show that $\overline{F}$ is recursive iff F is recursive. We cannot use the preceding equation as an explicit definition; for we cannot fill in ... until we know the value of the argument y. However, we have the explicit definitions

$$\overline{F}(y,\vec{x}) \simeq \mu z(Seq(z) \;\&\; lh(z) = y \;\&\; (\forall i < y)((z)_i = F(i,\vec{x}))),$$

$$F(y,\vec{x}) \simeq (\overline{F}(y+1,x))_y.$$

Given a total function G, we may define a total function F by induction on y as follows:

$$F(y,\vec{x}) \simeq G(\overline{F}(y,\vec{x}),\, y,\, \vec{x}).$$

We shall show that if G is recursive, then F is recursive. By the above, it is enough to show that $\overline{F}$ is recursive. But $\overline{F}$ has the inductive definition

$$\overline{F}(0,\vec{x}) \simeq <>,$$

$$\overline{F}(y+1,\vec{x}) = \overline{F}(y,\vec{x}) * <G(\overline{F}(y,\vec{x}),\, y\,,\vec{x})>.$$

An inductive definition of this sort is called a course–of–values inductive definition.

8. Indices

We are now going to assign codes to some of the elements in the operation of the basic machine. This will lead to some of the most important theorems of recursion theory.

First, a general remark on coding. Suppose that we want to code the members of I. We may be able to identify each member b of I with a finite sequence $a_1,...,a_k$ of objects which have already been coded. We can then assign to b the code $<x_1,...,x_k>$, where x_i is the code of a_i.

We begin by assigning codes to the instructions for the basic machine. We assign the code $<0,i>$ to the instruction INCREASE $\mathcal{R}i$; the code $<1,i,n>$ to the instruction DECREASE $\mathcal{R}i,n$; and the code $<2,n>$ to the instruction GO TO n. If P is a program consisting of N instructions with codes $x_1,...,x_N$, we assign the code $<x_1,...,x_N>$ to P.

We define

$$Ins(x) \leftrightarrow x = <0,(x)_1> \vee x = <1,(x)_1,(x)_2> \vee x = <2,(x)_1>,$$

$$Prog(x) \leftrightarrow Seq(x) \;\&\; (\forall i < lh(x))(Ins((x)_i) \;\&\; (((x)_i)_0 = 1 \to ((x)_i)_2 < lh(x))$$

$$\&\; (((x)_i)_0 = 2 \to ((x)_i)_1 < lh(x))).$$

Thus *Ins* is the set of codes of instructions and *Prog* is the set of codes of programs.

The action of the machine in executing A_k^P (described near the end of §3) with inputs $\mathfrak{x}$ is called the P–computation from $\mathfrak{x}$. If e is the code of P, then P mentions $\mathfrak{R}i$ only for $i < e$ by (2) of §8; so the contents of $\mathfrak{R}i$ are significant for this computation only for $i < e + k$. At any step in this computation, the register code of the machine is $<r_0,r_1,\ldots,r_{e+k\dot{-}1}>$, where r_i is the number in $\mathfrak{R}i$. If the computation stops after m steps, it has successive register codes $r_0, \ldots, r_m$. We then assign the code $r = <r_0,r_1,\ldots,r_m>$ to the computation. By (2) of §8, r is larger than any number which appears in a register during the computation. The output of the computation is $U(r)$, where U is the recursive real defined by

$$U(r) = ((r)_{lh(r)\dot{-}1})_0.$$

We define functions *Count* and *Reg* such that if e is the code of P and $x = <\mathfrak{x}>$, then after n steps in the P–computation from $\mathfrak{x}$, $Count(e,x,n)$ will be in the counter and $Reg(j,e,x,n)$ will be in $\mathfrak{R}j$. We define these functions by a simultaneous induction on n. Writing i for $(e)_{Count(e,n,x)}$,

$$Count(e,x,0) = 0,$$

$$Reg(j,e,x,0) = (x)_{\dot{j}\dot{1}} \qquad \text{if } j \leq lh(x) \;\&\; j \neq 0,$$

$$= 0 \qquad \text{otherwise,}$$

$$Count(e,x,n+1) = (i)_2 \qquad \text{if } (i)_0 = 1 \;\&\; Reg((i)_1,e,x,n) \neq 0,$$

$$= (i)_1 \qquad \text{if } (i)_0 = 2,$$

$$= Count(e,x,n) + 1 \qquad \text{otherwise,}$$

$$Reg(j,e,x,n+1) = Reg(j,e,x,n) + 1 \qquad \text{if } (i)_0 = 0 \;\&\; j = (i)_1,$$

$$= Reg(j,e,x,n) \dot{-} 1 \qquad \text{if } (i)_0 = 1 \ \&\ j = (i)_1,$$
$$= Reg(j,e,x,n) \qquad \text{otherwise.}$$

We define

$$Step(e,x,n) \leftrightarrow Count(e,x,n) \geq lh(e) \ \&\ (\forall i < n)(Count(e,x,i) < lh(e)).$$

Then in the above notation, $Step(e,x,n)$ means that the P–computation from $\vec{x}$ takes n steps.

