Chapter III

APPENDIX
Normal Form Theorem
S-m-n Theorem

This appendix presents the technical details of proving the *Kleene Normal Form Theorem*, and the *S-m-n Theorem* via the important technique of "arithmetization". These lead to basic results in recursive unsolvability and semi-recursiveness.

We also include a proof of Kleene's *recursion theorem* and an application (Rice's Theorem).

$\mathcal{P}$ has been defined in class as the closure of $\{\lambda x.0, \lambda x.x+1\} \cup \{\lambda \vec{x}_n.x_i : 1 \leq i \leq n \ \& \ n \in \mathbb{N}\}$ under *composition, primitive recursion* and *unbounded search*. We will assign "program codes" to each function, using our usual prime-power coding, namely,

$$\langle x_0, \ldots, x_n \rangle = \prod_{i \leq n} p_i^{x_i+1}$$

We recall the following definition, (1), of the "concatenation function".

$$x * y \stackrel{\text{def}}{=} x \cdot \prod_{i<lh(y)} p_{i+lh(x)}^{\exp(i,y)} \tag{1}$$

This yields a primitive recursive function $\lambda xy.x * y$. It is immediate that "$*$" deserves the name, because

$$\langle \vec{a} \rangle * \langle \vec{b} \rangle = \langle \vec{a}, \vec{b} \rangle$$

for all $\vec{a}$ and $\vec{b}$. We next "arithmetize" $\mathcal{P}$-functions and their "computations". We will assign "program codes" to each function. A program code—called a "Gödel number" or a "ϕ-index", or just an "index" in the literature—is, intuitively, a number in $\mathbb{N}$ that codes the "instructions" necessary to compute a $\mathcal{P}$-function.

If $i \in \mathbb{N}$ is $a^{\dagger}$ code for $f \in \mathcal{P}$, then we write

$$f = \{i\} \text{ Kleene's notation}$$

or

$$f = \phi_i^{\ddagger} \text{ Rogers' [Ro] notation}$$

Thus, either notation, $\{i\}$ or ϕ_i, denotes the function with code i.

The following table assigns inductively Gödel numbers (middle column) to all functions $\mathcal{P}$. In the table, $\widehat{f}$ indicates a code of f.

Function	**Code**	**Comment**
$\lambda x.0$	$\langle 0,1,0\rangle$	
$\lambda x.x+1$	$\langle 0,1,1\rangle$	
$\lambda \vec{x}_n.x_i$	$\langle 0,n,i,2\rangle$	
composition: $f(g_1(\vec{y}_m),\dots,g_n(\vec{y}_m))$	$\langle 1,m,\widehat{f},\widehat{g}_1,\dots,\widehat{g}_n\rangle$	f *must* be n-ary *all* g_i *must* be m-ary
primitive recursion from "basis" h and "iterated" part g	$\langle 2,n+1,\widehat{h},\widehat{g}\,\rangle$	h *must* be n-ary g *must* be $(n+2)$-ary
unbounded search: $(\mu y)f(y,\vec{x}_n)$	$\langle 3,n,\widehat{f}\,\rangle$	f *must* be $(n+1)$-ary

OK, we have been somewhat loose in our description above. "The following table assigns inductively", we have said, perhaps leading the reader to think that we are defining the codes by recursion on $\mathcal{P}$. Not so. After all, each function has infinitely many codes.

What is really involved here—see also below—is defining the set of all ϕ-indices, here called Φ, as a subset of $\{z : Seq(z)\}$.
Φ is the *smallest* set of "codes" that contains the "initial ϕ-indices"

$$\mathcal{I} = \{\langle 0,1,0\rangle, \langle 0,1,1\rangle\} \cup \{\langle 0,n,i,2\rangle : n > 0 \;\&\; 1 \le i \le n\}$$

and is closed under the following three operations:

(i) *Coding composition*: Input a and b_i $(i = 1,\dots,n)$ causes output

$$\langle 1,m,a,b_1,\dots,b_n\rangle$$

provided $(a)_1 = n$ and $(b_i)_1 = m$, for $i = 1,\dots,n$.

(ii) *Coding primitive recursion*: Input a and b causes output

$$\langle 2,n+1,a,b\rangle$$

provided $(a)_1 = n$ and $(b)_1 = n+2$.

†The indefinite article is appropriate here. Exactly as in "real life" a "computable" function has infinitely many different programs that compute it, a partial recursive function f has infinitely many different codes (see 0.3 later on).

‡That's where the name "ϕ-index" comes from.

