## 0.1 URM Computations and their Arithmetization

We now return to the systematic development of the basic theory of partial recursive functions, with a view of gaining an insight in the inherent limitations of the computing processes.

Instrumental to this study is a mathematical characterization of what is going on during a URM computation as well as a mathematical "coding", as a primitive recursive predicate, of the statement "*the URM $M$, when presented with input $x$ has a terminating computation, coded by the number $y$*" —the so-called Kleene-predicate. We achieve this "mathematization" via a process that [Göd31] invented in his paper on incompleteness of arithmetic, namely, *arithmetization*. The arithmetization of URM computations is our first task in this section. This must begin with a mathematically precise definition of "URM computation".

As an "agent" executes some URM's, $M$, instructions, it generates at each step *instantaneous descriptions* (*ID*s)—intuitively, "snapshots"—of a computation.

The information each such description includes is simply the values of each variable of $M$, and the label (instruction number) of *the instruction that is about to be executed next*—the so-called *current* instruction.

In this section we will *arithmetize* URMs and their computations—just as Gödel did in the case of formal arithmetic and its proofs (loc. cit.)—and prove a cornerstone result of computability, the "normal form theorem" of Kleene that, essentially, says that the URM programming language is rich enough to allow us write a *universal program for computable functions*. Such a program, $U$, receives two inputs: One is a URM description, $M$, and the other is "data", $x$. $U$ then simulates $M$ on the data, behaving exactly as $M$ would on input $x$.

Programmers may call such a universal program an *interpreter* or a *compiler*.

**0.1.1 Definition. (Codes for Instructions)** The instructions are coded—using prime-power coding $\langle x_0, \ldots, x_n \rangle$—as follows, where $X1^i$ is short for

$$X\overbrace{1\cdots 1}^{i \text{ ones}}, \qquad i \geq 0$$

In the metalanguage we use, of course, convenient names as $x_0, X', x, y, Z$, etc., for variables.

(1) $L:$ $X1^i \leftarrow a$ has code $\langle 1, L, i, a \rangle$.

(2) $L:$ $X1^i \leftarrow X1^i + 1$ has code $\langle 2, L, i \rangle$.

(3) $L:$ $X1^i \leftarrow X1^i \dot{-} 1$ has code $\langle 3, L, i \rangle$.

(4) $L:$ **if** $X1^i = 0$ **goto** $P$ **else goto** $R$ has code $\langle 4, L, i, P, R \rangle$.

(5) $L:$ **stop** has code $\langle 5, L \rangle$. □

**0.1.2 Remark.** So we have 5 types of instructions coded as numbers, and it will be convenient to have 5 predicates that test a number for being *which* instruction code. In what follows we employ shorthand such as $(\exists z, w)_{<u}$ for $(\exists z)_{<u}(\exists w)_{<u}$, and similarly for longer quantifier groupings, as well as for $\forall$.

$$typ1(z) \stackrel{Def}{\equiv} Seq(z) \wedge lh(z) = 4 \wedge (z)_0 = 1 \wedge (\exists L, i, a)_{\leq z} z = \langle 1, L+1, i, a \rangle$$
$$typ2(z) \stackrel{Def}{\equiv} Seq(z) \wedge lh(z) = 3 \wedge (z)_0 = 2 \wedge (\exists L, i)_{\leq z} z = \langle 2, L+1, i \rangle$$
$$typ3(z) \stackrel{Def}{\equiv} Seq(z) \wedge lh(z) = 3 \wedge (z)_0 = 3 \wedge (\exists L, i)_{\leq z} z = \langle 3, L+1, i \rangle$$
$$typ4(z) \stackrel{Def}{\equiv} Seq(z) \wedge lh(z) = 5 \wedge (z)_0 = 4 \wedge (\exists L, i, P, R)_{\leq z} z = \langle 4, L+1, i, P+1, R+1 \rangle$$
$$typ5(z) \stackrel{Def}{\equiv} Seq(z) \wedge lh(z) = 2 \wedge (z)_0 = 5 \wedge (\exists L)_{\leq z} z = \langle 5, L+1 \rangle$$

