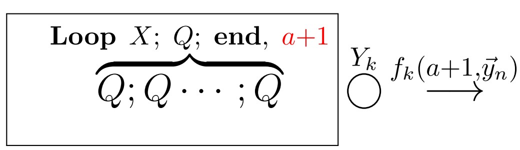
There are two subcases: X *is* in Q; OR X is *not* in Q.

(a) X is *not* in Q: Using the notation from (1), (1′), (2), and (2′), we show pictorially below —for $a > 0$— the dependency between $f_i(a+1, \vec{y}_n)$ and $f_m(a, \vec{y}_n)$, for $1 \le i, m \le n$.

$\vec{y}_n$ is an invariant "parameter" (as in (simultaneous) primitive recursion). To avoid cluttering the figure we **only show the output Y_i from the left box and Y_k from the right box, and don't show the Y_j with $j \neq i$ and $j \neq k$.**



By the definition of the **Loop** X-semantics (5.2.1), the above is the same as



Therefore,

$$f_k(a+1, \vec{y}_n) = g_k\Big(f_1(a, \vec{y}_n), \ldots, f_n(a, \vec{y}_n)\Big), \text{ for } k = 1, \ldots, n$$

and, for the basis where $a = 0$ (*loop skipped*),

$$f_k(0, \vec{y}_n) = y_i, \text{ for } k = 1, \ldots, n$$

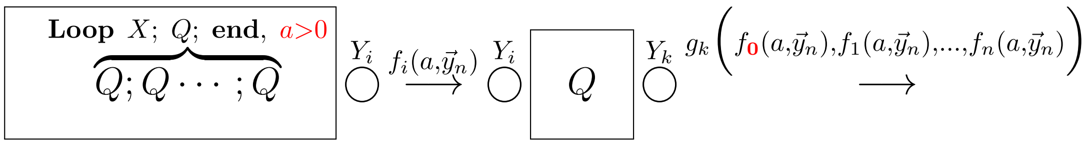
Moreover,

$f_X(a, \vec{y}_n) = a$, for all a since X —i.e., Y_0— is not changed by P

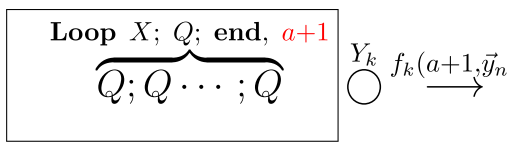
Thus $f_X = U_1^{n+1} \in \mathcal{PR}$ <u>and</u> the $f_k, k \geq 1$, are in $\mathcal{PR}$, the latter by closure under simultaneous primitive recursion.

(b) *X is* in Q:

So, let $X, \vec{Y}_n$ be all the variables of Q. Recall that X has the alias Y_0. The two figures above apply with trivial modifications to allow the presence of X (Y_0) in Q: See below.



By the definition of the **Loop** X-semantics (5.2.1), the above is the same as



Therefore,

$$f_k(a+1, \vec{y}_n) = g_k\Big(f_0(a, \vec{y}_n), f_1(a, \vec{y}_n), \ldots, f_n(a, \vec{y}_n)\Big), \text{ for } k = \mathbf{0}, 1, \ldots, n$$

and, for the basis where $a = 0$ (loop skipped),

$$f_k(0, \vec{y}_n) = y_i, \text{ for } k = \mathbf{0}, 1, \ldots, n$$

This concludes Case (b).

At the end of all this we have that, when P is a loop-closure, then $P_Z \in \mathcal{PR}$ for all Z in P. This concludes the Induction over L and also the proof of the Lemma. $\square$

We can now prove

5.5.3 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

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

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

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

and

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

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

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

5.6. 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

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

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 "$P; X; Y$").

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 $\lambda x.f_i(x)$

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

5.6.1. A Universal function for unary $\mathcal{PR}$ functions

Oct. 19, 2022

At the end of all this we got a *universal* or *enumerating* function $U^{(PR)}$ for *all* the unary functions functions in $\mathcal{PR}$.

That is the function of TWO arguments

$$U^{(PR)} = \lambda ix.f_i(x) \tag{2}$$

$$\underbrace{U^{(PR)}(\overset{prog}{i}, \overset{data}{x})}_{\text{programmable computer}} = f_i(x)$$

What do I mean by "Universal"?

5.6.1 Definition. $U^{(PR)}$ of (2) is *universal* or *enumerating* for all the unary functions of $\mathcal{PR}$ meaning it has two properties:

1. If $g \in \mathcal{PR}$ is unary, then there is an i such that

$$g = \lambda x.U^{(PR)}(i, x)$$

and

2. Conversely, for every $i \in \mathbb{N}$, $\lambda x.U^{(PR)}(i, x) \in \mathcal{PR}$. □

5.6.2 Theorem. *The function of two variables, $\lambda ix.U^{(PR)}(i,x)$ is computable informally.*

Proof. Here is how to calculate $U^{(PR)}(i,x)$ for each given i and a:

1. *Find the i-th P_Y^X in the enumeration (1) that we have built in (D) above. That is, the f_i in $List_3$.*

   This does NOT mean we HAVE an infinite List sitting there:

   It means: *build* $List_1$ and *simultaneously* the lists $List_2$ and $List_3$ and stop once you got the i-th element of the last List enumerated.

2. Now, run the P_Y^X you just found with input a into X. This terminates!

   After termination Y holds $f_i(a) = U^{(PR)}(i,a)$. □

Important. *We* repeat *for posterity TWO by-products of 5.6.1 and 5.6.2:*

- The informally computable *Enumeration function* $U^{(PR)}$ is total.
- $\lambda x.U^{(PR)}(i,x) = f_i$ for all i.

5.6.3 Theorem. *$U^{(PR)}$ is NOT primitive recursive.*

Proof. If it is, *then so is $\lambda x.U^{(PR)}(x,x)+1$* by Grzegorczyk operations. As this is a unary $\mathcal{PR}$ function, we must have an i such that

$$\overbrace{U^{(PR)}(x,x)+1}^{f_i(x),\ for\ some\ x} = U^{(PR)}(i,x), \text{ for all } x \tag{3}$$

Setting i into x in (3) we get the contradiction

$$U^{(PR)}(i,i)+1 = U^{(PR)}(i,i)$$ □

5.6.4 Remark. Thus $\lambda ix.U^{(PR)}(i,x)$ acts as the *COMPILER* of a *stored program computer*:

You give it a (pointer to a) **PROGRAM** i and *DATA* x and it *simulates* the **Program** (at address) i on the **Data** x!

We have just learnt in the above theorem that this compiler **CANNOT be programmed in the Loop-Programs Programming Language**! □

Chapter 6

A user-friendly Introduction to (un)Computability and Unprovability via "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). Its advent was strongly motivated, in the 1930s, by Hilbert's program, in particular by his belief that the *Entscheidungsproblem*, or *decision problem*, for axiomatic theories, that is, the problem "Is this formula a theorem of that theory?" was solvable by a mechanical procedure that was yet to be discovered.

Now, since antiquity, mathematicians have invented "mechanical procedures", e.g., Euclid's algorithm for the "greatest common divisor",[†] and had no problem recognising such procedures when they encountered them. But how do you mathematically *prove* the *nonexistence* of such a mechanical procedure for a particular problem? You need a *mathematical formulation* of what *is* a "mechanical procedure" in order to do that!

Intensive activity by many (Post [Pos36, Pos44], Kleene [Kle43],

[†]That is, the largest positive integer that is a common divisor of two given integers.

Church [Chu36b], Turing [Tur37], Markov [Mar60]) led in the 1930s to several alternative 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* calculable function is also calculable within any of these mathematical frameworks of calculability or computability.[†]

By the way, Church proved ([Chu36a, Chu36b]) that Hilbert's *Entscheidungsproblem* admits no solution by functions that are calculable within any of the known mathematical frameworks of computability. Thus, if we accept his "thesis", the Entscheidungsproblem admits no algorithmic solution, period!

The eventual introduction of computers further fueled the study of and research on the various mathematical frameworks of computation, "models of computation" as we often say, and "computability" is nowadays a vibrant and very extensive field.

[†]I stress that even if this sounds like a "completeness *theorem*" in the realm of computability, it is not. It is just an empirical belief, rather than a provable result. For example, Péter [Pé67] and Kalmár [Kal57], have argued that it is conceivable that the intuitive concept of calculability may in the future be extended so much as to transcend the power of the various mathematical models of computation that we currently know.

6.1. A leap of faith: Church's Thesis

The aim of Computability is to *mathematically capture* (for example, via URMs) the *informal* notions of "algorithm" and "computable function" (or "computable relation").

A lot of models of computation, that were very different in their syntactic details and semantics, have been proposed in the 1930s by many people (Post, Church, Kleene, Turing) and more recently by Shepherdson and Sturgis ([SS63]). They were all *proved to compute exactly the same number theoretic functions*—those in the set $\mathcal{P}$. The various models, and the gory details of why they all do the same job precisely, can be found in [Tou84].

This prompted Church to state his *belief*, famously known as "*Church's Thesis*", that

> Every *informal* algorithm (pseudo-program) that we propose for the computation of a function can be implemented (*made mathematically precise*, in other words) in each of the known models of computation. In particular, **it can be "programmed" as a URM**.

We note that at the present state of our understanding the concept of "algorithm" or "algorithmic process", ***there is no known way*** to define an "intuitively computable" function—via a pseudo-program of sorts—***which is outside of*** $\mathcal{P}$.†

Thus, as far as we know, $\mathcal{P}$ appears to *formalise* the ***largest***—i.e., most inclusive—set of "intuitively computable" functions known.

This "empirical" evidence supports Church's Thesis.

Church's Thesis —acronym CT— is not a theorem. It can never be, as it "connects" precise mathematical objects (URM, $\mathcal{P}$) with imprecise *informal* ones ("algorithm", "computable function").

†In the so-called relativised computability (with partial oracles) Church's Thesis fails [Tou86].

It is simply a belief that has overwhelming empirical backing, and should be only read as an ***encouragement to present algorithms in "pseudo-code"—that is, informally.***

In the literature, Rogers ([Rog67], a very advanced book) heavily relies on CT. On the other hand, [Dav58, Tou84, Tou12] never use CT, and give all the necessary constructions (implementations) in their full gory details —*that is the price to pay, if you avoid CT.*

Here is the template of **how** to use CT:

- We **completely** present —that is, no essential detail is missing— an algorithm in *pseudo-code*.
  ►BTW, "pseudo-code" does not mean "sloppy-code"!◄
- 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. The Universal and S-m-n Theorems

We note that

> Exactly the same technique (used for unary $\mathcal{PR}$ functions in the previous chapter) —building three algorithmically generated Lists— works if we apply it to all $M_{\mathbf{y}}^{\mathbf{x}}$ where M runs over all URMs.
>
> That is, List$_3$ enumerates all possible unary functions of $\mathcal{P}$ as $M_{\mathbf{y}}^{\mathbf{x}}$. We can indicate this listing of unary functions $M_{\mathbf{y}}^{\mathbf{x}}$ as
>
> $$\phi_0, \phi_1, \phi_2, \ldots, \phi_i, \ldots$$

Correspondingly, we have a *universal function* $\lambda ix.\phi_i(x)$ for $\mathcal{P}$ unary functions that we will denote by "h" —that is

$$h \stackrel{Def}{=} \lambda ix.\phi_i(x)$$

Since every $\lambda x.f(x) \in \mathcal{P}$ is an $M_{\mathbf{z}}^{\mathbf{x}}$, that is, a ϕ_i we have that

> Given a unary f in $\mathcal{P}$. Then, for some i, $h(i, x) = f(x)$, for all x.

Kleene's "*universal function theorem*" states that $h \in \mathcal{P}$.

6.2.1 Theorem. (Universal function theorem) *The universal function is computable.*

Proof. By CT:

Here is how the universal h is computed *in pseudo code*

- Given input i and x.
- Generate the listing of the $N_{\mathbf{z}}^{\mathbf{w}}$ long enough and stop as soon as the *i-th entry* was generated. Say, this entry is $M_{\mathbf{y}}^{\mathbf{x}}$.
- Now run program M with x inputed into the input program-variable $\mathbf{x}$. If and when M stops, then we return the value held in the program-variable $\mathbf{y}$ of M.

By CT, the three-bullet algorithm (pseudo-program) above can be implemented as a URM. *So h is partial computable.* □

Hmm. Can we *not* imitate the proof that $U^{(\mathcal{PR})}$ is *not primitive* recursive to show that $\lambda xy.h(x,y)$ is not partial recursive?

We cannot!

Suppose we went like this:

OK. *If* $\lambda xy.h(x,y)$ is in $\mathcal{P}$ (as we argued by CT)[†] then so is $\lambda x.h(x,x)+1$ by Grzegorczyk substitution. As h is universal, for some i and all x we have $h(i,x) = h(x,x)+1$ and specifically

$$h(i,i) = h(i,i) + 1 \tag{1}$$

A contradiction, *right*?

Nope. We cannot be sure that the two sides of (1) are *necessarily* defined. If undefined then (1) is true. No contradiction!

[†]WHAT?! Now you doubt Mr. Church?!

The notation "$\phi_i(x)$" is due to Rogers ([Rog67]).

Calling x the "program" for $\lambda y.\phi_x(y)$ is not exact, but **is eminently apt**: x is just a number, not a set of URM instructions; but this number is the *address* (location) of *a URM program for* $\lambda y.\phi_x(y)$. Given the address, we **can retrieve** the program from a list via a computational procedure, in a finite number of steps!

In the literature the address x in ϕ_x is called a ϕ-index. So, if $f = \phi_i$ then i is one of the infinitely many addresses where we can find how to program f.

Oct. 24, 2022

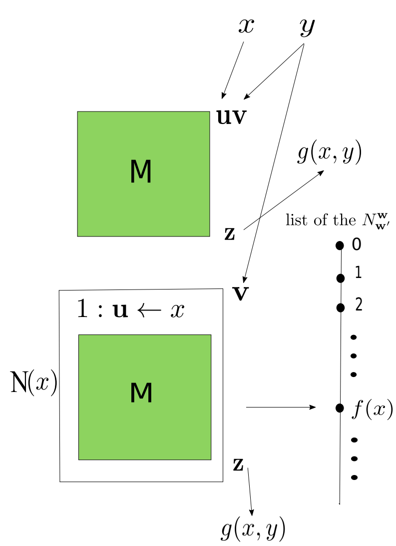
Another fundamental theorem in computability is the *Parametrisation* or *Iteration* or also "*S-m-n*" theorem of Kleene.

6.2.2 Theorem. (Parametrisation theorem) *For every $\lambda xy.g(x, y) \in \mathcal{P}$ there is a function $\lambda x.f(x) \in \mathcal{R}$ such that*

$$g(x, y) = \phi_{f(x)}(y), \textit{ for all } x, y \tag{1}$$

This says that given a program M that computes the function g as $M_{\mathbf{z}}^{\mathbf{uv}}$ with $\mathbf{u}$ receiving the input value x and $\mathbf{v}$ receiving the input value y, we can, for **any** fixed value x, *construct* a new program **located** in position $f(x)$ of *the algorithmic enumeration* of *all* $N_{\mathbf{w}'}^{\mathbf{w}}$ —the construction (of this address) effected by the *total* computable function f. The program at address $f(x)$ "knows" the value x, it is "hardwired" in its instructions, thus it does not receive the value x as a "read" *input*.

This hardwiring is effected by **adding** to program M a new first instruction, namely, $1 : \mathbf{u} \leftarrow x$. The original first instruction of M is now the 2nd of the modified program. Indeed all instructions of M are pushed down (their addresses increase by 1).



So the new program at location $f(x)$ of the listing, and the original program for $g = M_{\mathbf{z}}^{\mathbf{uv}}$ yield the same answer for the arbitrary fixed x, and all input values y "read" into the variable $\mathbf{v}$, as long as the the variable $\mathbf{u}$ gets the same value x in both programs.

Proof. Of the S-m-n theorem. The proof is encapsulated by the preceding figure.

It is clear that

1. We can construct program $N(x)^{\mathbf{v}}_{\mathbf{u}}$ given x and program $M^{\mathbf{uv}}_{\mathbf{u}}$.
2. We call its location in "List$_3$" "$f(x)$" to indicate dependency on x.

All that remains to argue is that this address, $\lambda x.f(x)$ is **total computable**. Well,

- Given $M^{\mathbf{uv}}_{\mathbf{z}}$.
- Given x.
- build $N(x)$ from M as indicated in the figure above.
- Go down the list of **<u>all</u>** $N^{\mathbf{w}}_{\mathbf{w}'}$ and keep comparing, until you find $N(x)^{\mathbf{v}}_{\mathbf{z}}$.
- Output the location, $f(x)$, of $N(x)^{\mathbf{v}}_{\mathbf{z}}$. You **WILL** find said location due to the underlined "all" above. So f is total.

By CT all informal computations here (building $N(x)$ from M and the process for finding $f(x)$ for the given x) can be done by URMs. Thus, $f \in \mathcal{R}$. □

6.3. Unsolvable "Problems" The Halting Problem

Some of the comments below (and Definition 6.3.1) occurred already in earlier posted Notes. We revisit and introduce some additional terminology (e.g., "decidable").

A number-theoretic *relation* is some ***set of*** n***-tuples*** —$n \geq 1$— from $\mathbb{N}$. A relation's outputs are **t** or **f** (or "yes" and "no"). However, a number-theoretic relation *must* have values ("outputs") also in $\mathbb{N}$.

Thus we *re-code* **t** and **f** as 0 and 1 respectively. This convention is preferred by Recursion Theorists (as people who do research in Computability like to call themselves) and is the opposite of the re-coding that, say, the C language employs (0 for **f** and non-zero for **t**).

6.3.1 Definition. (Computable or Decidable relations) "*A relation $Q(\vec{x}_n)$ is **computable**, or **decidable***" or "***solvable***" means that the 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 $\mathcal{R}_*$. Computable relations are also called *recursive*.

By the way, we call the function $\lambda \vec{x}_n . c_Q(\vec{x}_n)$ —which does the recoding of the outputs— the *characteristic function* of the relation Q ("c" for "characteristic"). □

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 the relation has a *decider*, i.e., the URM that *decides* membership of *any* tuple $\vec{x}_n$ in the relation.

6.3.2 Definition. (Problems) A "***Problem***" is 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***. □

6.3.3 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:

$$\phi_x(x) \downarrow \tag{1}$$

A different formulation uses the set

$$K = \{x : \phi_x(x) \downarrow\}^{\dagger} \tag{2}$$

that is, *the set of all numbers x, such that machine M_x on input x has a (halting!) computation.*

K we shall call the "**halting set**", and (1) we shall the "**halting problem**".

Clearly, (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!

6.3.4 Theorem. *The halting problem is unsolvable.*

Proof. We show, **by contradiction**, that $K \notin \mathcal{R}_*$.

Thus we start by *assuming the opposite*.

$$\text{Let } K \in \mathcal{R}_* \tag{3}$$

that is, 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 "$\uparrow$" for any undefined entry $\phi_x(y)$.

By 6.2.1 and the comments following it, **each** one argument function of $\mathcal{P}$ are in some row (as an array of outputs).

$$\begin{array}{cccccc}
\phi_0(0) & \phi_0(1) & \phi_0(2) & \ldots & \phi_0(i) & \ldots \\
\phi_1(0) & \phi_1(1) & \phi_1(2) & \ldots & \phi_1(i) & \ldots \\
\phi_2(0) & \phi_2(1) & \phi_2(2) & \ldots & \phi_2(i) & \ldots \\
\vdots & & & & & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \ldots & \phi_i(i) & \ldots \\
\vdots & & & & &
\end{array}$$

We will show that under the assumption (3) that we hope to contradict the flipped diagonal —flipping all $\uparrow$ red entries to $\downarrow$ and vice versa; (3) says we can tell via a URM *decider* whether $\phi_x(x)\downarrow$ or not— represents a *partial recursive function* and hence **must** fit the matrix along some row i since we have all ϕ_i captured in the matrix.

On the other hand, after flipping the diagonal the *modified-diagonal function* constructed, namely,

$$\overline{\phi_0(0)}, \overline{\phi_1(1)}, \overline{\phi_2(2)}, \ldots, \overline{\phi_i(i)}, \ldots$$

cannot fit.

We say we performed a "diagonalisation".

In more detail, or as most texts present this, we have defined the *flipped diagonal* for all x as

$$d(x) = \begin{cases} \downarrow & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases}$$

The above "diagonalisation" shows that said diagonal *does not fit as a row in the matrix*. We will get a contradiction if we also show that *it must fit*!

