Chapter 3

APPENDIX
Arithmetisation
Normal Form Theorems
Semi-Recursiveness and Unsolvability
S-m-n and Recursion Theorems

This appendix to Chapter 3 of [Tou84] retells in an alternative way the technical details of proving the *Kleene Normal Form Theorem*, and the *S-m-n Theorem* via the important technique of "arithmetisation". These lead to basic results in recursive unsolvability and semi-recursiveness. We also include a proof of Kleene's *recursion theorem* with an application to unsolvability (Rice's Theorem) and to proving that certain "recursive definitions" have partial recursive "solutions".

3.1. Computations and their Arithmetisation; The Kleene Predicate

$\mathcal{P}$ has been defined in class as the closure of $\{\lambda x.0, \lambda x.x+1\} \cup \{\lambda \vec{x}_n.x_i : 1 \leq i \leq n \wedge n \in \mathbb{N}\}$ 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 define in (1) below 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 "arithmetise" $\mathcal{P}$-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[†] code for $f \in \mathcal{P}$, then we write

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

or

$$f = \phi_i \text{[‡] Rogers's [Rog67] notation}$$

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**
$\lambda x.0$	$\langle 0,1,0 \rangle$	
$\lambda x.x+1$	$\langle 0,1,1 \rangle$	
$\lambda \vec{x}_n.x_i$	$\langle 0,n,i,2 \rangle$	$1 \le i \le n$
composition: $f(g_1(\vec{y}_m),\ldots,g_n(\vec{y}_m))$	$\langle 1,m,\widehat{f},\widehat{g}_1,\ldots,\widehat{g}_n \rangle$	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 and $n>0$

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"

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

and is closed under the following three operations:

[†]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 3.1.3 later on).

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

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

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

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

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

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

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

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

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

provided $(a)_1 = n + 1$ and $n > 0$.[†]

By the uniqueness of prime number decomposition, the rules defining the set Φ are *unambiguous*, hence we may define an interpretation (or *semantics*) function on Φ by recursion on Φ.

Indeed we define a *total* function $\lambda a.\{a\}$ (or $\lambda a.\phi_a$) for each $a \in \Phi$, that is, a function that maps codes into functions of $\mathcal{P}$:

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

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:

3.1.1 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. □

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

3.1.3 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. □

[†]By an obvious I.H. the other cases can fend for themselves, but, here, reducing the number of arguments must not result to 0 arguments, as we have decided not to allow 0-ary functions.

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

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

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

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

How do we compute a yes/no answer to the question "$\{e\}(\vec{a}) = b$?" for arbitrary $e, \vec{a}, b$? Just as in the case of the Ackermann function, we think of the question "$\{e\}(\vec{a}) = b$?" as a "call" to a function

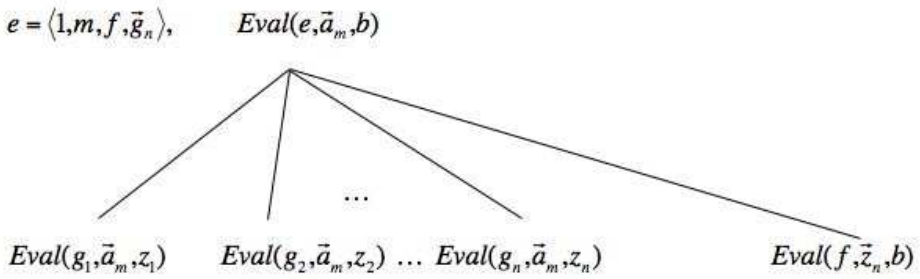
$$Eval(e, \vec{a}, b)$$

We can "program" *Eval* recursively:

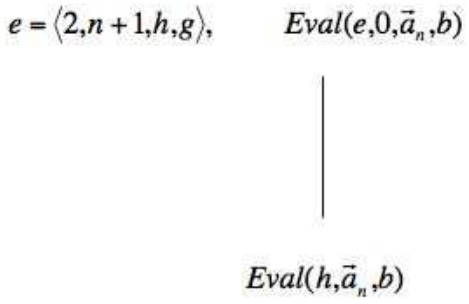
(i) If e codes an initial function, that is, $(e)_0 = 0$, then we directly check if said initial function on input $\vec{a}$ produces b and we answer "yes" or "no" accordingly.

