Lecture Notes #2.

0.1 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*.

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

Hilbert's *program* 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*.

*So what **IS** a "mechanical procedure"?*

Now, since antiquity, mathematicians have invented "mechanical procedures", e.g., Euclid's algorithm for the "greatest common divisor",[1] 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!

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

0.1.1 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.

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 is derivable".

Either way, a computable function is generated by a **finite devise** (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.

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

The **alphabet** of the language is

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

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:

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

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).

0.1.1.1 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.

This command is *syntactically illegal* (meaningless) if any of M or R exceed the label of the program's **stop** instruction.

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

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

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

The *semantics* of each command is given below.

0.1.1.2 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.

□

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 write $\vec{\mathbf{x}}$ and $\vec{a}$.

0.1.1.3 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*.

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* $\mathbb{N}^n$,

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

(1) We *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 thus *start* the computation.

So, the initialisation is NOT part of the computation!

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

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

We resume: Lecture NOTES #3. Sept.16

0.1.1.4 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*. □

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

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

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

BTW, you recall from MATH1019 that the symbol

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

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". *FORGET THAT!* 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 $f : \mathbb{N} \rightarrow N$ is *total* iff $\text{dom}(f) = \mathbb{N}^n$.

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

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

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

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

0.1.1.5 Example. Let M be the program

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

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

0.1.1.6 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).

0.1.1.7 Example. Let M be the program

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

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.

0.1.1.8 Example. Let M be the program

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

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

In Definition 0.1.1.4 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 *any* function in computability.

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

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

0.1.1.10 Example. Let M be the program

$$\begin{array}{l} 1 : \mathbf{x} \leftarrow 0 \\ 2 : \textbf{goto } 1 \\ 3 : \textbf{stop} \end{array}$$

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

0.1.1.11 Example. Let M be the program segment

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

What it does:

By the time the computation reaches instruction $k+3$, the program segment has set the value of $\mathbf{z}$ to 0, and has made the value of $\mathbf{x}$ equal to the value that $\mathbf{z}$ had when instruction $k-1$ was current.

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

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

where the 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. □

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

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

Because of Exercise 0.1.1.12 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 $\mathbf{x}'$ read-only:

- Add at the very beginning of M the instruction $1 : \mathbf{x} \leftarrow \mathbf{x}'$ of Exercise 0.1.1.12.
- 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, $M'^{\mathbf{x}',\mathbf{y}_1,\ldots,\mathbf{y}_n}_{\mathbf{z}} = M^{\mathbf{x},\mathbf{y}_1,\ldots,\mathbf{y}_n}_{\mathbf{z}}$, and M' does not change $\mathbf{x}'$.

Lecture #4. Sept. 21

0.1.1.13 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*.

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 a finite number of times:

0.1.1.14 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 0.1.1.14, that is:

0.1.1.15 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 0.1.1.15 as "*rigid*".

Indeed, note that it requires *all* the arguments of f to be substituted by some $g_i(\vec{z})$—unlike Example 0.1.1.13, where we *substituted* a function invocation (cf. terminology in 0.1.1.6) *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, "…", *must be the same*; in 0.1.1.15 it was $\vec{z}$.

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

We can rephrase 0.1.1.14, saying simply that

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

0.1.1.17 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 0.1.1.16.

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 $\mathcal{R}$*. □

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 *iteration*. The general case of iteration is called *primitive recursion*.

0.1.1.18 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*).

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

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

$f(3,\vec{y}) = g(2,\vec{y},g(1,\vec{y},g(0,\vec{y},h(\vec{y}))))$, 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 0.1.1.18 is computed by the pseudo code below:

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

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

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

0.1.1.19 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, $g = G^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}_{\mathbf{z}}$ while $h = H^{\vec{\mathbf{y}}}_{\mathbf{z}}$.
By earlier remarks we may assume:
(i) The only variables that H and G have in common are $\mathbf{z}, \vec{\mathbf{y}}$.
(ii) The variables $\vec{\mathbf{y}}$ are read-only in both H and G.
(iii) $\mathbf{i}$ is read-only in G.
(iv) $\mathbf{x}$ does not occur in any of H or G.

