March 19

0.1 The Iteration Theorem of Kleene

Suppose that i codes a URM program, M, that acts on input variables x and y to compute a function $\lambda xy.f(x,y)$. It is certainly trivial to modify program M to compute $\lambda x.f(x,a)$ instead. In computer programming terms, we replace an instruction such as "read y" by one that says "$y \leftarrow a$".

In URM terms, since the input variables $\mathbf{x}_1, \mathbf{x}_2, \ldots$ are initialized *before* the computation starts, the way to implement the suggested "decommissioning" of y as an input variable —opting rather to initialize it with the number a explicitly, *first thing* during the computation— is to do the following, assuming x and y of f are mapped to $\mathbf{x}_1$ and $\mathbf{x}_2$ of M:

(1) Remove $\mathbf{x}_2$ from the input variables list, $\mathbf{x}_1, \mathbf{x}_2$, and

(2) Modify M into M' by adding the instruction $\mathbf{x}_2 \leftarrow a$ as the very first instruction.

From the original code, i, a new code (depending on i and a) can be easily calculated. This is the intuition of Kleene's *iteration* or "S-m-n" theorem below.

The mathematical details are as follows, but first note that the function $g = \lambda x\vec{y}.\Pi_{i<x} f(i,\vec{y})$ is in $\mathcal{PR}$ if f is:

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

0.1.1 Definition. (Code Concatenation)

$$x * y \overset{\text{Def}}{=} x \cdot \Pi_{i<lh(y)} p_{i+lh(x)}^{\exp(i,y)} \qquad \square$$

0.1.2 Remark. Clearly, $\lambda xy.x*y$ is primitive recursive. The definition's aim is to achieve this—which it clearly does:

$$\langle a_1, \ldots, a_n \rangle * \langle b_1, \ldots, b_m \rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$$

If $Seq(x)$ or $Seq(y)$ fail, then the result of $x * y$ is irrelevant to us. $\square$

0.1.3 Exercise. What is $10 * 5$? $\square$

0.1.4 Exercise. What is $1 * z$? $z * 1$? $\square$

0.1.5 Definition. (Concatenating URMs) Given two URMs M and N of codes m and n. We denote their *concatenation* by MN and $m \frown n$ in terms of their codes. Note that MN means the superposition of the two URMs, in that order, with the **stop**-instruction removed from M and all the instructions of N adjusted to reflect that the first label of N now is $lh(m)$.

We define $m \frown n$ to be 0 if either of m or n is not a valid URM code. $\square$

0.1.6 Lemma. *Let $adj(n,k)$ ("adjust n") be the expression that codes a URM n* after *k was added to all its instruction numbers (and all if-statements were adjusted to still transfer to the same instructions as before). Let also $adj(n,k) = 0$ if n does not code a URM. Then the function $\lambda nk.adj(n,k)$ is primitive recursive.*

Proof. First let us define a less ambitious function, f, that adjusts one instruction due to adding k to the instruction number:

$$f(z,k) = \begin{cases} 3^k z & \text{if } (\exists L,i,a)_{\le z}\Big(z = \langle 1,L,i,a\rangle \vee \\ & \qquad\qquad\qquad z = \langle 2,L,i\rangle \vee \\ & \qquad\qquad\qquad z = \langle 3,L,i\rangle \vee \\ & \qquad\qquad\qquad z = \langle 5,L\rangle \Big) \\ 231^k z & \text{if } (\exists L,i,P,Q)_{\le z} z = \langle 4,L,i,P,Q\rangle \\ 0 & \text{otherwise} \end{cases}$$

Note that $231 = 3 \cdot 7 \cdot 11$. Clearly, $f \in \mathcal{PR}$. Finally,

$$adj(n,k) = \begin{cases} \Pi_{i<lh(n)} p_i^{f((n)_i,k)+1} & \text{if } URM(n) \\ 0 & \text{otherwise} \end{cases}$$

Clearly, adj is primitive recursive. □

0.1.7 Lemma. *$\lambda mn.m \frown n$ is primitive recursive.*

Proof.

$$m \frown n = \begin{cases} \left\lfloor \dfrac{m}{p_{lh(m)\dot-1}^{\exp(lh(m)\dot-1,m)}} \right\rfloor * adj(n, lh(m) \dot- 1) & \text{if } URM(m) \wedge URM(n) \\ 0 & \text{otherwise} \end{cases}$$

