Introduction to Computability Theory

Jeffery Zucker* Laurette Pretorius†
*Department of Computer Science and Systems, McMaster University, Hamilton, Ont. L8S 4K1, Canada
zucker@maccs.dcss.mcmaster.ca
†Department of Computer Science and Information Systems, University of South Africa, Pretoria
pretol@risc1.unisa.ac.za

Abstract

These are notes for a short introductory course on Computability Theory (or recursive function theory). The basic notion of computability is defined in terms of a simple imperative programming language.
Keywords: *Computability, Recursion, Primitive recursion, Church-Turing Thesis, Recursive enumerability*
Computing Review Categories: *F.1.1, F.4.1*

Contents

1 Introduction

Computability theory (also known as *recursive function theory* for historical reasons) originated in the 1930's in the research of Church, Gödel, Turing, Kleene and others, who formalised the notion of computable (or "recursive") function in different ways, for example, by Turing machines, lambda-calculus, definability by μ-recursive schemes, and definability by sets of equations. Corresponding to each of these formalisms is a "Church-Turing Thesis" which identifies computability by that formalism with intuitive effective computability. In the present exposition we follow a modern approach, using *computability by a simple imperative programming language* as our basic notion. This approach is directly inspired by, and follows closely, that of [1]. However, we take the notion of computability of *partial functions* ("partial recursiveness") as the basic notion. We have also benefitted from the by now classic references [2] and [3].

In the short course (10 hours) on which these notes are based, much important material had to be omitted. Nevertheless it is hoped that these notes may be useful for an introductory course (or half-course) in computability theory, or for self-study. In the latter case, the reader is encouraged to peruse the references for further topics.

2 Mathematical Preliminaries

We review some basic concepts concerning sets, relations, functions and predicates.

- **Sets and n-tuples**
  We write $a \in A$ to mean that a is an element of the set A. While the order in which the elements of a set $\{a_1, \cdots, a_n\}$ are written, is irrelevant, the order in an *n-tuple* $\vec{a} = (a_1, \cdots, a_n)$ is important. Indeed, $(a_1, \cdots, a_n) = (b_1, \cdots, b_n)$ iff $a_1 = b_1, \cdots, a_n = b_n$. If $A_1, \cdots, A_n$ are given sets, $A_1 \times \cdots \times A_n$ denotes the set of all n-tuples $(a_1, \cdots, a_n)$ such that $a_1 \in A_1, \cdots, a_n \in A_n$. We write A^n for $\underbrace{A \times \cdots \times A}_{n \text{ times}}$.

- **Natural numbers**
  $\mathcal{N} = \{0, 1, 2, \cdots\}$ is the set of *natural numbers.* By "*number*" we will mean natural number.

- **Relations**
  An n-ary relation on a set A is a subset of A^n, for $n = 1, 2, 3, \cdots$. When $n = 2$, we speak of a *binary relation* on A, and often use *infix* notation. Thus, for example, we write '$x < y$' for '$< (x, y)$', where '$<$' is the order relation on $\mathcal{N}$. If B and C are two n-ary relations on A, then their *union*,

intersection and *complement* are defined by:

$$\begin{array}{rcl} B \cup C & = & \{\vec{a} \in A^n | \vec{a} \in B \text{ or } \vec{a} \in C\}, \\ B \cap C & = & \{\vec{a} \in A^n | \vec{a} \in B \text{ and } \vec{a} \in C\}, \\ B \setminus C & = & \{\vec{a} \in A^n | \vec{a} \in B \text{ and } \vec{a} \notin C\}, \\ \bar{B} & = & A^n \setminus B. \end{array}$$

By "relation" we will generally mean *relation on* $\mathcal{N}$.

- **Functions**

Given two sets A and B, a *(partial) function*[1] $f : A \dot{\rightarrow} B$ is a subset of $A \times B$ such that for all $a \in A$ there is *at most one* $b \in B$ (denoted $f(a)$) such that $(a, b) \in f$. We define

$$\begin{array}{rl} & \boldsymbol{dom}(f) = \{a \in A | \exists b \in B : (a,b) \in f\} \\ \text{and} & \boldsymbol{ran}(f) = \{b \in B | \exists a \in A : (a,b) \in f\}, \end{array}$$

and write $f(a) \uparrow$ ("diverges") if $a \notin \boldsymbol{dom}(f)$, $f(a) \downarrow$ ("converges") if $a \in \boldsymbol{dom}(f)$, and $f(a) \downarrow b$ ("converges to b") if $a \in \boldsymbol{dom}(f)$ *and* $f(a) = b$. If $A = A_1 \times \cdots \times A_n$, we write $f(a_1, \cdots, a_n)$ and say f is a *function* of n *arguments*, or an n*-ary* function, or a function of *arity* n. (We call f *unary* if $n = 1$ and *binary* if $n = 2$.)

A function $f : A \dot{\rightarrow} B$ is *total* if $\boldsymbol{dom}(f) = A$ (written $f : A \rightarrow B$, without the dot). For our purposes, partial functions are the more basic concept, and totality of functions should *not* be assumed unless explicitly stated. In fact we will be concerned mainly with n-ary partial functions on $\mathcal{N}$, i.e. functions $f : \mathcal{N}^n \dot{\rightarrow} \mathcal{N}$, for some $n > 0$. By "function" we will generally mean *partial function on* $\mathcal{N}$, denoted by f,g,h, $\cdots$.

A function $f : A \dot{\rightarrow} B$ is called (a) *injective* or 1-1 if $\forall x, y \in \boldsymbol{dom}(f)$ $(f(x) = f(y) \Rightarrow x = y)$, (b) *surjective* or *onto* if $\boldsymbol{ran}(f) = B$, and (c) *bijective* or a *bijection* between A and B if it is total, 1-1 and onto. Two sets A and B are called *equinumerous*, written $A \simeq B$, if there is a bijection between them.

We will freely use "lambda-notation" informally, where, for example, $\lambda x, y \cdot (x^2 + y^2 + 1)$ denotes the function $f : \mathcal{N}^2 \rightarrow \mathcal{N}$ such that for all $x, y \in \mathcal{N}$, $f(x, y) = x^2 + y^2 + 1$.

For unary functions f and g, $f \circ g$ denotes their *composition* $\lambda x \cdot f(g(x))$.

- **Predicates**

Let $\mathbf{2}$=$\{0, 1\}$ be (identified with) the set of *truth values*, i.e. $0 = \mathit{false}$ and $1 = \mathit{true}$. A *predicate* on a set A is a total function $P : A \rightarrow \mathbf{2}$. An n-ary predicate on A is a predicate on A^n. Given $B \subseteq A$, the *characteristic function* or *characteristic predicate* of B on A is $\chi_B : A \rightarrow \mathbf{2}$ such that $\forall a \in A$

$$\chi_B(a) = \begin{cases} 1 & \text{if } a \in B \\ 0 & \text{otherwise.} \end{cases}$$

Conversely, given a predicate $P : A \rightarrow \mathbf{2}$, the *characteristic set* of P on A is the set $\mathcal{S}_P = \{a \in A \mid P(a) = 1\} \subseteq A$. Hence

$$\wp(A) \simeq \mathrm{PRED}(A)$$

where $\wp(A)$ is the power set (= the set of all subsets) of A and PRED(A) is the set of predicates on A.

We will usually take $A = \mathcal{N}$, *i.e.*, we will be working mainly with n-ary relations on $\mathcal{N}$ and n-ary predicates on $\mathcal{N}$ (for $n \geq 1$).

- **Basic set theory**

The following elementary concepts and results in set theory will clarify some of the later discussions. (They can be proved in classical set theory, with the Axiom of Choice. For some background on set theory, a good reference is [4].)

We define $A \subseteq B$ to mean A is a subset of B, i.e. $\forall x(x \in A \Rightarrow x \in B)$, and $A \subset B$ to mean A is a *proper* subset of B, i.e. $A \subseteq B$ but $A \neq B$.

A set A is *finite* if it is equinumerous with the set $\{1, \cdots, n\}$ for some $n \in \mathcal{N}$. (This includes the case $A = \emptyset$, the empty set, when $n = 0$.) Otherwise it is *infinite*.

Theorem 2.1 *A set is infinite iff it is equinumerous with a proper subset of itself.*

Theorem 2.2 (Countability) *Let A be a set. The following statements are equivalent:*
(a) There is a total injection $f : A \rightarrow \mathcal{N}$,
(b) $A = \emptyset$, *or there is a total surjection* $g : \mathcal{N} \rightarrow A$,
(c) A *is finite, or there is a bijection* $g : \mathcal{N} \rightarrow A$.
A *is called* countable *or* enumerable *if any of the above conditions holds.*

NOTES:

1. In (b) above, g is called an *enumeration with repetitions*, since g enumerates or lists A:

$$A = \{a_0, a_1, a_2, \cdots\}$$

where $a_i = g(i)$. Similarly, in (c), g is an *enumeration without repetitions*.

2. We will meet *constructive analogues* of the above notions and theorem, in §10 (on *recursive enumerability*).

3. By (c) above, if A is countable but not finite, then $A \simeq \mathcal{N}$, and A is called *countably infinite*. A set which is not countable is called *uncountable* (or *uncountably infinite*).

4. A subset of a finite set is finite, and a subset of a countable set is countable. Also, if $A \simeq B$ and A is finite, countable or uncountable (respectively), then so is B. Thus all

[1] This is a *set-theoretic* or "extensional" concept of function ("function-as-relation"). There is also a *constructive* or "intensional" concept of function ("function-as-rule"), which we prefer to call "algorithm". Note that a single function may have many distinct algorithms which compute it (or none at all, if it is not computable).

sets can be *classified by size* as (i) *finite*, or (ii) *countably infinite*, or (iii) *uncountably infinite*. Roughly speaking, countable infinity is the "smallest size" of infinity.

Let $\mathrm{TFN}^{(1)}$ be the class of total unary functions on $\mathcal{N}$.

Theorem 2.3 *The sets $TFN^{(1)}$, $\wp(\mathcal{N})$ and $PRED(\mathcal{N})$ are uncountably infinite.*

Proof: The proofs use a *diagonalisation method*, which we will encounter many times later in this paper, so they are worth giving here.

(a) Let $F = \{f_1, f_2, \cdots\}$ be any countable subset of $\mathrm{TFN}^{(1)}$. We will exhibit a function

$$f \in \mathrm{TFN}^{(1)} \setminus F,$$

i.e. a *witness* that $F \subset \mathrm{TFN}^{(1)}$. Define

$$f(n) = f_n(n) + 1.$$

Then for all n, $f(n) \neq f_n(n)$, and so $f \neq f_n$. Hence $f \notin F$.

(b) Let $S = \{X_1, X_2, \cdots\}$ be any countable subset of $\wp(\mathcal{N})$. We can similarly define a witness that $\mathcal{S} \subset \wp(\mathcal{N})$, namely $X =_{\mathrm{df}} \{n | n \notin X_n\}$, since for all n, $n \in X \Leftrightarrow n \notin X_n$, and so $X \neq X_n$.

(c) $\mathrm{PRED}(\mathcal{N})$ is uncountable: EXERCISE. □

- **Truth tables: basic operations on truth values** Let p and q be *boolean variables*, i.e. ranging over **2**. The operations *not*, *and*, and *or*, denoted by $\neg$, $\wedge$, and $\vee$, are defined by the truth tables

p	$\neg p$
1	0
0	1

and

p	q	$p \wedge q$	$p \vee q$
1	1	1	1
1	0	0	1
0	1	0	1

.