Clearly, $typ1(z)$ to $typ5(z)$ are primitive recursive, and so is

$$instr(z) \stackrel{Def}{\equiv} typ1(z) \vee typ1(z) \vee typ2(z) \vee typ3(z) \vee typ4(z) \vee typ5(z)$$

$instr(z)$ is true iff $z$ codes a URM instruction. □ 

In turn, we code a URM $M$ as an ordered sequence of numbers, each being a code for an instruction. Thus given a code $z$ [i.e., $z$ codes *something*: $Seq(z)$ is true] we can determine algorithmically whether $z$ codes some URM. This remark is made precise in Theorem 0.1.3 below.

**0.1.3 Theorem.** *The relation $URM(z)$ that holds precisely if $z$ codes a URM is in $\mathcal{PR}_*$.*

*Proof.* So, $URM(z)$ says that

- $z$ satisfies $Seq(z)$ and $lh(z) \geq 2$ (at least two instructions since **stop** refers to no variables)
- **stop** appears only once, as the $lh(z)$-th instruction
- The $i$-th instruction in the URM code $z = p_0^{(z)_0+1} \cdots p_i^{(z)_i+1} \cdots p_n^{(z)_n+1}$ has label $i$, but we must account in this correspondence that labels go "1, 2, 3, ...", but members of the code $z$ are enumerated as $(z)_j$, for $j = 0, 1, 2, \ldots$ Labels values are shifted 1-to-the-right.

  Thus, $\big((z)_i\big)_1 = i + 1$.
- Instruction type 4 does not branch to illegal line numbers (i.e., less than 1 or more than the label of **stop**).

$$
\begin{aligned}
URM(z) \equiv\ & Seq(z) \wedge lh(z) \geq 2 \wedge (z)_{lh(z)\dot{-}1} = \langle 5, lh(z)\rangle \wedge \\
& (\forall i)_{<lh(z)\dot{-}1}\Bigg(\Big(instr\big((z)_i\big) \wedge \neg typ5\big((z)_i\big)\Big) \wedge \big((z)_i\big)_1 = i + 1 \wedge \\
& \Big(typ4\big((z)_i\big) \to 1 \leq \big((z)_i\big)_3 \wedge 1 \leq \big((z)_i\big)_4 \wedge \big((z)_i\big)_3 \leq lh(z) \wedge \big((z)_i\big)_4 \leq lh(z)\Big)\Bigg) \qquad \square
\end{aligned}
$$

**0.1.4 Remark. (Normalizing Input/Output:)** There is clearly no loss of generality (why?) in assuming that any URM that computes a function of $n \geq 1$ inputs does so using $X1$ through $X1^n$ as *input* variables and $X$, i.e., $X1^0$ as the *output* variable. Such a URM will have at least three instructions, since the **stop** instruction does not reference any variables and no instruction refers to two distinct variables. □

**0.1.5 Definition.** An ID of a computation of a URM $M$ is an ordered sequence $L; a_0, a_1, \ldots, a_r$, where all of $M$'s variables —that is, all those referenced in instructions in $M$— appear ***among*** the $X, X1,\ X11, \ldots X1^r$, and the $a_i$ is the value of $X1^i$.

Thus, the formal variables $X, X1,\ X11, \ldots X1^r$ is a *superset* of the variables that are actually used.

In any discussion we pick and fix such a superset.

In *metanotation* we will denote these formal variables as $\mathbf{x_0}, \ldots, \mathbf{x_r}$, while $a_i$ denotes the current value of $\mathbf{x_i}$ *immediately before* instruction $L$ is executed. $L$ points precisely to the *current instruction*, meaning the immediately next to be executed.