The left hand operand of $*$ above represents the removal of the **stop**-instruction of the URM m prior to concatenation. It is immediate from the above and the preceding lemma that $\lambda mn.m \frown n$ is primitive recursive. □

0.1.8 Theorem. (Kleene's Iteration or "S-m-n" Theorem) *For each $m \ge 1$ and $n \ge 1$, there is a primitive recursive function $\lambda i\vec{y}_n.S^m_n(i,\vec{y}_n)$ such that, for all $i, \vec{x}_m, \vec{y}_n$,*

$$\phi_i^{(m+n)}(\vec{x}_m, \vec{y}_n) = \phi^{(m)}_{S^m_n(i,\vec{y}_n)}(\vec{x}_m) \tag{1}$$

Proof. The construction of S^m_n is guided by the introductory remarks of this section: If i codes a (normalised) URM M such that

$$M_{\mathbf{x_0}}^{\mathbf{x_1},\ldots,\mathbf{x_{m+n}}} = \phi_i^{(m+n)}$$

then we remove the n variables $\mathbf{x_{m+1}}, \ldots, \mathbf{x_{m+n}}$ from the *designated input-variable list* and add the instructions below at the top of program M.

$$\mathbf{x_{m+1}} \leftarrow y_1, \mathbf{x_{m+2}} \leftarrow y_2, \ldots, \mathbf{x_{m+n}} \leftarrow y_n$$

To be able to use of the tools we have developed in this section, we implement the above plan by concatenating the following URM, N, to the left of M. Note that N is *not* normalized, but NM is, since M is.

$$\begin{array}{rl} 1: & \mathbf{x_{m+1}} \leftarrow y_1 \\ & \vdots \\ n: & \mathbf{x_{m+n}} \leftarrow y_n \\ n+1: & \textbf{stop} \end{array} \tag{N}$$

The code for N is a function of $\vec{y}_n$ (recall that m, n are constants) which we will name $init(\vec{y}_n)$. Thus,

$$init(\vec{y}_n) = p_0^{\langle 1,1,m+1,y_1\rangle+1} p_1^{\langle 1,2,m+2,y_2\rangle+1} \cdots p_{n-1}^{\langle 1,n,m+n,y_n\rangle+1} p_n^{\langle 5,n+1\rangle+1} \tag{2}$$

It is immediate that $\lambda\vec{y}_n.init(\vec{y}_n)$ is primitive recursive. Thus, S^m_n, given below for all $i, \vec{y}_n$, is too by Lemma 0.1.7.

$$S^m_n(i, \vec{y}_n) = init(\vec{y}_n) \frown i \tag{3}$$

It is important to note that (by 0.1.7) if i fails the $URM(i)$ "test", then so does $S^m_n(i, \vec{y}_n)$ (indeed, equals 0; cf. 0.1.7) and thus both sides of (1) are equal (both sides contain the empty function acting on inputs). □

0.1.9 Remark. (1) It is important to note by inspecting (2) and (3) in the proof above that if $URM(i)$ holds, then S^m_n is strictly increasing with respect to each y_i variable. Of course, if $URM(i)$ fails, then S^m_n returns 0 no matter what the inputs y_i may be.

(2) A note on notation: In S^m_n the *upper* index, m, is a mnemonic tool for how many variables stayed "up" (in the ϕ argument), while the lower index, n, indicates how many variables were moved "down", to be hardwired into the "program" $S^m_n(i, \vec{y}_n)$ as it were.

These considerations led to the nickname of the iteration theorem as the "S-m-n theorem".

(3) *In practice, the S-m-n theorem is applied as follows*: If $\lambda\vec{x}_k y\vec{z}_r.f(\vec{x}_k, y, \vec{z}_r) \in \mathcal{P}$, then there is *a 1-1* $h \in \mathcal{PR}$, such that, for all $\vec{x}_k, y, \vec{z}_r$, we have

$$f(\vec{x}_k, y, \vec{z}_r) = \phi^{(k+r)}_{h(y)}(\vec{x}_k, \vec{z}_r)$$

By the assumption on f and the definition of ϕ-indices, there is an $i \in \mathbb{N}$, such that $f(\vec{x}_k, y, \vec{z}_r) = \phi^{k+r+1}_i(\vec{x}_k, \vec{z}_r, y)$. Note the permutation of variables, where y was moved to the end of the argument list of ϕ^{k+r+1}_i, to align with requirements of the S-m-n theorem. Can we do this? Yes, since we may chose the

URM i such that we have mapped the "mathematical" variables $\vec{x}_k, y, \vec{z}_r$ of f to the "formal" variables $X1, \ldots X1^{k+r+1}$ so that y's role is played by $X1^{k+r+1}$.