Strictly speaking, the above definition by cases does not **define** d since the "$\downarrow$" in the top case is not a value; it is *ambiguous*. Easy to fix:

Say,

$$d(x) = \begin{cases} 42 & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \tag{4}$$

Here is why the function in (4) is partial computable:

Given x, do:

- Use the decider for K (for $\phi_x(x) \downarrow$, that is) —assumed to exist by (3)— to test which condition obtains 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:

while $1 = 1$ **do**
end

By CT, the 2-bullet program has a URM realisation, so d is computable.

Say now

$$d = \phi_i \tag{5}$$

What can we say about $d(i) = \phi_i(i)$? Well, we have two cases:

Case 1. $\phi_i(i)\downarrow$. Then we are in the bottom case of (4). Thus $d(i)\uparrow$. But we also have $d(i) = \phi_i(i)$ by (5), and our case assumes $\phi_i(i)\downarrow$, that is, $d(i)\downarrow$; a contradiction.

Case 2. $\phi_i(i)\uparrow$. This leads to a contradiction too, since $d(i) = 42$ in this case, thus, $d(i)\downarrow$. But by (5) $d(i) = \phi_i(i)$, so we must also have $d(i)\uparrow$; contradiction once more.

So we reject (3). □

In terms of *theoretical significance*, the above is the most significant 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* (relation!) for which we want to see whether $\vec{y} \in P$ can be solved by a URM. We build a *reduction* that goes like this:

(1) Suppose that we have a URM M 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 the latter is unsolvable*, 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*?

since we can view a specification as just another formalism to express a function computation.

By CT, all such formalisms, programs or specifications, boil down to URMs, and hence the above asks whether two given URMs compute the same function —program equivalence.

Let us show now that the program equivalence problem cannot be solved by any URM.

6.3.5 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. The proof is by a reduction (see above), hence by contradiction. We will show that if we have a URM that solves it, "yes"/"no", then we have a URM that solves the halting problem too!

So

$$\text{Assume we have a (URM) } E \text{ for the } \textit{equivalence problem.} \qquad (\ddagger)$$

Let us use it to answer the question "$a \in K$"—that is, "$\phi_a(a) \downarrow$", for any a.

$$\text{So, fix an } a \qquad (2)$$

Consider these two computable functions given by:

For all x by:

$$Z(x) = 0$$

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 shown a URM M that computes it (in class). 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, using the universal function h start computing $h(a, a)$, which is the same as $\phi_a(a)$ (cf. 6.2.1). If this ever halts just print 0 and halt; otherwise let it loop forever.

[‡]If we set $P = \{(i, j) : \phi_i = \phi_j\}$, then this problem is the question "$(i, j) \in P$?" or "$P(i, j)$?".

By CT, $\widetilde{Z}$ is in $\mathcal{P}$, that is, it 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* $N^{\mathbf{w}}_{\mathbf{w}'}$. Thus $Z = \phi_i$ and $\widetilde{Z} = \phi_j$.

By the *assumption* (‡) above, we proceed to feed i and j to E. This machine will halt and answer "yes" (0) precisely when $\phi_i = \phi_j$; will halt and answer "no" (1) otherwise. But note that $\phi_i = \phi_j$ iff $\phi_a(a) \downarrow$. We have thus solved the halting problem! A contradiction to the existence of URM E. $\square$

Chapter 7

More (Un)Computability via Reductions

Oct. 31, 2022

This is Part II of our (Un)Computability notes. We introduce "half-computable" relations $Q(\vec{x})$ here. These play a central role in Computability.

The term "half-computable" 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., *exactly if* $\vec{a} \in Q$ 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 M into M' that *is* a decider of such a Q? *No, not in general*!

For example, our *halting set* K *does have* a verifier but *no decider*!

(The latter we know: having a decider means $K \in \mathcal{R}_*$ and we know that this NOT the case).

Why does a verifier *exist* for $x \in K$?

Well, $x \in K$ iff $\phi_x(x)\downarrow$ iff $h(x,x)\downarrow$.

A verifier for "$x \in K$" is *any URM M that computes $\lambda x.h(x,x)$*.

Since the "yes" of a verifier M is signalled by halting but the "no" is signalled 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)

7.1.1 Definition. (Semi-recursive or semi-decidable sets)
A relation $Q(\vec{x}_n)$ is *semi-decidable* or *semi-recursive* —what we called suggestively "half-computable" above—

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

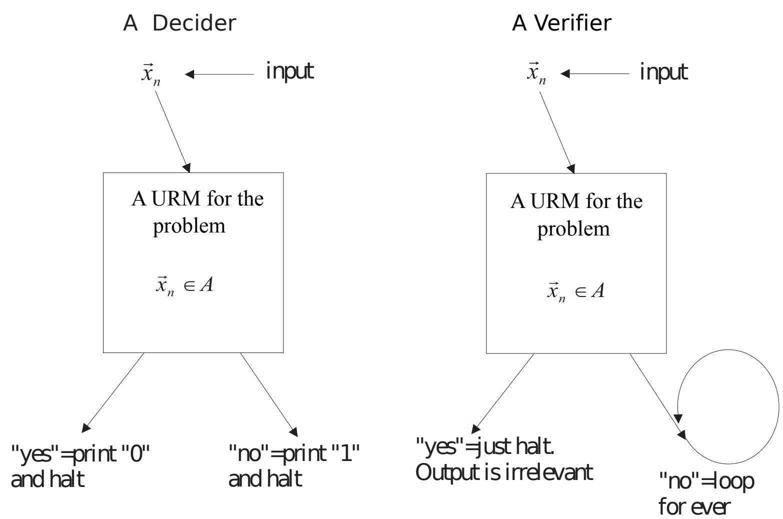
$$Q(\vec{x}_n) \equiv f(\vec{x}_n) \downarrow \tag{1}$$

Since $f \in \mathcal{P}$ is some $M_{\mathbf{y}}^{\vec{\mathbf{x}}_n}$, M is a verifier for Q.

The set of *all* semi-decidable relations we will denote by $\mathcal{P}_*$.[†] □

[†]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".

The following figure shows the two modes of handling a query, "$\vec{x}_n \in A$", by a URM.



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 $\lambda x.\phi_x(x) = \lambda x.h(x, x)$ and we know $h \in \mathcal{P}$.

Thus $x \in K \equiv h(x, x) \downarrow$ settles it. By Definition 7.1.1 (statement labeled (1)) we are done. $\square$

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 replace it by an "infinite loop",

while$(1 = 1)$
{
}

By CT we can certainly do that via a URM implementation.

A more elegant way (which still invokes CT) is to say, OK: Since $A \in \mathcal{R}_*$, it follows that c_A, its characteristic function, is in $\mathcal{R}$.

Define a new function f as follows:

$$f(\vec{x}_n) = \begin{cases} 0 & \text{if } c_A(\vec{x}_n) = 0 \\ \uparrow & \text{if } c_A(\vec{x}_n) = 1 \end{cases}$$

This is intuitively computable (the "$\uparrow$" is implemented by the same **while** as above).

Hence, by CT, $f \in \mathcal{P}$. But

$$\vec{x}_n \in A \equiv f(\vec{x}_n) \downarrow$$

because of the way f was defined. Definition 7.1.1 rests the case.

One more way to do this: Totally mathematical ("formal", as people say incorrectly[†]) this time!

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

$$f(\vec{x}_n) = \text{if } \underset{z}{\overset{c_A(\vec{x}_n)}{\downarrow}} = 0 \text{ then } \underset{u}{\overset{0}{\downarrow}} \text{ else } \underset{w}{\overset{\emptyset(\vec{x}_n)}{\downarrow}}$$

$\emptyset$ 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 $g = \lambda y\vec{x}_n.SZ(y) = \lambda y\vec{x}_n.1$.

In what follows we will favour the informal way (proofs by Church's Thesis) of doing things, most of the time. $\square$

An important observation following from the above examples deserves theorem status:

7.1.4 Theorem. *$\mathcal{R}_* \subset \mathcal{P}_*$ (or $\mathcal{R}_* \subsetneqq \mathcal{P}_*$).*

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 $\mathcal{R}_*$. $\square$

[†] "Formal" refers to syntactic proofs based on axioms. Our "*mathematical*" proofs are mostly *semantic*, depend on meaning, not just syntax. That is how it is in the majority of MATH publications.

What about $\overline{K}$, that is, the *complement*

$$\overline{K} = \mathbb{N} - K = \{x : \phi_x(x) \uparrow\}$$

of K?

The following general result helps us handle this question.

7.1.5 Theorem. *A relation $Q(\vec{x}_n)$ is recursive if **both** $Q(\vec{x}_n)$ and $\neg Q(\vec{x}_n)$ are semi-recursive.*

Before we proceed with the proof, a remark on notation is in order.

In "set notation" we write the complement of a set, A, of n-tuples as $\overline{A}$. This means, of course, $\mathbb{N}^n - A$, where

$$\mathbb{N}^n = \underbrace{\mathbb{N} \times \cdots \times \mathbb{N}}_{n \text{ copies of } \mathbb{N}}$$

In "relational notation" we write the same thing (complement) as

$$\neg A(\vec{x}_n)$$

Similarly,

"set notation": $A \cup B$, $\quad A \cap B$

"relational notation": $A(\vec{x}_n) \vee B(\vec{y}_m)$, $\quad A(\vec{x}_n) \wedge B(\vec{y}_m)$

Back to the proof.

Proof. 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. □

7.1.6 Remark. The above is really an "iff"-result, because $\mathcal{R}_*$ is *closed under complement (negation)* as we showed in class/Notes.

Thus, if Q is in $\mathcal{R}_*$, then so is $\overline{Q}$, by closure under $\neg$. By Theorem 7.1.4, both of Q and $\overline{Q}$ are in $\mathcal{P}_*$. □

†We can do that, i.e., M and M' exist, since both Q and $\overline{Q}$ are semi-recursive.

7.1.7 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. $\square$

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. (both undecidable) $x \in K$ (halting problem), $\phi_i = \phi_j$ (equivalence problem)

   and

2. (both not semi-recursive) $x \in \overline{K}$.

In fact, we will learn shortly that $\phi_i = \phi_j$ *is worse than undecidable*; just like $\overline{K}$ it *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". □

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 "subroutine" $z \in B$.
3. Compute $f(x)$.
4. Give the same answer for $x \in A$ (true or false) as you did for $f(x) \in B$.

When (1) (or, *equivalently*, (2)) holds, then, intuitively,

"A is easier than B to either decide or verify" since we can *solve* or "*half-solve*" $x \in A$ if we know how to *solve* or (only) *half-solve* $z \in B$.

This observation has a very precise counterpart (Theorem 7.2.3 below). But first,

[†] 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.

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), *throughout*, 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 $A \leq B$ 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.

Let $f \in \mathcal{R}$ effect the reduction.

(*i*) Let $z \in B$ be in $\mathcal{R}_*$.

Then for some $g \in \mathcal{R}$ we have

$$z \in B \equiv g(z) = 0$$

and thus

$$f(x) \in B \equiv g(f(x)) = 0 \tag{1}$$

But $\lambda x.g(f(x)) \in \mathcal{R}$ by composition, so (1) says that "$f(x) \in B$" is in $\mathcal{R}_*$. But that is the same as "$x \in A$".

(*ii*) Let $z \in B$ be in $\mathcal{P}_*$. By 7.2.2, so is $f(x) \in B$ in $\mathcal{P}_*$. But $f(x) \in B$ says $x \in A$. $\square$

Taking the "contrapositive", we have at once:

7.2.4 Corollary. *If $A \le B$ 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 $B \notin \mathcal{P}_*$*

We can now use K and $\overline{K}$ as "yardsticks" —or reference "problems"— and discover more undecidable and also *non semi-decidable* problems.

The idea of the corollary is applicable to the so-called "complete index sets".

7.2.5 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 $f \in \mathcal{C}$. Thus $f = \phi_i$ for some i. Then $i \in A$. But this is true of **all** ϕ_m that equal f.

We call A a *complete* (all) *index* (programs) set. $\square$

7.2.6 Example. The set $A = \{x : \mathrm{ran}(\phi_x) = \emptyset\}$ is not semi-recursive.

Recall that "range" for $\lambda x.f(x)$, denoted by $\mathrm{ran}(f)$, is *defined by*

$$\mathrm{ran}(f) \stackrel{Def}{=} \{x : (\exists y) f(y) = x\}$$

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.
Well, define

$$\psi(x, y) = \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \tag{2}$$

Here is how to compute ψ:

1. Given x, y, ignore y.
2. Fetch machine M at address x from the standard listing, and call it on input x. If it ever halts, then print "0" and halt everything.
3. If it never halts, then you will never return from the call, which is the correct specified in (2) behaviour for $\psi(x, y)$.

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,

$$\phi_{h(x)}(y) = \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} = \begin{cases} 0 & \text{if } x \in K \\ \uparrow & \text{if } x \in \overline{K} \end{cases} \tag{3}$$

or, rewriting (3) without arguments (as equality of functions, not equality of function calls)

$$\phi_{h(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \downarrow \\ \emptyset & \text{if } \phi_x(x) \uparrow \end{cases} \tag{3'}$$

In $(3')$, $\emptyset$ stands for $\lambda y. \uparrow$, the empty function.

Thus,

$$h(x) \in A \text{ iff } \operatorname{ran}(\phi_{h(x)}) = \emptyset \overbrace{\text{ iff }}^{\text{bottom case in } 3'} \phi_x(x) \uparrow \text{ iff } x \in \overline{K}$$

The above says $x \in \overline{K} \equiv h(x) \in A$, hence $\overline{K} \leq A$, and thus $A \notin \mathcal{P}_*$ by Corollary 7.2.4, part ii. $\square$

Nov. 2, 2022

Given a complete index set $A = \{x : \phi_x \in \mathcal{C}\}$. We want $K \leq A$ —or even $\overline{K} \leq A$.

The technique part is this:

Purpose: Use S-m-n to obtain $h \in \mathcal{R}$ such that

$$\phi_{h(x)} = \begin{cases} f & \text{if } x \in K \qquad (\textit{same as } \phi_x(x)\downarrow) \\ g & \text{if } x \in \overline{K} \qquad (\textit{same as } \phi_x(x)\uparrow) \end{cases}$$

Choice of f, g:

- Case where I want $K \leq A$. Then choose f to be in $\mathcal{C}$ but $g \notin \mathcal{C}$. So we have

$$h(x) \in A \overset{\textit{Def of } A}{\text{iff}} \phi_{h(x)} \in \mathcal{C} \overset{\textit{red type above}}{\text{iff}} \phi_{h(x)} = f \text{ iff } x \in K$$

- Case where I want $\overline{K} \leq A$. Then choose g to be in $\mathcal{C}$ but $f \notin \mathcal{C}$. So we have

$$h(x) \in A \overset{\textit{Def of } A}{\text{iff}} \phi_{h(x)} \in \mathcal{C} \overset{\textit{red type above}}{\text{iff}} \phi_{h(x)} = g \text{ iff } x \in \overline{K}$$

7.2.7 Example. The set $B = \{x : \phi_x$ 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,

$$h(x) \in B \text{ iff } \phi_{h(x)} \text{ has finite domain } \overbrace{\text{iff}}^{\underline{\text{bottom case}} \text{ in } 3'} \phi_x(x) \uparrow$$

The above says $x \in \overline{K} \equiv h(x) \in B$, hence $B \notin \mathcal{P}_*$ by Corollary 7.2.4, part ii. $\square$

7.2.8 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 7.2.6. Note that in $(3')$ the top case defines a constant function, but the bottom case defines a non-constant. Thus
   $$h(x) \in G \equiv \phi_x = \lambda y.0 \equiv x \in K$$
   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:
   $$h(x) \in I \overset{\emptyset \notin \mathcal{R}!}{\equiv} \phi_x = \lambda y.0 \equiv x \in K$$
   Thus $K \leq I$, therefore $I \notin \mathcal{R}_*$. $\square$

7.2.9 Example. (The Equivalence Problem, again) We now revisit the equivalence problem and show it is more unsolvable than we originally thought (cf. 6.3.5): 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.

Choose then for y a ϕ-index for the *empty function*.
So, if $\lambda xy.\phi_x = \phi_y$ is in $\mathcal{P}_*$ then so is

$$\phi_x = \emptyset$$

which is equivalent to

$$\mathrm{ran}(\phi_x) = \emptyset$$

and thus not in $\mathcal{P}_*$ by 7.2.6. □

7.2.10 Example. The set $C = \{x : \text{ran}(\phi_x) \text{ is finite}\}$ is not semi-decidable.

Here we cannot reuse (3′) above, because **both** cases —in the definition by cases— have functions of ***finite range***. We want one case to have a function of finite range, but the other to have i*nfinite range*.

Aha! This motivates us to choose a different "ψ" (hence a different "h"), and retrace the steps we took above.

OK, define

$$g(x,y) = \begin{cases} y & \text{if } \phi_x(x)\downarrow \\ \uparrow & \text{if } \phi_x(x)\uparrow \end{cases} \tag{ii}$$

Here is an algorithm for g:

- Given x, y.
- Use the universal program M for unary partial computable functions (computes the $\lambda xy.h(x,y)$) and start computing $h(x,x)$, that is, $\phi_x(x)$
- If this ever halts, then print "y" and halt everything.
- If it never halts then you will never return from the call, which is the correct behaviour for $g(x,y)$: 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)

$$\phi_{k(x)} = \begin{cases} \lambda y.y & \text{if } x \in K \\ \emptyset & \text{othw} \end{cases} \qquad (iii)$$

Hence,

$$k(x) \in C \text{ iff } \phi_{h(x)} \text{ has finite range } \overbrace{\text{iff}}^{\text{bottom case in } iii} x \in \overline{K}$$

That is, $\overline{K} \leq C$ and we are done. $\square$

7.2.11 Exercise. Show that $D = \{x : \mathrm{ran}(\phi_x) \text{ is infinite}\}$ is undecidable. $\square$

7.2.12 Exercise. Show that $F = \{x : \mathrm{dom}(\phi_x) \text{ is infinite}\}$ is undecidable. $\square$

Enough "negativity"! Here is an important "positive result" that helps to prove that certain relations *are* semi-decidable:

7.3. Some Positive Results

Nov. 7, 2022

7.3.1 Theorem. (Projection theorem) *A relation $Q(\vec{x}_n)$ is semi-recursive **iff** there is a recursive (decidable) relation $S(y, \vec{x}_n)$ such that*

$$Q(\vec{x}_n) \equiv (\exists y) S(y, \vec{x}_n) \tag{1}$$

Q is obtained by "projecting" S along the y-co-ordinate, hence the name of the theorem.

Proof. **If**-part. Let $S \in \mathcal{R}_*$, and Q be given by (1) of the theorem.
We show that some M semi-decides

$$\vec{x}_n \in Q \tag{2}$$

Here is how:

proc $Q(\vec{x}_n)$
$y \leftarrow 0$ /* Initialize "search" */
while $(c_S(y, \vec{x}_n) = 1)$ /* This call always terminates since $S \in \mathcal{R}_*$ */
{
$y \leftarrow y + 1$
}

By CT, there is a URM N that implements the above pseudo-code. Clearly, this URM semi-decides (2).

Did I say "search"? But of course! Trivially,

$$(\exists y) S(y, \vec{x}_n) \equiv \Big((\mu y) S(y, \vec{x}_n)\Big) \downarrow \tag{$*$}$$

But $\lambda \vec{x}_n.(\mu y)S(y, \vec{x}_n) \in \mathcal{P}$.† Hence $Q(\vec{x}_n)$ is semi-recursive by Definition 7.1.1 since, by $(*)$,

$$Q(\vec{x}_n) \equiv \Big((\mu y)S(y, \vec{x}_n)\Big) \downarrow$$

Only if-part. This is more interesting because it introduces *a new proof-technique*:

So, we now know that $Q \in \mathcal{P}_*$, and want to show that *there is an $S \in \mathcal{R}_*$ for which (1) above holds*:

Well, let M semi-decide $\vec{x}_n \in Q$.

Define $S(y, \vec{x}_n)$ as follows:

$$S(y, \vec{x}_n) \stackrel{\text{by Def}}{\equiv} \begin{cases} \textbf{true} & \text{if } M \text{ on input } \vec{x}_n \text{ halts in } \leq y \\ & \qquad\qquad \text{computation steps} \\ \textbf{false} & \text{otherwise} \end{cases}$$

We argue that $S(y, \vec{x}_n)$ is decidable. Indeed, here is how to decide it:

1. Enlist the help of *someone* who keeps track of computing **time** for M, from the time the URM's (program's) computation starts and onwards.

   In intuitive (non mathematical) terms, this "someone" could be the Operating System under which the program M is compiled and executed; or you or me.