Now we can form new predicates from old, for if P and Q are predicates on A, then so are $\neg P$, $P \wedge Q$, and $P \vee Q$, where for $x \in A$:

$$\neg P(x) = 1 - P(x),$$

$$(P \wedge Q)(x) = P(x) \wedge Q(x) = \begin{cases} 1 & \text{if } P(x) = 1 \\ & \text{and } Q(x) = 1 \\ 0 & \text{otherwise,} \end{cases}$$

$$(P \vee Q)(x) = P(x) \vee Q(x) = \begin{cases} 1 & \text{if } P(x) = 1 \\ & \text{or } Q(x) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The corresponding characteristic sets are

$$\mathcal{S}_{\neg P} = A \setminus \mathcal{S}_P = \{x \in A \mid \neg P(x)\},$$

$$\mathcal{S}_{P \wedge Q} = \mathcal{S}_P \cap \mathcal{S}_Q = \{x \in A \mid P(x) \wedge Q(x)\},$$

$$\mathcal{S}_{P \vee Q} = \mathcal{S}_P \cup \mathcal{S}_Q = \{x \in A \mid P(x) \vee Q(x)\}.$$

We will use De Morgan's laws:

$$\neg(p \wedge q) = \neg p \vee \neg q$$

$$\neg(p \vee q) = \neg p \wedge \neg q.$$

We define $p \Rightarrow q$ to mean $\neg p \vee q$ or $\neg(p \wedge \neg q)$.

- **Quantifiers**

We usually quantify over $\mathcal{N}$, so that $\forall x R(x)$ means $(\forall x \in \mathcal{N}) R(x)$ and $\exists x R(x)$ means $(\exists x \in \mathcal{N}) R(x)$. Quantifiers can also be *relativised* to predicates P on $\mathcal{N}$, thus:

$$(\forall x)_{P(x)} R(x) = \forall x \, [P(x) \Rightarrow R(x)]$$

and

$$(\exists x)_{P(x)} R(x) = \exists x \, [P(x) \wedge R(x)] \, .$$

In particular, we have *bounded* quantifiers:

$$(\forall x \leq n) P(x) = (\forall x)_{x \leq n} P(x),$$

$$(\forall x < n) P(x) = (\forall x)_{x < n} P(x),$$

$$(\exists x \leq n) P(x) = (\exists x)_{x \leq n} P(x),$$

$$(\exists x < n) P(x) = (\exists x)_{x < n} P(x).$$

De Morgan's laws for quantifiers are

$$\neg \forall x R(x) = \exists x \neg R(x),$$

$$\neg \exists x R(x) = \forall x \neg R(x),$$

$$\neg(\forall x)_{P(x)} R(x) = (\exists x)_{P(x)} \neg R(x),$$

$$\neg(\exists x)_{P(x)} R(x) = (\forall x)_{P(x)} \neg R(x).$$

- **Mathematical induction**

Let P be a predicate on $\mathcal{N}$. We give three different (but equivalent) formulations of this principle:

  - **Simple induction**

    If $P(0)$ *and* $\forall n \, [P(n) \Rightarrow P(n+1)]$
    then $\forall n P(n)$

  - **Course-of-values (CV) induction**

    If $\forall n \, [(\forall m < n \; P(m)) \Rightarrow P(n)]$
    then $\forall n P(n)$

  - **Least number principle**

    If $\exists n P(n)$
    then $\exists$ *least* $n P(n)$,
    that is, $\exists n \, [P(n) \wedge \forall m < n \; \neg P(m)]$.

EXERCISE: Prove that $\mathrm{PRED}(\mathcal{N})$ is uncountable (see Theorem 2.3).

3 Programs which Compute Functions

3.1 Programming language $\mathcal{G}$

The basis for our study of *computable functions* is the programming language $\mathcal{G}$ (for "goto"; it is called $\mathcal{S}$ in [1]).

3.1.1 Syntax and Informal Semantics

The syntax of $\mathcal{G}$ includes three classes of (program) variables:

- *input variables* $X_1, X_2, X_3, \cdots$,
- *auxiliary* or *local variables* $Z_1, Z_2, Z_3, \cdots$,
- the *output variable* Y,

and also

- *labels* $A_1, B_1, \cdots E_1, A_2, B_2, \cdots E_2, \cdots$.

We use $V, W, V', \cdots$ for any variable, $L, L_1, \cdots$ for any label, and often omit the subscript 1, e.g. 'X' means X_1, and 'A' means A_1.

Statements $S, \ldots$ have one of the following four forms:

$V{+}{+}$	(*increment*)
$V{-}{-}$	(*decrement*)
if $V{\neq}0$ goto L	(*conditional branch*)
skip	

An *instruction* has either of the two forms

	S	(*unlabelled statement*)
or	$[L]$ S	(*labelled statement*)

A *program* $\mathcal{P}$ is a list of instructions, possibly the *empty* list $\emptyset$.

In order to elucidate the *informal semantics* of $\mathcal{G}$-programs, we make the following assumptions. (The formal semantics are given later, in §3.1.3.):

- Auxiliary variables and the output variable Y are always *initialised* to 0.
- If V has the value 0, then instruction '$V{-}{-}$' *leaves* its value at 0.
- Execution of a program *halts* if *either* it has executed its last instruction, *or* it has executed an instruction '$\cdots$ goto L' without containing a label L.
- The label E will be used for an exit instruction, i.e. it will never be used to label a statement, and so 'goto E' will always mean "exit".

Note that variables can only take values in $\mathcal{N}$. We indicate the *value* of a variable by its lower case equivalent, e.g. x_1 denotes the *value* of X_1. More generally, lower case letters x_1, x_2, $\cdots$, k,m,n,r, $\cdots$, u,v, $\cdots$ will denote numbers (elements of $\mathcal{N}$).

Under the above informal semantics, it is clear that each $\mathcal{G}$-program computes a function on $\mathcal{N}$. This will be formalised later, in §4.1. This function is, in general, *partial*, since for some input values the programs may *diverge* (not halt).

For convenience we introduce abbreviating pseudo-instructions, called *macros*, and refer to the program texts they abbreviate as their *macro expansions*. For example, $\boxed{\text{goto } L}$ and $\boxed{V \leftarrow 0}$ are the macros for an *unconditional branch* and an *assignment of 0*, and have as macro expansions the program segments

```
    Z++
    if Z ≠ 0 goto L
```

and

```
[L] V--
    if V ≠ 0 goto L
```

respectively.

Note that when inserting macro expansions in a program, we have to be concerned with issues such as:

- initialisation of auxiliary variables,
- choosing auxiliary variables and labels not used in the main program, and
- replacing 'E' by the label for the statement immediately following the macro, if such a statement exists.

This is discussed more systematically in §4.2.

3.1.2 Examples of $\mathcal{G}$-programs

- **Identity function** $\lambda x \cdot x$
  1. First attempt:

```
[A] X--
    Y++
    if X ≠ 0 goto A
```

  However, this is incorrect since, for input 0, the program produces output 1 instead of 0.

  2. Second attempt:

```
[A] if X ≠ 0 goto B
    goto E
[B] X--
    Y++
    goto A
```

  The problem here is that the value of the input variable X is destroyed.

  3. Third attempt:

```
[A] if X ≠ 0 goto B
    goto C
[B] X--
    Y++
    Z++
    goto A
[C] if Z ≠ 0 goto D
    goto E
[D] Z--
    X++
    goto C
```

From this program we can get the assignment macro $\boxed{V \leftarrow W}$:

$$\boxed{\begin{array}{l} V \leftarrow 0 \\ \textit{Above program with } X \textit{ and } Y \\ \textit{replaced by } W \textit{ and } V \end{array}}$$

- **Sum function** $\lambda x_1, x_2 \cdot (x_1 + x_2)$

$$\boxed{\begin{array}{ll} & Y \leftarrow X_1 \\ & Z \leftarrow X_2 \\ [B] & \mathsf{if}\ Z \neq 0\ \mathsf{goto}\ A \\ & \mathsf{goto}\ E \\ [A] & Z{-}{-} \\ & Y{+}{+} \\ & \mathsf{goto}\ B \end{array}}$$

This program may now form the basis of the macro $\boxed{V \leftarrow W_1 + W_2}$ for *addition.*

- **Product function** $\lambda x_1, x_2 \cdot (x_1 * x_2)$

$$\boxed{\begin{array}{ll} & Z \leftarrow X_2 \\ [B] & \mathsf{if}\ Z \neq 0\ \mathsf{goto}\ A \\ & \mathsf{goto}\ E \\ [A] & Z{-}{-} \\ & \left.\begin{array}{l} Z_2 \leftarrow X_1 + Y \\ Y \leftarrow Z_2 \end{array}\right\} (*) \\ & \mathsf{goto}\ B \end{array}}$$

Note that the two statements in (*) may *not* be replaced by the single statement $Y \leftarrow X_1 + Y$, since the addition macro (as given above) does not work correctly for statements of the form $V \leftarrow W + V$. (We will see how to deal with this problem later, in §4.2.)

EXERCISE: Write $\mathcal{G}$-programs to compute:

1. The zero function $\lambda x \cdot 0$.
2. The everywhere diverging function $\lambda x \cdot \uparrow$.
3. The function $f(x) = \begin{cases} 1 & \text{if } x \text{ even} \\ 0 & \text{if } x \text{ odd.} \end{cases}$
4. The function $f(x) = \begin{cases} 1 & \text{if } x \text{ even} \\ \uparrow & \text{if } x \text{ odd.} \end{cases}$
5. The "*monus*" function
$$f(x_1, x_2) = x_1 \dot{-} x_2 = \begin{cases} x_1 - x_2 & \text{if } x_1 \geq x_2 \\ 0 & \text{otherwise.} \end{cases}$$
6. The predicate $\lambda x_1, x_2 \cdot (x_1 \leq x_2)$.

3.1.3 Formal Semantics for $\mathcal{G}$

We introduce the following notions:

- $\boldsymbol{var}(S)$ is the set of variables in statement S.
- $\boldsymbol{var}(\mathcal{P})$ is the set of variables in program $\mathcal{P}$.
- $\boldsymbol{lab}(\mathcal{P})$ is the set of labels in program $\mathcal{P}$.
- A *state* is a finite function from some set of variables to $\mathcal{N}$. We use the Greek lower case letters to denote states, e.g. $\sigma = \{(X,3), (Y,2), (Z,4)\}$.
- σ is a *state of progam* $\mathcal{P}$ iff $\boldsymbol{dom}(\sigma) \supseteq \boldsymbol{var}(\mathcal{P})$, i.e. σ assigns a value to each variable in $\mathcal{P}$.
- The *variant* $\sigma\{V/m\}$ of a state σ is the state τ which corresponds to σ except that $\tau(V) = m$. In other words, $\boldsymbol{dom}(\tau) = \boldsymbol{dom}(\sigma) \cup \{V\}$, and for all $W \in \boldsymbol{dom}(\tau)$,
$$\tau(W) = \begin{cases} \sigma(W) & \textit{if } W \not\equiv V \\ m & \textit{if } W \equiv V. \end{cases}$$
(NOTE: Here and elsewhere,'$\equiv$' denotes syntactic identity.)
- For a program $\mathcal{P}$, $|\mathcal{P}|$ denotes the *length* of $\mathcal{P}$, i.e., the number of instructions in $\mathcal{P}$; and $(\mathcal{P})_i$ denotes the i-th instruction of $\mathcal{P}$, for $1 \leq i \leq |\mathcal{P}|$.
- A *snapshot* or *instantaneous description* of $\mathcal{P}$, with $|\mathcal{P}| = \ell$, is a *pair* $s = (i, \sigma)$ where $1 \leq i \leq \ell + 1$ and σ is a *state* of $\mathcal{P}$. Intuitively, σ is the state just before the execution of $(\mathcal{P})_i$ if $1 \leq i \leq \ell$, or after completing the execution of $\mathcal{P}$ if $i = \ell + 1$. In the latter case, s is the *terminal snapshot* and σ the *terminal state* of $\mathcal{P}$.
- If (i, σ) is a *non-terminal snapshot* of $\mathcal{P}$, i.e. $i \leq |\mathcal{P}|$, then it has a *successor* (j, τ) (w.r.t. $\mathcal{P}$) , defined as follows:
  - Case 1: $(\mathcal{P})_i \equiv V{+}{+}$ and $\sigma(V) = m$. Then $j = i + 1$ and $\tau = \sigma\{V/m + 1\}$.
  - Case 2: $(\mathcal{P})_i \equiv V{-}{-}$ and $\sigma(V) = m$. Then $j = i+1$ and $\tau = \begin{cases} \sigma\{V/m - 1\} & \text{if } m > 0 \\ \sigma & \text{if } m = 0 \end{cases}$
  - Case 3: $(\mathcal{P})_i \equiv \mathsf{skip}$. Then $j = i + 1$ and $\tau = \sigma$.
  - Case 4: $(\mathcal{P})_i \equiv \mathsf{if}\ V \neq 0\ \mathsf{goto}\ L$. Then $\tau = \sigma$, and for j we have the two subcases:
    - $\sigma(V) = 0$. Then $j = i + 1$.
    - $\sigma(V) \neq 0$. Then j is the *least* number such that $(\mathcal{P})_j$ has label L, if $\mathcal{P}$ contains L. Otherwise, $j = \ell + 1$. (So if L occurs more than once in $\mathcal{P}$, then its first occurrence is used, and if L does not occur at all then $\mathcal{P}$ halts.)
- A *finite computation* of $\mathcal{P}$ is a list $s_1, s_2, \cdots, s_k$ of snapshots such that $s_1 = (1, \sigma_1)$ and for $i = 1, \cdots, k-1$, s_{i+1} is the successor (w.r.t. $\mathcal{P}$) of s_i, and s_k is terminal. An *infinite computation* of $\mathcal{P}$ is an infinite list $s_1, s_2, \cdots$ of snapshots such that $s_1 = (1, \sigma_1)$ and for $i = 1, 2, \cdots$, s_{i+1} is the successor (w.r.t. $\mathcal{P}$) of s_i .

  In both cases, we have a *computation of* $\mathcal{P}$ with *initial snapshot* $(1, \sigma_1)$ and *initial state* σ_1, or a *computation of* $\mathcal{P}$ *from* σ_1.

4 $\mathcal{G}$-Computable Functions

Computability theory is the study of computable functions. In our approach, the notion of *computability* is relative to the programming language $\mathcal{G}$. For this to be an interesting concept, we will have to show that it is stable, i.e. not dependent on slight changes in the definition of $\mathcal{G}$. Furthermore, we will have to link this with more traditional characterisations of computability. These will both be done later in the paper.

4.1 $\mathcal{G}$-computability

We formalise the fundamental notion: a *$\mathcal{G}$-program $\mathcal{P}$ computes an n-ary function f.*

- For any positive integer n and any n numbers $x_1, x_2, \cdots, x_n$, consider a *computation* $s_1, s_2, \cdots$ for $\mathcal{P}$ with *initial snapshot* $s_1 = (1, \sigma_1)$, where $\sigma_1 : \boldsymbol{var}(\mathcal{P}) \to \mathcal{N}$ is defined by

$$\begin{array}{ll} \sigma_1(X_i) = x_i & \text{for } i = 1, \cdots, n \\ \sigma_1(X_i) = 0 & \text{for } i > n \\ \sigma_1(Z_j) = 0 & \text{for all } Z_j \in \boldsymbol{var}(\mathcal{P}) \\ \sigma_1(Y) = 0. & \end{array}$$

  - Case 1: This computation is *finite*, with *terminal snapshot* $s_k = (\ell + 1, \sigma_k)$ (where $\ell = |\mathcal{P}|$), and $\sigma_k(Y) = y$. Then $f(x_1, x_2, \cdots, x_n) = y$.
  - Case 2: This computation is *infinite*. Then $f(x_1, \cdots, x_n) \uparrow$.
- If $\mathcal{P}$ computes the n-ary function f, then we write $f = \Psi_{\mathcal{P}}^{(n)}$ (and often drop the superscript '(n)' when $n = 1$). Note that $\mathcal{P}$ is not required to have *exactly* n input variables, and a particular $\mathcal{P}$ can compute different n-ary functions for different values of n. For example, the program given for the sum function in §3.1.2 yields the following:

$$\begin{aligned} \Psi_{\mathcal{P}}^{(2)}(x_1, x_2) &= x_1 + x_2 \\ \Psi_{\mathcal{P}}^{(1)}(x_1) &= x_1 \\ \Psi_{\mathcal{P}}^{(3)}(x_1, x_2, x_3) &= x_1 + x_2 \end{aligned}$$

- For any $\mathcal{P}$ and n, the function $\Psi_{\mathcal{P}}^{(n)}$ is *computable* by $\mathcal{P}$.
- An n-ary function f is *$\mathcal{G}$-computable* if $f = \Psi_{\mathcal{P}}^{(n)}$ for some $\mathcal{G}$-program $\mathcal{P}$.
- f is *total $\mathcal{G}$-computable* if f is $\mathcal{G}$-computable and total.
- A *$\mathcal{G}$-computable n-ary predicate* is a total $\mathcal{G}$-computable function $P : \mathcal{N}^n \to \mathbf{2}$.

From the $\mathcal{G}$-programs in §3.1.2 and §3.1.3 it follows that the functions $\lambda x \cdot 0$, $\lambda x \cdot x$, $\lambda x, y \cdot (x + y)$, $\lambda x, y \cdot (x * y)$, and $\lambda x, y \cdot (x \dot{-} y)$ are $\mathcal{G}$-computable.

- $\mathrm{FN}^{(n)}$ denotes the class of n-ary (partial) functions, and $\mathrm{FN} = \cup_n \mathrm{FN}^{(n)}$.
- $\mathrm{TFN}^{(n)}$ denotes the class of n-ary *total* functions, and $\mathrm{TFN} = \cup_n \mathrm{TFN}^{(n)}$.
- $\mathcal{G}\text{-COMP}^{(n)}$ is the class of $\mathcal{G}$-computable n-ary (partial) functions, and $\mathcal{G}\text{-COMP} = \cup_n \mathcal{G}\text{-COMP}^{(n)}$.
- $\mathcal{G}\text{-TCOMP}^{(n)}$ is the class of n-ary *total* $\mathcal{G}$-computable functions, and $\mathcal{G}\text{-TCOMP} = \cup_n \mathcal{G}\text{-TCOMP}^{(n)}$.

Clearly, the following inclusion relations hold:

$$\boxed{\begin{array}{ccc} \mathcal{G}\text{-COMP} & \subseteq & \mathrm{FN} \\ \cup & & \cup \\ \mathcal{G}\text{-TCOMP} & \subseteq & \mathrm{TFN} \end{array}}$$

The question as to whether the above "$\subseteq$" inclusions are proper, i.e. whether *all* functions are computable, still has to be answered.

NOTE: For historical reasons, total $\mathcal{G}$-computable functions are also called *recursive* functions, and $\mathcal{G}$-computable functions are also called *partial recursive* functions.

4.2 Macros for $\mathcal{G}$-computable functions

Once we have a $\mathcal{G}$-program $\mathcal{P}$ which computes an n-ary function f, we can augment our language $\mathcal{G}$ with a macro $\boxed{W \leftarrow f(V_1, V_2, \cdots, V_n)}$ for f derived from $\mathcal{P}$ as follows:

1. Assume
   - $\boldsymbol{var}(\mathcal{P}) \subseteq \{X_1, \cdots, X_n, Z_1, \cdots, Z_k, Y\}$,
   - $\boldsymbol{lab}(\mathcal{P}) \subseteq \{E, A_1, \cdots, A_l\}$,
   - for instructions of the form '$\mathsf{if}\ V \neq 0\ \mathsf{goto}\ A_i$' in $\mathcal{P}$, there is an instruction in $\mathcal{P}$ labelled A_i, and E is the only exit label.

   Clearly, $\mathcal{P}$ can easily be modified to meet these requirements. So let us put

$$\mathcal{P} \equiv \mathcal{P}(Y, X_1, \cdots, X_n, Z_1, \cdots, Z_k, E, A_1, \cdots, A_l)$$

2. Now *choose* m sufficiently large so that all variables and labels in the main program have indices less than m, and let

$$\begin{aligned} \mathcal{P}_m \equiv\ & \mathcal{P}(Z_m, Z_{m+1}, \cdots, Z_{m+n}, Z_{m+n+1}, \cdots, \\ & Z_{m+n+k}, E_m, A_{m+1}, \cdots, A_{m+l}) \end{aligned}$$

3. Then let macro $\boxed{W \leftarrow f(V_1, \cdots, V_n)}$ have the expansion

$$\boxed{\begin{array}{ll} & Z_m \leftarrow 0 \\ & Z_{m+1} \leftarrow V_1 \\ & \quad\vdots \\ & Z_{m+n} \leftarrow V_n \\ & Z_{m+n+1} \leftarrow 0 \\ & \quad\vdots \\ & Z_{m+n+k} \leftarrow 0 \\ & \mathcal{P}_m \\ [E_m] & W \leftarrow Z_m \end{array}}$$

Observe that

- we may have $W \equiv V_i$ for some $i \in \{1, 2, \cdots, n\}$, and
- if $f(v_1, \cdots, v_n) \uparrow$, then the macro for f will *not terminate* if it is entered in state σ such that $\sigma(V_i) = v_i$, $i = 1, 2, \cdots, n$. (Therefore the whole program

will not terminate.)

A useful extension of the language $\mathcal{G}$ is a generalisation of the conditional branch statement by means of the macro $\boxed{\textsf{if } P(V_1, \cdots, V_n) \textsf{ goto } L}$, where P is *any computable predicate*. The appropriate macro expansion is

$$\boxed{\begin{array}{l} Z \leftarrow P(V_1, \cdots, V_n) \\ \textsf{if } Z \neq 0 \textsf{ goto } L \end{array}}$$

EXAMPLE: If we want to use the statement '$\textsf{if } V = 0 \textsf{ goto } L$', we have to verify that the predicate

$$P(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

is computable. Indeed, the appropriate $\mathcal{G}$-program is

$$\boxed{\begin{array}{l} \textsf{if } X \neq 0 \textsf{ goto } E \\ Y{+}{+} \end{array}}.$$

4.3 Relative $\mathcal{G}$-computability

We extend the language $\mathcal{G}$ to include *oracle statements*, and *relativise* the concept of $\mathcal{G}$-program with respect to such statements.

Let $\vec{g} = g_1, \cdots, g_k$ be functions of arity $r_1, \cdots, r_k$. An *oracle statement* for g_i has the form

$$\boxed{V \leftarrow g_i(U_1, \cdots, U_{r_i})}.$$

For the semantics of such a statement, we can think of an *oracle* or "black box" for g_i, which, when given input values $\vec{u} = u_1, \cdots, u_{r_i}$ for $U_1, \cdots, U_{r_i}$ *either* produces the output value g_i for V (if $g_i(\vec{u}) \downarrow$) *or* "ticks over" indefinitely (if $g_i(\vec{u}) \uparrow$).