We take $h = \lambda y.S_1^{k+r}(i, y)$. Note that the italicized part, "a 1-1", above is a weakening of the observation (1) above. $\square$

0.2 Diagonalization Revisited; Unsolvability via Reductions

This section further develops the theory of computability and *un*computability by developing tools—in particular, *reducibility*—that are more sophisticated than the ones we encountered so far in this volume, toward discovering undecidable and non c.e. problems. We also demonstrate explicitly that diagonalization is at play in a number of interesting examples.

As we mentioned in the Preface and elsewhere already, the aim of computability is to "formalize" the concept of "algorithm" and then proceed to classify problems as decidable vs. undecidable and verifiable vs. unverifiable.

We continue taking—*by definition*—the term "algorithm" to *mean* URM program, and computable (partial) function to mean a URM-computable function, or equivalently, one that has a $\mathcal{P}$-derivation (using the number theoretic re-definition of $\mathcal{P}$ proved in class and in the text). Thus *proving* that such and such a problem $x \in A$ does *not* have an "algorithmic solution", or is not even verifiable, becomes mathematically precise: **By Definition**, we need to show that $A \notin \mathcal{R}_*$ or $A \notin \mathcal{P}_*$, respectively.

Church has gone a step further, and observing that all known formalisms of the concept of algorithm were proved to be equivalent (each produced the same computable functions), formulated

> **Church's Thesis.** Any partial function that can be informally demonstrated to be computable by some algorithm, **can be mathematically demonstrated to be programmable in any one of the known formalisms** (such as Turing machines, Markov algorithms, Post systems, URMs[*]).

Of course, this "thesis" is a belief based on empirical evidence, not a metatheorem. The difficulty (toward theoremhood) lies in the fact that in order to, say, demonstrate mathematically that the concepts of "algorithm" and URM coincide, we must already have a mathematical formulation of *algorithm*!

This is why we said above that we take *by definition* that algorithm means URM. We cannot do better than being arbitrary like this. We already mentioned that while the "Thesis" is widely adopted—indeed, some advanced books such as [Rog67] use it to shorten proofs that such and such a function is computable—the adoption is not universal; cf. [Kal57].

This volume will not take the shortcut of relying on Church's Thesis. Whenever we want to prove that f *is computable* we will do so mathematically, invariably using the two definitions of $\mathcal{P}$ as needed and closure properties of $\mathcal{P}$. Nevertheless, we will often also offer an intuitive argument that establishes the desired computability, just to get a feeling as to why things tick!

[*]Actually, the URM formalism postdates the formulation of Church's Thesis but is demonstrably equivalent to all the others.

0.2.1 More Diagonalization

We begin the development of the theory by revisiting the proof of the undecidability of the halting problem.

0.2.1 Theorem. (The Undecidability of the Halting Problem; Again) *$K \notin \mathcal{R}_*$.*

Proof. We will argue by contradiction, so we assume that $K \in \mathcal{R}_*$, that is, the relation $\phi_x(x)\downarrow$ is recursive. We view a one-argument function f as a sequence of values,

$$f(0), f(1), \ldots$$

where (informally), if $f(x)\uparrow$, then we take the symbol "$\uparrow$" as the yielded value.

On this understanding we form the infinite matrix below, of which the i-th row represents ϕ_i, for all i.

$$\begin{array}{cccc}
\phi_{\mathbf{0}}(\mathbf{0}) & \phi_0(1) & \phi_0(2) & \ldots \\
\phi_1(0) & \phi_{\mathbf{1}}(\mathbf{1}) & \phi_1(2) & \ldots \\
\vdots & \vdots & \vdots & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \ldots \\
\vdots & \vdots & \vdots &
\end{array}$$

We proceed to utilize the main diagonal

$$\phi_0(0), \phi_1(1), \ldots, \phi_i(i), \ldots$$

and build a function that *cannot be a row* of the above matrix. We simply change each entry that is $\uparrow$ to $\downarrow$, and vice versa:

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

The above captures the idea, but it is not a well-defined function since we have not said what the output of d is when it is defined. We resolve this "uncertainty" arbitrarily as follows:

$$d(x) = \begin{cases} 42 & \text{if } \phi_x(x)\uparrow \\ \uparrow & \text{if } \phi_x(x)\downarrow \end{cases} \tag{1}$$

Indeed, d does not match any row above as it differs from each row in the spot where it intersects the diagonal. Why do we care? Well, since $\mathcal{P}$ is closed under definition by cases (Exercise 28 in Problem Set #1), and since by assumption both $\phi_x(x)\downarrow$ and $\phi_x(x)\uparrow$ are recursive,

Pause. Why "both"?◀

it follows that $d \in \mathcal{P}$, i.e., $d = \phi_i$ for some i—i.e., d *must* be some row in the above matrix. We have a contradiction. □

An intuitive reason as to why the function d as defined in (1) is computable, is presented here by outlining a pseudo algorithm for the computation of $d(x)$: Let M be a URM that decides the predicate $\phi_x(x)\downarrow$. Given input x, run M on x. If it says "no", then print 42 and halt; if it says "yes", then get into a deliberate infinite loop.

Worth repeating. We chose d so that at input x it *differs* from $\phi_x(x)$, and thus it differs from ϕ_x; full stop. We have "**cancelled**" x as a possible ϕ-index of d.

Given that we have done this for *all* x, we have cancelled all possible ϕ-indices of d. Thus d is not computable. Since our assumption about $\phi_x(x)\downarrow$ also forced d to *be* computable, we managed to reject said assumption as it forced a contradiction.

A version of unbounded search is the following:

0.2.2 Definition. (Alternate Unbounded Search Operator) For any *total* function $\lambda y\vec{x}.g(y,\vec{x})$ the expression $(\tilde{\mu}y)g(y,\vec{x})$ stands for

$$\begin{cases}\min\{y : g(y,\vec{x}) = 0\} & \text{if the minimum exists}\\ \uparrow & \text{otherwise}\end{cases}$$

$\square$

It is immediate that (μy) and $(\tilde{\mu}y)$ coincide on total functions since—in $g(y,\vec{x}) = 0 \wedge (\forall z)_{<y}(g(z,\vec{x})\downarrow)$—the subformula $(\forall z)_{<y}(g(z,\vec{x})\downarrow)$ is true for such g and therefore its presence or absence in the formula is immaterial.

0.2.3 Definition. We say that a class of number-theoretic functions $\mathcal{C}$ is closed under $(\tilde{\mu}y)$ just in case for every *total* g in the class, $\lambda\vec{x}.(\tilde{\mu}y)g(y,\vec{x})$—which may fail to be total—is in the class. $\square$

0.2.4 Theorem. *$\mathcal{P}$ is closed under $(\tilde{\mu}y)$.*

Proof. By the preceding -remark, if $g \in \mathcal{R}$, then, for all $\vec{x}$, $(\tilde{\mu}y)g(y,\vec{x}) = (\mu y)g(y,\vec{x})$.

Thus $\lambda\vec{x}.(\tilde{\mu}y)g(y,\vec{x}) \in \mathcal{P}$. $\square$

Note that the requirement that $(\tilde{\mu}y)$ apply on total functions makes it a *semantic* rather than a *syntactic* operator: As we have seen, the problem of whether $\phi_i^{(n+1)}$ is in $\mathcal{R}$ or not is undecidable, indeed is not even c.e.

Thus, given i we cannot know whether writing $(\tilde{\mu}y)\phi^{(n+1)}(y,\vec{x}_n)$ *makes sense or not*: For we cannot *decide*, as Definition 0.2.2 requires, whether $\phi_i^{(n+1)}$ is total.

Pause. Hmm. Why can't we stop worrying about totalness and just *allow*—in defiance of Definition 0.2.2—$(\widetilde{\mu} y)$ to apply to all partial functions, including nontotal ones?◀

Well, a first approximation objection to the suggestion of defiance is that while $(\mu y)g(y, \vec{x})$ is correctly computed by the pseudo program below

```
        $y \leftarrow 0$
**while**
        $\neg g(y, \vec{x}) = 0$
        $y \leftarrow y + 1$
**end**
```

the *same program* does not compute $(\widetilde{\mu} y)g(y, \vec{x})$.