†You recall, of course, that $(\mu y)S(y, \vec{x}_n)$ is *defined* to *mean* $(\mu y)c_S(y, \vec{x}_n)$.

2. Given an input $y, \vec{x}_n$, the *System* keeps track of **elapsed computation time** during M's computation.

   This "time" could be in *time units*, like *seconds, nanoseconds*, etc., **or** in *instruction-execution units*, that is, the *number of instructions executed* —*with* repetitions, of course: instruction, say, $L: \ \ldots$, if embedded in a loop, may be executed *several* times. Each time counts!

   The system will halt the entire process (including exiting M even if M did not hit its stop instruction yet) as soon as y time units have elapsed.

   It is *absolutely important* to remember at this point that any URM M will continue computing *in a trivial manner* once it hits **stop**:

   This "trivial manner" consists of M going on "computing", specifically "executing" **stop** ad infinitum, and doing so by **changing nothing in any variable**.

3. **Output Decisions at time** y.

   Output will be as follows:

   - **true** (0) if M was executing **stop** —and probably doing so even at earlier steps, which explains the "$\le y$".
   - **false** (1) if M was **not** executing **stop** at the time the System halted everything.

     Comment. The above is the case where M needed MORE than y steps to finish its computation (if at all).

By CT, the above algorithm, M plus Operating System plus decisions on what to output, can be formalized into a URM, N, which

decides (true/false) S, i.e., $S \in \mathcal{R}_*$.

Now it is trivial that (1) holds (p.207), for we have the equivalences

$$Q(\vec{x}_n) \equiv \text{For some } y,\ M, \text{ on input } \vec{x}_n, \text{ halts in } \le y \text{ steps}$$

That is

$$Q(\vec{x}_n) \equiv \text{For some } y,\ S(y, \vec{x}) \text{ is true}$$

or

$$Q(\vec{x}_n) \equiv (\exists y) S(y, \vec{x})$$

$\square$

7.3.2 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. Call the universal function h on input x, y.
2. If the Universal program H for h halts, then
   - If the output of H equals z then halt everything (the "yes" output).
   - If the output is not equal to z, then enter an infinite loop (say "no", by looping).

By CT the above informal verifier can be formalised as a URM M.

But is it correct? Does it verify $\phi_x(y) = z$?

Yes. See **Comment** above. □

Chapter 8

Uncomputability; Part III

8.1. 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:

8.1.1 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:

- $A = \emptyset$

  OR

- $A = \mathrm{ran}(f)$, where $f \in \mathcal{R}$.

$\square$

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:

8.1.2 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 $A = \mathrm{ran}(f)$ —where $\lambda x.f(x)$ is recursive— is* semi-recursive *(and, trivially, nonempty).*

In short, *the semi-recursive sets are precisely the same as the c.e. or r.e. sets*. For $A \neq \emptyset$ this is the content of 8.1.2, while $\emptyset$ is r.e. by definition and known to us to be also semi-recursive —due to $\emptyset \in \mathcal{PR}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$.

Before we prove the theorem, here is an example:

8.1.3 Example. The set $\{0\}$ is c.e. Indeed, $f = \lambda x.0$, our familiar function Z, effects the enumeration with repetitions (lots of them!)

$$\begin{array}{llllll} x & = 0 & 1 & 2 & 3 & 4 \quad \ldots \\ f(x) & = 0 & 0 & 0 & 0 & 0 \quad \ldots \end{array}$$

$\square$

Proof. of the theorem.

(I) **We prove the first sentence of the theorem**. So, let $A \neq \emptyset$ be *semi-recursive*.

By the projection theorem (cf. 7.3.1) there is a **recursive** relation $Q(y, x)$ such that

$$x \in A \equiv (\exists y)Q(y, x), \text{ for all } x \tag{1}$$

Thus,

$$\text{for every } x \in A \text{ some } y \text{ makes } Q(y, x) \text{ true.} \tag{2}$$

and conversely,

$$\text{if } Q(y, x) \text{ holds for some } y, x \text{ pair, then } x \in A. \tag{2'}$$

(2) and (2′) *jointly rephrase (1)*.

So why not enumerate all POSSIBLE PAIRS y, x

$$(y = (z)_0, \quad x = (z)_1)$$

for each $z = 0, 1, 2, 3, \ldots$— and output x iff *we find that $Q(y, x)$ is true?*

We do exactly this!

Recall that $A \neq \emptyset$. So fix an $a \in A$.

$$f(z) = \begin{cases} (z)_1 & \text{if } Q((z)_0, (z)_1) \\ a & \text{othw} \end{cases} \tag{3}$$

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*.

The case "a" does two things:

- a is an f-output in A. So f's outputs are in A in both the upper and lower case in (3).
- Ensures we are *never* at a loss and declare $f(z) \uparrow$ whenever $Q((z)_0, (z)_1)$ is false.

(II) **Proof of the second sentence of the theorem**.

So, let $A = \mathrm{ran}(f)$ —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. $\square$

8.1.4 Corollary. *An $A \subseteq \mathbb{N}$ is semi-recursive iff it is r.e. (c.e.)*

Proof. For *nonempty* A this is Theorem 8.1.2. For empty A we note that this is r.e. by 8.1.1 but also semi-recursive by $\emptyset \in \mathcal{PR}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$. $\square$

Corollary 8.1.4 allows us to prove some non-semi-recursiveness results by good old-fashioned Cantor diagonalisation.

See below.

8.1.5 Theorem. *The complete index set $A = \{x : \phi_x \in \mathcal{R}\}$ is* not *semi-recursive.*

This sharpens the undecidability result for A that we established already (cf. 7.2.8 2.).

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., $Z \in \mathcal{R}$ and thus all ϕ-indices of Z are in A.

Thus, theorem 8.1.2 applies and **there is an $f \in \mathcal{R}$ that enumerates A**:

$$y \in A \equiv (\exists x) f(x) = y$$

*In words, a ϕ-index y is in A **iff** it* has the form $f(x)$ for some x.

Define

$$d = \lambda x.1 + \phi_{f(x)}(x) \tag{1}$$

Seeing that $\phi_{f(x)}(x) = h(f(x), x)$ —*you remember the universal h?*— we obtain $d \in \mathcal{P}$ and, by totalness, $d \in \mathcal{R}$.

Also,

$$d = \phi_{f(i)}, \text{ for some } i \tag{2}$$

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*. $\square$

One can take as d different functions, for example, either of $d = \lambda x.42 + \phi_{f(x)}(x)$ or $d = \lambda x.1 \dot{-} \phi_{f(x)}(x)$ works. And infinitely many other choices do!

8.2. Some closure properties of decidable and semi-decidable relations

We already know that

8.2.1 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 $\mathcal{P}_*$?

8.2.2 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. Let $Q(\vec{x}_n)$ be semi-decided by a URM M, and $S(\vec{y}_m)$ be semi-decided 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!

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!

   What do I do? Coding to the rescue!

   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 <u>PAIR</u> of values (z, y).*

   In fact I can <u>CODE</u> the pair as $w = \langle z, y \rangle = 2^{z+1}3^{y+1}$ —remember coding?— 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
$$(\exists y)_{<z} Q(y, \vec{x}) \equiv (\exists y)\Big(y < z \wedge Q(y, \vec{x})\Big) \tag{$*$}$$
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 so that $P(w_y, y, \vec{x})$ holds

Since all those w_y are finitely many (z many!) there is a value u as big as **ANY** of them (for example, take $u = \max(w_0, \ldots, w_{z-1})$).

Thus $(**)$ says (i.e., **is equivalent to**)

$$(\exists u)(\forall y)_{<z} (\exists w)_{\leq u} P(w, y, \vec{x})$$

The blue part of the above is **decidable** (by closure properties of $\mathcal{R}_*$, since $P \in \mathcal{R}_*$ —you may peek at 8.2.1). 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 $\overline{K} \notin \mathcal{P}_*$.

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 8.2.1), $\neg Q(y, x)$ 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 $x \in \overline{K}$! □

8.3. Computable functions and their graphs

Nov. 9, 2022

We prove a fundamental result here, that

8.3.1 Theorem. *$\lambda\vec{x}.f(\vec{x}) \in \mathcal{P}$ iff the graph $y = f(\vec{x})$ is in $\mathcal{P}_*$.*

Proof.

- ($\rightarrow$, that is, the **Only if**) Let $\lambda\vec{x}.f(\vec{x}) \in \mathcal{P}$. By an easy adaptation of the proof in Example 7.3.2 it follows that $y = f(\vec{x})$ is semi-computable.
- ($\leftarrow$, that is, the **If**) Let $y = f(\vec{x})$ be semi-computable.

  Here is an obvious idea: Let M be a **verifier** for $y = f(\vec{x})$. Program as follows:

  1. **for** $z = 0, 1, 2, \ldots$ **do**:
  2. **if** M **verified** $z = f(\vec{x})$ **then return** (z)

  Let us emphasise: The verifier M does not compute $f(\vec{x})$ but rather verifies when a *pair $z, \vec{x}$ belongs to the graph* of f. If we knew *a priori* how to compute $f(\vec{x})$ we would not need to deal with the graph and its verifier at all!

  Alas, the above idea does **not** work! For any z-value that is $< f(\vec{x})$ in the above search for the "correct" z[†] the verifier says "**no**" by **looping** forever! We will never reach the correct z, *if there is one.*[‡]

  We must be more sophisticated in **what** and **how** we are searching for:

[†]That is, such that $z = f(\vec{x})$.
[‡]It may well be that $f(\vec{x}) \uparrow$ for the given $\vec{x}$.

By (strong projection theorem)

$$y = f(\vec{x}) \equiv (\exists z) Q(z, y, \vec{x}) \tag{1}$$

for some decidable Q. The idea of how to find the correct y, **if any**, once we are *given* an $\vec{x}$, is to search (**simultaneously**!) for a z **and** y that "work" —i.e., they satisfy $Q(z, y, \vec{x})$ for the given $\vec{x}$.[†] So, informally, we search the sequence

$$w = 0, 1, 2, 3, \ldots$$

and stop as soon as we note that $Q((w)_0, (w)_1, \vec{x})$ is true —if this ever happens!

As $(w)_0$ plays the role of z and $(w)_1$ plays the role of y, we obviously report $(w)_1$ as our answer, **if and when we stop the search**.

Mathematically,

$$f(\vec{x}) = \Big((\mu w) c_Q((w)_0, (w)_1, \vec{x})\Big)_1$$

f is in $\mathcal{P}$ by closure properties. □

We can now settle

8.3.2 Theorem. *If $A = \mathrm{ran}(f)$ and $f \in \mathcal{P}$, then $A \in \mathcal{P}_*$.*

Proof. By 8.3.1 $y = f(x)$ is semi-recursive. By closure properties of $\mathcal{P}_*$, so is $(\exists x) y = f(x)$. But $(\exists x) y = f(x) \equiv y \in \mathrm{ran}(f)$, that is, $(\exists x) y = f(x) \equiv y \in A$ since $\mathrm{ran}(f) = A$. Done. □

[†]We saw this idea in the proof of Theorem 8.1.2 at the beginning of this note.

8.4. More Unsolvability; Some tricky reductions

This section highlights a more sophisticated reduction scheme that improves our ability to effect reductions of the type $\overline{K} \le A$.

8.4.1 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

$$g(x, y) = \begin{cases} 0 & \text{if } x \in \overline{K} \\ \uparrow & \text{othw} \end{cases}$$

The problem is that *if we plan next to say* "by <u>CT g is partial</u> recursive hence by S-m-n, etc.", *then the underlined part is wrong*.

$g \notin \mathcal{P}$, *provably*! For if it is computable, then so is $\lambda x.g(x, x)$ by Grz-Ops. But

$$g(x, x) \downarrow \text{ iff we have the top case, iff } x \in \overline{K}$$

Thus

$$x \in \overline{K} \equiv g(x, x) \downarrow$$

which proves that $\overline{K} \in \mathcal{P}_*$ using the verifier for "$g(x, x) \downarrow$". **Contradiction!** □

8.4.2 Example. (8.4.1 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

$$f(x,y) = \begin{cases} 0 & \text{if the computation } \phi_x(x) \text{ has not reached } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases}$$

The "othw" says, of course, that the computation of the call $\phi_x(x)$ —or $h(x,x)$, where h is the universal function— *did* halt in $\leq y$ steps.

Next task is to enable the 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** $h(x,x)$, that is, $\phi_x(x)$, and keep count of its computation steps
(2) **Return** 0 if $\phi_x(x)$ did **not** hit **stop** in y steps
(3) **Loop forever** if $\phi_x(x)$ **halted** in $\leq y$ steps

Of course, the "command" **Loop forever** means

"transfer to the subprogram" **while** 1=1 **do** { }

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

$$\phi_{k(x)}(y) = \begin{cases} 0 & \text{if } \phi_x(x) \text{ is still not at } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases} \tag{1}$$

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 $\phi_x(x)$ —i.e., $h(x,x)$— ended in m steps. Therefore,

- for step counts $y = 0, 1, 2, \ldots, m-1$ the computation of $h(x,x)$ 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 $h(x,x)$ 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:

$$\phi_x(x)\downarrow \Longrightarrow \phi_{k(x)} = \overbrace{(0, 0, \ldots, 0)}^{\text{length } m} \tag{3}$$

In

$$\phi_{k(x)} = \overbrace{(0, 0, \ldots, 0)}^{\text{length } m}$$

we depict the function $\phi_{k(x)}$ as an array of m output values.

Two things: *One*, in English, when $\phi_x(x) \downarrow$, the function $\phi_{k(x)}$ is NOT a constant! Not even total!

Two, m depends on x, of course, when said x brings us to case (B). Regardless, the non-constant / non total nature of $\phi_{k(x)}$ is still a fact; just the length m of the finite array $\overbrace{(0, 0, \ldots, 0)}^{\text{length } m}$ changes.

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 $\overline{K} \leq A$. □

8.4.3 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 words now for the bottom case, 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. Thus we have

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \uparrow \\ \text{nontotal function} & \text{if } \phi_x(x) \downarrow \end{cases} \tag{6}$$

and therefore

$$k(x) \in B \text{ iff } \phi_{k(x)} \text{ is total iff the top case of (6) applies iff } \phi_x(x) \uparrow$$

That is, $x \in \overline{K} \equiv k(x) \in B$, hence $\overline{K} \leq B$. $\square$

8.4.4 Example. An earlier Exercise asks you to prove that $D = \{x : \text{ran}(\phi_x) \text{ is infinite}\}$ is not recursive.

Actually, D is not **semi**-recursive either, a fact that furnishes an example of a set that neither it, nor its complement are semi-recursive!

We (heavily) piggy back on Example 8.4.2 above. We want to find $j \in \mathcal{R}$ such that

$$\phi_{j(x)} = \begin{cases} \text{inf. range} & \text{if } \phi_x(x) \uparrow \\ \text{finite range} & \text{if } \phi_x(x) \downarrow \end{cases} \tag{$*$}$$

OK, define ψ (almost) like f of Example 8.4.2 by

$$\psi(x, y) = \begin{cases} y & \text{if the computation } \phi_x(x) \text{ has still } \textit{not} \text{ hit } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases}$$

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 8.4.2. The only difference is that in the algorithm (0)–(3) we change "**Return** 0" to "**Return** y".

The question $\psi \in \mathcal{P}$ settled, by S-m-n there is a $j \in \mathcal{R}$ such that

$$\phi_{j(x)}(y) = \begin{cases} y & \text{if the computation } \phi_x(x) \text{ has not hit } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases} \tag{$\dagger$}$$

Analysis of ($\dagger$) 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 ($\dagger$), we have $\phi_{j(x)}(y) = y$, for all y, that is,

$$\phi_x(x) \uparrow \Longrightarrow \phi_{j(x)} = \lambda y.y \tag{1}$$

(II) Case where $\phi_x(x)\downarrow$. Let m = *smallest* y such that the call $\phi_x(x)$ —i.e., $h(x,x)$— ended in m steps. Therefore, as before we find that for $y = 0, 1, \ldots, m-1$ we have $\phi_{j(x)}(y) = y$, that is,

$$\begin{array}{rllll} \text{for } y & = 0, & 1, & \ldots, & m-1 \\ \phi_{j(x)}(y) & = 0, & 1, & \ldots, & m-1 \end{array}$$

and as before,

$$\begin{array}{rllll} \text{for } y & = m, & m+1, & m+2, & \ldots \\ \phi_{j(x)}(y) & = \uparrow, & \uparrow, & \uparrow, & \ldots \end{array}$$

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.235— above. Thus

$j(x) \in D$ iff $\text{ran}(\phi_{j(x)})$ is infinite, iff we have the top case, iff $\phi_x(x)\uparrow$

Thus $\overline{K} \leq D$ via j. $\square$

8.5. An application of the GraphTheorem

A definition like

$$f(x,y) = \begin{cases} 0 & \text{if } x \in K \\ \uparrow & \text{othw} \end{cases} \tag{1}$$

is a special case of a so-called "Definition by Positive Cases". That is

- The cases listed explicitly (here $x \in K$) are semi-recursive, but the "othw" is **not** semi-recursive (here $x \in \overline{K}$). Therefore, as the latter cannot be **verified**, we let the function output be undefined in this case.

In *any* definition by cases

$$g(\vec{x}) = \begin{cases} \vdots & \vdots \\ g_i(\vec{x}) & \text{if } R_i(\vec{x}) \\ \vdots & \vdots \end{cases}$$

we have

$$\text{If } R_i(\vec{x}) \text{ then } g_i(\vec{x})$$

that is, we only need **verify** $R_i(\vec{x})$ —even if it is (primitive)recursive— to select the answer $g_i(\vec{x})$. However, in the **(primitive)recursive case** the "othw" is the negation of $R_1(\vec{x}) \vee R_2(\vec{x}) \vee \ldots \vee R_m(\vec{x})$, where $R_m(\vec{x})$ is the last explicit condition/case. By closure properties of $\mathcal{R}_*$, the "othw" case is recursive as well.

- In a Definition by Positive Cases the g_i are partial recursive.

The general form of Definition by Positive Cases is

$$g(\vec{x}) = \begin{cases} \vdots & \vdots \\ g_i(\vec{x}) & \text{if } R_i(\vec{x}) \\ \vdots & \vdots \\ g_k(\vec{x}) & \text{if } R_k(\vec{x}) \\ \uparrow & \text{othw} \end{cases} \tag{2}$$

where the g_i are in $\mathcal{P}$ and the R_i are in $\mathcal{P}_*$.

Note that $\mathcal{P}_*$ is **not** closed under negation, thus the "othw" in (2) is not in general semi-recursive. This is so in the case of (1) where the "othw" is $x \in \overline{K}$.

Does a definition like (2) yield a partial recursive g?

Yes:

8.5.1 Theorem. *g in (2), under the stated conditions, is partial recursive.*

Proof. We use the graph theorem, so it suffices to prove

$$y = g(\vec{x}) \text{ is semi-recursive} \tag{3}$$

Now, (3) is true precisely when $g(\vec{x}) \downarrow$ **and** the output is the **number** y. For this to happen, *some* **explicit** condition $R_i(\vec{x})$ was true and $y = g_i(\vec{x})$ was also true. In short, $y = g_i(\vec{x}) \wedge R_i(\vec{x})$ was true. Thus we prove (3) by noting

$$y = g(\vec{x}) \equiv y = g_1(\vec{x}) \wedge R_1(\vec{x}) \vee y = g_2(\vec{x}) \wedge R_2(\vec{x}) \vee \ldots \vee y = g_k(\vec{x}) \wedge R_k(\vec{x})$$

The rhs of $\equiv$ is semi-recursive since each $R_i(\vec{x})$ is (given) and each $y = g_i(\vec{x})$ is ($g_i \in \mathcal{P}$ and 8.3.1) at which point we invoke closure properties of $\mathcal{P}_*$ (8.2.2). $\square$

The immediate import of 8.5.1 is that, for example, we can prove without using CT that functions given as in (1), p.237, are in $\mathcal{P}$.

Chapter 9

The Recursion Theorem and the Theorem of Rice

Nov. 14, 2022

This chapter concludes our computability material with two important results named in the chapter title.

The recursion theorem is actually the "2nd recursion theorem" (there is also a "1st recursion theorem" that we will not get into). Both recursion theorems are due to Kleene but the version of the 2nd given here (and its proof) is due to Rogers.

9.0.1 Theorem. (The 2nd Recursion Theorem) *Given a recursive unary function f, there is a number e such that $\phi_e = \phi_{f(e)}$.*

Note that the theorem does NOT say that $f(e) = e$. Rather it says that the two addresses e and $f(e)$ have programs that compute the same function ϕ_e.