(iii) *Coding unbounded search*: Input a causes output

$$\langle 3, n, a\rangle$$

provided $(a)_1 = n + 1$.

By the uniqueness of prime number decomposition, the following recursive definition that assigns functions to indices is *unambiguous*.

We define by recursion on Φ a *total* function $\lambda a.\{a\}$ (or $\lambda a.\phi_a$) for each $a \in \Phi$, that is, a function that maps codes to functions of $\mathcal{P}$:

$$\begin{aligned}
\{\langle 0,1,0\rangle\} &= \lambda x.0\\
\{\langle 0,1,1\rangle\} &= \lambda x.x+1\\
\{\langle 0,n,i,2\rangle\} &= \lambda \vec{x}_n.x_i\\
\{\langle 1,m,a,b_1,\dots,b_n\rangle\} &= \lambda \vec{y}_m.\{a\}(\{b_1\}(\vec{y}_m),\dots,\{b_n\}(\vec{y}_m))\\
\{\langle 2,n+1,a,b\rangle\} &= \lambda x\vec{y}_n.Prec(\{a\},\{b\})\\
\{\langle 3,n,a\rangle\} &= \lambda \vec{x}_n.(\mu y)\{a\}(y,\vec{x}_n)
\end{aligned}$$

In the above recursive definition we have used the abbreviation $Prec(\{a\},\{b\})$ for the function given (for all $x, \vec{y}_n$) by the primitive recursive schema with "h-part" $\{a\}$ and "g-part" $\{b\}$.

We can now make the intentions implied in the above table official:

0.1 Theorem. $\mathcal{P} = \{\{a\} : a \in \Phi\}$.

Proof. $\subseteq$-part. Induction on $\mathcal{P}$. The previous table encapsulates the argument diagrammatically.

$\supseteq$-part. Induction on Φ. It follows trivially from the recursive definition of $\{a\}$ and the fact that $\mathcal{P}$ contains the initial functions and is closed under composition, primitive recursion and unbounded search. □

0.2 Remark. (*Important*) Thus, $f \in \mathcal{P}$ iff for some $a \in \Phi$, $f = \{a\}$.

0.3 Example. Every function $f \in \mathcal{P}$ has infinitely many ϕ-indices. Indeed, let $f = \{\widehat{f}\}$. Since $f = \lambda \vec{x}_n.u^1_1(f(\vec{x}_n))$, we obtain $f = \{\langle 1, n, \langle 0,1,1,2\rangle, \widehat{f}\,\rangle\}$. Since $\langle 1, n, \langle 0,1,1,2\rangle, \widehat{f}\,\rangle > \widehat{f}$, the claim follows. □

0.4 Theorem. *The relation $x \in \Phi$ is primitive recursive.*

Proof. Let χ denote the characteristic function of $x \in \Phi$. Then

$$\begin{aligned}
\chi(0) &= 1\\
\chi(x+1) &= 0 \textbf{ if } \quad x+1 = \langle 0,1,0\rangle \vee x+1 = \langle 0,1,1\rangle \vee\\
&\quad (\exists n,i)_{\leq x}\Big(n>0 \;\&\; 0<i\leq n \;\&\; x+1=\langle 0,n,i,2\rangle\Big)\vee\\
&\quad (\exists a,b,m,n)_{\leq x}\Big(\chi(a)=0 \;\&\; (a)_1 = n \;\&\; Seq(b) \;\&\\
&\quad lh(b)=n \;\&\; (\forall i)_{<n}\big(\chi((b)_i)=0 \;\&\; ((b)_i)_1 = m\big) \;\&\\
&\quad x+1=\langle 1,m,a\rangle * b\Big)\vee\\
&\quad (\exists a,b,n)_{\leq x}\Big(\chi(a)=0 \;\&\; (a)_1=n \;\&\; \chi(b)=0 \;\&\\
&\quad (b)_1 = n+2 \;\&\; x+1=\langle 2,n+1,a,b\rangle\Big)\vee\\
&\quad (\exists a,n)_{\leq x}\Big(\chi(a)=0 \;\&\; (a)_1=n+1 \;\&\; x+1=\langle 3,n,a\rangle\Big)\\
&= 1 \textbf{ otherwise}
\end{aligned}$$

The above can easily be seen to be a course-of-values recursion. For example, if $H(x) = \langle \chi(0),\dots,\chi(x)\rangle$, then an occurrence of "$\chi(a) = 0$" above can be replaced by "$(H(x))_a = 0$", since $a \leq x$. $\square$