For the sake of argument, say, for a given $\vec{a}$, we have $g(0, \vec{a}) \uparrow$, but $g(1, \vec{a}) = 0$.

If so, the above pseudo program correctly computes $(\mu y)g(y, \vec{a})$ since the definition requires —for convergence—that $(\forall z < 1)g(z, \vec{a}) \downarrow$.

This is not the case for $(\widetilde{\mu} y)g(y, \vec{a})$, which ought to *return* $\min\{y : g(y, \vec{a}) = 0\} = 1$ (overlooking the nontotal-ness of g) but the program above loops forever, since the call to $g(0, \vec{a})$ does.

But wait! What if there *is* a really clever *alternative* program that *computes correctly* $\min\{y : g(y, \vec{a}) = 0\}$, for any computable g, *total or not*?

How can we establish that such a program does not exist (assuming we believe that it does not)? *By producing a partial recursive, nontotal, $\lambda xy.\psi(x, y)$, for which $\lambda x.(\widetilde{\mu} y)\psi(x, y) \notin \mathcal{P}$!*

0.2.5 Theorem. *There is a nontotal $\lambda xy.\psi(x, y) \in \mathcal{P}$ such that $\lambda x.(\widetilde{\mu} y)\psi(x, y) \notin \mathcal{P}$.*

Proof. The proof just firms up the "what if" discussion above that cast some initial doubt on the appropriateness of applying $(\widetilde{\mu} y)$ to nontotal functions. So let us define ψ by

$$\psi(x, y) = \begin{cases} 0 & \text{if } \big(y = 0 \land \phi_x(x) \downarrow\big) \lor\ y = 1 \\ \uparrow & \text{otherwise} \end{cases}$$

Given that the predicate $\phi_x(x) \downarrow$ is semi-recursive, closure properties of $\mathcal{P}_*$ establish the top condition in the definition of ψ as semi-recursive. By *definition by positive cases* we have that $\psi \in \mathcal{P}$.

Let us evaluate

$$(\widetilde{\mu} y)\psi(x, y) \tag{1}$$

There are just two possible output values: The search returns 0 if $\phi_x(x) \downarrow$, while it returns 1 if $\phi_x(x) \uparrow$. Thus $\lambda x.(\widetilde{\mu} y)\psi(x, y)$ is χ_K and therefore is not in $\mathcal{P}$). □

Incidentally, note that χ_K, being a characteristic function, it is total, even though ψ is not.

0.2.6 Proposition. *The problem which requires us to determine for a given URM program i and input x whether a predetermined output y is attained is undecidable.*

We opted to say the above in English, in the first instance. Mathematically we are saying that $\lambda ix.\phi_i(x) = y$ is not in $\mathcal{R}_*$.

Proof. If the stated predicate is in $\mathcal{R}_*$ then so is $\lambda x.\phi_x(x) = y$ by closure properties. We will use a straightforward diagonalization to see that the latter cannot be.

$$\begin{array}{cccc}
\phi_{\mathbf{0}}(\mathbf{0}) & \phi_0(1) & \phi_0(2) & \dots \\
\phi_1(0) & \phi_{\mathbf{1}}(\mathbf{1}) & \phi_1(2) & \dots \\
\vdots & \vdots & \vdots & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \dots \\
\vdots & \vdots & \vdots &
\end{array}$$

Define the new diagonal so that it differs from the one above at every place.

$$d(x) = \begin{cases} y+1 & \text{if } \phi_x(x) = y \\ y & \text{otherwise} \end{cases}$$

Thus, d is *not* a row in the above infinite matrix. On the other hand, since we *assumed* that $\lambda x.\phi_x(x) = y$ is recursive, we have that $d \in \mathcal{R}$ (it is total) hence $d = \phi_i$ for some i and hence *must* be a row. A contradiction. □

0.2.7 Corollary. *$\lambda ixy.\phi_i(x) = y$ is not in $\mathcal{R}_*$.*

Proof. Otherwise we would contradict the preceding proposition, via Grzegorczyk operations. □

The next result, also based on a variant of diagonalisation, has a *computational complexity* flavour: *There are arbitrarily hard-to-compute recursive functions*! Recall our concept of complexity of computable functions, Φ, introduced in the context of the dovetailing technique.