Proof. Define ψ by

$$\psi(x,y) = \begin{cases} \phi_{f(\phi_x(x))}(y) & \text{if } \phi_x(x)\downarrow \\ \uparrow & \text{othw} \end{cases} \tag{1}$$

By the universal function (h) theorem and Grz Ops, $\lambda xy.\phi_{f(\phi_x(x))}(y) = \lambda xy.h(f(\phi_x(x)), y) \in \mathcal{P}$.

So (1) is in $\mathcal{P}$ and thus (1) is a definition by *semi-recursive* or "positive" cases. As such, $\psi \in \mathcal{P}$.

By S-m-n, let $g \in \mathcal{R}$ be such that $\psi(x,y) = \phi_{g(x)}(y)$, for all y, or, equivalently

$$\phi_{g(x)} = \begin{cases} \phi_{f(\phi_x(x))} & \text{if } \phi_x(x)\downarrow \\ \emptyset & \text{othw} \end{cases} \tag{2}$$

Let a be a program (address) for g and take $e = \phi_a(a)$. Of course, $\phi_a(a)\downarrow$. Thus

$$\phi_e = \phi_{\phi_a(a)} = \phi_{g(a)} \overset{\textit{top of (2) by } \phi_a(a)\downarrow}{=} \phi_{f(\phi_a(a))} = \phi_{f(e)}$$

$\square$

9.0.2 Corollary. (Kleene's Original Version) *If $\lambda xy.\psi(x,y) \in \mathcal{P}$, then there is an $e \in \mathbb{N}$ such that $\psi(e,y) = \phi_e(y)$, for all y.*

Proof. By S-m-n, let $g \in \mathcal{R}$ such that $\psi(x,y) = \phi_{g(x)}(y)$, for all x, y. By 9.0.1 just pick e so that $\phi_{g(e)} = \phi_e$. $\square$

A major application of the recursion theorem (there are many other *major* applications that are beyond the scope of these Notes) is to provide an easy and short proof of *Rice's theorem* that states "Every nontrivial complete index set is undecidable (not recursive)".

Rice defined a complete index set $A = \{x : \phi_x \in \mathcal{C}\}$ to be trivial iff A is one of $\emptyset$ or $\mathbb{N}$. Else he called it nontrivial —i.e., when $\emptyset \neq A \neq \mathbb{N}$.

In popular language, and viewing (as it is normal practice) a set A as a "property" of its members, Rice's theorem below says that

a property of programs is decidable *iff all* programs have it or *no* program has it.

The following proof of Rice's theorem is attributed by Rogers ([Rog67]) to Wolpin.

9.0.3 Theorem. (Rice) *The complete index set* $A = \{x : \phi_x \in \mathcal{C}\}$ *is recursive iff it is trivial.*

Proof. IF. So let A be trivial. Done since both $\emptyset$ and its complement $\mathbb{N}$ are recursive (indeed primitive recursive).

ONLY IF Let A be recursive. We will argue that it must be trivial by contradiction, so let instead A be nontrivial.

$$\emptyset \neq A \neq \mathbb{N} \tag{1}$$

By (1) let $a \in A$ and $b \in \mathbb{N} - A$. Define a function f by

$$\text{For all } x, \quad f(x) = \begin{cases} b & \text{if } x \in A \\ a & \text{othw} \end{cases} \tag{2}$$

Since, by assumption on A, (2) is a definition by *recursive cases*, $f \in \mathcal{P}$. Being total I have

$$f \in \mathcal{R} \tag{3}$$

By construction of f I have,

$$\text{For all } x, \quad x \in A \text{ iff } f(x) \notin A \tag{4}$$

By 9.0.1 let $e \in \mathbb{N}$ be such that

$$\phi_e = \phi_{f(e)} \tag{5}$$

Thus

$$e \in A \text{ iff } \phi_e \in \mathcal{C} \overset{(5)}{\text{iff}} \phi_{f(e)} \in \mathcal{C} \text{ iff } f(e) \in A$$

We obtained $e \in A \equiv f(e) \in A$ which contradicts (4). $\square$

Chapter 10

A Subset of the URM Language; FA and NFA

This Note turns to a special case of the URM programming language that we call *Finite Automata*, in short *FA*.

This part presents (almost) a balance of How To and Limitations of Computing topics.

Main feature of the latter will be the so-called "*Pumping Lemma*".

10.1. The FA

The FA (*programming language*)[†] is introduced informally as a modified and restricted URM.

"FA" is an acronym for "**Finite Automaton**" (plural "finite automata").

This new URM model will have explicit "**read**" instructions.[*]

Secondly, *any specific URM under this model* ***will ONLY have ONE variable*** that we may call generically "**x**".

This variable will always be of *type* *single-digit*; it cannot hold arbitrary integers, rather it can only hold *single digits* as *values*.

[†]Note that some texts look at it as a "machine", hence the terminology "automaton".

[*]In Notes #2 we explained why explicit *read* instructions are theoretically as redundant as explicit *write* instructions are.

The FA has no instructions other than
1) "**read**" —**unlike the FULL URM**— and
2) a simplified **if-goto** instruction.

In the absence of a **stop** instruction, how does a computation halt?

We postulate that our modified URMs halt simply by reading something *that does not belong*, that is, it saw in the input stream an object that is *not* a member of the *input alphabet* of permissible digits.

Such an "illegal" symbol *serves as an end-marker* of the useful stream digits that constitute the *input string* over the given alphabet. As such it is often called an "*end-of-file*" marker, in short, **eof**.

This *eof*-marker is any "illegal" symbol, that is, a symbol not in the *particular FA's INPUT ALPHABET*.

*Thus the modified URM halts if **IFF** it runs out of input, as this is signaled by it reading something* NOT in its input alphabet.

Our insistence on a URM-like model for the automaton *will be confined in this brief motivational introduction* and is only meant to illustrate the indebtedness of the finite automata model to the general URM model.

As it is with the URM, all FA instructions are labelled.

The FA has, for *each* label L, a **group** of instructions as follows.

The typical group-instruction of an automaton.

$$L: \begin{cases} \textbf{read} \quad \textbf{Comment} \text{ Get the next, } \underline{\text{unread}}\text{, symbol; assign to } \mathbf{x} \\ \textbf{if } \mathbf{x} = a \textbf{ then goto } M \\ \textbf{if } \mathbf{x} = a' \textbf{ then goto } M' \\ \vdots \\ \textbf{if } \mathbf{x} = a^{(n)} \textbf{ then goto } M^{(n)} \\ \textbf{if } \mathbf{x} = \mathit{eof} \textbf{ then halt} \end{cases}$$

where L and $M, M', \ldots, M^{(n)}$ are labels —*not necessarily distinct*— and $a, a', \ldots, a^{(n)}$ are **all** the possible digit values in the context of a specific FA (program), that is, $\{a, a', \ldots, a^{(n)}\}$ *IS* the *input alphabet*.

The empty string, λ, will NEVER be part of a FA's input alphabet.

For any particular *FA (program)* —a particular FA, as we say (omitting "program")— its labels, in practice, *are not restricted to be numerical* **nor even to be consecutive (if numerical)**.

► However, *one* instruction's placement is significant.

It is often identified by a label such as "0", or "q_0", or some such symbol and **is placed at the very beginning of the program**.

This instruction's label is called the **initial state** of the specific automaton. Indeed, all labels in an automaton are called states in the literature.

ONE STARTS an FA computation with the instruction pointed at by the initial state.

Pause. A finite automaton does not care about the order of its other instructions, since they will be reachable by the goto-structure as needed wherever they are.◄

The semantics of the "typical" instruction above is:

- Read into the variable $\mathbf{x}$ the first unread digit-value from some "external (to the FA) input stream" that is waiting to be read.
- Then move to the *next instruction as is determined* by the $a^{(i)}$s (or the *eof*) in the if-cases above (p.247).

In order to have the FA make a decision about the input string that it just read, we (this is part of the design of the particular FA program) partition the instruction-labels of any given FA into two types: **accepting** and **rejecting**.

Their role is as follows: Such an FA, when it has halted,

Pause. *When* or *if*?◀

will have finished scanning a sequence of digits —*a string over its alphabet*.

This string is *accepted* if the program halted while in an *accepting state*, otherwise the input is *rejected*.

10.1.1 Definition. (The Language of an FA; Regular Sets) The language decided by a FA M is called in the literature "**the Language accepted by** M". It is, of course,

$$L(M) \stackrel{Def}{=} \{x : x \text{ is accepted by automaton } M\}$$

The accepted language we also call it a "**Regular Set**". □

Since an FA cannot "write", i.e., cannot change the contents of $\mathbf{x}$ — because it does not have any of the instructions $\mathbf{x} \leftarrow c$, $\mathbf{x} \leftarrow \mathbf{x} + 1$, $\mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$— we need the *type* of state the FA is in at the end of scanning to "code" the yes/no (accept/reject) answer.

10.2. Deterministic Finite Automata and their Languages

10.2.1 Example. Consider the FA below that operates over the input alphabet $\{0, 1\}$

$$0: \begin{cases} \mathbf{read} \\ \mathbf{if}\ \mathbf{x} = 0\ \mathbf{then}\ \mathbf{goto}\ 0 \\ \mathbf{if}\ \mathbf{x} = 1\ \mathbf{then}\ \mathbf{goto}\ 1 \\ \mathbf{if}\ \mathbf{x} = \mathit{eof}\ \mathbf{then}\ \mathbf{halt} \end{cases}$$

$$1: \begin{cases} \mathbf{read} \\ \mathbf{if}\ \mathbf{x} = 0\ \mathbf{then}\ \mathbf{goto}\ 1 \\ \mathbf{if}\ \mathbf{x} = 1\ \mathbf{then}\ \mathbf{goto}\ 0 \\ \mathbf{if}\ \mathbf{x} = \mathit{eof}\ \mathbf{then}\ \mathbf{halt} \end{cases}$$

What does this program do? Once we have the graph model, we will elaborate on what the above automaton actually does. LATER!

In particular we will look into two cases:

- When only state 0 is accepting.
- When only state 1 is accepting.

□

10.2.1. FA as Flow-Diagrams

Moving away from the URM-like programming language for automata, we next consider a "flow chart" or "flow diagram" formalisation.[†] This is achieved by first abstracting an instruction

$$L: \ \mathbf{read;\ if\ x} = a \ \mathbf{then\ goto}\ M \tag{1}$$

as the configuration below:

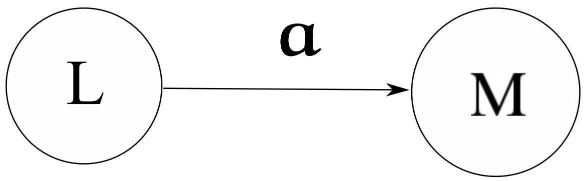


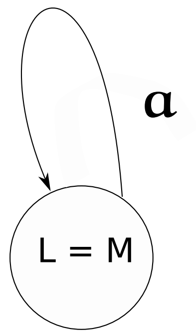
Figure capturing (1) above

Thus the "read" part is implicit, while the labeled arrow that connects the states L and M denotes exactly the semantics of (1). What is just read —a— is the arrow label.

[†]Defining the FA *form* as a flow chart.

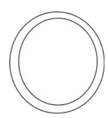
Therefore, an entire automaton can be viewed as a *directed graph* — that is, a finite set of (possibly) labeled circles, the *states*, and a finite set of arrows, the *transitions*, the latter labeled by members of the automaton's input alphabet.

An arrow label a in the figure above represents "**if x** $= a$ **then goto** M". The arrows or *edges* interconnect the states. If $L = M$, then we have the configuration

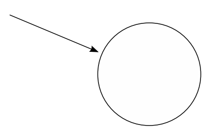


where the optional label could be L, or M, or $L = M$ (as above), or nothing.

In the Flow Chart Model we depict the partition of states into *accepting* and *rejecting* by using two concentric circles for each accepting state as below.



The special start state is denoted by drawing an arrow, that comes from nowhere, pointing to the state.



To summarise and firm up:

10.2.2 Definition. (FA as Flow Diagrams) A *finite automaton*, in short, *FA*, over the FINITE *input alphabet* Σ is a **finite directed graph of circular nodes** —the *states*— and interconnecting edges —the *transitions*— the latter being labeled by members of Σ.

We impose a restriction to the automaton's structure:

▶ For *every* state L and *every* $a \in \Sigma$, there will be *precisely* one *edge*, labeled a, leaving L and pointing to some state M (possibly, $L = M$).

We say the automaton is *fully specified* (corresponding to the italics in the part "For every state L and every $a \in \Sigma$, *there will be* ...") and *deterministic* (corresponding to the italics in the part "there will be *precisely one* edge, ...").

This graph depiction of a FA is called its *flow diagram* and is akin to a programming "*flow chart*". □

10.2.3 Remark. (1) Thus, full specification makes the *transition function total* —that is, for any state-input pair (L, a) as argument, it will yield some state as "*output*".

On the other hand, *determinism* ensures that the transition function is indeed a *function* (single-valued).

(2) **On Digits.** Each "legal" input symbol is a member of the alphabet Σ, and vice versa. In the preamble of this chapter we referred to such legal symbols as "digits" in the interest of preserving the *inheritance* from the URM, the latter being a number-theoretic programming language.

But what *is* a "digit"? In binary notation it is one of 0 or 1. In decimal notation we have the digits $0, 1, \ldots, 9$. In *hexadecimal* notation[†] we add the "digits" a, b, c, d, e, f that have "values", in that order, $10, 11, 12, 13, 14, 15$.

The objective is to have single-symbol, *atomic*, digits to avoid ambiguities in string notation.

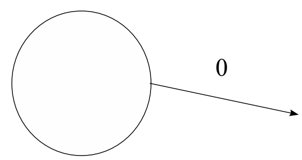
Thus, a "digit" is an atomic symbol (unlike "10" or "11").
We will drop the terminology "digit" from now on.

Thus our automata alphabets are *finite sets of symbols* —any length-ONE symbols, period. □

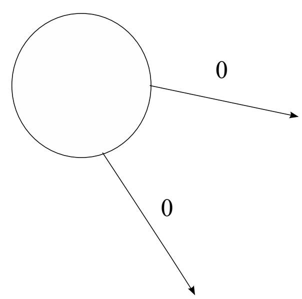
[†]Base 16 notation.

10.2.4 Example. Thus, if our alphabet is $A = \{0, 1\}$, then we cannot have the following configurations be part of a FA.

Nontotal Transition Function

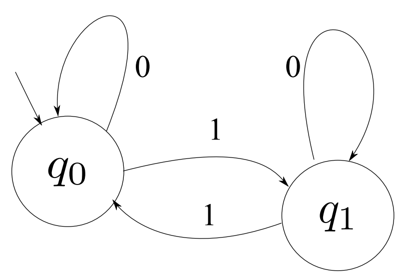


Non-determinism



$\square$

10.2.5 Example. The FA of the example of 10.2.1, in flow diagram form but with no decision on which state(s) is/are accepting is given below:



We wrote q_0 and q_1 for the states "0" and "1" of 10.2.1. $\square$

Another way to define a FA without the help of flow diagrams is as follows:

10.2.6 Alternative Definition. (FA —Algebraically) A *finite automaton*, *FA*, is a toolbox $M = (Q, A, q_0, \delta, F)$,[‡] where

(1) Q is a finite set of states.

(2) A is a finite set of symbols; the *input alphabet*.

(3) $q_0 \in Q$ is the distinguished *start state*.

(4) $\delta : Q \times A \to Q$ is a *total function*, called the *transition function*.

(5) $F \subseteq Q$ is the set of accepting states; $Q - F$ is the set of rejecting states. □

[‡] "M" is generic; for "machine".

10.2.7 Remark. Let us compare Definitions 10.2.2 and 10.2.6.

(1) The set of states corresponds with the nodes of the graph (flow diagram) model. It is convenient —but not theoretically necessary in general— to actually *name* (label) the nodes with names from Q.

(2) The A in the flow diagram model is not announced separately, but can be extracted as the set of all edge labels.

(3) q_0 —the start state by any name; q_0 being generic— in the graph model is recognised/indicated as the node pointed at by an arrow that emanates from no node.

(4) $\delta : Q \times A \to Q$ in the graph model is given by the arrow structure: Referring to the figure at the beginning of 10.2.1, we have $\delta(L, a) = M$. □

How does a FA compute? From the URM analogy, *we understand the computation of a FA consisting of successive*

- read moves
- attendant changes of state based on *current* state-symbol pair.
- until the program *halts* (by reading the *eof*).
- At that point we proclaim that the string *formed by the stream of symbols read* is *accepted* or *rejected* according as the halted machine is in an *accepting* or *rejecting* state.

To formalise/mathematise FA computations as described above, we use snapshots or *Instantaneous Descriptions* (of a computation), in short *ID*s.

The IDs of the FA are very simple, since the machine (program) is incapable of altering the input stream.

You do not need to keep track of how the contents of variables change.

10.2.8 Remark. (Digression into the Prerequisite!) We recall from discrete mathematics, that a *binary relation* R is a set of *ordered pairs* and we prefer to write aRb instead of $(a,b) \in R$ or $R(a,b)$. For example, we write $a \le b$ if R is $\le$.

We also recall that the so-called *transitive closure* of a relation R, denoted R^+, is defined by

$$aR^+b \stackrel{Def}{\equiv} aRa_1Ra_2 \ldots \overbrace{a_iRa_{i+1}}^{a\text{ “step”}} \ldots a_{m-1}Rb, \text{ for some } a_i, i = 0, \ldots, m-1$$

where we think of a as a_0 and b as a_m.

In other words,

aR^+b is true iff a can *reach* b in a finite number of one or more consecutive steps of the type "a_iRa_{i+1}", for $i = 0, \ldots, m-1$ as above.

We note that

for all i, $a_iRa_{i+1}Ra_{i+2}$ is short for a_iRa_{i+1} **and** $a_{i+1}Ra_{i+2}$

just as $a \le b \le c$ means $a \le b$ **and** $b \le c$.

The *reflexive transitive closure* of R is denoted by R^* and is defined by

$$aR^*b \stackrel{Def}{\equiv} a = b \vee aR^+b$$

The following notations also are useful:

$$aR^mb \stackrel{Def}{\equiv} aRa_1Ra_2Ra_3Ra_4 \ldots a_{m-2}Ra_{m-1}Rb$$

that is, exactly m copies of R occur in the **R-chain** —or just "chain" if R is understood—

$$aRa_1Ra_2Ra_3Ra_4 \ldots a_{m-2}Ra_{m-1}Rb$$

Finally, "$aR^{<m}b$" means "aR^nb **and** $n < m$". □

Nov. 16, 2022

10.2.9 Definition. (FA Computations; Acceptance)
Let $M = (Q, A, q_0, \delta, F)$ be a FA, and x be an input string —that is, a string over A that is presented as a *stream of (atomic) input symbols from* A.

An *M-ID* or simply *ID* related to x is a string of the form tqu, where $q \in Q$ is the state in the snapshot, and $x = tu$.

Intuitively, the expression tqu means that the computing agent, the FA, is in state q and that *the next input to process* is the *first symbol* of u.

$$\overbrace{t}^{\text{processed}} \quad q \quad \underbrace{u}_{\text{to be processed}}$$

If $u = \lambda$ —and hence the ID is simplified to tq— then M has halted (has read eof; no more input).

Formally, an ID of the form tq has *no next ID*. We call it a *terminal ID*.

However, an ID of form $tqau'$, where $a \in A$, has a *unique* next ID; this one: $ta\widetilde{q}u'$, *just in case* $\delta(q, a) = \widetilde{q}$.

Comment on full specification here!

We write

$$tqau' \vdash_M ta\widetilde{q}u'$$

or, simply (if M is understood)

$$tqau' \vdash ta\widetilde{q}u'$$

and pronounce it "(ID) $tqau'$ *yields* (ID) $ta\widetilde{q}u'$".

We say that M *accepts the string* x iff, for some $q \in F$, we have $q_0x \vdash_M^* xq$ —or, in words, ID q_0x reaches the terminal accepting ID xq in a finite number of *zero* or *more* steps.

Pause. *Zero* steps? Yes! If $x = \lambda$, then it is accepted *without taking any step* since

$$\overbrace{q_0x\P}^{\textit{initial}} = \overbrace{xq_0\P}^{\textit{terminal \& accepting}}$$

where I added "eof" as "¶" for emphasis.

The *language accepted by the FA* M is denoted generically by $L(M)$ and is the subset of A^* —this is notation for the set of **all** strings over the alphabet A[§]— given by $L(M) = \{x : (\exists q \in F)q_0x \vdash_M^* xq\}$.