We think of† a *computation* as a *sequence of equations* like $\{e\}(\vec{a}) = b$. Such an equation is intuitively read as "the program e, when it executes on input $\vec{a}$, produces output b". An equation will be *legitimate* iff

(i) it states an input-output relation of some initial function (i.e., case where $(e)_0 = 0$), or

(ii) it states an input-output relation according to ϕ-indices e such that $(e)_0 \in \{1,2,3\}$, using results (i.e., equations) that *already appeared in the sequence.*

For example, in order to state $(\mu y)\{e\}(y,\vec{a}_n) = b$ one *must* ensure that *all* the following equations, where the r_i's are *all* non-zero,

$$\{e\}(b,\vec{a}_n) = 0, \{e\}(0,\vec{a}_n) = r_0,\dots,\{e\}(b-1,\vec{a}_n) = r_{b-1}$$

have already appeared in the sequence. In our coding, every equation $\{a\}(\vec{a}_n) = b$ will be denoted by a triple $\langle e,\vec{a}_n,b\rangle$ that codes, in that order, the ϕ-index, the input and the output. We will collect (code) all these triples into a single code, $u = \langle \dots,\langle e,\vec{a}_n,b\rangle,\dots\rangle$.

Before proceeding, let us recall that $Seq(z) \leftrightarrow x>1 \;\&\; (\forall y)_{\leq x}(\forall z)_{\leq x}(y|x \;\&\; Pr(y) \;\&\; Pr(z) \;\&\; z<y \rightarrow z|x)$ and $lh(z) = (\mu y)_{\leq z}\neg p_y|z$. We also define the following two primitive recursive predicates:

† "We think of" indicates our determination to avoid a rigorous definition. The integrity of our exposition will not suffer from this.

(1) $\lambda uv.u \in v$ ("v is a term in the (coded) sequence u")

(2) $\lambda uvw.v <_u w$ ("v occurs *before* w in the (coded) sequence u")

Primitive recursiveness follows from the equivalences

$$v \in u \leftrightarrow Seq(u)\ \&\ (\exists i)_{\leq u}(u)_i = v$$
$$v <_u w \leftrightarrow v \in u\ \&\ w \in u\ \&\ (\exists i,j)_{\leq u}((u)_i = v\ \&\ (u)_j = w\ \&\ i < j)$$

We are now ready to define the relation "$Computation(u)$" which holds iff "u codes a computation according to the previous understanding". This involves a lengthy formula. In the interest of readability, comments enclosed in { }-brackets are included on the left margin, to indicate the case under consideration.

$$\begin{aligned}
Computation(u) \leftrightarrow\ & Seq(u)\ \&\ (\forall v)_{\leq u}\Big[v \in u \rightarrow \\
\{\lambda x.0\}\quad & (\exists x)_{\leq u} v = \langle\langle 0,1,0\rangle, x, 0\rangle \vee \\
\{\lambda x.x+1\}\quad & (\exists x)_{\leq u} v = \langle\langle 0,1,1\rangle, x, x+1\rangle \vee \\
\{\lambda \vec{x}_n.x_i\}\quad & (\exists x,n,i)_{\leq u}\{Seq(x)\ \&\ n = lh(x)\ \&\ i < n\ \& \\
& v = \langle\langle 0,n,i+1,2\rangle\rangle * x * \langle (x)_i\rangle\} \vee \\
\{composition\}\quad & (\exists x,y,\widehat{f},\widehat{g},m,n,z)_{\leq u}\{Seq(x)\ \&\ Seq(y)\ \&\ Seq(\widehat{f})\ \& \\
& Seq(\widehat{g})\ \&\ n = lh(x)\ \&\ n = lh(\widehat{g})\ \&\ m = lh(y)\ \& \\
& (\widehat{f})_1 = n\ \&\ (\forall i)_{<n}(Seq((\widehat{g})_i)\ \&\ ((\widehat{g})_i)_1 = m)\ \& \\
& v = \langle\langle 1,m,\widehat{f}\rangle * \widehat{g}\rangle * y * \langle z\rangle\ \& \\
& \langle\widehat{f}\rangle * x * \langle z\rangle <_u v\ \& \\
& (\forall i)_{<n}\langle(\widehat{g})_i\rangle * y * \langle (x)_i\rangle <_u v\} \vee \\
\{prim.\ recursion\}\quad & (\exists x,y,\widehat{h},\widehat{g},n,c)_{\leq u}\{Seq(\widehat{h})\ \&\ (\widehat{h})_1 = n\ \&\ Seq(\widehat{g})\ \& \\
& (\widehat{g})_1 = n+2\ \&\ Seq(y)\ \&\ lh(y) = n\ \&\ Seq(c)\ \& \\
& lh(c) = x+1\ \&\ v = \langle\langle 2,n+1,\widehat{h},\widehat{g}\rangle, x\rangle * y * \langle (c)_x\rangle\ \& \\
& \langle\widehat{h}\rangle * y * \langle (c)_0\rangle <_u v\ \&\ (\forall i)_{<x}\langle\widehat{g}, i\rangle * y * \langle (c)_i, (c)_{i+1}\rangle <_u v\} \vee \\
\{(\mu y)f(y,\vec{x}_n)\}\quad & (\exists \widehat{f},y,x,n,r)_{\leq u}\{Seq(\widehat{f})\ \&\ (\widehat{f})_1 = n+1\ \& \\
& Seq(x)\ \&\ lh(x) = n\ \&\ Seq(r)\ \&\ lh(r) = y\ \& \\
& v = \langle\langle 3,n,\widehat{f}\rangle\rangle * x * \langle y\rangle\ \& \\
& \langle\widehat{f}, y\rangle * x * \langle 0\rangle <_u v\ \& \\
& (\forall i)_{<y}(\langle\widehat{f}, i\rangle * x * \langle (r)_i\rangle <_u v\ \&\ (r)_i > 0)\}\Big]
\end{aligned}$$