(ii) If $(e)_0 \in \{1,2,3\}$, then we have cases shown in tree form below. In each case the *root call* is $Eval(e, \vec{a}, b)$ and the "children" represent the recursive calls *needed* —according to the definition of the ϕ-indices and their semantics given earlier (pages 2 and 3)— to determine

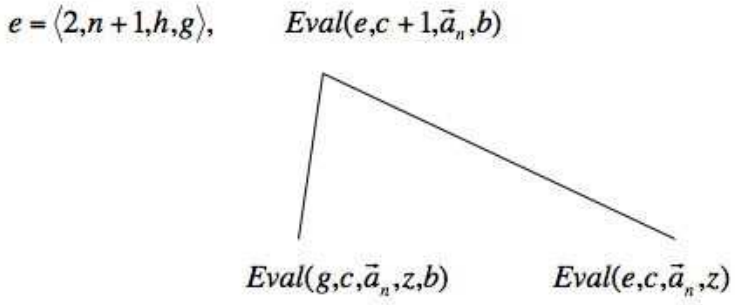
the (yes/no) value of the root.



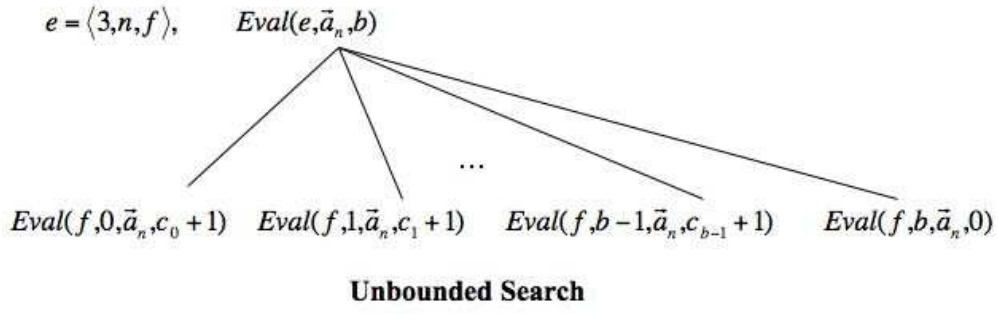
Composition



Primitive Recursion: Basis Case



Primitive Recursion: Iteration Case



Thus, the computation of $Eval(e, \vec{a}, b)$ can be arranged in a tree —a computation tree— with its root (1st "call") labelled $Eval(e, \vec{a}, b)$ and all the other nodes labelled by the relevant calls.

Clearly, the name "$Eval$" does not add any information to the call "$Eval(e, \vec{a}, b)$" and we might as well choose any other name, for example "$George$", and still carry out all the calls perfectly well.

That is, all we need to know for each call are the relevant arguments, $e, \vec{a}, b$. We will thus drop the name "$Eval$" and label the tree nodes by the call-arguments coded as single numbers, that is, $\langle e, \vec{a}, b\rangle$. As a matter of fact we can, and will, code the entire computation tree by a single number via the inductive definition below:

3.1.5 Definition. (Coding Trees) Let $T, T_0, T_1, \ldots$ name trees and $\widehat{T}, \widehat{T_0}, \widehat{T_1}, \ldots$ name their codes respectively. We code a tree T as follows:

(1) If T has a single node labelled v, then $\widehat{T} = \langle v\rangle$.

(2) Let T be the tree denoted by the ordered tuple $v, T_0, T_1, \ldots, T_{m-1}$, where v labels its root and the T_i, $i = 0, \ldots, m-1$, are all the trees hanging from the root (*sub-trees*) from left to right. Then $\widehat{T} = \langle v, \widehat{T_0}, \widehat{T_1}, \ldots, \widehat{T_{m-1}}\rangle$. □

3.1.6 Remark. A *computation tree*, of course, is either a single node v (where $(v)_0 = 0$) or is an ordered sequence $v, T_0, \ldots, T_{l-1}$ where v is one of

(1) $v = \langle\langle 1, m, f, \vec{g}_k\rangle, \vec{a}_m, b\rangle$ ($l = k+1$ in this case; cf. figure for composition, p.5)