An ID of the form q_0x is called a *start-ID*. □

[§] A^+, by definition, is $A^* - \{\lambda\}$.

10.2.10 Remark.

(I) Of course, $\vdash_M^*$ is the *reflexive transitive closure* of $\vdash_M$ and therefore $I \vdash_M^* J$ —where I (**not** necessarily a start-ID) and J (**not** necessarily terminal) are IDs— means that $I = J$ **or**, for some IDs I_m, $m = 1, \ldots, n-1$, we have an $\vdash_M$-chain

$$I \vdash_M I_1 \vdash_M I_2 \vdash_M I_3 \vdash_M \ldots \vdash_M I_{n-1} \vdash_M J \tag{1}$$

We say that we have an M-computation from I to J iff we have $I \vdash_M^* J$. We say simply *computation* if the "M-" part is understood.

(II) Is the Graph Model just a *static* way to *depict* a FA? No, it is MUCH more!

There is a tight relationship between computations and paths in a FA depicted as a graph. Compare Figure 10.1 and Display (1) below. They both say the same thing regarding part of the computation of some FA on the segment of the input from a_1 to a_n.

Indeed consider (1) below —viewed within the Algebraic model, say, of some FA M.

We have $\delta(p_i, a_i) = p_{i+1}$, for $i = 1, 2, \ldots, n$ which *tracks precisely* the FA moves depicted graphically in the segment of the flow diagram representation of the same FA shown in Figure 10.1.

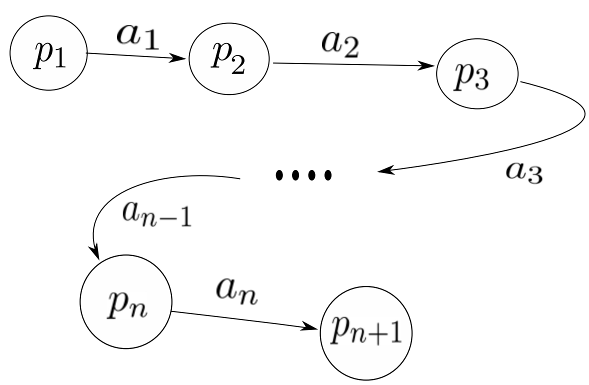


Figure 10.1: FA Computation Path

$$\overbrace{\ldots}^{t}\ \overset{p_1}{\overset{\downarrow}{a_1}}\ \overset{p_2}{\overset{\downarrow}{a_2}}\ \overset{p_3}{\overset{\downarrow}{a_3}}\ \ldots\ \overset{p_i}{\overset{\downarrow}{a_i}}\ \ldots\ \overset{p_n}{\overset{\downarrow}{a_n}}\ \overbrace{\ldots}^{u} \tag{1}$$

Thus, the concatenation of the labels a_i of the path in Figure

10.1 denote the string “ consumed” when starting at a_1 on state p_1.

If now p_1 is the *start state* and p_{n+1} is *accepting*, and moreover if $t = u = \lambda$ —and hence (1) above records that

$$\overset{\downarrow}{p_1}\, a_1 a_2 \ldots a_n \vdash^* a_1 a_2 \ldots a_n\, p_{n+1}$$

— we then have the important remark below:

A string $x = a_1 a_2 \ldots a_n$ over the input alphabet belongs to $L(M)$ —the Language Accepted (Decided) by the FA M; cf. 10.1.1— iff *it is formed by concatenating the labels of a path such as the one in Fig. 10.1, where $p_1 = q_0$ (start state) and p_{n+1} is* accepting.

We see that the flowchart model of a FA is more than a static depiction of an automaton's "vital" parameters, Q, A, q_0, δ, F.

> *Rather,* all computations, *including* accepting computations, *are also* encoded *within the model as certain paths.*

$\square$

The last few paragraphs were important. Let as summarise:

10.2.11 Definition. (Graph acceptance) Let M be a FA of start-state "p_1" over the alphabet Σ.

Let $x = a_1 a_2 \ldots a_n$ be a string over Σ.

Then x is accepted by M —equivalently $x \in L(M)$ (cf. 10.1.1)— iff x is the label of a *computation path* in the graph version of M in the sense that x is obtained by concatenating the names a_1, a_2, ..., a_n *OF THE EDGES* of said computation path (cf. Fig. 10.1) that starts at p_1 and ends at an *accepting* state p_{n+1}. The latter state has just scanned *eof* thus it caused M to halt. □

Armed with Definition 10.2.11, let us consider an example and shed more light on what exactly is eof.

10.2.12 Example.
Compilers, that is, **Systems Programs** that read programs written in a high level programming language like C and translate them into assembly language have several subtasks.

One of them is delegated to the so-called "scanner" or "token scanner" of the compiler and is the task of picking up variables (also special symbols like "++", "begin", "end") from the program source.

To "pick up" a variable, the scanner has to "*recognise*" that it saw one! Well, an automaton can do that!

Assume (as typically is the case) that the syntax of a variable is a string that

- *begins with a letter*

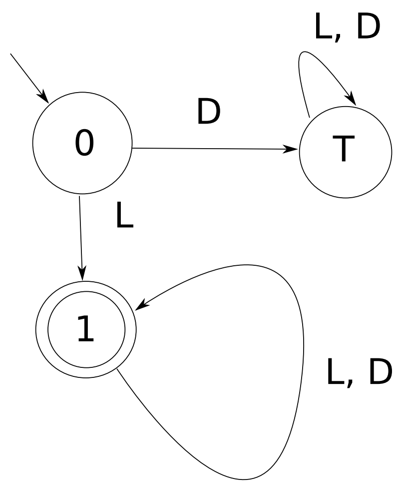
  and

- *continues with letters or digits*.

To simplify the example and not get lost in details, we denote the input alphabet of the automaton that we will build here $\Sigma = \{L, D\}$ where the symbol L stands for any **letter** (in real life, one uses the members of the set {A, B, C, . . . , Z; a, b,. . . , z}, sometimes augmented by some special symbols like \$ and *underscore*).

Similarly the symbol D in our alphabet stands for **digit** (in real life, one has here the set of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$).

Using the characterisation of acceptance in 10.2.11, here is our design:



The only paths to state "1" (accepting) are labelled with L, followed by zero or more L and/or D in any order. That's the right syntax we want!

What is the role of state "T"?

T for trap! We do not want the first symbol of a variable to be other than L *AND* we want the FA to be fully specified (total δ). So, if it is D what we have picked up at state 0, then we go to trap, never to exit from it (inputs L or D keep you in T, which is NOT an accepting state!)

▶ What if input is λ? We do not want that to be accepted either!

We are good since "0" —the start state— is NOT accepting. If λ was the string provided as input (not something starting with D), *then immediately 0 "sees" eof and halts. "0" being not accepting, λ is rejected!*

Finally, let us familiarise a bit more with eof.

This is not a unique end marker but is *context dependent*. In the context of variable names, in something like

$$LLLDDD++$$

(in C++) the first + is eof as it is not in the alphabet of our scanner FA! Ditto if we had

$$LDDD := (LDLDDD + LLL)$$

in, say, Pascal. The first variable "$LDDD$" has ":" as eof. The second one "$LDLDDD$" has "+" as eof. The third one "LLL" has ")" as eof. $\square$

10.2.13 Proposition. *If M is a FA, then $\lambda \in L(M)$ iff q_0 —the start state— is an accepting state.*

Proof. First, say $\lambda \in L(M)$.

By 10.2.11, we have a path labeled λ from q_0 to some accepting p.

Since there are no symbols in λ to consume *the only application of "read"* gave us eof and we are still at q_0. Thus $q_0 = p$ must be accepting.

Conversely, let q_0 is accepting.

The input stream looks like $\lambda¶$, where I generically indicated eof by "¶" for emphasis/visibility. This ¶ is scanned by q_0 and halts the machine right away.

But q_0 is accepting and λ is what was consumed *before* hitting eof. Thus λ is accepted: $\lambda \in L(M)$. □

10.2.14 Example.

Here is another example that we promised. Refer to Example 10.2.5. Consider the case where q_0 is accepting. Then the only possible acceptable strings x will have an even number of 1s —even parity— since to go from q_0 back to q_0 we need to consume a 1 going and a 1 coming.

But do we get an arbitrary string *otherwise*? Yes, since between any two consecutive 1s —and before the first 1 and after the last 1 we can consume any number of 0s.

Clearly, if q_1 was the accepting state instead, then we have an odd number of 1s in the accepting path since to end on q_1 as accepting state we need one 1, or three, or five, ... We add two 1s every time to leave q_1 and to go back. □

10.2.15 Remark. BTW, for any M, the set $L(M)$ —considered as a set of *numbers* since the symbols in the alphabet are essentially *digits*— is decidable!

The question $x \in L(M)$ is decided by the FA M itself: $x \in L(M)$ iff we have an accepting computation of M with input x. Cf. 10.2.11.

Wait! Is not decidability defined in terms of URMs? Yes, but an FA is a special case of a URM! □

Chapter 11

FA and NFA; Part II

11.0.1 Definition. If M is a FA, then its $L(M)$ is called the **regular set** associated with M, or even the **regular language** *recognised/accepted* (**decided**, actually) by M. □

This chapter continues from where Chapter 10 left but we will present first a few more simple examples of automata* that decide/accept some given set of strings over some alphabet.

*Plural of automaton.

11.1. Examples

11.1.1 Example. We want to specify (to "program"!) an automaton M over $\Sigma = \{0, 1\}$, such that $L(M) = \{0^n 1 : n \geq 0\}$.

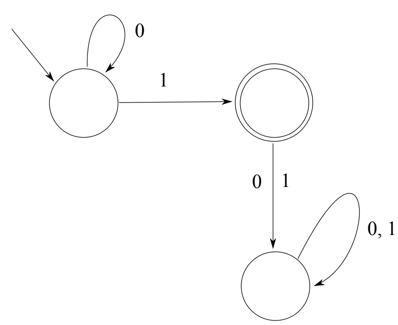
We recall that, for any string x, $x^0 \stackrel{Def}{=} \lambda$, while

$$x^{n+1} \stackrel{Def}{=} x^n * x \stackrel{\text{induction!}}{=} \overbrace{x * x * \ldots * x}^{n+1 \text{ copies of } x}$$

where I denoted *concatenation* by $*$. Thus the strings in $\{0^n 1 : n \geq 0\}$ are

$$1, 01, 001, 0001, 00001, \ldots \tag{1}$$

We readily see that the following automaton's **only accepting paths** will follow zero or more times the "loop" labeled 0 (attached to the start state), and then follow the edge labeled 1 to end up with an accepting state.



The state at the very bottom is a trap state. What is the need for it?

Well, the FA must be fully specified, so I am obliged to say what the accepting state does when it sees *one or the other legal input*.

And remember: Accepting states do NOT stop the machine! Any state stops the machine IFF it has just scanned *eof*.

A new thing we learnt in the above example is that in depicting an automaton as a graph we do *not* necessarily need to *name* the states!

Of course, as in all mathematical arguments, we will of course assign names to objects (in particular to states) **if we need to refer to them** in the course of the argument —it is convenient to refer to them by name!

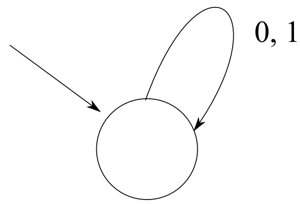
The reader should also note the use of *two shorthand notations in labeling*:

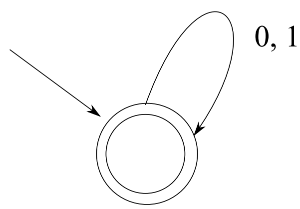
One, we used two labels on the vertical down-pointing edge.

This abbreviates the use of *two* edges going from the accepting to the trap state, one labeled 0, the other 1.

We could also have used the label "0, 1" both at the left or right of the arrow, "," serving as a separator. This latter notational convention was used in labeling the loop attached to the trap state. □

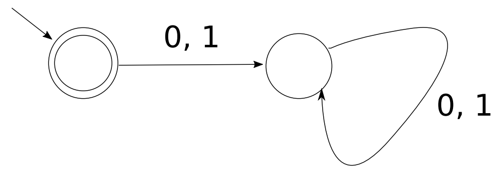
11.1.2 Example. The two FAs below, each over the input alphabet $\{0, 1\}$, accept the languages $\emptyset$ (the top one) and $\{0, 1\}^*$ (the bottom one).





$\square$

11.1.3 Example. The FA below over the input alphabet $\{0, 1\}$ accepts the language $\{\lambda\}$.



Indeed, we saw in Chapter 10 that making the start (initial) state also accepting we do accept λ. Moreover, the FA above accepts nothing else since any input symbol leads to the rejecting *trap* state. $\square$

11.2. Some Closure Properties of Regular Languages

11.2.1 Theorem. *The set of all regular languages over an alphabet Σ is closed under complement. That is, if $L \subseteq \Sigma^*$ is regular, then so is $\overline{L} \stackrel{Def}{=} \Sigma^* - L$.*

Proof. Let $L = L(M)$ for some FA M over input alphabet Σ and state alphabet Q. Moreover, let $F \subseteq Q$ be the set of accepting states of M.

We need a FA that recognises/decides $\overline{L}$.

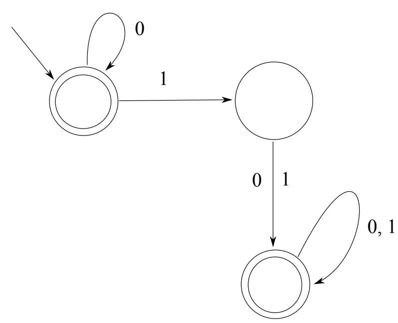
Trivially, *we want to swap the "yes" (accepting state) and "no" (rejecting state) behaviour of M,* changing nothing else.

Thus, $\overline{L} = L(\widetilde{M})$, where the FA $\widetilde{M}$ is the same as M, except that $\widetilde{M}$'s set of accepting states is $Q - F$. □

What makes the above proof tick is that FA are "**total**": Every input string will be scanned all the way to its eof. Only the yes/no decision changes.

What does "total" have to do with this? On the blackboard!

11.2.2 Example. The automaton that accepts the complement of the language in Example 11.1.1 is found without comment below, just following the construction of the $L(M)$ complement for some FA M, given above.



□

11.2.3 Theorem. *The set of all regular languages over an alphabet Σ is closed under union. That is, if $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ are regular, then so is $L \cup L'$.*

Proof. This proof will *wait* until after the introduction of NFA which make the proof much easier! □

11.2.4 Corollary. *The set of all regular languages over an alphabet Σ is closed under intersection. That is, if $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ are regular, then so is $L \cap L'$.*

Proof. $L \cap L' = \overline{\overline{L} \cup \overline{L'}}$. □

11.3. *Proving Negative Results for FA; Pumping Lemma*

Nov. 21, 2022

Is there a FA M such that $L(M) = \{0^n 1^n : n \geq 0\}$?

How can we tell?

Surely, not by trying each FA (*infinitely many*) out there as a possible fit for this language!

The following theorem, known as the *pumping lemma* can be used to prove "negative" results such as: There is no FA M such as $L(M) = \{0^n 1^n : n \geq 0\}$. In short, the language $\{0^n 1^n : n \geq 0\}$ is **not regular**.

11.3.1 Theorem. (Pumping Lemma) *If the language S is regular, i.e., $S = L(M)$ for some FA M, then there is a constant C that we will refer to as* A *pumping constant such that for any string $x \in S$, if $|x| \geq C$, then we can decompose it as $x = uvw$ so that*

(1) $v \neq \lambda$

(2) $uv^i w \in S$, *for all* $i \geq 0$ *—by definition,* $v^0 = \lambda$

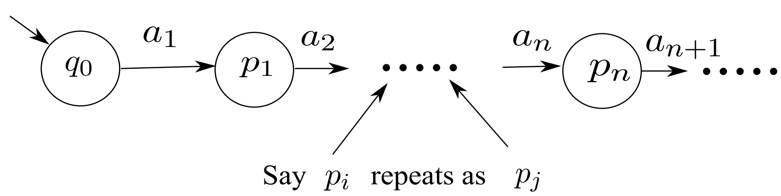
and

(3) $|uv| \leq C$.

A pumping constant is *not* uniquely determined by S.

Proof. So, let $S = L(M)$ for some FA M of n states. We will show that if we take $C = n$[†] this will work.

Let then $x = a_1 a_2 \cdots a_n \cdots a_m$ be a string of S. As chosen, it satisfies $|x| \geq C$. An accepting computation path of M with input x looks like this:



where $p_1, p_2, \ldots$ denotes a (notationally) convenient renaming[‡] of the states visited after q_0 in the computation.

[†]You see why C is not unique, since for any S that is an $L(M)$ we can have infinitely many different M that accept S. Can we not?

[‡]Why rename? What is wrong with $q_1, q_2, \ldots$? Well, the set Q is *given* as something like $\{q_0, q_1, q_2, q_3, \ldots\}$ using some arbitrary fixed enumeration order without repetition for its members. Now, it would be wrong to expect that the arbitrary input x caused the FA to walk precisely along q_1, q_2, q_3, etc., *after* it saw the first symbol of x.

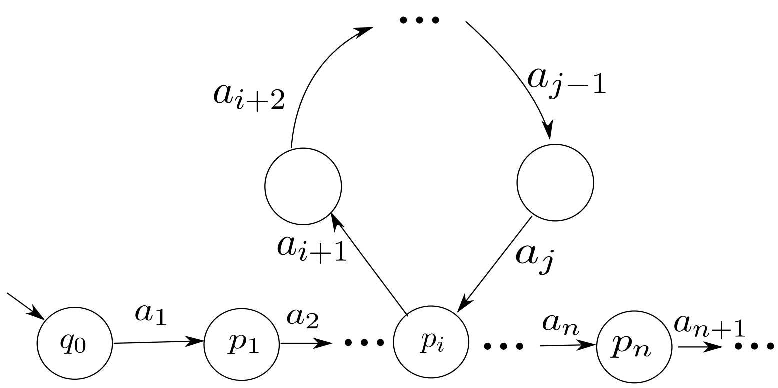
In the sequence

$$q_0, p_1, p_2, \ldots, p_n$$

we have named $n+1$ states, while we only have n states in the FA's "Q".

Thus, *at least two names "p_i and p_j" OR "q_0 and p_j" refer to the same state* "q_r" —as we say, two states repeat.

We may redraw the computation above as follows taking *without loss of generality* that $p_i = p_j$ indicating the repeating state $p_i = p_j$:



We can now partition x into u, v and w parts from the picture above: We set

$$u = a_1 a_2 \ldots a_i$$

$$v = a_{i+1} a_{i+2} \ldots a_j$$

and

$$w = a_{j+1} a_{j+2} \ldots a_m$$

Note that

(1) $v \neq \lambda$, since *there is at least one edge (labelled a_{i+1})* emanating from p_i on the sub-path that connects this state to the (identical) state p_j. *In short a_{i+1} is part of v.*

(2) We may utilize the loop v *zero* or *times* (along with u in the front and w at the tail) to always obtain a NEW *accepting* path. Thus, all of uv^iw belong to $L(M)$ —i.e., S.

(3) *Since $|uv| = j \le n$, we have also verified that $|uv| \le C$.* □

The repeating pair p_i, p_j may occur anywhere between q_0 and p_n. A different "graphical proof" is common in the literature.

Let $x = a_1a_2 \ldots a_m$ as above.

Below we show the *original* x as an input stream array

$$x = a_1 \ldots a_{i+1} \ldots a_j a_{j+1} \ldots a_n \ldots a_m$$

where the repeating $p_i = p_j$ is shown.

$$x = \overbrace{\boxed{a_1}\boxed{a_2}\boxed{\ldots}\boxed{a_i}}^{u}\underbrace{\overset{\overset{p_i}{\downarrow}}{\boxed{a_{i+1}}}\boxed{\ldots}\boxed{a_j}}_{v}\overbrace{\underset{\underset{p_j}{\uparrow}}{\boxed{a_{j+1}}}\boxed{\ldots}\boxed{a_n}\boxed{a_{n+1}}\boxed{\ldots}\boxed{a_m}}^{w}$$

Observe:

1. By determinism, the subcomputation that starts at symbol a_{j+1} (blue) —while in state p_j ($= p_i$)— will end at the eof after consuming the string w and will be uniquely at (accepting) state p_m.

2. After consuming the prefix $a_1 \ldots a_i$ of x the FA is uniquely at state p_i.

3. By determinism, the subcomputation that starts at symbol a_{i+1} (red) in state p_i, will consume v and end at p_j —uniquely, today, tomorrow and in 10^{350000} years from now— ready to process a_{j+1} (blue).