By inspection of the formula to the right of "$\leftrightarrow$" above we see that $Computation(u)$ is primitive recursive.

0.5 Definition (*The Kleene T-predicate*). For each $n \in \mathbb{N}$, $T^{(n)}(a, \vec{x}_n, z)$ stands for $Computation((z)_1)$ & $\langle a, \vec{x}_n, (z)_0 \rangle \in (z)_1$. □

The above discussion yields immediately:

0.6 Theorem (*Kleene Normal Form Theorem*).
(1) $y = \{a\}(\vec{x}_n) \equiv (\exists z)(T^{(n)}(a, \vec{x}_n, z)\ \&\ (z)_0 = y)$
(2) $\{a\}(\vec{x}_n) = \big((\mu z)T^{(n)}(a, \vec{x}_n, z)\big)_0$
(3) $\{a\}(\vec{x}_n) \downarrow \equiv (\exists z)T^{(n)}(a, \vec{x}_n, z)$.

0.7 Remark. (*Very important*) The right hand side of 0.6(2), above, is *meaningful* for all $a \in \mathbb{N}$, while the left hand side is only meaningful for $a \in \Phi$.

We now extend the symbols $\{a\}$ and ϕ_a to be meaningful for all $a \in \mathbb{N}$. *In all cases, the meaning is given by the right hand side of (2).*

Of course, if $a \notin \Phi$, then $(\mu z)T^{(n)}(a, \vec{x}_n, z) \uparrow$, *for all* $\vec{x}_n$, since $T^{(n)}(a, \vec{x}_n, z)$ will be false under the circumstances. Hence also $\{a\}(\vec{x}_n) \uparrow$, as it should be intuitively. In computer programmer's jargon: "If the 'program' a is 'syntactically incorrect', then it will not 'run'. Thus, it will 'define' the *everywhere undefined function*."

We can now define a $\mathcal{P}$-counterpart to $\mathcal{R}_*$ and $\mathcal{PR}_*$ and consider its closure properties.

0.8 Definition. (*Semi-recursive relations* or *predicates*) A relation $P(\vec{x})$ is *semi-recursive* iff for some $f \in \mathcal{P}$, the equivalence

$$P(\vec{x}) \leftrightarrow f(\vec{x}) \downarrow \tag{1}$$

holds (for all $\vec{x}$, of course). Equivalently, we can state that $P = \mathrm{dom}(f)$.

The set of all semi-recursive relations is denoted by $\mathcal{P}_*$[†]

If $f = \{a\}$ in (1) above, then we say that "a is *a* semi-recursive index of P".

If P has one argument (i.e., $P \subseteq \mathbb{N}$) and a is one of its semi-recursive indices, then we write $P = W_a$ ([Ro]). □

We have at once

0.9 Corollary. (Normal Form Theorem for semi-recursive relations)
$P(\vec{x}_n) \in \mathcal{P}_*$ *iff, for some* $a \in \mathbb{N}$,

$$P(\vec{x}_n) \leftrightarrow (\exists z)T^{(n)}(a, \vec{x}_n, z).$$

Proof. only if-part. This is 0.6(3).