In this way, the notion of $\mathcal{G}$-computable and the function classes $\mathcal{G}$-COMP and $\mathcal{G}$-TCOMP can be relativised to obtain the notion *$\mathcal{G}$-computable in $\vec{g}$*, and the function classes $\mathcal{G}$-COMP$(\vec{g})$ and $\mathcal{G}$-TCOMP$(\vec{g})$. If a function is total $\mathcal{G}$-computable in $\vec{g}$, then it is also said to be *recursive in $\vec{g}$*. A *relativised* version of the diagram in §4.1 is

$$\boxed{\begin{array}{ccc} \mathcal{G}\text{-COMP}(\vec{g}) & \subseteq & \text{FN} \\ \cup & & \cup \\ \mathcal{G}\text{-TCOMP}(\vec{g}) & \subseteq & \text{TFN} \end{array}}$$

Once again, the question as to the properness of the "$\subseteq$" inclusions still needs to be answered.

Proposition 4.1 *(a) $\mathcal{G}$-COMP $\subseteq \mathcal{G}$-COMP$(\vec{g})$*
(b) $\mathcal{G}$-COMP $= \mathcal{G}$-COMP$(\emptyset)$
(c) If $\vec{g} \subseteq \vec{h}$, then $\mathcal{G}$-COMP$(\vec{g}) \subseteq \mathcal{G}$-COMP$(\vec{h})$.
Proof: Clear from the definition. □

Theorem 4.1 (Transitivity) *(a) If $f \in \mathcal{G}$-COMP$(\vec{g})$, and $g_1, \cdots, g_k \in \mathcal{G}$-COMP, then $f \in \mathcal{G}$-COMP.*
More generally:
(b) If $f \in \mathcal{G}$-COMP$(\vec{g})$, $g_1, \cdots, g_k \in \mathcal{G}$-COMP$(\vec{h})$, then $f \in \mathcal{G}$-COMP$(\vec{h})$,
(c) If $f \in \mathcal{G}$-COMP$(\vec{g}, \vec{h})$, $g_1, \cdots, g_k \in \mathcal{G}$-COMP$(\vec{h})$, then $f \in \mathcal{G}$-COMP$(\vec{h})$.
Proof:(a). Replace the oracle statement for g_i by the macro expansion for g_i $(i = 1, \cdots, k)$ in the (relative) $\mathcal{G}$-program for f.
(b), (c). Similarly. □

4.4 Construction of $\mathcal{G}$-computable functions

We are now going to take a different approach to computability. Namely, we will take a set of computable *initial functions*, together with general methods for constructing new computable functions from old. Initial functions will be introduced in §5.1, while this section, building on our theory of relative computability, contains two methods for forming new computable functions from old.

4.4.1 Composition

Given a k-ary function g and n-ary functions $h_1, \cdots, h_k$ we define the *composition* of g and $h_1, \cdots, h_k$ as the n-ary function

$$f(\vec{x}) \simeq g(h_1(\vec{x}), \cdots, h_k(\vec{x})) \tag{1}$$

where $\vec{x} \equiv x_1, \cdots, x_n$, and "$\simeq$" means that the left hand side of (1) is defined iff the right hand side of (1) is, in which case they are equal. Indeed, $f(\vec{x}) \downarrow y$ (say) iff there exists $z_1, \cdots, z_k$ such that $h_1(\vec{x}) \downarrow z_1 \wedge \cdots \wedge h_k(\vec{x}) \downarrow z_k \wedge g(\vec{z}) \downarrow y$.

Proposition 4.2 *In* (1), *if g and $\vec{h}$ are total, then so is f.*
Proof: EXERCISE. □

Theorem 4.2 *In* (1), *f is $\mathcal{G}$-computable in g, $h_1, \cdots, h_k$. Hence if g, $h_1, \cdots, h_k$ are $\mathcal{G}$-computable, then so is f.*
Proof: Using oracles for g,$h_1, \cdots, h_k$, we can construct a (relative) $\mathcal{G}$-program for f:

$$\boxed{\begin{array}{l} Z_1 \leftarrow h_1(X_1, \cdots, X_n) \\ \quad\vdots \\ Z_k \leftarrow h_k(X_1, \cdots, X_n) \\ Y \leftarrow g(Z_1, \cdots, Z_k) \end{array}}$$

The second part of the statement follows from Theorem 4.1*(a)*. □

4.4.2 Primitive Recursion

A unary function f, defined by

$$\begin{cases} f(0) & = & k \\ f(x+1) & = & h(x, f(x)) \end{cases} \tag{2}$$

with k fixed, and h a binary function, is said to be defined by *primitive recursion (without parameters)*.

Lemma 4.1 *For any $k \in \mathcal{N}$, the constant function $\lambda x \cdot k$ is $\mathcal{G}$-computable.*
Proof: For $k = 0$, either the empty program or the program $\boxed{\textsf{skip}}$ computes the function. For $k > 0$, the

following program may be used:

$$\boxed{\left.\begin{array}{l} Y{+}{+} \\ \vdots \\ Y{+}{+} \end{array}\right\} (k \text{ times})} \quad \square$$

These programs can form the basis of the macro $\boxed{Y \leftarrow k}$.

Proposition 4.3 *In* (2)*, if h is total, then so is f.*

Proof: By *induction on x* we can show that $\forall x(f(x)\downarrow)$. $\square$

Theorem 4.3 *In* (2)*, f is $\mathcal{G}$-computable in h. Hence if h is $\mathcal{G}$-computable, then so is f.*

Proof: Using an oracle for h, we can construct a (relative) $\mathcal{G}$-program for f:

$$\boxed{\begin{array}{ll} & Y \leftarrow k \\ [A] & \mathsf{if}\ X = 0\ \mathsf{goto}\ E \\ & Y \leftarrow h(Z,Y) \\ & Z{+}{+} \\ & X{-}{-} \\ & \mathsf{goto}\ A \end{array}}$$

As before, the second part of the statement follows from Theorem 4.1*(a)*. $\square$

The above is actually a special case of the more general concept of definition by *primitive recursion with parameters.* An $(n+1)$-ary function f, defined by

$$\left\{\begin{array}{rcl} f(\vec{x},0) & \simeq & g(\vec{x}) \\ f(\vec{x},t+1) & \simeq & h(\vec{x},t,f(\vec{x},t)) \end{array}\right. \tag{3}$$

with *parameters* $\vec{x} \equiv x_1,\cdots,x_n$ (where g and h have arities n and $n+1$ respectively), is said to be defined from g and h by *primitive recursion (with parameters).*

Proposition 4.4 *In* (3)*, if g and h are total, then so is f.*

Proof: By *induction on t* we can show that $\forall t(f(\vec{x},t)\downarrow)$. $\square$

Theorem 4.4 *In* (3)*, f is $\mathcal{G}$-computable in g,h. Hence if g,h are $\mathcal{G}$-computable, then so is f.*

Proof: Using oracles for g and h, the following (relative) $\mathcal{G}$-program computes f:

$$\boxed{\begin{array}{ll} & Y \leftarrow g(X_1,\cdots,X_n) \\ [A] & \mathsf{if}\ X_{n+1} = 0\ \mathsf{goto}\ E \\ & Y \leftarrow h(X_1,\cdots,X_n,Z,Y) \\ & Z{+}{+} \\ & X_{n+1}{-}{-} \\ & \mathsf{goto}\ A \end{array}} \quad \square$$

EXERCISE: Prove Proposition 4.2.

4.5 Effective calculability

A function is *effective* or *effectively calculable* or *algorithmic* iff there is a algorithm to compute it. This is an *intuitive*, not a mathematical notion, since it depends on the intuitive notion of *algorithm.* The classes of *effective functions* and *total effective functions* are denoted by EFF and TEFF respectively.

Clearly,

$$\boxed{\begin{array}{ccccc} \mathcal{G}\text{-COMP} & \subseteq & \text{EFF} & \subseteq & \text{FN} \\ \cup & & \cup & & \cup \\ \mathcal{G}\text{-TCOMP} & \subseteq & \text{TEFF} & \subseteq & \text{TFN} \end{array}}$$

A function f is *effective in $\vec{g}$* iff there is an *algorithm* for f which uses an "oracle" or "black box" for $\vec{g}$. EFF($\vec{g}$) and TEFF($\vec{g}$) denote the classes of *functions effective in $\vec{g}$* and *total functions effective in $\vec{g}$* respectively. The relativised version of the above diagram is

$$\boxed{\begin{array}{ccccc} \mathcal{G}\text{-COMP}(\vec{g}) & \subseteq & \text{EFF}(\vec{g}) & \subseteq & \text{FN} \\ \cup & & \cup & & \cup \\ \mathcal{G}\text{-TCOMP}(\vec{g}) & \subseteq & \text{TEFF}(\vec{g}) & \subseteq & \text{TFN} \end{array}}$$

As before, the question as to the properness of the above "$\subseteq$" inclusions needs to be answered.

5 Primitive Recursiveness

Having described (in §4.4) two ways of systematically forming new functions from existing ones, we introduce the class of initial functions, and the concepts of *primitive recursive (PR) closedness*, and *primitive recursive functions.*

5.1 PR-closed classes

The three *initial functions* are the *zero function* $\boldsymbol{Z} = \lambda x \cdot 0$, the *successor function* $\boldsymbol{S} = \lambda x \cdot (x+1)$, and the *projection functions* $\boldsymbol{U}_i^n = \lambda x_1 \cdots x_n \cdot x_i$ for $n \geq 0$, $1 \leq i \leq n$, of which the *identity function* $\boldsymbol{U}_1^1 = \lambda x \cdot x$ is a special case.

A class $\mathcal{C}$ of functions is *PR-closed* iff (i) $\mathcal{C}$ contains the initial functions, and (ii) $\mathcal{C}$ is *closed under composition and primitive recursion*, i.e. any function obtained from functions in $\mathcal{C}$ by *composition* or *primitive recursion* is also in $\mathcal{C}$.

Examples of PR-closed classes:

- FN (trivially).
- **Proposition 5.1** *TFN is PR-closed.*

  Proof: By definition, the initial functions are total. From Propositions 4.2, 4.3, and 4.4 it follows that totality is preserved by composition and primitive recursion. $\square$
- **Proposition 5.2** *$\mathcal{G}$-COMP is PR-closed.*

  Proof: The $\mathcal{G}$-programs $\boxed{\mathsf{skip}}$, $\boxed{\begin{array}{l} Y \leftarrow X \\ Y{+}{+} \end{array}}$, and $\boxed{Y \leftarrow X_i}$ compute the zero, successor, and projection functions respectively. By Theorems 4.2, 4.3, and 4.4 it follows that the class $\mathcal{G}$-COMP is closed under composition and primitive recursion. □

- **Proposition 5.3** *$\mathcal{G}$-TCOMP is PR-closed.*
  Proof: By Propositions 5.1 and 5.2 the classes TFN and $\mathcal{G}$-COMP are PR-closed. Hence their intersection $\mathcal{G}$-TCOMP is PR-closed. □

5.2 Primitive recursive functions

A function f is *primitive recursive* (PR) iff it is obtained from the *initial functions* by a finite number of applications of *composition* and *primitive recursion*. In other words, f is primitive recursive iff there is a *finite sequence* of functions $f_1, f_2, \cdots, f_n$ such that $f_n = f$, and for $i = 1, \cdots, n$, either f_i is an *initial function*, or f_i is obtained from some f_j's, for $j < i$, by *composition* or *primitive recursion*. Such a sequence is called a *PR derivation* of f, of length n.