If $\vec{x}$ is a k–tuple, $T_k(e,\vec{x},y)$ mean that e is the code of a program P and y is the code of the P–computation from $\vec{x}$. Thus

$$T_k(e,\vec{x},y) \leftrightarrow Prog(e) \ \&\ Seq(y) \ \&\ Step(e,<\vec{x}>,lh(y)\dot{-}1) \ \&$$
$$(\forall i < lh(y))((y)_i = \overline{Reg}(e+k,e,<\vec{x}>,i)).$$

If e is the code of a program P, $\vec{x}$ is a k–tuple, and A_k^P has an output when applied to the inputs $\vec{x}$, then $\{e\}(\vec{x})$ is that output; otherwise $\{e\}(\vec{x})$ is undefined. Clearly

$$\{e\}(\vec{x}) \simeq U(\mu y T_k(e,\vec{x},y)).$$

This equation is called the Normal Form Theorem.

We say that e is an <u>index</u> of F if $F(\vec{x}) \simeq \{e\}(\vec{x})$ for all $\vec{x}$.

8.1. PROPOSITION. A function is recursive iff it has an index.

Proof. If F is recursive and e is the code of a program which computes F, then e is an index of F. The converse follows from the Normal Form Theorem. □

8.2. ENUMERATION THEOREM (KLEENE). For each k, $\{e\}(x_1,\ldots,x_k)$ is a recursive function of $e,x_1,\ldots,x_k$.

Proof. By the Normal Form Theorem. □

By the Normal Form Theorem, $\{e\}(\vec{x})$ is defined iff there is a y such that $T_k(e,\vec{x},y)$. By the meaning of T_k, this y is then unique; and $\{e\}(\vec{x}) = U(y)$. We call y the <u>computation number</u> of $\{e\}(\vec{x})$. Since y is greater than every number appearing in a register during the P–computation from $\vec{x}$, it is greater than the x_i and $\{e\}(\vec{x})$.

Recall that the results of the last three section depended only on the fact that the class of recursive functions was recursively closed. Thus every recursively closed class contains U and the T_k, and hence, by the Normal Form Theorem, each of the functions $\{e\}$. Hence by 8.1:

8.3. PROPOSITION. The class of recursive functions is the smallest recursively closed class. □

The importance of 8.3 is that it gives us a method of proving that every recursive function has a property P; we have only to show that the class of functions having property P is recursively closed.

We define

$$\{e\}_s(\vec{x}) \simeq U(\mu y(y \leq s \,\&\, T_k(e,\vec{x},y))).$$

Clearly $\{e\}(\vec{x}) \simeq z$ iff $\{e\}_s(\vec{x}) \simeq z$ for some s; in this case, $\{e\}_s(\vec{x}) = z$ for all $s \geq y$, where y is the computation number of $\{e\}(\vec{x})$. Thus $\{e\}_s$ may be thought of as the sth approximation to $\{e\}$. If $\{e\}_s(\vec{x})$ is defined, each x_i is $< s$; so $\{e\}_s$ is a finite function.

8.4. PROPOSITION. The relations P and Q defined by

$$P(e,s,\vec{x},z) \leftrightarrow \{e\}_s(\vec{x}) \simeq z$$

and

$$Q(e,s,\vec{x}) \leftrightarrow \{e\}_s(\vec{x}) \text{ is defined}$$

are recursive.

Proof. We have

$$P(e,s,\vec{x},z) \leftrightarrow (\exists y \leq s)(T_k(e,\vec{x},y) \,\&\, U(y) = z),$$

$$Q(e,s,\vec{x}) \leftrightarrow (\exists y \leq s)\, T_k(e,\vec{x},y). \,\square$$

We shall now use indices to extend 6.5 to partial functions.

8.5. PROPOSITION. Let $R_1,\ldots,R_n$ be recursive relations such that for every $\vec{x}$, exactly one of $R_1(\vec{x}),\ldots,R_n(\vec{x})$ is true. Let $F_1,\ldots,F_n$ be recursive functions, and define a function F by

$$F(\vec{x}) \simeq F_1(\vec{x}) \quad \text{if } R_1(\vec{x}),$$

$$\simeq \ldots$$

$$\simeq F_n(\vec{x}) \quad \text{if } R_n(\vec{x}).$$

Then F is recursive.

Proof. Let f_i be an index of F_i. Using 6.5, define a total recursive function G by

$$\begin{aligned} G(\vec{x}) &= f_1 \quad \text{if } R_1(\vec{x}), \\ &= \ldots \\ &= f_n \quad \text{if } R_n(\vec{x}). \end{aligned}$$

We can then define F by $F(\vec{x}) \simeq \{G(\vec{x})\}(\vec{x})$. □

8.6. PARAMETER THEOREM. If F is a $(k+m)$–ary recursive function, there is a recursive total function S such that

$$(1) \qquad \{S(y_1,\ldots,y_m)\}(\vec{x}) \simeq F(\vec{x},y_1,\ldots,y_m)$$

for all $\vec{x},y_1,\ldots,y_m$.