Thus, all of $uv, uvvw, uvvvw, \ldots, uv^nw, \ldots$ are in $L(M)$. Of course, so is $x = uvw$ (given).

11.3.2 Example. The language over $\{0, 1\}$ given as $L = \{0^n 1^n : n \geq 0\}$ is not regular.

Suppose it is. Then the pumping lemma holds for L, so let C be an appropriate pumping constant and consider the string $x = 0^C 1^C$ of L. We can then decompose x as uvw with $|uv| \leq C$ so that we can "pump" $v \neq \lambda$ as much as we like and the obtained $uv^i w$ will all be in L.

We will prove the statement in red false, so we cannot pump; *but then L cannot be regular!*[§]

"*The red statement*" is false due to the observations:

1. By $|uv| \leq C$, uv (and hence v) lie entirely in the 0^C-part of the chosen $x = 0^C 1^C$.
2. So, if we *pump down* —or above with $i \geq 2$ (i.e., use v^0 or $v^i, i \geq 2$) we obtain $uw \in L$ or $uv^i w \in L, i \geq 2$.

But $uw = 0^K 1^C$ where $K < C$ since $|v| \geq 1$. However such unbalanced 0-1 strings cannot be in L, by specification, so we contradicted the pumping lemma. □

[§]All sufficiently long strings of regular languages can be pumped by 11.3.1 and stay in L.

All proofs by Pumping Lemma 11.3.1 are by contradiction and they prove *non acceptability* by *any* FA (or, equivalently, NFA to be introduced in Section 11.4.1).

11.3.3 Example. *We introduced FA as special URMs that cannot write.*

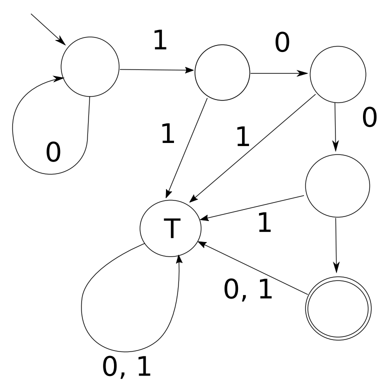
Is it then an immediate conclusion that they cannot compute functions?

Not at all! Such a general conclusion is false!

For example, we can agree that by "*compute* $f(x)$" we mean " decide the graph $y = f(x)$".

For example, we can "compute" $\lambda x.3$ by accepting all strings, but no others, of the form $0^n 1000$ over the alphabet $\{0, 1\}$.

That is, we use 1 as a separator between input $n \geq 0$ (depicted as 0^n) and output 3 (depicted as 000), then the following FA **decides** (*accepts/recognises*) the language $L = \{0^n 1000 : n \geq 0\}$.



□

11.3.4 Example. FA cannot compute $\lambda x.x + 1$.

"*Surely*", you say, "*how can they add 1 if they cannot do arithmetic or write anything at all?*"

Wrong reason!

Again, how about deciding the *"graph"-language* over $A = \{0, 1\}$, given by $T = \{0^n 10^{n+1} : n \geq 0\}$?

Here "0^n" represents input n, *"0^{n+1}" represents output $n + 1$* and 1 is a separator as in the previous example.

▶ Alas, no FA can do this.

Say T is FA-decidable, and let C be an appropriate pumping constant. Choose $x = 0^C 10^{C+1}$. Splitting x as uvw with $|uv| \leq C$ *we see that* 1 *is to the right of* v.

v is all zeros.

Thus, uw (using v^0) is not in T since the "$n/n+1$ relation" between the 0s to the left and those to the right of 1 is destroyed —we have $0^K 10^{C+1}$ with $K < C$ in the language by the PL! This contradicts the assumption that T is FA-decidable. □

11.3.5 Exercise. Indeed FA cannot even compute the identity function, $\lambda x.x$, as it should be clear from the proof in 11.3.2. Adapt that proof to show the graph language for $\lambda x.x$, namely, $\{0^n 10^n : n \geq 0\}$ is not regular. □

11.3.6 Example. The set over the alphabet $\{0\}$ given by $P = \{0^q : q$ is a prime number$\}$ is not FA-decidable.

> A string of 0s is in language P iff it has prime length: Note $|0^Q| = Q$.

Assume the contrary, and let C be an appropriate pumping constant. Let $Q \geq C$ be prime.

We show that considering the string $x = 0^Q$ will lead us to a contradiction. Well, as x is longer than C, let us write —according to 11.3.1— $x = uvw$.

$$\boxed{\text{Note that } |x| = |uvw| = |u| + |v| + |w|.}$$

By PL, we must have that *all* numbers $|u| + i|v| + |w|$, for $i \geq 0$ *are prime*. These numbers have the form

$$ai + b \tag{1}$$

where $a = |v| \geq 1$ and $b = |u| + |w|$. Can REALLY *ALL these numbers in (1)* (for all i) *be prime*?

Here is WHY NOT, and hence our contradiction. We consider cases:

- Case where $b = 0$. This is impossible, since the numbers in (1) now have the form ai. But, e.g., $a4$ is not prime.
- Case where $b > 0$. We have Subcases!
  - Subcase $b > 1$. Then taking $i = b$, one of the numbers of the form (1) is $(a + 1)b$. But $(a + 1)b$ is not prime (recall that $a + 1 \geq 2$ since $a \geq 1$).

– Subcase $b = 1$. Then take $i = 2 + a$ to obtain the number (of type (1)) $a(2 + a) + 1 = a^2 + 2a + 1 = (a + 1)^2$. But this is not prime! $\square$

The preceding shows that *we can have a set that is sufficiently complex and thus fails to be FA-decidable even over a single-symbol alphabet*.

Here is another such case.

11.3.7 Example. Consider $Q = \{0^{n^2} : n \geq 0\}$ over the alphabet $A = \{0\}$. It will *not* come as a surprise that Q is not FA-decidable.

For suppose it is. Then, if C is an appropriate pumping constant, consider $x = 0^{C^2}$.

- Clearly, $x \in Q$ and is long enough.

So, split it as $x = uvw$ with $|uv| \leq C$ and $v \neq \lambda$.

Now, by 11.3.1,

$$uvvw \in Q \tag{1}$$

But

$$C^2 = |uvw| \overset{|v|\geq 1}{<} |uvvw| \leq |uvw| + |uv| \leq C^2 + C \overset{by\ +1}{<} C^2 + 2C + 1 = (C+1)^2$$

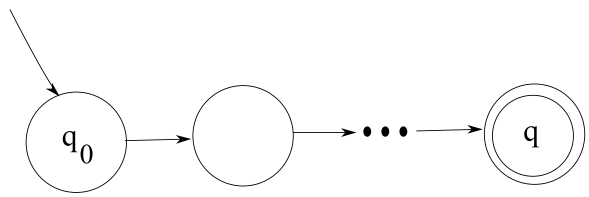
Thus, the number $|uvvw|$ is NOT a perfect square *being between two successive ones*.

But this will not do, because by (1), for some n, we *must* have $uvvw = 0^{n^2}$ and thus $|uvvw| = n^2$ —a perfect square after all! □

11.4. Nondeterministic Finite Automata

The FA formalism provides us with tools to finitely define certain languages:

Such a language —defined as an $L(M)$ over some alphabet A, for some FA M— contains a string x iff there is an *accepting path* —within the FA— *whose labels from left to right form x*.



The computation above, that is, the path labeled x within the FA, is uniquely determined by x since the automaton is deterministic.

Much is to be gained in *theoretical and practical flexibility* if we **relax both** "deterministic" requirements NFA 1) and NFA 2) below

NFA 1) Every state is defined on all inputs from the input alphabet (totaleness)

NFA 2) No state has two different responses (i.e., does not send the process *to either of two different* states) *for the same input*.

AND moreover we profit —*theoretically and practically*— from *ADDING* the feature

NFA 3) The automaton can have *empty-moves*, that is, λ*-moves*, *meaning* it can go from state q to state p *WITHOUT CONSUMING ANY INPUT*.

An *empty move* from q to p is depicted in the flow diagram as:

$$\textcircled{q} \xrightarrow{\lambda} \textcircled{p}$$

11.4.1 Definition. (NFA) A so relaxed FA —that is also *augmented* by the feature "NFA 3)" above— is called *Non Deterministic Finite Automaton*, in short, *NFA*.

An NFA M accepts a string x iff there is a *path from its start state* (generically depicted as) "q_0" *to some accepting state* p whose edge-labels concatenated from q_0 toward p in order *form the string* x.

Of course empty moves do not contribute to the path name!

IMPORTANT! *Every FA is also an NFA* —but NOT vice versa— since the enhancements in *NFA 1) – NFA 3)* above are NOT compulsory!

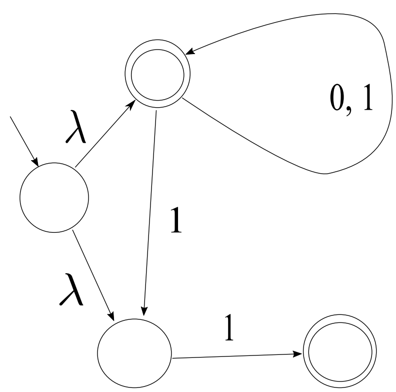
□

11.4.2 Example. The displayed flow diagram below, over the alphabet $\{0,1\}$, incorporates all the liberties in notation and *conventions introduced in Definition 11.4.1* and the *items NFA 1) – NFA 3) preceding the definition*.

We have two λ moves, and the string "1" can be accepted *in two distinct ways*: One is to follow the top λ move, and then go *once* around the loop, consuming input 1. The other is to follow the bottom λ move, and then follow the transition labeled 1 to the accepting state at the bottom (reading 1 in the process).

Folklore jargon —not based on science or theory— will have us speak of *guessing* when we describe what the diagram does with an input.

For example, to accept the input 00 one would say that *the NFA guesses* that it should follow the upper λ, and then it would go twice around the top loop, on input 0 in each case.



This diagram is an example of a *nondeterministic finite automaton*, or NFA;

- it has λ moves,
- its transition relation —as depicted by the arrows— is not a function (e.g., the top accepting state has *two distinct responses on*

input 1),

- nor is it total.

  For example, the bottom accepting state is not defined on any input; nor is the start state: λ *is not an input!* □

Returning to the issue of *guessing*, we emphasize that this use of this term is an unfortunate habit in the literature.

Nobody and Nothing guesses Anything!

A NFA simply provides the *mathematical framework* within which *we* can formulate and verify an *existential MATHEMATICAL statement* of the type

$$\textit{for a given input } x\textit{, an accepting path} \text{ exists} \tag{1}$$

Given an acceptable input, *the NFA does NOT actually guess* "correct" moves (from among a set of choices), either in a hidden manner (*consulting the Oracle in Delphi, for example!*), or in an explicit computational manner (e.g., parallelism, backtracking) toward *finding* an accepting path for said x.

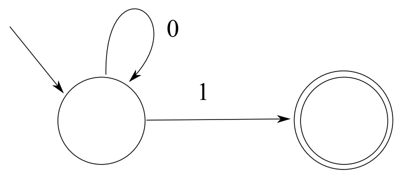
Simply, the NFA formalism allows *us* to *state* —and provides tools so that we can *verify*— the statement (1) above by verifying an accepting path exists! (11.4.1)

This is analogous with the fact that *the language of logic allows us to state statements such as* $(\exists y)\mathscr{F}(y,x)$, *and offers tools to us to prove them*.

In the case of NFA, an independent agent, which *could be ourselves* or a FA —YES, we will see that *every NFA can be simulated by some FA!*— can effect the verification that indeed an accepting path labeled x exists.

Nov. 23, 2022

11.4.3 Example. The following is a NFA but not a FA (why? Compare with 11.1.1), which accepts the language $\{0^n 1 : n \geq 0\}$.

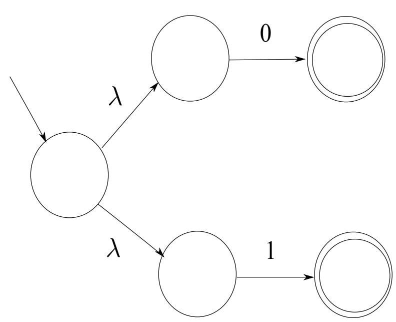


□

11.4.4 Example. NFA are much easier to construct than FA, partly because of the convenience of the λ moves, and the ability to "guess" (cf. earlier discussion about "guessing").

Also, partly due to *lack of concern for totalness*: we *do not have to worry about "installing" a trap state*.

For example, the following NFA over $A = \{0, 1\}$ decides/recognises just its alphabet A and nothing else as we can trivially see that there are just two accepting paths: one named "0" and one named "1".



□

11.5. From FA to NFA and Back

We noted earlier that any FA is a NFA (Def. 11.4.1), thus *the NFA are at least as powerful as the FA*.

They can do all that the deterministic model can do.

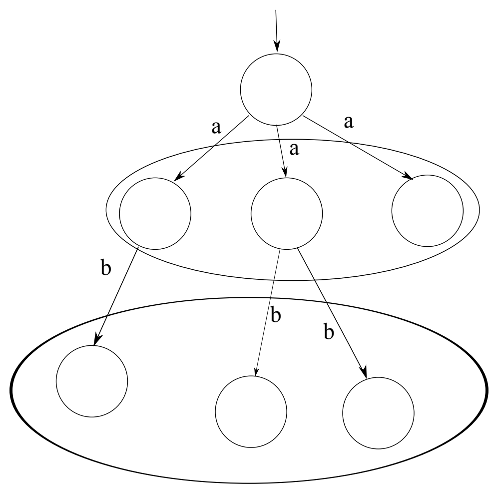
It is *a bit of a surprise* that the opposite is also true: *For every NFA M we can* construct *a FA N, such that $L(M) = L(N)$.*

Thus, in the case of these very simple machines, nondeterminism ("guessing") buys *ONLY convenience*, <u>but not real power</u>.

How does one simulate a NFA on an input x?

The most straightforward idea is to trace *ALL possible paths* labeled x (due to nondeterminism they may be more than one —or none at all) *in parallel* and accept iff one (*or more*) of those is accepting.

The principle of this idea is illustrated below.



Say, the input to the NFA M is $x = ab\ldots$ Suppose that a leads the start state —which is at "level 0"— to three states; we draw all three. These are at level 1.

We repeat for *each state at level 1* on input b:

Say, for the sake of discussion, that, of the three states at level 1, *the first leads to one state* on input b, *the second leads to two* and *the third leads to none*.

We draw these three states obtained on input b; they are at level 2. Etc.

An FA can keep track of all the states at the various levels since **at ANY level, they can be no more than the totality of states of the NFA M!**

Thus, the amount of information at each level is *independent of the input size* —i.e., it is a constant— and moreover can *be coded as a single FA-state* (*depicted in the figure by an ellipse*) that uses a *"compound" name*, *consisting of all the NFA state names at that level*.

This has led to the idea that *the simulating FA must have as states nodes whose names are* sets of *state names of the original NFA*.

Clearly, for this construction, *state names are important*, through which we can keep track of and describe what we are doing.

Here are the details:

11.5.1 Definition. **(a-successors)** Let M be a NFA over an input alphabet Σ, q be a state, and $a \in \Sigma$.

A state p is *AN a-successor* of q iff there is an edge from q to p, *labeled a*. □

In a NFA a-successors need not be unique, nor need to exist —for all pairs (q, a).

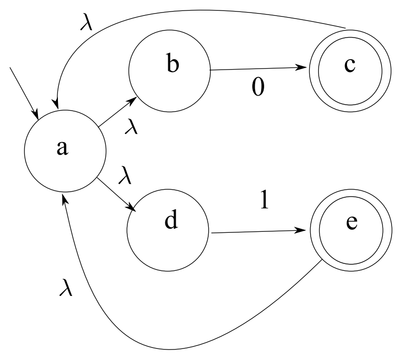
On the other hand, *in a FA they exist and are unique*.

11.5.2 Definition. **(λ-closure)** Let M be a NFA with state-set Q and let $S \subseteq Q$. The λ-closure of S, denoted by $\lambda(S)$, is defined to be *the smallest set that includes S* but *also includes all $q \in Q$, such that* there is a path, named λ *—we call such a path a "λ-path"— from some $p \in S$ to q.*

When we speak of the λ-closure of ONE state q, we mean that of the set $\{q\}$ and write $\lambda(q)$ rather than $\lambda(\{q\})$. □

Note that a path named λ will have **all** its edges named λ since the concatenation of a sequence of strings is λ iff each string in the sequence is.

11.5.3 Example. Consider the NFA below.



We compute some λ-closures: $\lambda(a) = \{a, b, d\}$; $\lambda(c) = \{c, a, b, d\}$. $\square$

11.5.4 Theorem. *Let M be a NFA with state set Q and input alphabet Σ. Then there is a FA N that has as state set a subset of $\mathcal{P}(Q)$ —the power set of Q— and the same input alphabet as that of M.*

N satisfies $L(M) = L(N)$.

We say that two automata M and N (whether both are FA or both are NFA, or we have one of each kind) are *equivalent* iff $L(M) = L(N)$.

Thus, the above says that *for any NFA there is an equivalent FA*.

In fact, this can be strengthened as the proof shows: We can *construct* the equivalent FA.

We show how *in the definition below, BEFORE we start the proof proper.*

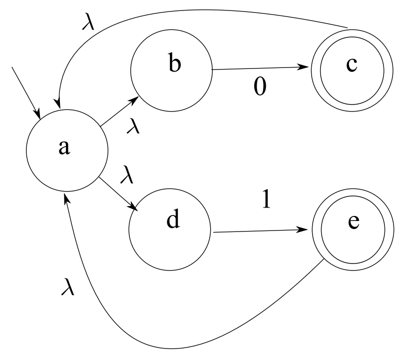
11.5.5 Definition. (NFA to FA Construction)

- The start state of N is $\lambda(q_0)$, where q_0 is the start state of NFA M.
- A state of the FA N is accepting **iff** its *name* contains at least one accepting state *name* of the NFA M.
- Let S be a state of N and let $a \in \Sigma$. The unique a-successor of S in N is constructed as follows:

  (1) Construct the set of *all* a-successors in M of *all component-names* of S. Call T this set of a-successors.

  (2) Construct $\lambda(T)$; this is *the* a-successor of state S in N. □

As an illustration, we compute some *0-successors* in the FA constructed as above if the given NFA is that of Example 11.5.3, reproduced also below:



(I) For state $\{a, b, d\}$ step (1) yields $\{c\}$. Step (2) yields the λ-closure of $\{c\}$: The state $\{c, a, b, d\}$ is the 0-successor. This state is accepting in the FA since the NFA has c as accepting.

(II) For state $\{c, a, b, d\}$ step (1) yields $\{c\}$. Step (2) yields the λ-closure of $\{c\}$: The state $\{c, a, b, d\}$ is the 0-successor; that is, *the 0-edge loops back to where it started*: at state $\{c, a, b, d\}$.

We do NOT draw a new copy of $\{c, a, b, d\}$!

Proof. Of 11.5.4.

With the FA N constructed as in 11.5.5 from the NFA M, we need to prove two things:

Direction 1. $L(M) \subseteq L(N)$.

Direction 2. $L(N) \subseteq L(M)$.

- $L(M) \subseteq L(N)$ *direction*:

Let

$$x = a_1 a_2 \cdots a_n \in L(M) \tag{1}$$

$$\text{Prove that } x \in L(N) \tag{2}$$

Without loss of generality, we have an accepting path in M that is labeled as follows:

$$x = \lambda^{j_1} a_1 \lambda^{j_2} a_2 \lambda^{j_3} a_3 \cdots \lambda^{j_n} a_n \lambda^{j_{n+1}} \tag{3}$$

where each λ^{j_i} depicts $j_i \geq 0$ consecutive path edges, each labeled λ, *where $j_i = 0$ in this context means that the j_i group has no λ-moves —that is, there is NO "λ^{j_i}" between a_{i-1} and a_i.*

An accepting path for the exact string —λ and all— in (3) is the zig-zag path depicted in Fig. 11.2 below.

To prove (2) we need a path in FA N *the edges of which are labeled by the a_i in $x = a_1 a_2 \ldots a_n$ in the indicated order*, while the nodes are states of N with the first node being the initial one, and that last one is an accepting state.