All IDs have the same length, and we say that ID $I_1 = L; a_0, \ldots, a_r$ *yields* ID $I_2 = P; b_0, \ldots, b_r$, in symbols $I_1 \vdash I_2$, *exactly when one* of (i)–(v) below is the case:

(i) $L$ labels "$\mathbf{x_i} \leftarrow c$", and $I_1$ and $I_2$ are identical, except that $b_i = c$ and $P = L + 1$.

(ii) $L$ labels "$\mathbf{x_i} \leftarrow \mathbf{x_i} + 1$", and $I_1$ and $I_2$ are identical, except that $b_i = a_i + 1$ and $P = L + 1$.

(iii) $L$ labels "$\mathbf{x_i} \leftarrow \mathbf{x_i} \dot{-} 1$", and $I_1$ and $I_2$ are identical, except that $b_i = a_i \dot{-} 1$ and $P = L + 1$.

(iv) $L$ labels "if $\mathbf{x_i} = 0$ goto $R$ else goto $Q$", and $I_1$ and $I_2$ are identical, except that $P = R$ if $a_i = 0$, while $P = Q$ otherwise.

(v) $L$ labels "**stop**", and $I_1$ and $I_2$ are identical.

Thus when **stop** is reached the computation continues forever trivially, *changing nothing.*

A *terminating computation* of $M$ with input $a_1, \ldots, a_k$ is a sequence of IDs $I_0, \ldots, I_n$ such that for all $i < n$ we have $I_i \vdash I_{i+1}$ and for some $j \leq n$, $I_j$ has as 0th component the label of **stop**. Moreover, $I_0$ is *initial*; that is,

$$I_0 = 1; 0, a_1, \ldots, a_k, \underbrace{0, \ldots, 0}_{r-k \text{ 0s}}$$

The above reflects the normalizing convention of 0.1.4, and the standard convention of *implicitly* —i.e., not as part of the computation— setting all the non-input variables to 0, i.e., the $\mathbf{x_{k+1}}, \ldots, \mathbf{x_r}$ and $\mathbf{x_0}$, *before* the computation "starts".

The *length* or *run time* of the computation is its number of *steps* $I_i \vdash I_{i+1}$: That is, $n$. □

We code an ID $I = L; a_0, \ldots, a_r$ as $code(I) = \langle L, a_0, \ldots, a_r \rangle$ and a terminating computation $I_0, \ldots, I_n$ by $\langle code(I_0), \ldots, code(I_n) \rangle$.

So, how long need an ID of a URM $M$ coded as $z$ be?

Just long enough so that we do not omit any variables actually used in $M$!

Given the ID's format (0.1.5), it suffices that it is as long as the *largest* index $j$ of of any variable $y_j$ —this $y_j$ is *among* the $\mathbf{x_i}$— that is used by the URM; *plus two* (why two?).

Since the maximum $j$ is $\max\{\big((z)_i\big)_2 : i < lh(z)\}$, and we have $\big((z)_i\big)_2 < z$, we adopt for ID length the *generous*, but simple, bound $\leq z + 1$ (that is, $(z-1)+2$).

Thus, all IDs of a URM coded as $z$ will have length $z + 1$.

**0.1.6 Definition.** The relation "$yield(z,u,v)$" says "the URM with code $z$ causes $u \vdash v$, where each of $u$ and $v$ are codes of IDs". □