More formally, a *PR derivation* of a function f is a sequence of labelled function symbols of the form:

$$\begin{array}{rl} & f_1 \leftarrow L_1 \\ & f_2 \leftarrow L_2 \\ & \vdots \\ f = & f_n \leftarrow L_n \end{array}$$

where for each $i = 1, \cdots, n$ one of the following cases applies:

- Case 1: f_i is an *initial function*, and label L_i is (correspondingly) one of '$\boldsymbol{Z}$', '$\boldsymbol{S}$' or '$\boldsymbol{U}^n_j$'.
- Case 2: f_i is obtained from an ℓ-ary function f_j, and m-ary functions $f_{k_1}, \cdots, f_{k_\ell}$ by *composition*, for $j, k_1, \cdots, k_\ell < i$, and the label L_i is '$f_j, f_{k_1}, \cdots, f_{k_\ell}$ $(\mathsf{compos} : \ell, m)$'.
- Case 3*a*: f_i is obtained from f_j and f_k, for $j, k < i$ by *recursion* with m parameters ($m > 0$), and the label L_i is 'f_j, f_k $(\mathsf{rec} : m)$'.
- Case 3*b*: f_i is obtained from f_k, for $k < i$ by recursion without parameters, and initial value c, and the label L_i is 'c, f_k $(\mathsf{rec} : 0)$'.

(We are not distinguishing here between functions and their symbols). The class of primitive recursive functions, and the class of n-ary primitive recursive functions are denoted by PR and $\mathrm{PR}^{(n)}$ respectively.

Lemma 5.1 *PR is PR-closed.*

Proof: Follows from the definition. □

Lemma 5.2 *Let $\mathcal{C}$ be any PR-closed class of functions. Then PR $\subseteq \mathcal{C}$.*

Proof: We can show that $f \in \mathrm{PR} \Rightarrow f \in \mathcal{C}$, by CV induction on the length of a PR-derivation of f. We distinguish three cases:

- Case 1: f is an *initial function*. Then $f \in \mathcal{C}$, since $\mathcal{C}$ is PR-closed.
- Case 2: f is obtained from earlier functions $g_1, \cdots, g_k$ in the derivation by *composition*. Then $g_1, \cdots, g_k$ have *shorter* PR-derivations (i.e. the initial parts of the PR-derivation of f ending with them), and so by the *induction hypothesis* they are in $\mathcal{C}$. Hence again, since $\mathcal{C}$ is PR-closed, $f \in \mathcal{C}$.
- Case 3: f is obtained from earlier functions in the derivation by *primitive recursion*. This is similar to case 2. □

Theorem 5.1 *PR is the* smallest PR-closed *class. In other words: (i) PR is PR-closed; and (ii) PR is contained in every PR-closed class.*

Proof: By Lemmas 5.1 and 5.2. □

Corollary 5.1 *PR $\subseteq$ TFN.*

Proof: By Proposition 5.1, TFN is PR-closed, and so by Theorem 5.1, PR $\subseteq$ TFN. □

Corollary 5.2 *PR $\subseteq$ $\mathcal{G}$-COMP.*

Proof: By Proposition 5.2, $\mathcal{G}$-COMP is PR-closed , and so by Theorem 5.1, PR $\subseteq$ $\mathcal{G}$-COMP. □

Corollary 5.3 *PR $\subseteq$ $\mathcal{G}$-TCOMP.*

Proof: By Corollaries 5.1 and 5.2, or since, by Proposition 5.3, $\mathcal{G}$-TCOMP is PR-closed. □

So clearly,

$$\boxed{\begin{array}{cccccccc} & & \mathcal{G}\text{-COMP} & \subseteq & \text{EFF} & \subseteq & \text{FN} \\ & & \cup & & \cup & & \cup \\ \text{PR} & \subseteq & \mathcal{G}\text{-TCOMP} & \subseteq & \text{TEFF} & \subseteq & \text{TFN} \end{array}}$$

Once again, the question as to the properness of the "$\subseteq$" inclusions still needs to be answered.

Examples of PR functions:

- **Sum function** $f = \lambda x, y \cdot (x + y)$
  This function has the well-known recursive definition:

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

  However, we must put it in the form required by §4.4.2 (3):

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

  where $g, h \in$ PR (with one parameter: x). So let us take $g(x) = x$, and $h(x, y, z) = z + 1$. Putting

$$g(x) = \boldsymbol{U}^1_1(x), \text{ and } h(x, y, z) = \boldsymbol{S}(\boldsymbol{U}^3_3(x, y, z)),$$

  a PR-derivation for f is

$$\begin{array}{rl} & f_1 \leftarrow \boldsymbol{U}^1_1 \\ & f_2 \leftarrow \boldsymbol{S} \\ & f_3 \leftarrow \boldsymbol{U}^3_3 \\ & f_4 \leftarrow f_2, f_3 \ (\mathsf{compos} : 1, 3) \\ f = & f_5 \leftarrow f_1, f_4 \ (\mathsf{rec} : 1). \end{array}$$

- **Product function** $f = \lambda x, y \cdot (x * y)$
  Recursive definition:

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

Required form:

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

where $g, h \in$ PR (with one parameter: x). Putting $g(x) = \boldsymbol{Z}(x)$, and

$$\begin{aligned} h(x,y,z) &= z + x \\ &= \boldsymbol{sum}(z,x) \\ &= \boldsymbol{sum}(\boldsymbol{U}_3^3(x,y,z), \boldsymbol{U}_1^3(x,y,z)), \end{aligned}$$

a PR-derivation for f is

$$\begin{aligned} & \vdots \\ sum = \quad & f_5 \leftarrow \cdots \\ & f_6 \leftarrow \boldsymbol{Z} \\ & f_7 \leftarrow \boldsymbol{U}_3^3 \\ & f_8 \leftarrow \boldsymbol{U}_1^3 \\ & f_9 \leftarrow f_5, f_7, f_8 \ (\mathsf{compos} : 2, 3) \\ f = \quad & f_{10} \leftarrow f_6, f_9 \ (\mathsf{rec} : 1). \end{aligned}$$

- **Factorial** $f = \lambda x \cdot x!$
  Recursive definition:

$$\begin{cases} f(0) &= 1 \\ f(x+1) &= f(x) * (x+1) \end{cases}$$

Required form:

$$\begin{cases} f(0) &= 1 \\ f(x+1) &= h(x, f(x)) \end{cases}$$

where $h \in$ PR (with no parameters). Putting

$$\begin{aligned} h(x,y) &= y * (x+1) \\ &= \boldsymbol{prod}(y, \boldsymbol{S}(x)) \\ &= \boldsymbol{prod}(\boldsymbol{U}_2^2(x,y), \boldsymbol{S}(\boldsymbol{U}_1^2(x,y))), \end{aligned}$$

we can obtain an appropriate PR-derivation, as before.

Clearly, we require an easier way to show that functions are PR! In §6 we address this problem, but before we do that, we conclude this section by generalising the notion of primitive recursive function to *relative primitive recursive function.*

5.3 Relative primitive recursiveness

Let $\vec{g} = g_1, \cdots, g_n$ be any functions. A function f is *primitive recursive in* $\vec{g}$ iff f is obtained from the *initial functions* and/or $g_1, \cdots, g_n$ by a finite number of applications of *composition* and *recursion.* Equivalently, f is PR in $\vec{g}$ iff there is a finite sequence of functions $f_1, \cdots, f_n$ such that $f_n = f$ and, for $i = 1, \cdots, n$, *either* f_i is an *initial function*, *or* f_i is one of the g_j's, *or* f_i is obtained from some f_j's ($j < i$) by *composition* or *primitive recursion.* Such a sequence is called a *PR-derivation of f from $\vec{g}$*, and PR($\vec{g}$) denotes the class of functions PR in $\vec{g}$.

Proposition 5.4 *(a) $PR \subseteq PR(\vec{g})$*
(b) $PR = PR(\emptyset)$
(c) If $\vec{g} \subseteq \vec{h}$, then $PR(\vec{g}) \subseteq PR(\vec{h})$.
Proof: Clear from the definition. □

Theorem 5.2 (Transitivity) *(a) If $f \in PR(\vec{g})$ and $g_1, \cdots, g_k \in PR$, then $f \in PR$.*
More generally:
(b) If $f \in PR(\vec{g})$ and $g_1, \cdots, g_k \in PR(\vec{h})$, then $f \in PR(\vec{h})$,
(c) If $f \in PR(\vec{g}, \vec{h})$ and $g_1, \cdots, g_k \in PR(\vec{h})$, then $f \in PR(\vec{h})$.
Proof:(*a*). *Prepend* a PR-derivation of f from $\vec{g}$ to PR-derivations of $g_1, \cdots, g_k$.
(*b*), (*c*). Similarly. □

Lemma 5.3 *$PR(\vec{g})$ is PR-closed and contains $\vec{g}$.*
Proof: Follows from the definition. □

Lemma 5.4 *Let $\mathcal{C}$ be any PR-closed class of functions which contains $\vec{g}$. Then $PR(\vec{g}) \subseteq \mathcal{C}$.*
Proof: We can show that $f \in \mathrm{PR}(\vec{g}) \Rightarrow f \in \mathcal{C}$, by CV induction on the length of the PR-derivation from $\vec{g}$ of f. □

Theorem 5.3 *$PR(\vec{g})$ is the* smallest *PR-closed class which contains $\vec{g}$. In other words, (i) $PR(\vec{g})$ is PR-closed and contains $\vec{g}$; and (ii) $PR(\vec{g})$ is contained in every PR-closed class which contains $\vec{g}$.*
Proof: By Lemmas 5.3 and 5.4. □

Corollary 5.4 *$PR(\vec{g}) \subseteq \mathcal{G}\text{-}COMP(\vec{g})$*
Proof: Since $\mathcal{G}$-COMP($\vec{g}$) contains $\vec{g}$ and is PR-closed. □

Note that PR($\vec{g}$) need not consist of total functions only, since the g_i might not be total! So if $TPR(\vec{g})$ is the class of *total* functions PR in $\vec{g}$, then the *relativised* version of the diagram in §5.2 is

$$\begin{array}{ccccccc} \mathrm{PR}(\vec{g}) & \subseteq & \mathcal{G}\text{-COMP}(\vec{g}) & \subseteq & \mathrm{EFF}(\vec{g}) & \subseteq & \mathrm{FN} \\ \cup & & \cup & & \cup & & \cup \\ \mathrm{TPR}(\vec{g}) & \subseteq & \mathcal{G}\text{-TCOMP}(\vec{g}) & \subseteq & \mathrm{TEFF}(\vec{g}) & \subseteq & \mathrm{TFN} \end{array}$$

As before, the question as to the properness of the above "⊆" inclusions needs to be answered.

6 Some Techniques for Defining PR Functions

6.1 Explicit definability

We introduce a convenient method for showing that certain functions are PR.

We must first define a certain class of formal expressions. Given a sequence $\vec{g} \equiv g_1, \cdots, g_m$ of functions of arity $r_1, \cdots, r_m$, and a sequence $\vec{x} \equiv x_1, \cdots, x_n$ of *indeterminates*, the class $\boldsymbol{Expr}(\vec{g}, \vec{x})$ of *expressions* in $\vec{g}, \vec{x}$ is *defined inductively* by:

1. $x_i \in \boldsymbol{Expr}(\vec{g}, \vec{x}) \quad (i = 1, \cdots, n)$,

2. $\bar{0} \in \boldsymbol{Expr}(\vec{g},\vec{x})$, where $\bar{0}$ a symbol for the number 0,
3. If $E \in \boldsymbol{Expr}(\vec{g},\vec{x})$, then so is $\bar{\boldsymbol{S}}(E)$, where $\bar{\boldsymbol{S}}$ is a symbol for the successor function $\boldsymbol{S}$,
4. If $E_1,\cdots,E_{r_i} \in \boldsymbol{Expr}(\vec{g},\vec{x})$, then so is $\bar{g}_i(E_1,\cdots,E_{r_i})$ $(i=1,\cdots,m)$, where $\bar{g}_i$ is a symbol for the function g_i.