0.2.8 Theorem. *For any a priori chosen recursive function $\lambda x.g(x)$, we can construct an $f \in \mathcal{R}$ such that,* for any i, *if $f = \phi_i$, then $g(x) < \Phi_i(x)$ for all $x \geq i$.*

Thus, we have *a priori* arbitrarily chosen a *level of computational "difficulty"*, g. We may choose a horrendously "big" g [e.g., $A_x(x)$, where A is the Ackermann function]. Then we show how to find a function f, which **no matter how we program it** (via a URM i), such a program will take *more* than $g(x)$ "steps" to terminate on *almost all inputs* x, indeed on all $x \geq i$.

Proof. We want to build an f that for $i \leq x$ cannot be computed within $\leq g(x)$ steps.

Thus, we need to meet two requirements:

(1) Ensure that the f we build is recursive.

(2) Ensure that *all* ϕ-indices i that satisfy

$$i \leq x \textit{ and } \Phi_i(x) \leq g(x)$$

are *cancelled*.

Let us thus set

$$I(x) \overset{\text{Def}}{=} \{i : i \leq x \land \Phi_i(x) \leq g(x)\}$$

Given that $\Phi_i(x) \leq y$ is recursive, so is $\Phi_i(x) \leq g(x)$ since $g \in \mathcal{R}$, and thus we have that $\lambda ix.i \in I(x)$ is recursive. So is the predicate $I(x) \neq \emptyset$, being equivalent to $(\exists i)_{\leq x}\Phi_i(x) \leq g(x)$. We define f, for all x, as follows:

$$f(x) = \begin{cases} 1 + \sum_{i \in I(x)} \phi_i(x) & \text{if } I(x) \neq \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

It is clear that $f \in \mathcal{P}$ from Exercise 28 of Problem Set #1. But we need to work a bit more to show it is total, before we show that it has property (2) above. Let us define, for all i, x, the function h:

$$h(i,x) \overset{\text{Def}}{=} \text{if } i \in I(x) \text{ then } \phi_i(x) \text{ else } 0$$

By the earlier remark on $i \in I(x)$ we have that $h \in \mathcal{P}$. Since whenever $i \in I(x)$ holds we have $\phi_i(x) \downarrow$ (why?), it follows that $h \in \mathcal{R}$. Thus the totalness of f is established as soon as we rewrite (3) as follows, since $\sum_{i \in I(x)} \phi_i(x) = \sum_{i \leq x} h(i,x)$.

$$f(x) = \begin{cases} 1 + \sum_{i \leq x} h(i,x) & \text{if } I(x) \neq \emptyset \\ 1 & \text{otherwise} \end{cases}$$

We finally turn to establish property (2) for f. Let then $f = \phi_k$ for some k (as it must since it is recursive) and pick any $x \ge k$. Can it be that $\Phi_k(x) \le g(x)$?

No, for otherwise $k \in I(x)$ holds and therefore $f(x) = 1 + \ldots + \phi_k(x) + \ldots > \phi_k(x)$. A contradiction. □

The reader will note that the claim that we "can construct" an f with the stated properties is apt.

0.2.9 Corollary. *There is no $\lambda x.g(x) \in \mathcal{R}$ such that every recursive ϕ_i is expressed as $\phi_i = \lambda x.d\big((\mu y)_{\le g(x)} T(i,x,y)\big)$.*

Proof. Exercise! □

The corollary says that there is no upper bound on the complexities of the recursive functions.

You may omit this -enclosed part.

It is noteworthy that there are arbitrarily hard to compute 0-1 valued recursive functions, that is, arbitrarily hard to compute recursive predicates. The following and its proof is due to [Blu67].

0.2.10 Theorem. *For any a priori chosen recursive function $\lambda x.g(x)$, we can construct 0-1 valued $f \in \mathcal{R}$ —in essence, a predicate in $\mathcal{R}_*$— such that,* for any *i, if $f = \phi_i$, then $g(x) < \Phi_i(x)$ a.e.*

Proof. With a 0-1 valued function we have to employ a more tricky index cancellation process, following [Blu67]. Adding all the ϕ_i —for the i we want to cancel—and then adding 1 on top of that will not work. We define instead as follows:

$$f(x) = \begin{cases} 1 \dot{-} \phi_k(x) & \text{if } k \text{ is the } \textit{smallest} \text{ uncancelled } i \text{ in } I(x); \\ & \text{now } \textit{cancel} \text{ the } k \text{ that was employed above;} \\ 1^{\dagger} & \text{if no uncancelled } i \text{ exists in } I(x) \end{cases} \tag{1}$$

Let us leave for last the rather dull verification that (1) can be made mathematically precise toward showing that $f \in \mathcal{R}$. That f is 0-1 valued is obvious.