(2) $v = \langle\langle 2, n+1, h, g\rangle, c, \vec{a}_n, b\rangle$ (here $l = 1$ or $l = 2$)

(3) $v = \langle\langle 3, n, f\rangle, \vec{a}_n, b\rangle$ (here $n > 0$)

and each of the sub-trees T_i is a computation tree with root as indicated in the figures on pp.5–6. □

Before proceeding, let us recall that

$$Seq(z) \leftrightarrow z > 1 \wedge (\forall x)_{\leq z}(\forall y)_{\leq z}(y|z \wedge Pr(y) \wedge Pr(x) \wedge x < y \rightarrow x|z)$$

and $lh(z) = (\mu y)_{\leq z} rem(z, p_y) > 0$.

We are now ready to show that the predicate "$Tree(u)$" which holds iff "u codes a computation tree" is primitive recursive.

3.1.7 Remark. We will break the predicate into sub-predicates to enhance readability. Thus, let $u = \langle v, t_0, \ldots, t_m \rangle$, where each t_i denotes a tree code, while v codes a call.

(a) The predicate $I(u)$ will be true iff the t_i-part is empty, and v is a call to an initial function.

(b) The predicate $C(u)$ will be true iff v is a call to composition, and the roots of each t_i are correct (cf. composition figure, p.5).

(c) The predicate $P(u)$ will be true iff v is a call to primitive recursion (two cases!), and the roots of each t_i are correct (cf. primitive recursion figure, p.5).

(d) The predicate $U(u)$ will be true iff v is a call to unbounded search, and the roots of each t_i are correct (cf. unbounded search figure, p.6).

There is also a common or "factored out" part in all cases: Let us call this $F(u)$. It occurs as a conjunct "$F(u) \wedge$" in the definition of $Tree(u)$ and is not part of the I, C, P, U.

$$F(u) \leftrightarrow Seq(u) \wedge (\forall i)_{<lh(u)} Seq((u)_i)$$

That is, "u is a code of codes". □ 

We now define each of I, C, P, U. The definitions below, in each case, make it abundantly clear —by closure properties— that we have a predicate in $\mathcal{PR}_*$. Each case involves a lengthy formula. In the interest of readability, comments enclosed in { }-brackets may be included on the left margin, to indicate the sub-case under consideration.

The $I(u)$: $u = \langle v \rangle$, thus $v = (u)_0$.

$$\begin{aligned} I(u) \longleftrightarrow lh(u) = 1 \wedge \Big(& \\ \{\lambda x.0\} \quad & (\exists x)_{\leq u} (u)_0 = \langle \langle 0,1,0 \rangle, x, 0 \rangle \vee \\ \{\lambda x.x+1\} \quad & (\exists x)_{\leq u} (u)_0 = \langle \langle 0,1,1 \rangle, x, x+1 \rangle \vee \\ \{\lambda \vec{x}_n.x_i\} \quad & (\exists x,n,i)_{\leq u} \{ Seq(x) \wedge n = lh(x) \wedge i < n \wedge \\ & (u)_0 = \langle \langle 0, n, i+1, 2 \rangle \rangle * x * \langle (x)_i \rangle \} \Big) \end{aligned}$$

The $C(u)$: $u = \langle v, t_0, \ldots, t_n \rangle$, where (cf. 3.1.6) $v = \langle \langle 1, m, f, \vec{g}_n \rangle, \vec{a}_m, b \rangle$. Thus $v = (u)_0$. We can represent the vectors $\vec{g}_n = g_0, \ldots, g_{n-1}$, $\vec{a}_m = a_0, \ldots, a_{m-1}$, and $t_0, \ldots, t_n$ by the single numbers a, g and t respectively, so that $v = \langle \langle 1, m, f \rangle * g \rangle * a * \langle b \rangle$ and $u = \langle v \rangle * t$.