**0.1.7 Lemma.** $yield(z,u,v) \in \mathcal{PR}_*$.

*Proof.*

$$
\begin{aligned}
&yield(z,u,v) \equiv (\exists k)_{<z}(\exists L)_{<lh(z)}\Big(L+1 = (u)_0 \wedge \Big\{ \\
&(\exists a)_{<z}\big((z)_L = \langle 1, L+1, k, a\rangle \wedge v = 2p_{k+1}^{a+1}\left\lfloor u/p_{k+1}^{\exp(k+1,u)}\right\rfloor {}^{*}\big) \vee \\
&\big((z)_L = \langle 2, L+1, k\rangle \wedge v = 2p_{k+1}u\big) \vee \\
&\big((z)_L = \langle 3, L+1, k\rangle \wedge v = 2(\text{if } (u)_{k+1} = 0 \text{ then } u \text{ else } \lfloor u/p_{k+1}\rfloor)\big) \vee \\
&(\exists P, R)_{<lh(z)}\big((z)_L = \langle 4, L+1, k, P+1, R+1\rangle \wedge \\
&\qquad\qquad v = \text{if } (u)_{k+1} = 0 \text{ then } \left\lfloor u/2^{L+2}\right\rfloor 2^{P+2} \\
&\qquad\qquad\quad \text{else } \left\lfloor u/2^{L+2}\right\rfloor 2^{R+2}\big) \vee \\
&\big((z)_L = \langle 5, L+1\rangle \wedge v = u\big)\Big\}\Big)
\end{aligned}
$$

□

**0.1.8 Theorem.** *For any $n \geq 1$, the relation $Comp^{(n)}(z,y)$, which is true iff $y$ codes a terminating computation of the $n$-input normalised URM coded by $z$ is primitive recursive.*

*Proof.* By the remark on p. 4, which normalises the input/output convention, it must be that $lh(y) \geq 3$. By the previous discussion ID length bound, and the fact that the variables of the ID are ***among*** the $\mathbf{x_i}$, we also have $lh((y)_i) = z+1$ for all $i < lh(y)$.

In the course of the proof we will want to keep our quantifiers bounded by some primitive recursive function so that the placement of $Comp^{(n)}(z,y)$ in $\mathcal{PR}_*$ can be achieved.

Observe next that

$$
\begin{aligned}
Comp^{(n)}(z,y) \equiv URM(z) \wedge Seq(y) \wedge (\forall i)_{<lh(y)}\Big(&Seq((y)_i) \wedge lh((y)_i) = z+1 \wedge \\
&((y)_i)_0 > 0\Big) \wedge lh(y) \geq 3 \wedge (\forall j)_{<lh(y)\dot{-}1} yield(z,(y)_j,(y)_{j+1}) \wedge \\
\{\textbf{Comment.}\ \text{The last ID has the label of } z\text{'s } \textbf{stop}.\}\ &((y)_{lh(y)\dot{-}1})_0 = lh(z) \wedge \\
\{\textbf{Comment.}\ \text{The initial ID.}\}\ &((y)_0)_0 = 1 \wedge ((y)_0)_1 = 0 \wedge \\
&(\forall i)_{\leq z}\big(n+1 < i \to ((y)_0)_i = 0\big)
\end{aligned}
$$

*The effect of "$L+1 :\ X1^k \leftarrow a$" on ID $u = \langle L+1, \ldots\rangle$ is to change $L+1$ to $L+2$ (effected by the factor 2) and change the current value of $\mathbf{x_k}$, which is $(u)_{k+1}$ since $L+1$ is at position 0 of $u$, stored in the ID as a factor $p_{k+1}^{\exp(k+1,u)}$. This factor we *remove* by dividing $u$ by it and then reset $\mathbf{x_k}$ to $a$, this being accomplished by *inserting* the factor $p_{k+1}^{a+1}$ instead.

**0.1.9 Corollary. (The Kleene $T$-predicate)** *For each $n \geq 1$, the Kleene predicate $T^{(n)}(z, \vec{x}_n, y)$ that is true precisely when the $n$-input URM $z$ with input $\vec{x}_n$ has a* terminating computation *$y$, is primitive recursive.*

*Proof.* By earlier remarks, $T^{(n)}(z, \vec{x}_n, y) \equiv Comp^{(n)}(z, y) \wedge \big((y)_0\big)_2 = x_1 \wedge \big((y)_0\big)_3 = x_2 \wedge \ldots \wedge \big((y)_0\big)_{n+1} = x_n$. □

Recalling that for any predicate $R(y, \vec{x})$, $(\mu y)R(y, \vec{x})$ is alternative notation for $(\mu y)\chi_R(y, \vec{x})$ we have:

**0.1.10 Corollary. (The Kleene Normal Form Theorem)**

(1) *For any input/output normalized URM $M$ of code $z$ (0.1.1) and $n$ inputs, we have that $M_X^{X1,\ldots X1^n}$ is defined on the input $\vec{x}_n$ iff $(\exists y)T^{(n)}(z, \vec{x}_n, y)$.*

(2) *There is a primitive recursive function $d$ such that for any $\lambda\vec{x}_n.f(\vec{x}_n) \in \mathcal{P}$ there is a number $z$ and we have for all $\vec{x}_n$:*

$$f(\vec{x}_n) = d\big((\mu y)T^{(n)}(z, \vec{x}_n, y)\big)$$

*Proof.* Statement (1) is immediate as "$(\exists y)T^{(n)}(z, \vec{x}_n, y)$" says that there is a terminating computation of $M$ (coded as $z$) on input $\vec{x}_n$.

For (2), let $f = M_X^{X1,\ldots X1^n}$, where $M$ is a normalized URM of code $z$. The role of $d$ is to extract, from a terminating computation's *last* ID, its 1st component. Thus, for all $y$, we let $d(y) = \big((y)_{lh(y)\dot{-}1}\big)_1$. □

In what follows, the term *computation* will stand for *terminating computation*. Note that the "complete" equality† in the corollary, (2), becomes standard equality, $=$, iff we do have a (terminating) computation.

†That is, equality on $\mathbb{N}$ extended by stipulating that $\uparrow = \uparrow$.

**0.1.11 Definition. ($\phi$-Notation of [Rog67])** We denote by $\phi_z^{(n)}$ the partial recursive $n$-ary function computed by a URM of code $z$, as $M_X^{X1,\ldots,X1^n}$. That is, $\phi_z^{(n)} = \lambda\vec{x}_n.d\big((\mu y)T^{(n)}(z,\vec{x}_n,y)\big)$. We usually write $\phi_z$ for $\phi_z^{(1)}$ and $T(z,x,y)$ for $T^{(1)}(z,x,y)$. □

**0.1.12 Remark.** If $f = \phi_i^{(n)}$, for some URM code $i$, then we call $i$ a $\phi$-index of $f$. □

We now readily obtain the very important number-theoretic characterization of $\mathcal{P}$, a class that was originally defined in **??** via the URM formalism. This result is a direct consequence of 0.1.10 and is the direct analog of Theorem **??**, about $\mathcal{PR}$.

**0.1.13 Corollary. (Number-Theoretic Characterization of $\mathcal{P}$)** *$\mathcal{P}$ is the closure of the same $\mathcal{I}$ that we used for $\mathcal{PR}$, under composition, primitive recursion, and unbounded search.*

*Proof.* If we temporarily call $\widetilde{\mathcal{P}}$ the closure that we mentioned in the corollary, then since $\mathcal{P}$ contains $\mathcal{I}$ and is closed under the three stated operations, we immediately have $\widetilde{\mathcal{P}} \subseteq \mathcal{P}$.

Conversely, ignoring closure under $(\mu y)$ for a moment, we get $\mathcal{PR} \subseteq \widetilde{\mathcal{P}}$. Thus $\lambda \vec{x}_n.d((\mu y)T^{(n)}(z, \vec{x}_n, y)) \in \widetilde{\mathcal{P}}$, for all $z$. This shows $\mathcal{P} \subseteq \widetilde{\mathcal{P}}$. □

The preceding corollary provides an alternative formalism—that is, a *syntactic*, *finite description* other than via URM programs—for the functions of $\mathcal{P}$: Via $\mathcal{P}$-derivations, which can be defined totally analogously with the case of **??**, by adding the operation of unbounded search. Both types of derivation are special cases of the general case of **??**.