[†]We are making this symbol up. It is not standard in the literature.

if-part. $(\exists z)T^{(n)}(a,\vec{x}_n,z) \leftrightarrow (\mu z)T^{(n)}(a,\vec{x}_n,z)\downarrow$.
But $\lambda\vec{x}_n.(\mu z)T^{(n)}(a,\vec{x}_n,z) \in \mathcal{P}$. $\square$

Rephrasing the above (hiding the "a", and remembering that $\mathcal{P}\mathcal{R}_* \subseteq \mathcal{R}_*$) we have

0.10 Corollary. (Strong Projection Theorem) $P(\vec{x}_n) \in \mathcal{P}_*$ *iff, for some* $Q(\vec{x}_n,z) \in \mathcal{R}_*$,

$$P(\vec{x}_n) \leftrightarrow (\exists z)Q(\vec{x}_n,z).$$

Here is a characterization of $\mathcal{P}_*$ that is identical, *in form*, to the characterizations of $\mathcal{P}\mathcal{R}_*$ and $\mathcal{R}_*$.

0.11 Corollary. $P(\vec{x}_n) \in \mathcal{P}_*$ *iff, for some* $f \in \mathcal{P}$,

$$P(\vec{x}_n) \leftrightarrow f(\vec{x}_n) = 0.$$

Proof. only if-part. Say $P(\vec{x}_n) \leftrightarrow g(\vec{x}_n)\downarrow$. Take $f = \lambda\vec{x}_n.0 \cdot g(\vec{x}_n)$.

if-part. Let $f = \{a\}$. By 0.6(1), $f(\vec{x}_n) = 0 \leftrightarrow (\exists z)\big(T^{(n)}(a,\vec{x}_n,z)\ \&\ (z)_0 = 0\big)$. We are done by strong projection. $\square$

We immediately obtain

0.12 Corollary. $\mathcal{R}_* \subseteq \mathcal{P}_*$.

Intuitively, for a predicate $R \in \mathcal{R}_*$ we have an algorithm (one that computes χ_R) that for any input $\vec{x}$ *will halt* and answer "yes" ($= 0$) or "no" ($= 1$) to the question "$\vec{x} \in R$?"

For a predicate $Q \in \mathcal{P}_*$ we are *only* guaranteed the existence of a weaker algorithm (for $f \in \mathcal{P}$ such that $\mathrm{dom}(f) = Q$). It will halt iff the answer to the question "$\vec{x} \in Q$?" is "yes" (and halting will amount to "yes"). If the answer is "no" it will never tell, because it will (as we say for non halting) "loop for ever" (or diverge). Hence the name "semi-recursive" for such predicates.

0.13 Theorem. $R \in \mathcal{R}_*$ *iff both* R *and* $\neg R$ *are in* $\mathcal{P}_*$.

Proof. only if-part. By 0.11 and closure of $\mathcal{R}_*$ under $\neg$.

if-part. Let i and j be semi-recursive indices of R and $\neg R$ respectively, that is

$$R(\vec{x}_n) \leftrightarrow (\exists z)T^{(n)}(i,\vec{x}_n,z)$$
$$\neg R(\vec{x}_n) \leftrightarrow (\exists z)T^{(n)}(j,\vec{x}_n,z)$$

Define

$$g = \lambda \vec{x}_n.(\mu z)\big(T^{(n)}(i, \vec{x}_n, z) \vee T^{(n)}(j, \vec{x}_n, z)\big)$$

Trivially, $g \in \mathcal{P}$. Hence, $g \in \mathcal{R}$, since it is total (Why?). We are done by noticing that $R(\vec{x}_n) \leftrightarrow T^{(n)}(i, \vec{x}_n, g(\vec{x}_n))$. $\square$

(*Unsolvable Problems*) A *problem* is a question "$\vec{x} \in R$?" for any predicate R. "The problem $\vec{x} \in R$ is *recursively unsolvable*", or just *unsolvable*, means that $R \notin \mathcal{R}_*$, that is, intuitively, there is *no* algorithmic solution to the problem.

The "halting problem" has central significance in recursion theory. It is the question whether "program x will ever halt if it starts computing on input x". That is, we set $K = \{x : \{x\}(x) \downarrow\}$. The halting problem is $x \in K$.[†] We denote the complement of K by $\overline{K}$.