(More on inductive definitions may be found in [3], §55.) Since each expression in $\vec{g}$, $\vec{x}$ represents an *explicit definition* of an n-ary function, we define an (n-ary) function f to be *explicitly definable from* $\vec{g}$ iff $\bar{f}(\vec{x}) \in \boldsymbol{Expr}(\vec{g},\vec{x})$, where $\bar{f}$ is a symbol for f.

NOTES:

1. The constant function $\boldsymbol{C}^n_k = \lambda\vec{x}\cdot k$ is explicitly defined from $\vec{g}$ by the *numeral* $\bar{k} =_{\text{df}} \underbrace{\bar{\boldsymbol{S}}(\cdots\bar{\boldsymbol{S}}}_{k \text{ times}}(\bar{0})\cdots)$.
2. In general we will not distinguish between functions and their symbols, or between numbers and their numerals.

Theorem 6.1 *If f is explicitly definable from $\vec{g}$, then $f \in PR(\vec{g})$. Hence if f is explicitly definable from PR functions, then $f \in PR$.*

Proof: The first part of the statement is proved by induction on the complexity of the expression defining f from $\vec{g}$. The second part from Theorem 5.2*(a)* □

Corollary 6.1 *In particular, we can define new PR functions from old by:*
(a) permuting arguments, e.g. $f(x,y) = g(y,x)$
(b) using dummy arguments, e.g. $f(x,y,z) = g(x,y)$
(c) identifying arguments, e.g. $f(x) = g(x,x)$
(d) substituting numerals for args., e.g. $f(x) = g(\bar{2},x)$
(e) any combination of the above.

Proof: *(a)* $f \in \mathrm{PR}(\vec{g})$ since

$$f(x,y) = g(\boldsymbol{U}^2_2(x,y), \boldsymbol{U}^2_1(x,y)).$$

(b)-(e) Similarly. □

EXAMPLE: If $f(x,y,z) = g(x,h(z,k(x)),\bar{2})$, then f is explicitly definable from g,h, k. Putting $\vec{x} \equiv x_1,x_2,x_3$,

$$f(\vec{x}) = g(\boldsymbol{U}^3_1(\vec{x}), h(\boldsymbol{U}^3_3(\vec{x}), k(\boldsymbol{U}^3_1(\vec{x})), \boldsymbol{C}^3_2(\vec{x}))),$$

which suggests a PR-derivation of f from g,h,k.

So from now on, we will freely use *explicit definitions*, as well as *infix* and *postfix* notation, to show that functions are PR.

More examples of PR functions:

- **Exponential** $\lambda x,y\cdot x^y$
  Defined by *primitive recursion on the second argument*: $\begin{cases} x^0 &=& 1 \\ x^{y+1} &=& x^y * x. \end{cases}$
- **Predecessor** $\boldsymbol{pd}(x) = \begin{cases} x-1 & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases}$
  Defined by prim. rec.: $\begin{cases} \boldsymbol{pd}(0) &=& 0 \\ \boldsymbol{pd}(x+1) &=& x. \end{cases}$
- **Monus** $x\dot{-}y = \begin{cases} x-y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$
  Defined by prim. rec. on the second argument: $\begin{cases} x\dot{-}0 &=& x \\ x\dot{-}(y+1) &=& \boldsymbol{pd}(x\dot{-}y). \end{cases}$
- **Absolute difference** $\lambda x,y\cdot|x-y|$
  Defined by *explicit definition* from $\dot{-}$ and $+$ which are both PR:
  $$|x-y| = (x\dot{-}y) + (y\dot{-}x).$$
- **Zero predicate (characteristic function of 0)**
  $\boldsymbol{zero}(x,y) = \begin{cases} 1 & \text{if } x=0 \\ 0 & \text{otherwise} \end{cases}$
  Defined by prim. rec.: $\begin{cases} \boldsymbol{zero}(0) &=& 1 \\ \boldsymbol{zero}(x+1) &=& 0 \end{cases}$
  or by expl. def. from $\boldsymbol{monus}$: $\boldsymbol{zero}(x) = 1\dot{-}x$.
- **Characteristic function of positive integers**
  $\boldsymbol{pos}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{otherwise} \end{cases}$
  Defined by prim. rec.: $\begin{cases} \boldsymbol{pos}(0) &=& 0 \\ \boldsymbol{pos}(x+1) &=& 1. \end{cases}$
- **Equality predicate (char. fn. of equality)**
  $\boldsymbol{eq}(x,y) = \begin{cases} 1 & \text{if } x=y \\ 0 & \text{otherwise} \end{cases}$
  Defined by expl. def.: $\boldsymbol{eq}(x,y) = \boldsymbol{zero}(|x-y|)$.
- **Less-than-or-equal predicate**
  $$\boldsymbol{leq}(x,y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$$
  Defined by expl. def.: $\boldsymbol{leq}(x,y) = \boldsymbol{zero}(x\dot{-}y)$.

Theorem 6.2 *Let P and Q be n-ary predicates. If we define the predicates $R_1(\vec{x}) \Leftrightarrow \neg P(\vec{x})$, $R_2(\vec{x}) \Leftrightarrow P(\vec{x}) \wedge Q(\vec{x})$, and $R_3(\vec{x}) \Leftrightarrow P(\vec{x}) \vee Q(\vec{x})$, then $R_1 \in PR(P)$ and R_2, $R_3 \in PR(P,Q)$. More informally: the predicate $\neg P$ is PR in P, and the predicates $P \wedge Q$, and $P \vee Q$ are PR in P,Q. Hence if $P,Q \in PR$, then so are $\neg P$, $P \wedge Q$, $P \vee Q$.*

Proof: $R_1(\vec{x}) = \boldsymbol{zero}(P(\vec{x}))$, $R_2(\vec{x}) = P(\vec{x}) * Q(\vec{x})$, and $R_3(\vec{x}) = \boldsymbol{pos}(P(\vec{x}) + Q(\vec{x}))$. Alternatively, for R_3, by De Morgan's law, $P \vee Q \Leftrightarrow \neg(\neg P \wedge \neg Q)$.. □

Hence

- **Less predicate** $\lambda x,y\cdot x<y$
  is PR since $x<y \Leftrightarrow \neg(y \leq x)$.

Proposition 6.1 (Definition by cases) *Suppose f is defined by*

$$f(\vec{x}) \simeq \begin{cases} g(\vec{x}) & \text{if } P(\vec{x}) \\ h(\vec{x}) & \text{otherwise.} \end{cases}$$

Then $f \in PR(g,h,P)$. Hence if $g,h,P \in PR$, then so is f.

Proof: $f(\vec{x}) \simeq g(\vec{x}) * P(\vec{x}) + h(\vec{x}) * \boldsymbol{zero}(P(\vec{x}))$. □

Proposition 6.2 *Let P be an n-ary predicate, and $f_1,\cdots,f_n$ m-ary functions. Suppose that Q is defined by $Q(\vec{x}) \Leftrightarrow P(f_1(\vec{x}),\cdots,f_n(\vec{x}))$. Then $Q \in PR(P,f_1,\cdots,f_n))$. Hence if $P,f_1,\cdots,f_n \in PR$, then so is Q.*

Proof: By composition. □

Corollary 6.2 *Suppose that Q is defined by $Q(\vec{x}) \Leftrightarrow (f_1(\vec{x}) = f_2(\vec{x}))$. Then $Q \in PR(f_1, f_2)$. Hence if $f_1, f_2 \in PR$, then so is Q.*

Note that (in Propositions 6.1 and 6.2 and Corollary 6.2) if the f's are total, then Q is a predicate.

EXERCISES:

1. Does the converse of Theorem 6.1 hold (i.e. $f \in \mathrm{PR}(\vec{g}) \Rightarrow f$ explicitly definable from $\vec{g}$)? If so, prove it. If not, state a modified result which *is* true, and prove it.
2. **(Generalised definition by cases)** Let, for some $n \geq 2$, $g_1, \cdots, g_n$ be functions and $P_1, \cdots, P_{n-1}$ predicates. For the function f, as defined below, show that $f \in \mathrm{PR}(g_1, \cdots, g_n, P_1, \cdots, P_{n-1})$. Hence if $\vec{g}, \vec{P} \in \mathrm{PR}$, then so is f. (Hint: Induction on n with basis $n = 2$).

$$f(\vec{x}) \simeq \begin{cases} g_1(\vec{x}) & \text{if} \quad P_1(\vec{x}) \\ g_2(\vec{x}) & \text{if} \quad \neg P_1(\vec{x}) \wedge P_2(\vec{x}) \\ g_3(\vec{x}) & \text{if} \quad \neg P_1(\vec{x}) \wedge \neg P_2(\vec{x}) \wedge P_3(\vec{x}) \\ \vdots & \\ g_{n-1}(\vec{x}) & \text{if} \quad \neg P_1(\vec{x}) \wedge \cdots \wedge \neg P_{n-2}(\vec{x}) \wedge \\ & \qquad P_{n-1}(\vec{x}) \\ g_n(\vec{x}) & \text{if} \quad \neg P_1(\vec{x}) \wedge \cdots \wedge \neg P_{n-1}(\vec{x}). \end{cases}$$

6.2 Finite sums and products

Theorem 6.3 *Let f be an $(n+1)$-ary function. If*

$$\begin{aligned} g(y, \vec{x}) &= \textstyle\sum_{z<y} f(z, \vec{x}), \\ \textit{and} \quad h(y, \vec{x}) &= \textstyle\prod_{z<y} f(z, \vec{x}), \end{aligned}$$

then $g, h \in PR(f)$. Hence if $f \in PR$, then so are g, h.

Proof: Define g, h by *primitive recursion* on y:

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

and

$$\begin{cases} h(0, \vec{x}) &= 1 \\ h(y+1, \vec{x}) &= h(y, \vec{x}) * f(y, \vec{x}). \end{cases} \qquad \square$$

Corollary 6.3 *If*

$$\begin{aligned} g'(y, \vec{x}) &= \textstyle\sum_{z=0}^{y} f(z, \vec{x}), \\ \textit{and} \quad h'(y, \vec{x}) &= \textstyle\prod_{z=0}^{y} f(z, \vec{x}), \end{aligned}$$

then $g', h' \in PR(f)$.

Proof: $g'(y, \vec{x}) = g(y+1, \vec{x})$, and $h'(y, \vec{x}) = h(y+1, \vec{x})$. $\square$

Corollary 6.4 *If*

$$\begin{aligned} g''(y, \vec{x}) &= \textstyle\sum_{z=1}^{y} f(z, \vec{x}), \\ \textit{and} \quad h''(y, \vec{x}) &= \textstyle\prod_{z=1}^{y} f(z, \vec{x}), \end{aligned}$$

then $g'', h'' \in PR(f)$.

EXERCISE: Prove Corollary 6.4.

6.3 Bounded quantification

Theorem 6.4 *Let P be an $(n+1)$-ary predicate. If*

$$\begin{aligned} Q(y, \vec{x}) &= (\exists z < y) P(z, \vec{x}), \\ \textit{and} \quad R(y, \vec{x}) &= (\forall z < y) P(z, \vec{x}), \end{aligned}$$

then $Q, R \in PR(P)$. Hence if $P \in PR$, then so are Q and R.