Here is how to do this:

▶ It suffices to show that for each level $i = 0, 1, 2, \ldots$, the N-path *of N-states and labelled edges*, consists of the indicated ellipses in Fig. 11.2 —and partially shown in Fig. 11.1— which contain in their name the enclosed "*horizontal*" M-nodes shown. *Why "suffices"? Read on!*

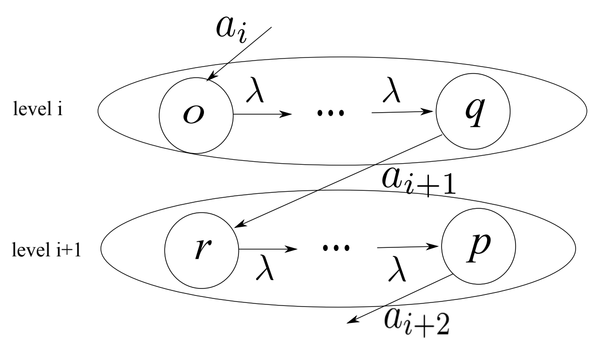


Figure 11.1: Idea for $L(M) \subseteq L(N)$ proof

▶ Regarding the above Figure, if we assume that at level i the elliptical N-node indeed includes all the indicated M-nodes in its name (*these are M-nodes from the M-computation!*), then so does the N-node at level $i+1$ —that is, the *a_{i+1}-successor of the N-node at level i.*

This is so by Def. 11.5.5 since at level $i+1$ we have the λ-closure of all a_{i+1}-successors —in M. But r is ONE such successor and thus all horizontal nodes will be in the name too! ◀

Now, clearly, $\lambda(q_0)$, THE start state of N —depicted by the level-0 ellipse in Fig. 11.2— will contain all horizontal nodes shown.

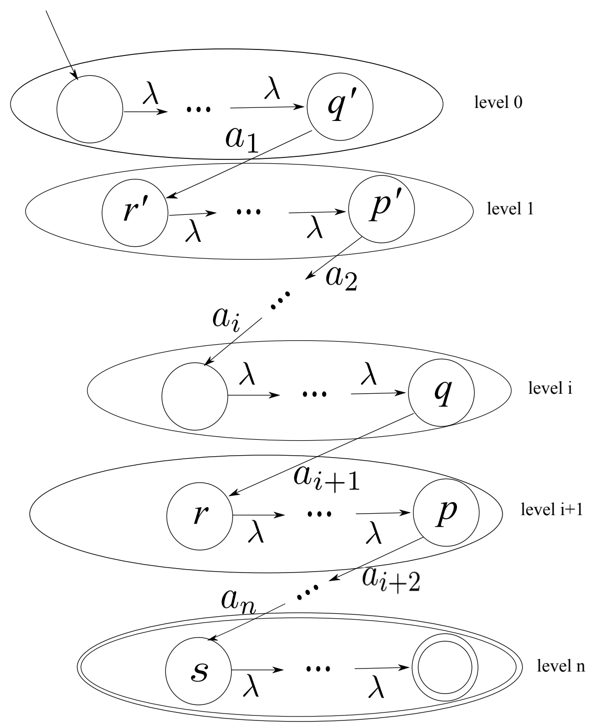


Figure 11.2: Equivalence of NFA and FA

Then —by (1) and (3) on p.313— the next ellipse (N-state) at *level 1* must contain r' (by Def. 11.5.5) and hence also *all horizontal nodes (as sub-names) shown at level 1*.

By the "Induction step" in the ► ◄-passage on p.314 all depicted elliptical nodes of N in Fig. 11.2 contain the nodes from M shown.

Clearly Fig. 11.2 depicts and FA N-computation that consumes $x = a_1a_2 \ldots a_n$ and ends with an accepting N-state. It is *ACCEPTING* because it *contains in its name an accepting M-state*. All in all: $x \in L(N)$. We proved (2) (p.313).

- *$L(N) \subseteq L(M)$ direction*:

$$\text{So let } x = a_1a_2 \ldots a_n \in L(N) \text{ this time.} \tag{$\dagger$}$$

We will argue that also

$$x \in L(M) \tag{$\ddagger$}$$

We will reuse Fig. 11.2.

Observe that by ($\dagger$) we have a path in N from the elliptical start-state to some *accepting elliptical N-state*.

We will construct an *accepting path for x in the NFA M*.

In our construction we start from the end (accepting N-state) and *proceed BACKWARDS towards the N start state*.

All the work is shown in Fig. 11.2 where now we retrace the M-path backwards.

OK. The accepting N state must have an *accepting NFA state* in its name.

▶ How did this get there?

– Either as an a_n-successor in NFA M of some name *found in the elliptical state immediately above*,

or, more generally,

– It is at the end of a λ-path starting at an *a_n-successor in NFA M* —here named "s"— found in the last ellipse. *This general case is depicted in Fig. 11.2*.

To understand *how the construction propagates UPWARDS* (BACKWARDS) imagine that $a_n = a_{i+2}$.

Then the question is "whose a_{i+2}-successor (in M) is s? *Well, we named it p in Fig. 11.2*.

The next question is: "How did p get in the name of the ellipse at level $n - 1 = i + 1$?"

Well, as above,

– Either as an a_{i+1}-successor in NFA M of some name *found in the elliptical state immediately above* —at level $n - 2 = i$, or, more generally,

– p is at the end of a λ-path starting at an *a_{i+1}-successor r in NFA M* found in the last ellipse. *This general case is depicted in Fig. 11.2*.

Continuing the construction like this we find that the presence of q' in the start state of N is either that it is the same as the state "q_0" of the NFA M, OR q' is connected to q_0 by a backwards λ-path, in general, as depicted in Fig. 11.2.

We have just constructed a path labelled

$$\lambda^{j_1} a_1 \lambda^{j_2} a_2 \lambda^{j_3} a_3 \cdots \lambda^{j_n} a_n \lambda^{j_{n+1}} = a_1 a_2 a_3 \cdots a_n$$

in the NFA M from its q_0 to some accepting state!

Thus $x \in L(M)$. $\square$

In theory, to construct a FA for a given NFA we draw all the states of the latter —*named by ALL subsets of the state-set Q of the NFA*— and then determine the interconnections via edges, for each state-pair of the FA and each member of the input alphabet Σ.

In practice we may achieve significant economy of effort if we start building the FA "*from the start state down*": That is, *starting with the start state (level 0)* we determine *all* its (elliptical) a-successors, *for each $a \in \Sigma$*.

At the end of this step we will have drawn all states at "level 1".

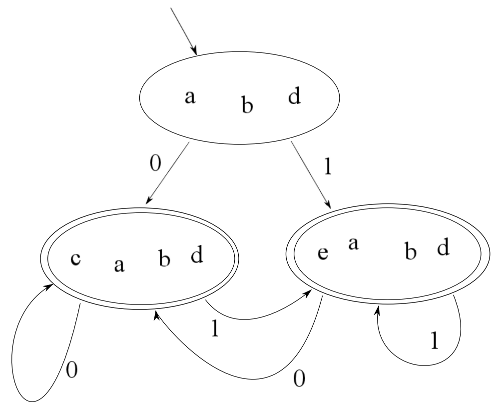
In the next step for each state at level 1, draw its a-successors, for each $a \in \Sigma$. And so on.

This sequence of steps *terminates* since there are only a *finite number of states in the FA* and *we cannot keep writing* new *ones*

Sooner or later we will stop introducing *new* states: edges will point *"back" to existing states*.

See the following example.

11.5.6 Example. We convert the NFA of 11.5.3 to a FA. See below, and review the above comment and the proof of 11.5.4, in particular the three bullets on p.311, to verify that the given is correct, and follows procedure.



Nov. 28, 2022

You will notice the aforementioned economy of effort achieved by our process. We have only three states in the FA as opposed to the predicted 32 ($= 2^5$) of the proof of Theorem 11.5.4. *But what happened to the other states?* Why are they not listed by our procedure?

Because *OUR procedure* only constructs FA states that *are accessible FROM the start state via a computation path*.

These are the only ones that can possibly participate in an accepting path. The others —the non-accessible ones— are irrelevant to accepting computations —indeed to any computations that start with the start state— and can be omitted without affecting the set decided by the FA. □

11.5.7 Example. Suppose that we have converted a NFA M into a FA N.

Let a be in the input alphabet.

What is the a-successor of the state named $\emptyset$ *in* N?

Well, there are no states in $\emptyset$ *to start the deterministic a-successor process* of Def. 11.5.5!

So the set of successor states we get is empty; we are back to $\emptyset$.

Thus, the set of a-successors (*in* M) of states from $\emptyset$ is itself the empty set. In other words, the a-successor of $\emptyset$ *in* N is $\emptyset$. *The edge labeled a loops back to it*.

Therefore, in the context of the NFA-to-FA conversion, $\emptyset$ is a *trap state* in N. □

Chapter 12

Regular Expressions

The FA and NFA of the previous Notes provide finite descriptions of regular languages, since an FA/NFA M is finite (a graph, say) and a regular language is an $L(M)$ for some M.

The next section proposes *another type of finite description* of regular languages.

12.1. Regular Expressions

Regular expressions are familiar to users of the UNIX operating system.

They are *names* for regular sets as we will see.

- Do they name ALL regular sets, i.e., all sets of the type $L(M)$ where M is a FA (or NFA, equivalently)?
- Do they name any NON regular sets?

We will see that we must answer YES, NO.

Regular Expressions are more than "just names" as they *embody enough information* —as we will see— to be *mechanically transformable* into an NFA (and thus to a FA as well).

12.1.1 Definition. (Regular expressions over Σ) Given the *finite alphabet of atomic symbols* Σ, we form the *extended alphabet*

$$\Sigma \cup \{\emptyset, +, \cdot, *, (,)\} \tag{1}$$

where the symbols $\emptyset, +, \cdot, *, (,)$ (not including the comma separators) are all abstract or *formal*[*] —are just names— and *do not occur in* Σ.

In particular, "$\emptyset$" in this alphabet is just a symbol —do NOT interpret it! (Yet!)

So are "+", "$\cdot$", "$*$" and the brackets. *All these symbols will be* interpreted *shortly*.

The set of *regular expressions over* Σ is *a set of strings over the augmented alphabet above*, given inductively by

> **Regular expressions are specific names, formed as strings over the alphabet (1) as follows :**

(1) Every member of $\Sigma \cup \{\emptyset\}$ is a regular expression.

Examples for case (1): If $\Sigma = \{0, 1\}$ then $0, 1$, and $\emptyset$, all viewed as *abstract symbols* with no interpretation are each a regular expression.

(2) If α and β are *(names of) regular expressions*, then so is the string $(\alpha + \beta)$

(3) If α and β are *(names of) regular expressions*, then so is the string $(\alpha \cdot \beta)$

(4) If α is a *(name of) regular expression*, then so is the string (α^*)

[*] Employed to define form or structure.

The letters α, β, γ are used as *metavariables* (*syntactic variables*) in this definition. They will stand for *arbitrary regular expressions* (*we may add primes or subscripts to increase the number of our metavariables*). □

12.1.2 Remark.

(i) We emphasize that regular expressions are built starting from the *objects* contained in $\Sigma \cup \{\emptyset\}$.

We *also emphasize* that we have *NOT* talked about *semantics* yet, that is, we *did NOT say YET* what *sets* these expressions will *name*, nor, what "+, "$\cdot$" and "*" *mean*.

(ii) We *will often omit the "dot"* in $(\alpha \cdot \beta)$ and write simply $(\alpha\beta)$.

(iii) We *will also omit the outermost brackets so* $(\alpha \cdot \beta)$ *is simply* $\alpha\beta$.

(iv) We assign the *highest priority* to *, the *next lower* to $\cdot$ and the lowest to $+$.

We will let $\alpha \circ \alpha' \circ \alpha'' \circ \alpha'''$ group ("associate") from right to left, *for any* $\circ \in \{+, \cdot, ^*\}$.

Given these priorities, we may omit some brackets, as is usual.

Thus, $\alpha + \beta\gamma^*$ means $\Big(\alpha + \big(\beta(\gamma^*)\big)\Big)$

and $\alpha\beta\gamma$ means $(\alpha(\beta\gamma))$. □

We next define what sets these expressions name (semantics).

12.1.3 Definition. (Regular expression semantics)
We define the *semantics* of any regular expression over Σ by recursion on the Definition 12.1.1.

We use the notation $L(\alpha)$ **to indicate *the set* named *by*** α.

(1) $L(\emptyset) = \emptyset$, where the left "$\emptyset$" is the symbol in the augmented alphabet (1) above, while the right "$\emptyset$" is the *name of the empty set in ordinary MATH*.

(2) $L(a) = \{a\}$, for each $a \in \Sigma$

(3) $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$

(4) $L(\alpha \cdot \beta) = L(\alpha)L(\beta)$ —where for two languages (sets of strings!) L and L', LL' —the *concatenation of the SETS* in this order— stands for $\{xy : x \in L \wedge y \in L'\}$.

(5) $L(\alpha^*) = \Big(L(\alpha)\Big)^*$[†] —where for any set S —finite or not— S^* denotes *the set of all strings*

$$x_1 x_2 \ldots x_n, \text{ for } n \geq 0, \text{ and where all (strings) } x_i \in S$$

where $n = 0$ means that $x_1 x_2 \ldots x_n = \lambda$.

Thus, in particular, we have always $\lambda \in S^*$.

□

[†]The $*$ in S^* is called the Kleene closure. So S^* is the Kleene closure of S.

12.1.4 Example. Let $\Sigma = \{0, 1\}$. Then $L\Big((0+1)^*\Big) = \Sigma^*$. Indeed, this is because $L\Big(0+1\Big) = L(0) \cup L(1) = \{0\} \cup \{1\} = \{0, 1\} = \Sigma$. $\square$

12.1.5 Example. We note that $L(\emptyset^*) = \Big(L(\emptyset)\Big)^* = \emptyset^* = \{\lambda\}$.

Why so?

Because Σ^* is λ along with the set of all strings formed using symbols from Σ.

$\emptyset$ has no symbols to form strings with. So all we got is λ.

See last "red" comment in Def. 12.1.3.

Because of the above, we add "λ" as a *DEFINED NAME* —not in the original alphabet— for the set $\{\lambda\}$. $\square$

Of course, two regular expressions α and β over the same alphabet Σ are equal, written $\alpha = \beta$, iff they are so *as strings*.

We also have another, *semantic*, concept of regular expression "equality":

12.1.6 Definition. (Regular expression equivalence) We say that two regular expressions α and β over the same alphabet Σ are *equivalent*, written $\alpha \sim \beta$, iff they *name the same set/language*, that is, iff $L(\alpha) = L(\beta)$. $\square$

12.1.7 Example. Let $\Sigma = \{0,1\}$. Then $(0+1)^* \sim \big(0^*1^*\big)^*$.
Indeed, $L\big((0+1)^*\big) = \Sigma^*$, by 12.1.4.

So, if anything, we do have

$$L\Big((0+1)^*\Big) \supseteq L\Big((0^*1^*)^*\Big)$$

Now —for $L\Big((0+1)^*\Big) \subseteq L\Big((0^*1^*)^*\Big)$— the set

$$L\Big((\underbrace{0^*1^*}_{A})^*\Big)$$

is A^* where

$$A = L(0^*1^*) = \{0^n1^m : n \geq 0 \wedge m \geq 0\}$$

because

$$L(0^*) = L(0)^* = \{0\}^* = \{0^n : n \geq 0\}$$

and similarly for

$$L(1^*) = L(1)^* = \{1\}^* = \{1^m : m \geq 0\}$$

It should be clear that *any string of 0s and 1s can be built using as building blocks* 0^n1^m *judiciously choosing* n *and* m *values*.

E.g., $01^{10}0^{11}$ can be thought of as

$$0^11^0\,0^01^{10}\,0^{11}1^0$$

More generally, to show that an arbitrary string over Σ,

$$\ldots 0^k \ldots 1^r \ldots \tag{1}$$

is in A^* view (1) as

$$\ldots 0^k1^0 \ldots 0^01^r \ldots$$

But then the statement between the signs simply says that $\Sigma^* \subseteq L\Big((0^*1^*)^*\Big)$. Done. □

By the above example, $\alpha \sim \beta$ *does NOT imply* $\alpha = \beta$.

12.2. From a Regular Expression to NFA and Back

There is a *mechanical procedure* (*algorithm*), which from a given regular expression α *constructs* a NFA M so that $L(\alpha) = L(M)$, and conversely:

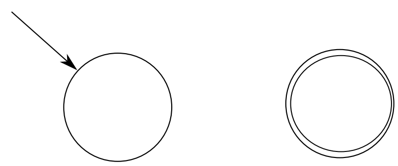
Given a NFA M constructs a regular expression α so that $L(\alpha) = L(M)$.

We split the procedure into two directions. *First, we go from regular expression to a NFA*.

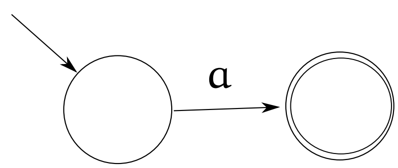
12.2.1 Theorem. (Kleene) *For any regular expression α over an alphabet Σ we can construct a NFA M with input alphabet Σ so that $L(\alpha) = L(M)$.*

Proof. Induction over the Inductive of Definition 12.1.1 —that is, on the formation of a regular expression α according to the said definition. For the basis we consider the cases

- $\alpha = \emptyset$; the NFA below works

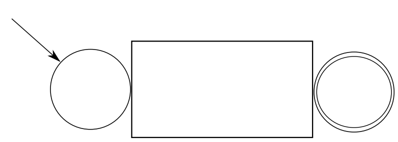


- $\alpha = a$, where $a \in \Sigma$; the NFA below works



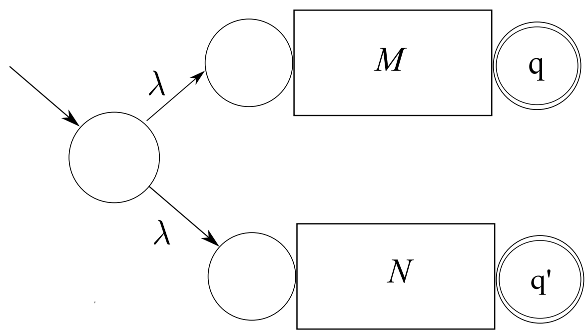
Both of the above NFA have EXACTLY ONE accepting state. *Our construction maintains this property throughout.*

That is, **all the NFA we construct in this proof will have that form**, namely

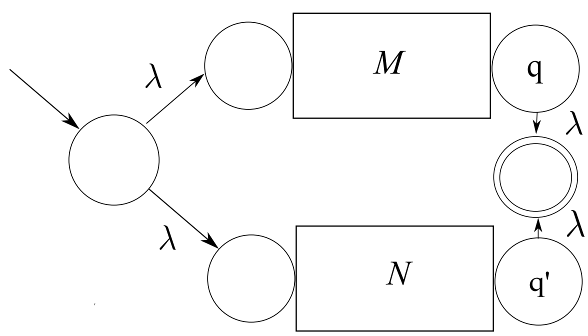


Assume now (*the I.H. on regular expressions!*) that we have built NFA for α and β —M and N— so that $L(\alpha) = L(M)$ and $L(\beta) = L(N)$. *Moreover, these M and N have the form above.* For the induction step we have three cases:

- To build a NFA for $\alpha + \beta$, that is, one that accepts the language $L(M) \cup L(N)$. The NFA below works since the accepting paths are precisely those from M and those from N.



However, to maintain the single accepting state form, we modify it as the NFA below.

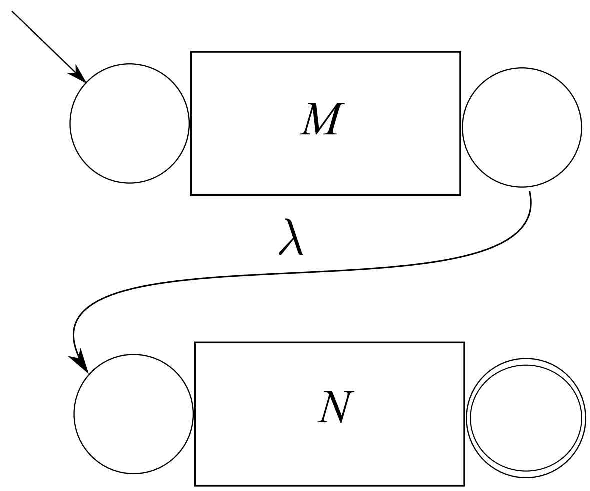


- To build a NFA for $\alpha\beta$, that is, one that accepts the language $L(M)L(N)$.