0.14 Theorem. (Unsolvability of the halting problem) *The halting problem is unsolvable.*

Proof. It suffices to show that $\overline{K}$ is not semi-recursive. Suppose instead that i is a semi-recursive index of the set. Thus,

$$x \in \overline{K} \leftrightarrow (\exists z) T^{(1)}(i, x, z)$$

or, making the part $x \in \overline{K}$—that is, $\{x\}(x) \uparrow$—explicit

$$\neg(\exists z) T^{(1)}(x, x, z) \leftrightarrow (\exists z) T^{(1)}(i, x, z) \tag{1}$$

Substituting i into x in (1) we get a contradiction. $\square$

$K \in \mathcal{P}_*$, of course, since $\{x\}(x) \downarrow \leftrightarrow (\exists z) T^{(1)}(x, x, z)$. We conclude that the inclusion $\mathcal{R}_* \subseteq \mathcal{P}_*$ is proper, i.e., $\mathcal{R}_* \subset \mathcal{P}_*$.

0.15 Theorem. (Closure properties of $\mathcal{P}_*$)
$\mathcal{P}_*$ *is closed under* $\vee, \&, (\exists y)_{<z}, (\exists y), (\forall y)_{<z}$. *It is* not *closed under either* $\neg$ *or* $(\forall y)$.

Proof. We will rely on the normal form theorem for semi-recursive relations and the strong projection theorem.

Given semi-recursive relations $P(\vec{x}_n)$, $Q(\vec{y}_m)$ and $R(y, \vec{u}_k)$ of semi-recursive indices p, q, r respectively.

($\vee$:)

$$\begin{aligned} P(\vec{x}_n) \vee Q(\vec{y}_m) &\leftrightarrow (\exists z) T^{(n)}(p, \vec{x}_n, z) \vee (\exists z) T^{(m)}(q, \vec{y}_m, z) \\ &\leftrightarrow (\exists z)\big(T^{(n)}(p, \vec{x}_n, z) \vee T^{(m)}(q, \vec{y}_m, z)\big) \end{aligned}$$

[†] "K" is a reasonably well reserved symbol for the set $\{x : \{x\}(x) \downarrow\}$. Unfortunately, K is also used for the "first projection" of a "pairing function", but the context easily decides which is which.

(&:)

$$\begin{aligned} P(\vec{x}_n)\ \&\ Q(\vec{y}_m) &\leftrightarrow (\exists z)T^{(n)}(p,\vec{x}_n,z)\ \&\ (\exists z)T^{(m)}(q,\vec{y}_m,z)\\ &\leftrightarrow (\exists w)\big((\exists z)_{<w}T^{(n)}(p,\vec{x}_n,z)\ \&\ (\exists z)_{<w}T^{(m)}(q,\vec{y}_m,z)\big)\end{aligned}$$

Breaking the pattern established by the proof for $\vee$ we may suggest a simpler proof: $P(\vec{x}_n)\ \&\ Q(\vec{y}_m) \leftrightarrow \Big((\mu z)T^{(n)}(p,\vec{x}_n,z) + (\mu z)T^{(m)}(q,\vec{y}_m,z)\Big)\downarrow$. Yet another proof, involving the decoding function $\lambda iz.(z)_i$ is

$$\begin{aligned} P(\vec{x}_n)\ \&\ Q(\vec{y}_m) &\leftrightarrow (\exists z)T^{(n)}(p,\vec{x}_n,z)\ \&\ (\exists z)T^{(m)}(q,\vec{y}_m,z)\\ &\leftrightarrow (\exists z)\big(T^{(n)}(p,\vec{x}_n,(z)_0)\ \&\ T^{(m)}(q,\vec{y}_m,(z)_1)\big)\end{aligned}$$

There is a technical reason (soon to manifest itself) that we want to avoid "complicated" functions like $\lambda iz.(z)_i$ in the proof.

$((\exists y)_{<z}$:)

$$\begin{aligned}(\exists y)_{<z}R(y,\vec{u}_k) &\leftrightarrow (\exists y)_{<z}(\exists w)T^{(k+1)}(r,y,\vec{u}_k,w)\\ &\leftrightarrow (\exists w)(\exists y)_{<z}T^{(k+1)}(r,y,\vec{u}_k,w)\end{aligned}$$

$((\exists y)$:)

$$\begin{aligned}(\exists y)R(y,\vec{u}_k) &\leftrightarrow (\exists y)(\exists w)T^{(k+1)}(r,y,\vec{u}_k,w)\\ &\leftrightarrow (\exists z)(\exists y)_{<z}(\exists w)_{<z}T^{(k+1)}(r,y,\vec{u}_k,w)\end{aligned}$$