For now, we view the description in (1) as a reasonably complete guideline on how to "program" f and embark on proving its claimed complexity.

Let then $f = \phi_r$, for some r, and let us argue by contradiction.

$$\text{Since } f = \phi_r, \text{ the } \phi\text{-index } r \text{ is never cancelled.} \tag{2}$$

If the claim "$g(x) < \Phi_r(x)$ a.e." is false, then there is an infinite sequence of inputs above r,

$$r < x_1 < x_2 < x_3 < \ldots < x_m < \ldots <$$

[†]Could have used output 0; either is fine.

on which $\Phi_r(x_i) \le g(x_i)$, for each x_i: $i = 1, 2, \ldots$.

Now, input x_i, for each i, satisfies $r < x_i$ too. *Moreover*, r is uncanceled—that is, there are available indices to cancel in $I(x_i)$; cf. (1). So we *cancel* some $j < r$ (why $j < r$?) *at this step*, and set $f(x_i) = 1 \dot{-} \phi_j(x_i)$.

From the above follows that we will have an *infinite sequence* of indices, j_i, that we will cancel, one for each x_i:

$$\begin{array}{cccccc} j_1 & < & j_2 & < & j_3 & < \ldots < r \\ \uparrow & & \uparrow & & \uparrow & \\ \text{due to } x_1 & & \text{due to } x_2 & & \text{due to } x_3 & \end{array}$$

The inequalities are clear: Since r cannot be cancelled, it must be $j_1 < r$, and indeed j_1 is the smallest available. For x_2, index j_1 is already cancelled, so the next larger uncanceled index, j_2, is cancelled. Always, it must be that the indices we cancel are below r, as the latter is never cancelled.

This leads us to the absurdity that we have an infinite ascending sequence of integers between j_1 and r. We conclude that $\Phi_r(x) > g(x)$ a.e. as claimed.

But why is f recursive? We build f together with $\lambda x.c(x)$, the latter a function that stores, via prime power coding, the cancelled indices $i \le x$, *after* $f(x)$ has been defined. We start with c, but first we recall the notation $x \in z$ from the Ackerman lectures. The function c is given by a primitive recursion.

$$\begin{cases} c(0) & = 1 \\ c(x+1) & = c(x) * \Big\langle \text{if } (\exists y)_{\le x}\big(y \in I(x) \wedge \neg y \overset{\overset{\text{uncancelled}}{\downarrow}}{\in} c(x)\big) \\ & \quad \text{then } (\mu y)_{\le x}\big(y \in I(x) \wedge \neg y \in c(x)\big) \text{ else } 1 \Big\rangle \end{cases}$$

$\lambda yx.y \in I(x)$ being in $\mathcal{R}_*$, we conclude that $c \in \mathcal{R}$. We return to f:

$$f(x) = \begin{cases} 1 \dot{-} \phi_{(\mu y)_{\le x}\left(y \in I(x) \wedge \neg y \in c(x)\right)}(x) & \text{if } (\exists y)_{\le x}\big(y \in I(x) \wedge \neg y \in c(x)\big) \\ 1 & \text{if } \neg(\exists y)_{\le x}\big(y \in I(x) \wedge \neg y \in c(x)\big) \end{cases}$$

Since $\lambda xy.\phi_x(y) \in \mathcal{P}$ and by Exercise 28 of Problem Set #1, $f \in \mathcal{P}$. It is also defined on any x since the index of ϕ is in $I(x)$. □

The above theorem establishes the existence of arbitrarily hard to compute *recursive predicates*. Does this mean that there are recursive predicates that are *not* in $\mathcal{PR}_*$? Yes! Exercise!

Bibliography

[Blu67] E. Blum, *A machine-independent theory of the complexity of recursive functions*, ACM **14** (1967), 322–336.

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

[Rog67] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.