**0.1.14 Remark.** (1) The *normal form theorem* says, in particular, that every *unary* function that is computable in the technical sense of **??**—or, equivalently, 0.1.13—can be expressed as an unbounded search followed by a composition, *using a toolbox of just* two *primitive recursive functions(!)*: $d$ and $\lambda zxy.\chi_T(z,x,y)$. This representation, or "normal form", is parametrized by $z$, which denotes a URM $M$ that computes the function in a normalized manner: as $M_X^{X1}$. *Thus what we had set out to do at the beginning of this section is now done:* The two-input URM **U** that computes $\lambda zx.d\big((\mu y)T(z,x,y)\big)$—a computable function by 0.1.13—*is* universal, in precise the same way that *compilers*[‡] of practical computing are: The universal URM **U** accepts two inputs—a program $M$, coded as a number $z$, *and* data for said program, $x$. It then "interprets" and acts exactly as program $z$ would on $x$, i.e., as $M_X^{X1}$.

(2) From Definition 0.1.1 it is clear that *not* every $z \in \mathbb{N}$ represents a URM. *Nevertheless, "$\lambda x.d\big((\mu y)T(z,x,y)\big)$" in Definition 0.1.11 is meaningful for* all *natural numbers $z$ regardless of whether they code a URM or not,*


*and* is in $\mathcal{P}$, by the latter's closure properties.

Thus, if $z$ is *not* a URM code, then $T(z,x,y)$ will simply be false, for all $x$, and all $y$; thus we will have $\phi_z(x)\uparrow$ for all $x$. *This is perfectly fine*! Indeed, it is consistent with the phenomenon where a real-life computer program that is *not* syntactically correct (like our $z$ here) will not be translated by the compiler and thus will not run. Therefore, for any input it will decline to offer an output; the corresponding function will be totally undefined.

Due to these considerations we *extend* the concept of $\phi$-index to all of $\mathbb{N}$, and correspondingly remove the hedging from Definition 0.1.11: "computed by a URM of code $z$, as $M_X^{X1,\dots,X1^n}$".

We now say: *For* all $z \in \mathbb{N}$, $\phi_z^{(n)}$ *denotes the function* $\lambda\vec{x}_n.d\big((\mu y)T^{(n)}(z,\vec{x}_n,y)\big)$.

(3) In view of the above redefinition, Definition **??** can now be rephrased as "$\lambda\vec{x}_n.f(\vec{x}_n) \in \mathcal{P}$ iff, for some $z \in \mathbb{N}$, $f = \phi_z^{(n)}$"—not just "for some $z$ that is a URM code". □

[‡]A "compiler" translates "high level" programs written in C, Pascal, etc., into machine language so they can be "understood" by a computer, and therefore be executed on given input data.

**0.1.15 Exercise.** Prove that every function of $\mathcal{P}$ has infinitely many $\phi$-indices.

*Hint.* There are infinitely many ways to modify a program and yet have all programs so obtained compute the same function. □

**0.1.16 Example.** The nowhere-defined function, $\emptyset$, is in $\mathcal{P}$, as it can be obtained from any invalid code. For example,

$$\emptyset = \lambda x.d\big((\mu y)T(0,x,y)\big)$$

**Pause.** Why is 0 not a URM code?◀

It can, however, also be obtained from a program that compiles all right. Setting $\widetilde{S} = \lambda yx.x+1$ we note:

(1) $\lambda x.(\mu y)\widetilde{S}(y,x) \in \mathcal{P}$ by **??** and **??**.

(2) By the techniques of **??** we can write a program for $\emptyset = \lambda x.(\mu y)\widetilde{S}(y,x)$.

As a side-effect we have that $\mathcal{PR} \neq \mathcal{P}$ and $\mathcal{R} \neq \mathcal{P}$. □

# Bibliography

[Göd31] K. Gödel, *Über formal unentsceidbare sätze der pricipia mathematica und verwandter systeme i*, Monatshefte für Math. und Physic **38** (1931), 173–198, (Also in English in Davis [**?**, 5–38]).

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