We can now see that the following URM program, let us call it F, computes f defined as in 0.1.1.18 from h and g, where $\boxed{H'}$ is program H with the **stop** instruction removed, $\boxed{G'}$ is program G that has the **stop** instruction removed, and instructions renumbered (and if-statements adjusted) as needed:

$$
\begin{array}{ll}
 & \boxed{H'}^{\vec{\mathbf{y}}}_{\mathbf{z}} \\
r : & \mathbf{i} \leftarrow 0 \\
r+1 : & \textbf{if } \mathbf{x} = 0 \textbf{ goto } k+m+2 \textbf{ else goto } r+2 \\
r+2 : & \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1 \\
 & \boxed{G'}^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}_{\mathbf{z}} \\
k : & \mathbf{i} \leftarrow \mathbf{i} + 1 \\
k+1 : & \mathbf{w}_1 \leftarrow 0 \\
\vdots & \\
k+m : & \mathbf{w}_m \leftarrow 0 \\
k+m+1 : & \textbf{goto } r+1 \\
k+m+2 : & \textbf{stop}
\end{array}
$$

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 $f = F^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$. □

Lecture #5 (Sept. 23)

We have at once:

0.1.1.20 Proposition. *If f, g, h relate as in Definition 0.1.1.18 and h and g are in $\mathcal{P}$, then so is f. We say that $\mathcal{P}$ is* closed under primitive recursion.

0.1.1.21 Corollary. *If f, g, h relate as in Definition 0.1.1.18 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 saw that if h and g is total, then so is f.

So, $f \in \mathcal{R}$. □

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

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

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

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

In short, the k satisfies

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

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

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

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

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

where $(\forall y)_{y<x} R(y, \ldots)$ is short for $(\forall y)(y < x \to R(y, \ldots))$. We call the operation $(\mu y) g(y, \vec{a})$ —equivalently, the **program segment** "**while** $g(\mathbf{x}, \vec{a}) \neq 0$ **do**"— *unbounded search*.

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

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

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

  or

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

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

0.1.1.22 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.

$1:$ $\mathbf{x} \leftarrow 0$
$\boxed{G'}$
$k:$ **if** $\mathbf{z} = 0$ **goto** $k+l+3$ **else goto** $k+1$
$k+1:$ $\mathbf{w}_1 \leftarrow 0$ {**Comment.** Setting *all non-input variables* to 0; cf. 0.1.1.19.}
$\vdots$
$k+l:$ $\mathbf{w}_l \leftarrow 0$ {**Comment.** Setting *all non-input variables* to 0; cf. 0.1.1.19.}
$k+l+1:$ $\mathbf{x} \leftarrow \mathbf{x}+1$
$k+l+2:$ **goto** 2
$k+l+3:$ **stop** {**Comment.** Read answer off $\mathbf{x}$. This is the last $\mathbf{x}$-value used by G'}

□

The result of Example 0.1.1.22 yields at once:

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

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

$$\begin{array}{ll} 1: & \mathbf{w}_1 \leftarrow \mathbf{w}_1 + 1 \\ \vdots & \\ i: & \mathbf{z} \leftarrow \mathbf{w}_i \ \{\textbf{Comment.}\ \text{Cf. Exercise } 0.1.1.12\} \\ \vdots & \\ n: & \mathbf{w}_n \leftarrow 0 \\ n+1: & \textbf{stop} \end{array}$$

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

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 [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; cf. [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].

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 3rd ed., Addison-Wesley, Boston, 2007.

[Kal57] L. Kalmár, *An argument against the plausibility of Church's thesis*, Constructivity in Mathematics, *Proc. of the Colloquium*, Amsterdam, 1957, pp. 72–80.

[Kle36] S.C. Kleene, *General recursive functions of natural numbers*, Math. Annalen **112** (1936), 727–742.

[Kle43] ———, *Recursive predicates and quantifiers*, Transactions of the Amer. Math. Soc. **53** (1943), 41–73, [Also in [Dav65], 255–287].

[LP98] Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, NJ, 1998.

[Mar60] A. A. Markov, *Theory of algorithms*, Transl. Amer. Math. Soc. **2** (1960), no. 15.

[P67] 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.

[Sip97] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing, Boston, 1997.

[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.

[Tur37] Alan M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math Soc. **2** (1936, 1937), no. 42, 43, 230–265, 544–546, [Also in [Dav65], 115–154].