Both of the $\exists$-cases can be handled by the decoding function $\lambda iz.(z)_i$. For example,

$$\begin{aligned}(\exists y)R(y,\vec{u}_k) &\leftrightarrow (\exists y)(\exists w)T^{(k+1)}(r,y,\vec{u}_k,w)\\ &\leftrightarrow (\exists z)T^{(k+1)}(r,(z)_0,\vec{u}_k,(z)_1)\end{aligned}$$

$((\forall y)_{<z}$:)

$$\begin{aligned}(\forall y)_{<z}R(y,\vec{u}_k) &\leftrightarrow (\forall y)_{<z}(\exists w)T^{(k+1)}(r,y,\vec{u}_k,w)\\ &\leftrightarrow (\exists v)(\forall y)_{<z}(\exists w)_{<v}T^{(k+1)}(r,y,\vec{u}_k,w)\end{aligned}$$

Think of v above as the successor $(+1)$ of the maximum of some set of w-values, $w_0,\ldots,w_{z-1}$, that "work" for $y=0,\ldots,z-1$ respectively. The usual overkill proof of the above involves $(z)_i$ (or some such decoding scheme) as follows:

$$\begin{aligned}(\forall y)_{<z}R(y,\vec{u}_k) &\leftrightarrow (\forall y)_{<z}(\exists w)T^{(k+1)}(r,y,\vec{u}_k,w)\\ &\leftrightarrow (\exists w)(\forall y)_{<z}T^{(k+1)}(r,y,\vec{u}_k,(w)_y)\end{aligned}$$

Regarding closure under $\neg$ and $\forall y$, K provides a counterexample to $\neg$, and $\neg T^{(1)}(x,x,y)$ provides a counterexample to $\forall y$. $\square$

(*Recursively enumerable predicates*) A predicate $R(\vec{x}_n)$ is *recursively enumerable*, or *r.e.*, iff $R = \emptyset$ or, for some $f \in \mathcal{R}$ of one variable, $R = \{\vec{x}_n : (\exists m) f(m) = \langle \vec{x}_n \rangle\}$, or, equivalently

$$R(\vec{x}_n) \leftrightarrow (\exists m) f(m) = \langle \vec{x}_n \rangle \tag{1}$$

By (1), R every r.e. relation is semi-recursive. The converse is also true.

0.16 Theorem. *Every semi-recursive R is r.e.*

Proof. Let a be a semi-recursive index of R. If $R = \emptyset$ then we are done. Suppose then $R(\vec{a}_n)$ for some $\vec{a}_n$. We define a function f by cases

$$f(m) = \begin{cases} \langle (m)_0, \dots, (m)_{n-1} \rangle & \textbf{if } T^{(n)}(a, (m)_0, \dots, (m)_{n-1}, (m)_n) \\ \langle \vec{a}_n \rangle & \textbf{otherwise} \end{cases}$$

It is trivial that f is recursive and satisfies (1) above. Indeed, our f is in $\mathcal{PR}$. $\square$

Suppose that i codes a "program" that acts on input variables x and y to compute a function $\lambda xy.f(x,y)$. It is certainly trivial to modify the program to compute $\lambda x.f(x,a)$ instead. In computer programming terms, we replace a "command" such as "read y" by one that says "$y := a$" (copy the value of a into y). From the original code, a new code (depending on i and a) ought to be trivially calculated.

This is the essence of Kleene's *iteration* or "S-m-n" theorem below.

0.17 Theorem. (Kleene's S-m-n *or* iteration theorem) *There is a primitive recursive function $\lambda xy.\sigma(x,y)$ such that for all i, x, y,*

$$\{i\}(\langle x, y \rangle) = \{\sigma(i,y)\}(x)$$

Proof. Let a be a ϕ-index of $\lambda x.\langle x, 0 \rangle$ and b a ϕ-index of $\lambda x.3x$.

Next we find a primitive recursive $\lambda y.h(y)$ such that for all x and y

$$\{h(y)\}(x) = \langle x, y \rangle \tag{$*$}$$

To achieve this observe that

$$\langle x, 0 \rangle = \{a\}(x)$$

and

$$\langle x, y+1\rangle = 3\langle x, y\rangle = \{b\}(\langle x, y\rangle)$$

Thus, it *suffices* to take

$$\begin{aligned} h(0) &= a \\ h(y+1) &= \langle 1, 1, b, h(y)\rangle \end{aligned}$$