The "z" in the definition of $C(u)$ below codes the outputs $z_0, \ldots, z_{n-1}$ of each recursive call $\langle g_i, \vec{a}_m, z_i \rangle = \langle (g)_i \rangle * a * \langle (z)_i \rangle$ (root of t_i for $i < n$ —cf. figure on p.5).

$$\begin{aligned} C(u) \longleftrightarrow (\exists a,z,f,g,m,n,b)_{\leq u} \Big\{ & lh(u) = n+2 \wedge Seq(a) \wedge Seq(z) \wedge Seq(f) \wedge \\ & Seq(g) \wedge n = lh(z) \wedge n = lh(g) \wedge m = lh(a) \wedge \\ & (f)_1 = n \wedge (\forall i)_{<n} (Seq((g)_i) \wedge ((g)_i)_1 = m) \wedge \\ \{\text{Root (main) call}\} \quad & (u)_0 = \langle \langle 1, m, f \rangle * g \rangle * a * \langle b \rangle \wedge \\ \{\text{Root of } t_n\} \quad & ((u)_{n+1})_0 = \langle f \rangle * z * \langle b \rangle \wedge \\ \{\text{Root of } t_i,\ i < n\} \quad & (\forall i)_{<n} ((u)_{i+1})_0 = \langle (g)_i \rangle * a * \langle (z)_i \rangle \Big\} \end{aligned}$$

The $P(u)$: There are two cases (see figure regarding primitive recursion on p.5):

(A) $u = \langle v, t\rangle$, where (cf. 3.1.6) $v = \langle\langle 2, n+1, h, g\rangle, 0, \vec{a}_n, b\rangle$. We can represent the vector $\vec{a}_n = a_0, \ldots, a_{n-1}$, by the single number a, so that $v = \langle\langle 2, n+1, h, g\rangle, 0\rangle * a * \langle b\rangle$. The single call of v is the root of t —$(t)_0 = \langle h\rangle * a * \langle b\rangle$.

(B) $u = \langle v, t_0, t_1\rangle$. We will set $t = \langle t_0, t_1\rangle$, hence $u = \langle v\rangle * t$. Here $v = \langle\langle 2, n+1, h, g\rangle, c+1\rangle * a * \langle b\rangle$ and the two calls of v are $(t)_0 = \langle g, c\rangle * a * \langle z, b\rangle$ and $(t)_1 = \langle\langle 2, n+1, h, g\rangle, c\rangle * a * \langle z\rangle$ for some appropriate z (cf. p.5).

$$
\begin{aligned}
P(u) \longleftrightarrow (\exists n, h, g, c, a, b, z)_{\le u}\Big\{ & Seq(h) \wedge (h)_1 = n \wedge Seq(g) \wedge (g)_1 = n+2 \wedge \\
& Seq(a) \wedge lh(a) = n \wedge \\
\{\text{Basis Case}\} \quad & \big(lh(u) = 2 \wedge (u)_0 = \langle\langle 2, n+1, h, g\rangle, 0\rangle * a * \langle b\rangle \wedge \\
\{\text{The only call}\} \quad & ((u)_1)_0 = \langle h\rangle * a * \langle b\rangle \vee \\
\{\text{Iteration Case}\} \quad & lh(u) = 3 \wedge (u)_0 = \langle\langle 2, n+1, h, g\rangle, c+1\rangle * a * \langle b\rangle \wedge \\
\{\text{1st call}\} \quad & ((u)_1)_0 = \langle g, c\rangle * a * \langle z, b\rangle \wedge \\
\{\text{2nd call}\} \quad & ((u)_2)_0 = \langle\langle 2, n+1, h, g\rangle, c\rangle * a * \langle z\rangle\big)\Big\}
\end{aligned}
$$

The $U(u)$: $u = \langle v, t_0, \ldots, t_b\rangle$, where (cf. 3.1.6) $v = \langle\langle 3, n, f\rangle, \vec{a}_n, b\rangle$. Thus $v = (u)_0$. We can represent the vectors $\vec{a}_n = a_0, \ldots, a_{n-1}$, and $t_0, \ldots, t_b$ by the single numbers a and t respectively, so that $v = \langle\langle 3, n, f\rangle\rangle * a * \langle b\rangle$ and $u = \langle v\rangle * t$.

The recursive calls —cf. figure regarding unbounded search on p.6— are, for $i < b$, $((t)_i)_0 = \langle f, i, \vec{a}_n, c_i + 1\rangle$ and $((t)_b)_0 = \langle f, b, \vec{a}_n, 0\rangle$.