Proof. To simplify the notation, we suppose that $m = 1$ and write y for y_1. Suppose that $\vec{x}$ is a k–tuple. Let the program P_y consist of y INCREASE $\mathcal{R}(k+1)$ instructions followed by the macro of a program P which computes F. Then P_y computes the function G defined by $G(\vec{x}) \simeq F(\vec{x},y)$. If we take $S(y)$ to be the code of the program for the basic machine which by 4.1 is equivalent to P_y, then (1) holds.

It remains to give a definition of S which shows that it is recursive. First we define

$$\begin{aligned} F(i,y) &= <1,(x)_1,(x)_2 + y> && \text{if } (i)_0 = 1, \\ &= <2,(x)_1+y> && \text{if } (x)_0 = 2, \\ &= i && \text{otherwise.} \end{aligned}$$

If i is the code of an instruction I, $F(i,y)$ is the code of the instruction obtained from I by increasing every instruction number by y. Let e be the code of P. Then we define $S(y) = S_1(y) * S_2(y)$, where

$$S_1(y) = \mu z(Seq(z) \;\&\; lh(z) = y \;\&\; (\forall i < y)((z)_i = <0,k+1>))$$

and

$S_2(y) = \mu z(Seq(z)\ \&\ lh(z) = lh(e)\ \&\ (\forall i < lh(z))((z)_i = F((e)_i,y)))$. □

An implicit definition of a function F has the form $F(\vec{x}) \simeq$ _____ where now _____ may contain F as well as previously defined symbols and the variables in $\vec{x}$. Of course, this is not really a definition of F; it merely tells us to search for an F which satisfies the equation $F(\vec{x}) \simeq$ _____. Thus $F(\vec{x}) \simeq F(\vec{x})$ is satisfied by every F, and $F(\vec{x}) \simeq F(\vec{x}) + 1$ is satisfied only by the function whose domain is the empty set.

Let us rewrite our implicit definition as $F(\vec{x}) \simeq G(F,\vec{x})$. We would like to show that if G is recursive, then this has at least one recursive solution. Unfortunately, G is not a function in our sense because of the argument F. We therefore replace F as an argument to G by an index of F.

8.7. Recursion Theorem (Kleene). If G is recursive, there is a recursive function F with an index f such that $F(\vec{x}) \simeq G(\vec{x},f)$ for all $\vec{x}$.

Proof. Since $\{y\}(\vec{x},y)$ is a recursive function of $\vec{x},y$, the Parameter Theorem implies that there is a recursive total function S such that $\{S(y)\}(\vec{x}) \simeq \{y\}(\vec{x},y)$ for all $\vec{x}$ and y. Define a recursive function H by

$$H(\vec{x},y) \simeq G(\vec{x},S(y))$$

and let h be an index of H. Let $F = \{S(h)\}$, $f = S(h)$, so that F has index f. Then

$$F(\vec{x}) \simeq \{S(h)\}(\vec{x}) \simeq \{h\}(\vec{x},h) \simeq H(\vec{x},h) \simeq G(\vec{x},S(h)) \simeq G(\vec{x},f).\ \square$$

The Recursion Theorem is often useful for showing that functions are recursive. For example, suppose that we define

$$(2) \qquad \begin{aligned} F(0,x) &\simeq G(x), \\ F(y+1,x) &\simeq H(F(y,2x),y,x), \end{aligned}$$

where G and H are total recursive functions. This is a legitimate definition by induction on y; it uniquely defines a function F, which is total. Our previous methods will not show that F is recursive, since they allow $F(y,x)$ but not $F(y,2x)$ on the right side. We define a recursive L by

$$L(y,x,f) \simeq G(x) \qquad \text{if } y = 0,$$
$$\simeq H(\{f\}(y \dot{-} 1,2x),\ y \dot{-} 1,\ x) \qquad \text{otherwise.}$$

Then we use the Recursion Theorem to obtain a recursive F with an index f such that $F(y,x) \simeq L(y,x,f)$. Clearly F satisfies (2); so the function defined by (2) is recursive.

9. Church's Thesis

We have already remarked that it is clear that every recursive function is computable. The statement that every computable function is recursive is known as Church's Thesis. It was proposed by Church about 1934 and has since come to be accepted by almost all logicians. We shall discuss the reasons for this.

Since the notion of a computable function has not been defined precisely, it may seem that it is impossible to give a proof of Church's Thesis. However, this is not necessarily the case. We understand the notion of a computable function well enough to make some statements about it. In other words, we can write down some axioms about computable functions which most people would agree are evidently true. It might be possible to prove Church's Thesis from such axioms. However, despite strenuous efforts, no one has succeeded in doing this (although some interesting partial results have been obtained).

We are thus reduced to trying to give arguments for Church's Thesis which seem to be convincing. We shall briefly examine these arguments.

The first argument is that all the computable functions which have been produced have been shown to be recursive, using, for the most part, the techniques which we have already described. Moreover, all the known techniques for producing new computable functions from old ones (such as definition by induction or by cases) have been shown to lead from recursive functions to recursive functions.