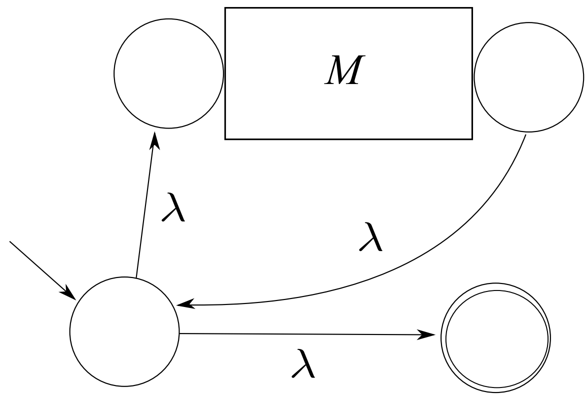
  The NFA below works —since *the accepting paths are precisely those formed by concatenating an* accepting path of M *(labeled by some $x \in L(M)$)* with *an λ-move and then with* an accepting path of N *(labeled by some $y \in L(N)$)*;

  in that left to right order.

  The λ that connects M and N will not affect the path name: $x\lambda y = xy$.



- To build a NFA for α^*, that is, one that accepts the language $L(M)^*$. The NFA below, that we call P, works. That is, $L(P) = L(M)^*$.



□

12.2.2 Theorem. (Kleene) *For any FA or NFA M with input alphabet Σ we can construct a regular expression α over Σ so that $L(\alpha) = L(M)$.*

Proof. Given a FA M (if an NFA is given, *then we convert it to a FA first*).

We will construct an α with the required properties. The idea is to express $L(M)$ in terms of simple to describe (indeed, regular themselves) sets of strings over Σ *by repeatedly using* the *operations $\cdot$, $\cup$ and Kleene star*, a finite number of times.

These regular sets —NAMEABLE by RegEXs— are called by Kleene "R_{ij}^k", where $k \leq n$ and where the state set of the FA is

$$q_1, q_2, \ldots, q_n \text{ —the same "}n\text{" as above}$$

It turns out that "$\bigcup_j R_{1j}^n$" is the set of all FA-acceptable strings, the union taken *over all accepting q_j*.

We will see that a simple regular EXPRESSION can name the above mentioned finite union of regular sets.

So let $Q = \{q_1, q_2, \ldots, q_n\}$ be the set of states of M, where q_1 is the start state.[†] We will refer to the set of M's accepting states as F.

We next define several *SETS* of strings (over Σ) —denoted by R^k_{ij}, for $k = 0, 1, \ldots, n$ and each i and j ranging from 1 to n.

$$R^k_{ij} = \{x \in \Sigma^* : x \text{ labels a path from } q_i \text{ to } q_j \text{ and every } q_m \text{ in this path, other than the endpoints } q_i \text{ and } q_j\text{, satisfies } m \le k\} \tag{1}$$

A superscript of $k = n$ removes the *restriction* on the path x,

$$q_i \overset{x}{\frown} q_j \tag{2}$$

since *every state q_m satisfies $m \le n$ anyway!*

Thus R^n_{ij} contains ALL strings that name FA-paths from q_i to q_j —no restriction on where these paths pass through.

[†]We start numbering states from 1 rather than 0 for technical convenience; see the blue sentence at the top of next page.

So if we manage to get *regular expression names* α_{ij}^k, for *each* R_{ij}^k, then the set of strings $L(M)$ accepted by the given FA is

$$L(M) = \bigcup_{q_j \in F} R_{1j}^n$$

and we can *name it as finite sum* of α_{1j}^n's. *Why "finite"?*

How do we prove that each R_{ij}^k is nameable by a regular expression? By *induction* (on k) *along the inductive definition of these sets*!

See below Kleene's *formulation* of the inductive definition of the R_{ij}^k *and of their names*.

We first note that for $k = 0$ we get very small finite sets.

Indeed, since state numbering starts at 1, the condition $m \le 0$ is false and therefore in R^0_{ij} we have the cases:

- if we have $i \neq j$, then the condition (2) on p.339 can hold precisely when $x = a \in \Sigma$ for some a —since there can be no nodes in the interior of x.

  That is, we have precisely the case:

$$q_i \xrightarrow{a} q_j \tag{$\dagger$}$$

- The case $i = j$ also *adds* λ in the set, since, when we have ONE state:

$$q_i = q_j \tag{$\ddagger$}$$

  "*I can go from q_i to q_j DETERMINISTICALLY without consuming ANY input*" —*algebraically, think of $q_i\lambda \vdash^* \lambda q_i$.*

To summarize, for all i and j we have

$$R^0_{ij} = \begin{cases} \{a \in \Sigma : \text{ Case } (\dagger)\} & \text{if } i \neq j \\ \{\lambda\} \cup \{a \in \Sigma : \text{ Case } (\dagger)\} & \text{if } i = j \end{cases} \tag{3}$$

Since every *finite* set of strings *can be named by a regular expression* (Exercise!),

$$\text{there are RegEx: } \alpha^0_{ij} \text{ such that } L(\alpha^0_{ij}) = R^0_{ij}, \text{ for all } i, j \tag{4}$$

For example, say $A = \{3, 5, 8, \lambda\}$. This is a finite set. It is NOT an alphabet (contains λ).

Then the RegEX $3 + 5 + 8 + \lambda = 3 + 5 + 8 + \emptyset^*$ NAMES A.

Why? Because $A = \{3\} \cup \{5\} \cup \{8\} \cup \{\lambda\}$.

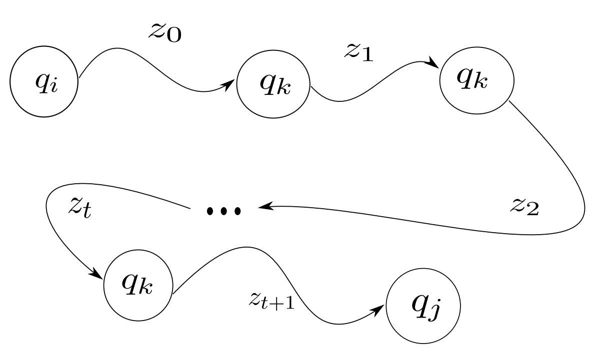
Next note that the R_{ij}^k can be COMPUTED **recursively/inductively using k as the recursion/induction variable** and i, j as parameters, and taking (3) on page 341 as the basis of the recursion.

> To see this, *consider a string $x \in \Sigma^*$ placed in R_{ij}^k, where $k > 0$. By Definition (1), p.339,*
>
> *This x labels a path from q_i to q_j whose internal nodes q_m satisfy $m \le k$.*
>
> We explore the structure of this path below.

It is possible that all q_m *(other than q_i and q_j) that occur in the path x have $m < k$.*

In such case, this x also belongs to (is placed in) R_{ij}^{k-1}.

If on the other hand we DO have q_k's appear in the interior of the path labeled x, one or more times, then we have the picture below.



where the q_k occurrences start immediately after the path named z_0 and are connected by paths named z_i, for $i = 1, \ldots, t$. Thus, $x = z_0 z_1 z_2 \ldots z_t z_{t+1}$. Noting that $z_0 \in R_{ik}^{k-1}$, $z_i \in R_{kk}^{k-1}$ —for $i = 1, \ldots, t$— and $z_{t+1} \in R_{kj}^{k-1}$, we have that $x \in R_{ik}^{k-1} \cdot \left(R_{kk}^{k-1}\right)^* \cdot R_{kj}^{k-1}$. We have established, for all $k \geq 1$ and all i, j, that

$$R_{ij}^{k} = R_{ij}^{k-1} \cup R_{ik}^{k-1} \cdot \left(R_{kk}^{k-1}\right)^{*} \cdot R_{kj}^{k-1} \tag{4}$$

Explanation. Noting that

$$\left(R_{kk}^{k-1}\right)^{*} = \{\lambda\} \cup R_{kk}^{k-1} \cup$$
$$R_{kk}^{k-1} R_{kk}^{k-1} \cup R_{kk}^{k-1} R_{kk}^{k-1} R_{kk}^{k-1} \cup R_{kk}^{k-1} R_{kk}^{k-1} R_{kk}^{k-1} R_{kk}^{k-1} \cup \ldots$$

the set of paths, from q_i to q_j depicted in the following part of (4):

$$R_{ik}^{k-1} \cdot \left(R_{kk}^{k-1}\right)^{*} \cdot R_{kj}^{k-1}$$

may contain

one interior q_k case corresponds to λ
two interior q_k case corresponds to R_{kk}^{k-1}
three interior q_k case corresponds to $R_{kk}^{k-1} R_{kk}^{k-1}$
four interior q_k case corresponds to $R_{kk}^{k-1} R_{kk}^{k-1} R_{kk}^{k-1}$
five interior q_k case corresponds to $R_{kk}^{k-1} R_{kk}^{k-1} R_{kk}^{k-1} R_{kk}^{k-1}$
etc.

Now take the I.H. that for $k - 1 \geq 0$ (fixed!) and all values of i and j we have regular expressions α_{ij}^{k-1} such that $\boxed{L(\alpha_{ij}^{k-1}) = R_{ij}^{k-1}}$ —that is, α_{ij}^{k-1} NAMES the set R_{ij}^{k-1}.

The above *is true for $k - 1 = 0$, i.e., $k = 1$*.

We see that we can construct —from the α_{ij}^{k-1}— *regular expressions* for the R_{ij}^{k} that we will name them with the short name α_{ij}^{k}.

Indeed, using the I.H. and (4), *we have that the RegEX* α^k_{ij} *, for all* i, j *and the fixed* k*, is a SHORT NAME for the rhs in (5)*

$$\alpha^k_{ij} = \alpha^{k-1}_{ij} + \alpha^{k-1}_{ik}\big(\alpha^{k-1}_{kk}\big)^*\alpha^{k-1}_{kj} \tag{5}$$

WHY? Because said rhs NAMES the set R^k_{ij} due to the I.H. as we argued above.

Along with the basis (3) that the R^0_{ij} sets *CAN be named (being finite)*, this induction proves that *all* the R^k_{ij} can be named by regular expressions, which we may construct, from the basis up.

Finally, the set $L(M)$ can be so named. Indeed,

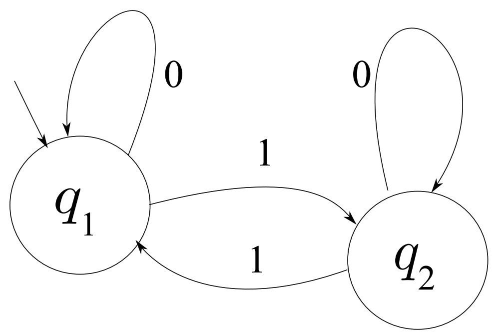
$$L(M) = \bigcup_{q_j \in F} R^n_{1j}$$

Therefore, as a RegEX:

$$\sum_{q_j \in F} \alpha^n_{1j} = \overbrace{\alpha^n_{1j_1} + \alpha^n_{1j_2} + \ldots + \alpha^n_{1j_m}}^{\textit{finitely many terms}}$$

The above is a finite union (F is finite!) of sets named by α^n_{1j} with $q_j \in F$. Thus we may construct its name as the "sum" (using "+", that is) of the names α^n_{1j} with $q_j \in F$. $\square$

12.2.3 Example. Consider the FA below.



We will compute regular expressions for:

- all sets R^0_{ij}
- all sets R^1_{ij}
- all sets R^2_{ij}

Recall the definition of the R^k_{ij}, *here for* $k = 0, 1, 2$ *and* i, j *ranging in* $\{1, 2\}$ (cf. proof of 12.2.2):

$\{x : q_i \overset{x}{\frown} q_j$, where no state in this computation x,

other than possibly the *end-points* q_i and q_j, that has index higher than $k\}$

This leads —as we saw— to the recurrence:

$$R^k_{ij} = R^{k-1}_{ij} \cup R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj}$$

Below I employ the abbreviated (regular expression) *name* "λ" for $\emptyset^*$.

SET	RegEx
R^0_{11}	$\lambda + 0$
R^0_{12}	1
R^0_{21}	1
R^0_{22}	$\lambda + 0$

Superscript 1 now:

SET	RegEx: By Direct Substitution
$R^1_{11} = R^0_{11} \cup R^0_{11}(R^0_{11})^* R^0_{11}$	$\lambda + 0 + (\lambda + 0)(\lambda + 0)^*(\lambda + 0)$
$R^1_{12} = R^0_{12} \cup R^0_{11}(R^0_{11})^* R^0_{12}$	$1 + (\lambda + 0)(\lambda + 0)^* 1$
$R^1_{21} = R^0_{21} \cup R^0_{21}(R^0_{11})^* R^0_{11}$	$1 + 1(\lambda + 0)^*(\lambda + 0)$
$R^1_{22} = R^0_{22} \cup R^0_{21}(R^0_{11})^* R^0_{12}$	$\lambda + 0 + 1(\lambda + 0)^* 1$

Using the previous table, the reader will have no difficulty to fill in the regular expressions under the heading "RegEx: By Direct Substitution" in the next table.

To make things easier it is best to simplify the regular expressions of the previous table, meaning, finding simpler, equivalent ones. For example, $L\big(\lambda+0+(\lambda+0)(\lambda+0)^*(\lambda+0)\big) = \{\lambda, 0\} \cup \{\lambda, 0\}\{\lambda, 0\}^*\{\lambda, 0\} = \{\lambda, 0\} \cup \underbrace{\underbrace{\{\lambda, 0\}\{\lambda, 0, 00, 000, \ldots\}}_{\{0\}^*}\{\lambda, 0\}}_{\{0\}^*} = \{0\}^*$, thus

$$\lambda + 0 + (\lambda + 0)(\lambda + 0)^*(\lambda + 0) \sim 0^*$$

Superscript 2:

SET	RegEx: By Direct Substitution
$R^2_{11} = R^1_{11} \cup R^1_{12}(R^1_{22})^* R^1_{21}$	Exercise
$R^2_{12} = R^1_{12} \cup R^1_{12}(R^1_{22})^* R^1_{22}$	Exercise
$R^2_{21} = R^0_{21} \cup R^0_{22}(R^1_{22})^* R^1_{21}$	Exercise
$R^2_{22} = R^1_{22} \cup R^1_{22}(R^1_{22})^* R^1_{22}$	Exercise

$\square$

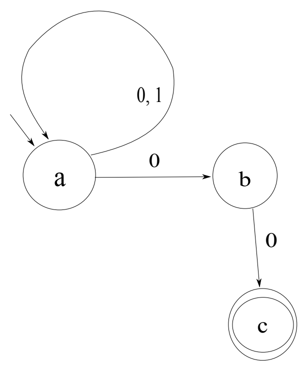
12.3. Another Example

12.3.1 Example. Let us show another NFA to FA conversion.

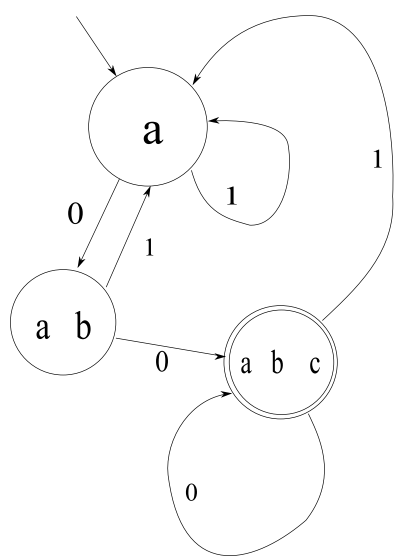
OK, given the following NFA which clearly decides the language over $\Sigma = \{0, 1\}$ given by the RegEx

$$(0 + 1)^* 00$$

that is, the language containing ALL strings that end in two 0s.



The DETERMINISTIC FA equivalent to the above is the following:



□

12.4. A RegEX ⇒ NFA ⇒ FA Example

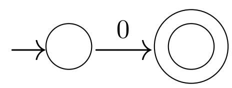
Consider again the RegEX over $\Sigma = \{0, 1\}$ below

$$\alpha = (0^*1^*)^* \tag{1}$$

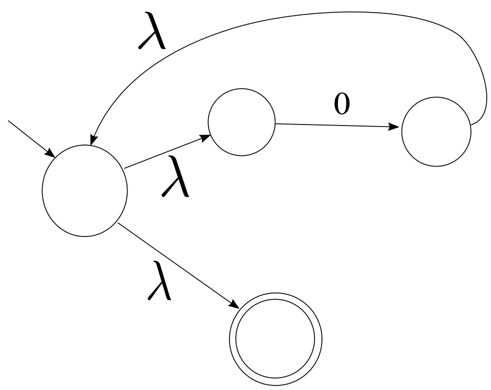
We will first build an NFA from it using a shortcut of Kleene's construction, and then we will apply our *NFA ⇒ FA process* to build an equivalent FA.

If we follow Kleene's proof/construction verbatim, then the first step would be to build an NFA for 0,

0 :



then build the Kleene closure of (2) as



0^* :

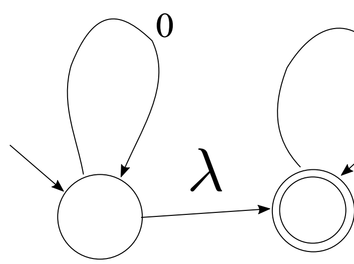
One then builds identically the NFA for 1^* in two steps (only the label 0 changes to label 1) and continues with the NFA for the concatenation the NFA for 0^* —above— with the NFA for 1^* (not shown) and finally builds the NFA for (1) in the way the proof of Kleene's theorem goes.

In this simple case we proceed guided by the definition of string acceptance in our shortcut construction.

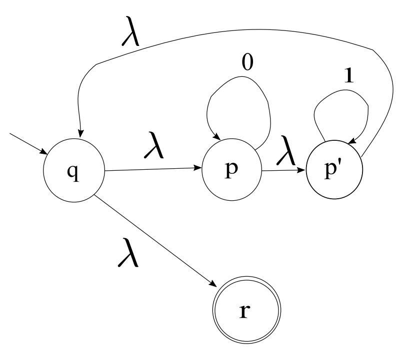
First, the following is clearly an NFA that accepts all strings described by 0^*1^*. These are all strings of the form

$$\{0^n1^m : n \geq 0 \wedge m \geq 0\} \tag{2}$$

We see by inspection that the NFA below accepts precisely the strings in (2) since *the only possible accepting-path labels* —0^n1^m— that we can get in the design below, and, indeed, *we get all of them*, for al $n \geq 0, m \geq 0$.



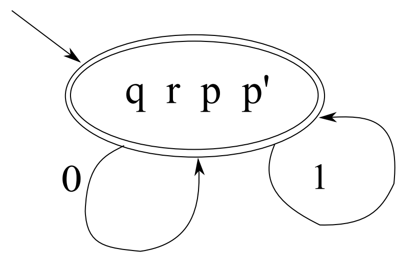
According to the proof of Kleene's theorem we get the NFA for the RegEX (1) as follows from the NFA above:



(2)

where we added state names for the next step, NFA⟹FA.

The FA is the following.



(2)

Note that q is the start state but all of p, r and p' are in $\lambda(q)$. Moreover, see how the 0-successor of the FA state "q, r, p, p'" is computed: We find that only p from the NFA above has a 0 successor, and that is p.

But you can easily compute that $\lambda(p) = \{q, r, p, p'\}$. So on input 0 the FA goes back to $\{q, r, p, p'\}$.

Similarly exactly, the 1-successor of $\{q, r, p, p'\}$ in the FA is $\{q, r, p, p'\}$.

Incidentally, the FA above proves again in a different way that *the language of* the RegEX $(0 + 1)^*$, which the FA trivially decides is the *same as the language of* the RegEX $(0^*1^*)^*$.

Bibliography

[Chu36a] Alonzo Church, *A note on the Entscheidungsproblem*, J. Symbolic Logic **1** (1936), 40–41, 101–102.

[Chu36b] ______, *An unsolvable problem of elementary number theory*, Amer. Journal of Math. **58** (1936), 345–363, (Also in Davis [Dav65, 89–107]).

[Dav58] M. Davis, *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

[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 Davis [Dav65, 5–38]).

[Grz53] A. Grzegorczyk, *Some classes of recursive functions*, Rozprawy Matematyczne **4** (1953), 1–45.

[Kal57] L. Kalmár, *An argument against the plausibility of Church's thesis*, Constructivity in Mathematics, *Proc. of the Colloquium*, Amsterdam, 1957, pp. 72–80.

[Kle43] S.C. Kleene, *Recursive predicates and quantifiers*, Transactions of the Amer. Math. Soc. **53** (1943), 41–73, (Also in Davis [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.

[Ṕ67] Rózsa Péter, *Recursive Functions*, Academic Press, New York, 1967.

[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.

[Rog67] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[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.

[Tou86] G. Tourlakis, *Some reflections on the foundations of ordinary recursion theory, and a new proposal*, Zeitschrift für math. Logik **32** (1986), no. 6, 503–515.

[Tou12] G. Tourlakis, *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 Davis [Dav65, 115–154].).