Coding the c_i by the number c, and noting that $(t)_i = (u)_{i+1}$, we have $((u)_{i+1})_0 = \langle f, i\rangle * a * \langle (c)_i + 1\rangle$ and $((u)_{b+1})_0 = \langle f, b\rangle * a * \langle 0\rangle$. Thus,

$$
\begin{aligned}
U(u) \longleftrightarrow (\exists n, f, a, b, c)_{\le u}\Big\{ & lh(u) = b+2 \wedge Seq(f) \wedge (f)_1 = n+1 \wedge n > 0 \wedge \\
& Seq(a) \wedge lh(a) = n \wedge Seq(c) \wedge lh(c) = b \wedge \\
\{\text{The root call}\} \quad & (u)_0 = \langle\langle 3, n, f\rangle\rangle * a * \langle b\rangle \wedge \\
\{\text{The recursive calls}\} \quad & (\forall i)_{<b}((u)_{i+1})_0 = \langle f, i\rangle * a * \langle (c)_i + 1\rangle \wedge \\
& ((u)_{b+1})_0 = \langle f, b\rangle * a * \langle 0\rangle\Big\}
\end{aligned}
$$

3.1.8 Theorem. $Tree(u) \in \mathcal{PR}_*$.

Proof.

$$
Tree(u) \leftrightarrow F(u) \wedge \big(I(u) \vee C(u) \vee P(u) \vee U(u)\big) \wedge (\forall i)_{<lh(u)}\big(i > 0 \rightarrow Tree((u)_i)\big) \tag{1}
$$

Wait a minute! What sort of definition is (1)?

It is a course-of-values recursion, for if $\lambda u.\chi(u)$ denotes the characteristic function of $Tree(u)$, then

$$
\begin{aligned}
\chi(0) &= 1 \\
\chi(u+1) &= 0 \quad \text{if } F(u+1) \wedge \big(I(u+1) \vee C(u+1) \vee P(u+1) \vee U(u+1)\big) \wedge
\end{aligned}
$$

$$\begin{aligned} &(\forall i)_{<lh(u+1)}\big(i>0 \to \chi((u+1)_i)=0\big)\\ =1 \quad &\text{otherwise} \end{aligned}$$

That this is a course-of-values recursion follows from $(y)_i < y$. □

3.1.9 Definition. (The Kleene T-predicate) For each $n \in \mathbb{N}$, $T^{(n)}(e, \vec{x}_n, z)$ stands for "there is a computation tree coded by z such that it has as root $\langle e, \vec{x}_n, y\rangle$ —and therefore verifies $\{e\}(\vec{x}_n) = y$— for an appropriate y". □

3.1.10 Theorem. *$\lambda e\vec{x}_n y.T^{(n)}(e, \vec{x}_n, z)$ is primitive recursive.*

Proof.

$$T^{(n)}(e, \vec{x}_n, z) \leftrightarrow Tree(z) \wedge (\exists y)_{<z}(z)_0 = \langle e, \vec{x}_n, y\rangle$$ □

3.1.11 Theorem. (Kleene Normal Form Theorem) *There is a primitive recursive function $\lambda x.ret(x)$ ("ret" for retrieve) such that*

(1) $y = \{a\}(\vec{x}_n) \equiv (\exists z)(T^{(n)}(a, \vec{x}_n, z) \wedge ret(z) = y)$
(2) $\{a\}(\vec{x}_n) = ret\big((\mu z)T^{(n)}(a, \vec{x}_n, z)\big)$
(3) $\{a\}(\vec{x}_n)\downarrow \equiv (\exists z)T^{(n)}(a, \vec{x}_n, z)$.

Proof. Use $ret = \lambda z.\big((z)_0\big)_{lh\big((z)_0\big)\dot{-}1}$. □

3.1.12 Remark. (Very important) The right hand side of 3.1.11(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 compile and hence it will not 'run'. Thus, it will 'define' the *everywhere undefined function.*" □

3.2. Semi-Recursive Predicates; Unsolvability

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

3.2.1 Definition. (Semi-recursive relations) 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 $\mathcal{P}_*$[†]

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

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 $P = W_a$ ([Rog67]). □

We have at once

3.2.2 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. Let $P(\vec{x}_n) \leftrightarrow f(\vec{x}_n)\downarrow$, with $f \in \mathcal{P}$. By Theorem 3.1.1 (and 3.1.12), $f = \{a\}$ for some $a \in \mathbb{N}$. Now invoke 3.1.11(3).