Proof:

$$\begin{aligned} R(y, \vec{x}) &= \textstyle\prod_{z<y} P(z, \vec{x}), \\ \text{and} \quad Q(y, \vec{x}) &= \boldsymbol{pos}(\textstyle\sum_{z<y} P(z, \vec{x})), \end{aligned}$$

or alternatively, $Q(y, \vec{x}) \Leftrightarrow \neg(\forall z < y) \neg P(z, \vec{x})$. $\square$

Corollary 6.5 *If*

$$\begin{aligned} Q'(y, \vec{x}) &= (\exists z \leq y) P(z, \vec{x}), \\ \textit{and} \quad R'(y, \vec{x}) &= (\forall z \leq y) P(z, \vec{x}), \end{aligned}$$

then $Q', R' \in PR(P)$. Hence if $P \in PR$, then so are Q' and R'.

Corollary 6.6 *If*

$$\begin{aligned} Q''(y, \vec{x}) &\simeq (\exists z < f(y, \vec{x})) P(z, \vec{x}), \\ \textit{and} \quad R''(y, \vec{x}) &\simeq (\forall z < f(y, \vec{x})) P(z, \vec{x}), \end{aligned}$$

then $Q'', R'' \in PR(f, P)$. Hence if $f, P \in PR$, then so are Q'' and R''.

Intuitively, *bounded quantification* is *effective* in P since there are only finitely many cases to check, while *unbounded quantification*, in general, is *not*.

EXERCISE: Prove Corollaries 6.5 and 6.6.

6.4 Bounded minimalisation

Theorem 6.5 *Let P be an $(n+1)$-ary predicate. Define $f(y, \vec{x}) = (\mu z < y) P(z, \vec{x})$, meaning "the least $z < y$ such that $P(z, \vec{x})$ holds, if such z exists, 0 otherwise". Then $f \in PR(P)$. Hence if $P \in PR$, then so is f.*

Proof: Put

$$g(y, \vec{x}) = \sum_{z<y} \prod_{t<z} \boldsymbol{zero}(P(t, \vec{x})) \tag{4}$$

. Clearly, $g \in \mathrm{PR}(P)$. We distinguish two cases:

- Case 1: There exists $t < y$ such that $P(t, \vec{x})$ is true, i.e. $P(t, \vec{x}) = 1$.
  Let t_0 be the least such t. Then, for any $t < t_0$, $P(t, \vec{x}) = 0$ so that $\boldsymbol{zero}(P(t, \vec{x})) = 1$, and $\boldsymbol{zero}$ $(P(t_0, \vec{x})) = 0$. So for all z,

$$\prod_{t<z} \boldsymbol{zero}(P(t, \vec{x})) = \begin{cases} 1 & \text{if } z < t_0 \\ 0 & \text{if } z \geq t_0. \end{cases}$$

  Therefore,

$$\sum_{z<y} \prod_{t<z} \boldsymbol{zero}(P(t, \vec{x})) = \underbrace{1 + \cdots + 1}_{t_0 \text{ times}} + 0 + 0 + \cdots = t_0 \tag{5}$$

- Case 2: For all $t < y$, $P(t, \vec{x})$ is false, i.e. $P(t, \vec{x}) = 0$. Clearly, $\boldsymbol{zero}(P(t, \vec{x})) = 1$. So for all $z < y$,

$$\prod_{t<z} \boldsymbol{zero}(P(t, \vec{x})) = 1.$$

Therefore,

$$\sum_{z<y} \prod_{t<z} \boldsymbol{zero}(P(t, \vec{x})) = \underbrace{1 + \cdots + 1}_{y \text{ times}} = y. \quad (6)$$

From (4), (5) and (6) we obtain

$$g(y, \vec{x}) = \begin{cases} \text{"least } z < y \text{ such that } P(z, \vec{x}) \\ \quad \text{if such } z \text{ exists"} \\ y \quad \text{otherwise.} \end{cases}$$

Finally, we define

$$f(y, \vec{x}) = \begin{cases} g(y, \vec{x}) & \text{if } Q(y, \vec{x}) \\ 0 & \text{otherwise,} \end{cases}$$

with $Q(y, \vec{x}) = (\exists z < y) P(z, \vec{x})$. Therefore, by definition by cases, $f \in \text{PR}(g, Q, P)$; by Theorem 6.3, $g \in \text{PR}(P)$; and by Theorem 6.4, $Q \in \text{PR}(P)$. So $f \in \text{PR}(P)$. □

Corollary 6.7 *If $f(y, \vec{x}) = (\mu z \le y) P(z, \vec{x})$, then $f \in PR(P)$.*

Corollary 6.8 *If $f(y, \vec{x}) \simeq (\mu z \le g(y, \vec{x})) P(z, \vec{x})$, then $f \in PR(g, P)$.*

6.5 A note on unbounded minimalisation

Let P be an $(n+1)$-ary predicate, and f an n-ary function defined by

$$f(\vec{x}) \simeq \mu y P(\vec{x}, y), \quad (7)$$

meaning "the least y such that $P(\vec{x}, y)$ holds, if such y exists, and $\uparrow$ otherwise". Clearly, f is *not* necessarily total, so f does *not*, in general, belong to $\text{PR}(P)$. Intuitively, however, $f \in \text{EFF}(P)$ since the following algorithm, which uses an oracle for P, computes f:

> "Test $P(\vec{x}, 0), P(\vec{x}, 1), P(\vec{x}, 2), \cdots$
> until y is found such that $P(\vec{x}, y)$.
> Then halt, with output y."

NOTES:

1. The n-ary function

$$g(\vec{x}) = \begin{cases} \mu y P(\vec{x}, y) & \text{if } \exists y P(\vec{x}, y) \\ 0 & \text{otherwise} \end{cases}$$

is total, but *not* (in general) effective in P.

2. In (7), $f \in \mathcal{G}\text{-COMP}(P)$. Hence if $P \in \mathcal{G}\text{-COMP}$, then so is f. The reader may try to prove this now, or wait for Proposition 12.1.

6.6 More examples

We conclude with some further examples of PR functions and predicates:

- **integer division or quotient**

$$\begin{aligned} \boldsymbol{quot}(x, y) &= \lfloor x/y \rfloor \\ &= \mu z[z * y \le x \wedge (z+1) * y > x] \\ &= (\mu z \le x)[(z+1) * y > x]. \end{aligned}$$

- **remainder** $\boldsymbol{rem}(x, y) = x \dot{-} \boldsymbol{quot}(x, y) * y$.
- **divisibility predicate**
  $y|x \Leftrightarrow \boldsymbol{rem}(x, y) = 0$, or alternatively,
  $y|x \Leftrightarrow \exists z(x = y * z) \Leftrightarrow (\exists z \le x)(x = y * z)$.
- **primality predicate**

$$\begin{aligned} \boldsymbol{prime}(x) &\Leftrightarrow x > 1 \wedge \neg \exists y[1 < y \wedge y|x] \\ &\Leftrightarrow x > 1 \wedge \neg(\exists y < x)[1 < y \wedge y|x]. \end{aligned}$$

- **prime number sequence**
  Let p_n denote the n-th prime, with $p_0 = 0$. Is $\lambda n \cdot p_n \in \text{PR}$? The primitive recursive definition

$$\begin{cases} p_0 &= 0 \\ p_{n+1} &= \mu y[\boldsymbol{prime}(y) \wedge y > p_n] \end{cases}$$

  is problematic as it stands, since (i) μ is unbounded, and (ii) it assumes the existence of a prime $> p_n$, or equivalently, the existence of infinitely many primes. Euclid comes to the rescue.

  Theorem 6.6 (Euclid) *There are infinitely many primes. More precisely,*

$$\forall x \exists p[\boldsymbol{prime}(p) \wedge x < p \le (x! + 1)].$$

  Proof: Let $y = x!+1$. For $2 \le k \le x$, $\boldsymbol{rem}(y, k) = 1$. Hence for $2 \le k \le x$, $k \nmid y$. But y has at least one prime factor p. So $x < p \le y$. □

  Since this theorem also gives a PR bound for each new prime, it suggests the following definition by primitive recursion:

$$\begin{cases} p_0 = 0 \\ p_{n+1} = (\mu y \le (p_n! + 1))[\boldsymbol{prime}(y) \wedge y > p_n] \end{cases}$$

  which, by Corollary 6.8, is PR.

EXERCISES:

1. Show that the following functions and predicates are PR:
   (a) $\boldsymbol{even}(x)$ (x is even)
   (b) $\boldsymbol{min}(x, y)$
   (c) $\boldsymbol{perfsq}(x)$ (x is a perfect square)
   (d) $\boldsymbol{sqrt}(x)$ (integral square root of x)
   (e) $\boldsymbol{gcd}(x, y)$.
2. Show that every finite subset of $\mathcal{N}$ is PR.
3. Is every co-finite subset of $\mathcal{N}$ PR? (A set is co-finite if its complement is finite.)
4. Let $f(x)$ = "the number of 1's in the binary representation of x". Show that $f \in \text{PR}$.
5. For any total function f of one argument, define $g(n, x) = f^n(x)$ (the n-th iterated composition of

f). Is $g \in \mathrm{PR}(f)$?

7 PR Codings of Finite Sequences of Numbers

In the previous sections we elucidated the concepts of primitive recursiveness and $\mathcal{G}$-computability. In this section we discuss coding devices based on primitive recursive functions, and then use them to code $\mathcal{G}$-programs as numbers so that they can serve as inputs to other programs — or to themselves!

Theorem 7.1 (Fundamental Theorem of Arithmetic) *Every number > 1 can be represented* uniquely *(apart from order) as a product of primes.*

Hence for $x > 1$, we can write

$$x = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k} \tag{8}$$

for *unique* $k > 0$, $e_1, \cdots, e_k$, where p_i = i-th prime ($p_1 = 2$), $e_i \geq 0$ for $1 \leq i \leq k$, and $e_k > 0$.

Lemma 7.1 *(a) For $a \geq 2$, $n < a^n$.*
(b) $n \leq p_n$.

Proof: By induction on n. □

Hence in (8):

$$\left.\begin{array}{ll} e_i < p_i^{e_i} \leq x & (1 \leq i \leq k) \\ k \leq p_k \leq x & \end{array}\right\} \tag{9}$$

7.1 PR coding of pairs of numbers

We define

$$\boldsymbol{pair}(x, y) = \langle x, y \rangle = 2^x(2y+1) \dot{-} 1,$$

which is clearly PR.

Proposition 7.1

$$\forall z \exists ! x, y(\langle x, y \rangle = z) \tag{10}$$

Proof: We want $z = \langle x, y\rangle$ i.e. $z + 1 = 2^x(2y+1)$. By the fundamental theorem of arithmetic, $z + 1 = 2^x 3^{a_2} 5^{a_3} \cdots = 2^x u$ for *unique* x and u, where u is *odd* (possibly 1). Put $u = 2y + 1$. So y is also uniquely determined (possibly 0). □

NOTE: Proposition 7.1 determines two *inverse functions* satisfying (10), i.e. the functions *left inverse* $\boldsymbol{\ell}(z)$ and *right inverse* $\boldsymbol{r}(z)$, which satisfy

$$\begin{array}{rcl} \boldsymbol{\ell}(\langle x, y\rangle) & = & x, \\ \boldsymbol{r}(\langle x, y\rangle) & = & y, \\ \text{and} \quad \langle \boldsymbol{\ell}(z), \boldsymbol{r}(z)\rangle & = & z. \end{array}$$

Lemma 7.2 $x, y \leq \boldsymbol{pair}(x, y)$.