Now that we have an h satisfying $(*)$, we note that

$$\sigma(i, y) \stackrel{\text{def}}{=} \langle 1, 1, i, h(y)\rangle$$

will do. □

0.18 Corollary. *There is a primitive recursive function $\lambda iy.k(i, y)$ such that, for all i, x, y,*

$$\{i\}(x, y) = \{k(i, y)\}(x).$$

Proof. Let a_0 and a_1 be ϕ-indices of $\lambda z.(z)_0$ and $\lambda z.(z)_1$ respectively. Then $\{i\}((z)_0, (z)_1) = \{\langle 1, 1, i, a_0, a_1\rangle\}(z)$ for all z, i. Take $k(i, y) = \sigma(\langle 1, 1, i, a_0, a_1\rangle, y)$. □

0.19 Corollary. *There is for each $m > 0$ and $n > 0$ a primitive recursive function $\lambda i\vec{y}_n.S^m_n(i, \vec{y}_n)$ such that, for all $i, \vec{x}_m, \vec{y}_n$,*

$$\{i\}(\vec{x}_m, \vec{y}_n) = \{S^m_n(i, \vec{y}_n)\}(\vec{x}_m).$$

Proof. Let a_r $(r = 0, \ldots, m-1)$ and b_r $(r = 0, \ldots, n-1)$ be ϕ-indices so that $\{a_r\} = \lambda xy.(x)_r$ $(r = 0, \ldots, m-1)$ and $\{b_r\} = \lambda xy.(y)_r$ $(r = 0, \ldots, n-1)$.

Set $c(i) = \langle 1, 2, i, a_0, \ldots, a_{m-1}, b_0, \ldots, b_{n-1}\rangle$, for all $i \in \mathbb{N}$, and let d be a ϕ-index of $\lambda\vec{x}_m.\langle\vec{x}_m\rangle$.

Then,

$$\begin{aligned} \{i\}(\vec{x}_m, \vec{y}_n) &= \{c(i)\}(\langle\vec{x}_m\rangle, \langle\vec{y}_n\rangle) \\ &= \{k(c(i), \langle\vec{y}_n\rangle)\}(\langle\vec{x}_m\rangle) \qquad \text{by 0.18} \\ &= \{\langle 1, m, k(c(i), \langle\vec{y}_n\rangle), d\rangle\}(\vec{x}_m) \end{aligned}$$

Take $\lambda i\vec{y}_n.S^m_n = \langle 1, m, k(c(i), \langle\vec{y}_n\rangle), d\rangle$. □

0.20 Corollary. (Kleene's recursion theorem) *If $\lambda z\vec{x}.f(z, \vec{x}_n) \in \mathcal{P}$, then for some e,*

$$\{e\}(\vec{x}_n) = f(e, \vec{x}_n) \text{ for all } \vec{x}_n.$$

Proof. Let $\{a\} = \lambda z \vec{x}_n . f(S_1^n(z,z), \vec{x}_n)$. Then

$$\begin{aligned} f(S_1^n(a,a), \vec{x}_n) &= \{a\}(a, \vec{x}_n) \\ &= \{S_1^n(a,a)\}(\vec{x}_n) \qquad \text{by 0.19} \end{aligned}$$

Take $e = S_1^n(a,a)$. □

0.21 Definition. A *complete index set* is a set $A = \{x : \{x\} \in \mathcal{Q}\}$ for some $\mathcal{Q} \subseteq \mathcal{P}$.

A is *trivial* iff $A = \emptyset$ or $A = \mathbb{N}$ (correspondingly, $\mathcal{Q} = \emptyset$ or $\mathcal{Q} = \mathcal{P}$). Otherwise it is *non trivial*. □

0.22 Theorem. (Rice) *A complete index set is recursive iff it is trivial.*

Thus, "algorithmically" we can only "decide" trivial properties of "programs".

Proof. (The idea of this proof is attributed in [Ro] to G.C. Wolpin.)

if-part. Immediate, since $\chi_\emptyset = \lambda x.1$.

only if-part. By contradiction, suppose that $A = \{x : \{x\} \in \mathcal{Q}\}$ is non trivial, yet $A \in \mathcal{R}_*$. So, let $a \in A$ and $b \notin A$. Define f by

$$f(x) = \begin{cases} b & \textbf{if } x \in A \\ a & \textbf{if } x \notin A \end{cases}$$

Clearly,

$$x \in A \text{ iff } f(x) \notin A, \text{ for all } x \tag{1}$$

By the recursion theorem, there is an e such that $\{f(e)\} = \{e\}$ (apply 0.20 to $\lambda xy.\{f(x)\}(y)$).

Thus, $e \in A$ iff $f(e) \in A$, contradicting (1). □