if-part. By 3.1.11(3), $P(\vec{x}_n) \leftrightarrow \{a\}(\vec{x}_n)\downarrow$. But $\{a\} \in \mathcal{P}$. □

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

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

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

Proof. For the *only if* take $Q(\vec{x}_n, z)$ to be $\lambda\vec{x}_n z.T^{(n)}(a, \vec{x}_n, z)$ for appropriate $a \in \mathbb{N}$. For the *if* take $f = \lambda\vec{x}_n.(\mu z)Q(\vec{x}_n, z)$. Then $f \in \mathcal{P}$ and $P(\vec{x}_n) \leftrightarrow f(\vec{x}_n)\downarrow$. □

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

3.2.4 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 3.1.11(1), $f(\vec{x}_n) = 0 \leftrightarrow (\exists z)\big(T^{(n)}(a, \vec{x}_n, z) \wedge ret(z) = 0\big)$. We are done by strong projection. □

3.2.5 Remark. The expression "$f(\vec{x}_n) = 0 \cdot g(\vec{x}_n)$" is shorthand for

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

Intuitively, f is computable: Run a program for g with input $\vec{x}_n$. If it ever halts, then stop everything and return 0.

A formal reason as to why $f \in \mathcal{P}$ is $f(\vec{x}_n) = z\big(g(\vec{x}_n)\big)$ where z is the zero (initial) function. □

We immediately obtain

3.2.6 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.

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

Proof. *only if*-part. By 3.2.6 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 $R(\vec{x}_n) \leftrightarrow T^{(n)}(i, \vec{x}_n, g(\vec{x}_n))$. □

(*Unsolvable Problems*) A *problem* is a question "$\vec{x} \in R$?" for some 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 $\overline{K}$.

3.2.8 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. □

$K \in \mathcal{P}_*$, 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., $\mathcal{R}_* \subset \mathcal{P}_*$.

3.2.9 Theorem. (Closure properties of $\mathcal{P}_*$) *$\mathcal{P}_*$ is closed under $\vee, \wedge, (\exists y)_{<z}, (\exists y), (\forall y)_{<z}$. 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.

$(\wedge:)$

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

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

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

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 $\forall y$. $\square$

3.3. Recursively Enumerable (r.e.) Predicates

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

3.3.1 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

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

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

3.4. The S-m-n and Recursion Theorems

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.

3.4.1 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

$$h(0) = a$$
$$h(y+1) = \langle 1, 1, b, h(y)\rangle$$

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. □

3.4.2 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 $k(i,y) = \sigma(\langle 1,1,i,a_0,a_1\rangle, y)$. □

3.4.3 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 $i \in \mathbb{N}$. This is a code for

$$\lambda xy.\{i\}((x)_0,\ldots,(x)_{m-1},(y)_0,\ldots,(y)_{n-1})$$

Let d be a ϕ-index of $\lambda\vec{x}_m.\langle\vec{x}_m\rangle$.

Then,

$$\begin{aligned}\{i\}(\vec{x}_m,\vec{y}_n) &= \{c(i)\}(\langle\vec{x}_m\rangle,\langle\vec{y}_n\rangle)\\ &= \{k(c(i),\langle\vec{y}_n\rangle)\}(\langle\vec{x}_m\rangle) && \text{by 3.4.2}\\ &= \{\langle 1, m, k(c(i),\langle\vec{y}_n\rangle), d\rangle\}(\vec{x}_m)\end{aligned}$$

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

3.4.4 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 $\{a\} = \lambda z\vec{x}_n.f(S^n_1(z,z),\vec{x}_n)$. Then

$$\begin{aligned}f(S^n_1(a,a),\vec{x}_n) &= \{a\}(a,\vec{x}_n)\\ &= \{S^n_1(a,a)\}(\vec{x}_n) && \text{by 3.4.3}\end{aligned}$$

Take $e = S^n_1(a,a)$. □

3.4.5 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*. □

3.5. Two Applications of the Recursion Theorem

3.5.1 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 [Rog67] to G.C. Wolpin.)

if-part. Immediate, since $\chi_\emptyset = \lambda x.1$ and $\chi_{\mathbb{N}} = \lambda x.0$.

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 3.4.4 to $\lambda xy.\{f(x)\}(y)$).

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

The second application is about self-referential (recursive) definitions of functions F such as the one below

$$F(\vec{x}_n) = f\Big(\ldots F\Big(\ldots F(\ldots)\ldots\Big)\ldots F\Big(\ldots F\big(\ldots F(\ldots)\ldots\big)\ldots\Big)\ldots\Big) \tag{1}$$

where nesting of occurrences of F can be anything.

We are interested in just those cases that, as we say, the right hand side of (1) —as a function of $\vec{x}_n$— is partial recursive in F.

3.5.2 Definition. We say that a function is *partial recursive in* F iff it is in the closure of $I \cup \{F\}$ under composition, primitive recursion and (μy). Here I denoted by "I" the standard initial functions of $\mathcal{P}$.

For short, a function is partial recursive in F iff is obtained by a finite number of partial recursive operations using as initial functions F and those in I. □

3.5.3 Remark. It follows from 3.5.2 that if $F \in \mathcal{P}$, then a function that is partial recursive in F is just partial recursive.

In particular, if we replace F throughout the right hand side of (1) by a partial recursive function $\{e\}$ of the same arity as F, then we end up with a partial recursive function. □

In (1) F acts as a "function variable" to solve for. A solution h for F is a specific function that makes (1) true for all $\vec{x}_n$ if we replace all occurrences of F by h.

We show that if the right hand side of (1) is partial recursive in F, then (1) always has a partial recursive solution for F. That is,

$$(\exists e)\Big(\text{if we replace } F \text{ in (1) by } \lambda\vec{x}_n.\{e\}(\vec{x}_n), \text{ then the resulting relation is true for all } \vec{x}_n\Big) \tag{2}$$

Indeed, the function $\lambda z \vec{x}_n . G(z, \vec{x}_n)$ given below is partial recursive by 3.5.3.

$$G(z, \vec{x}_n) = f\Big(\ldots \{z\}\Big(\ldots \{z\}(\ldots) \ldots \Big) \ldots \{z\}\Big(\ldots \{z\}\big(\ldots \{z\}(\ldots) \ldots \big) \ldots \Big) \ldots \Big) \tag{3}$$

By the recursion theorem there is an e such that

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

Thus, (3) yields

$$\{e\}(\vec{x}_n) = G(e, \vec{x}_n) = f\Big(\ldots \{e\}\Big(\ldots \{e\}(\ldots) \ldots \Big) \ldots \{e\}\Big(\ldots \{e\}\big(\ldots \{e\}(\ldots) \ldots \big) \ldots \Big) \ldots \Big) \tag{4}$$

That is, setting the "function variable F" equal to $\{e\}$ we have solved (1), and with a $\mathcal{P}$-solution at that!

3.5.4 Example. Here is a second solution to the question "$\lambda n x . A_n(x) \in \mathcal{R}$?".

$A_n(x)$ is given by

$$A_n(x) = \begin{cases} x + 2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ A_{n \dot{-} 1}\big(A_n(x \dot{-} 1)\big) & \text{otherwise} \end{cases}$$

We re-write the above using F as a function variable and setting $F(n, x) = A_n(x)$.

Thus, F is "given" by

$$F(n, x) = \begin{cases} x + 2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ F\big(n \dot{-} 1, F(n, x \dot{-} 1)\big) & \text{otherwise} \end{cases} \tag{5}$$

(5) has the form (1) and all assumptions are met. Thus, for some e, $F = \{e\}$ works. But is this $\{e\}$ the same as $A_n(x)$? Yes, provided (5) has a unique solution! That (5) indeed does have a unique total solution is an easy (double) induction exercise that shows $F(n, x) = F'(n, x)$ for all n, x if

$$F'(n, x) = \begin{cases} x + 2 & \text{if } n = 0 \\ 2 & \text{else if } x = 0 \\ F'\big(n \dot{-} 1, F'(n, x \dot{-} 1)\big) & \text{otherwise} \end{cases} \tag{5'}$$

□

Bibliography

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

[Tou84] G. Tourlakis. *Computability.* Reston Publishing Company, Inc., Reston, Virginia, 1984.