Proof: In (10), $x < 2^x \leq 2^x(2y+1) = z + 1$, and $y < 2y + 1 \leq 2^x(2y+1) = z + 1$. So $x, y \leq z$. □

Proposition 7.2 $\boldsymbol{\ell}, \boldsymbol{r} \in PR$.

Proof:

$$\begin{array}{rl} & \boldsymbol{\ell}(z) = (\mu x \leq z)(\exists y \leq z)(z = \langle x, y\rangle) \\ \text{and} & \boldsymbol{r}(z) = (\mu y \leq z)(\exists x \leq z)(z = \langle x, y\rangle). \end{array} \quad \square$$

Theorem 7.2 (Simultaneous or mutual primitive recursion) *Let*

$$\begin{array}{rcl} f_1(x, 0) & = & g_1(x) \\ f_2(x, 0) & = & g_2(x) \\ f_1(x, t+1) & = & h_1(x, t, f_1(x, t), f_2(x, t)) \\ f_2(x, t+1) & = & h_2(x, t, f_1(x, t), f_2(x, t)). \end{array}$$

Then $f_1, f_2 \in PR(g_1, g_2, h_1, h_2)$.
Hence if $g_1, g_2, h_1, h_2 \in PR$, then so are f_1, f_2.

Proof: We put $f(x, t) = \langle f_1(x, t), f_2(x, t)\rangle$ and show that $f \in \mathrm{PR}(g_1, g_2, h_1, h_2)$. Let

$$f(x, 0) = \langle g_1(x), g_2(x)\rangle = g(x) \ \ (say)$$

and

$$\begin{array}{rcl} f(x, t+1) & = & \langle h_1(x, t, f_1(x, t), f_2(x, t)), \\ & & \quad h_2(x, t, f_1(x, t), f_2(x, t))\rangle \\ & = & \langle h_1(x, t, \boldsymbol{\ell}(f(x, t)), \boldsymbol{r}(f(x, t))), \\ & & \quad h_2(x, t, \boldsymbol{\ell}(f(x, t)), \boldsymbol{r}(f(x, t)))\rangle \\ & = & h(x, t, f(x, t))) \ \ (say) \end{array}$$

where
$h(x, t, z) =_{\mathrm{df}} \langle h_1(x, t, \boldsymbol{\ell}(z), \boldsymbol{r}(z)), h_2(x, t, \boldsymbol{\ell}(z), \boldsymbol{r}(z))\rangle$.
So $f \in \mathrm{PR}(g, h)$, $g \in \mathrm{PR}(g_1, g_2)$, $h \in \mathrm{PR}(h_1, h_2)$ by explicit definition. Therefore, $f \in \mathrm{PR}(g_1, g_2, h_1, h_2)$. Finally, $f_1(x, t) = \boldsymbol{\ell}(f(x, t))$ and $f_2(x, t) = \boldsymbol{r}(f(x, t))$. So $f_1 \in \mathrm{PR}(f)$. Therefore, by transitivity,
$f_1 \in \mathrm{PR}(g_1, g_2, h_1, h_2)$. Similarly,
$f_2 \in \mathrm{PR}(g_1, g_2, h_1, h_2)$. □

7.2 PR coding of finite sequences of numbers

We define the *code* or *Gödel number (gn)* of a sequence $a_1, \cdots, a_n$ $(n \geq 0)$ as the number

$$[a_1, \cdots, a_n] = \prod_{i=1}^{n} p_i^{a_i}.$$

Proposition 7.3 *For fixed n,*

$$\lambda x_1, \cdots, x_n \cdot [x_1, \cdots, x_n] \in PR.$$

Proof: Clear. □

Theorem 7.3 (Uniqueness of components)

$$[a_1, \cdots, a_n] = [b_1, \cdots, b_n] \Rightarrow a_i = b_i \ \ (i = 1, \cdots, n).$$

Proof: By the fundamental theorem of arithmetic. □

NOTES:

1. $[a_1, \cdots, a_n, 0] = [a_1, \cdots, a_n]$, so trailing 0's make no difference.
2. $[0] = [0, 0] = [0, 0, 0] = \cdots = 2^0 3^0 5^0 \cdots = 1$, so 1 codes any sequence of 0's. We also assume that 1 codes the *empty sequence* [ ].

The following two functions are, in a sense, *inverses* of the gn function. Let $x = [a_1, \cdots, a_n]$. We define

$$(x)_i = \begin{cases} a_i & \text{if } 1 \le i \le n \\ 0 & \text{otherwise} \end{cases}$$

and for $x \neq 0$,

$$\begin{aligned} \boldsymbol{Lt}(x) &= \textit{length} \text{ of the sequence represented by } x \\ &= k \text{ when } x = [a_1, \cdots, a_k] \text{ with } a_k \neq 0 \end{aligned}$$

and put $\boldsymbol{Lt}(0) = 0$. Note that $(x)_i$ is *well-defined*, since for example, if $x = [a_1, a_2] = [a_1, a_2, 0, 0]$, then $(x)_4 = 0$ under either interpretation.

Proposition 7.4

(a) $([a_1, \cdots, a_n])_i = \begin{cases} a_i & \textit{if } 1 \le i \le n \\ 0 & \textit{otherwise} \end{cases}$

(b) $[(x)_1, \cdots, (x)_n] = x$ *if* $n \ge \boldsymbol{Lt}(x)$.

Proof: From the definitions. □

Theorem 7.4 $\lambda x, i \cdot (x)_i$, $\boldsymbol{Lt} \in PR$.

Proof: *(a)* $(x)_i = (\mu y < x)\neg(p_i^{y+1} | x)$.

(b) $\boldsymbol{Lt}(x) = \mu k[(x)_k \neq 0 \wedge (\forall j > k)((x)_j = 0)]$. But to apply the results of §6.3 and §6.4, we need bounds for k and j. So from (9), $\boldsymbol{Lt}(x) =$
$(\mu k < x)[(x)_k \neq 0 \wedge (\forall j < x)(k < j \Rightarrow (x)_j = 0)]$. □

NOTE 3: For later use we define

$$\boldsymbol{concat}(x, y) = x^\frown y = \text{concatenation of } x \text{ and } y,$$

where x and y are viewed as gn's of finite sequences.

Proposition 7.5 $\boldsymbol{concat} \in PR$.

Proof: Suppose that

$$\begin{aligned} x &= p_1^{a_1} \cdots p_k^{a_k}, \quad k = \boldsymbol{Lt}(x), \quad a_i = (x)_i, \quad a_k \neq 0 \\ y &= p_1^{b_1} \cdots p_\ell^{b_\ell}, \quad \ell = \boldsymbol{Lt}(y), \quad b_i = (y)_i, \quad b_\ell \neq 0. \end{aligned}$$

So

$$\begin{aligned} x^\frown y &= p_1^{a_1} \cdots p_k^{a_k} \cdot p_{k+1}^{b_1} \cdots p_{k+\ell}^{b_\ell} \\ &= x * \textstyle\prod_{i=1}^{\boldsymbol{Lt}(y)} p_{\boldsymbol{Lt}(x)+i}^{(y)_i}. \end{aligned}$$

□

EXERCISES:

1. **(CV recursion)** For any function f, write

$$\begin{cases} \tilde{f}(0) &= 1, \\ \tilde{f}(n) &= [f(0), \cdots, f(n-1)] \text{ if } n \neq 0. \end{cases}$$

   Now, given a function g, suppose f is defined by $f(n) = g(\tilde{f}(n))$. (The point is that the value of f at n depends explicitly on the *values* of f at i for *all* $i < n$, not just on $f(n-1)$, as with definition by primitive recursion.) Show that $f \in \text{PR}(g)$. (Hence if $g \in \text{PR}$, then so is f.)

2. **(Fibonacci sequence)** Let $F(0) = 0$, $F(1) = 1$, $F(n+2) = F(n) + F(n+1)$. Show that $F \in \text{PR}$.

7.3 Gödel numbering of the $\mathcal{G}$ programming language

Let S be a set. A *Gödel numbering (GN)* or *effective numbering* of S is a 1-1 map $\# : S \to \mathcal{N}$ such that for all $x \in S$, we can effectively (or algorithmically) find $\#(x) \in \mathcal{N}$, and for all $n \in \mathcal{N}$, we can effectively determine whether $n \in \boldsymbol{ran}(\#)$, and if so, effectively find the $x \in S$ such that $\#(x) = n$. Note that if S has a GN, then S is *countable* (by Theorem 2.2).

It is often convenient to make $\#$ *surjective*, in which case it has a bijective *inverse* $\#^{-1} : \mathcal{N} \to \mathcal{S}$ that is an *effective enumeration* of S. Moreover, we can move effectively from *surjective GN's* of S to *effective enumerations* of S, and vice versa, defining either one or the other, whichever is more convenient. Indeed, we have already defined *surjective GN's*, and hence *effective enumerations*, of $\mathcal{N}^2$ (§7.1) and $\mathcal{N}^*$, the set of all finite sequences from $\mathcal{N}$ (§7.2).

We are now ready to code $\mathcal{G}$-programs as numbers.

- **Effective enumeration of all variables**

$$\begin{array}{cccccccc} Y, & X_1, & Z_1, & X_2, & Z_2, & X_3, & Z_3, & \cdots \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & \cdots \end{array}$$

  For example, $\#(X_2) = 4$.

- **Effective enumeration of all labels**

$$\begin{array}{cccccccc} A_1, & B_1, & C_1, & D_1, & E_1, & A_2, & B_2, & \cdots \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & \cdots \end{array}$$

  For example, $\#(B_2) = 7$.

- **Gödel numbering of all instructions**
  For convenience we *replace* 'skip' by '$V \leftarrow V$' for any variable V. Then the Gödel numbering of instruction I is $\#(I) = \langle a, \langle b, c \rangle \rangle$ where
  - $a = \begin{cases} 0 & \text{if } I \text{ is unlabelled} \\ \#(L) & \text{if } I \text{ has label } L \end{cases}$
  - $b = \begin{cases} 0 & \text{if } I \text{ is} & V \leftarrow V \\ 1 & \text{"} & V{+}{+} \\ 2 & \text{"} & V{-}{-} \\ \#(L') + 2 & \text{"} & \text{if } V \neq 0 \text{ goto } L' \end{cases}$
  - $c = \#(V) - 1$ if the variable in I is V.

  The associated *effective enumeration* of all instructions is obtained as follows: Given $q \in \mathcal{N}$, we let $a = \boldsymbol{\ell}(q)$, $b = \boldsymbol{\ell}(\boldsymbol{r}(q))$, $c = \boldsymbol{r}(\boldsymbol{r}(q))$. Then, the statement
  - is unlabelled if $a = 0$, and the statement has the label with number a if $a \neq 0$.
  - is $\begin{cases} V \leftarrow V & \text{if} & b = 0 \\ V{+}{+} & \text{"} & b = 1 \\ V{-}{-} & \text{"} & b = 2 \\ \text{if } V \neq 0 \text{ goto } L & \text{"} & b > 2 \end{cases}$
    where the label L is such that $\#(L) = b - 2$.
  - uses variable V with $\#(V) = c + 1$.

- **Gödel numbering of programs**
  Let $\mathcal{P} = (I_1, \cdots, I_k)$ be a program. We define

$$\#(\mathcal{P}) = [\#(I_1), \cdots, \#(I_k)] - 1$$

  which is *surjective* and, therefore, gives an *effective enumeration* of programs. But note that the unlabelled statement '$Y \leftarrow Y$' has Gödel numbering 0, and hence we can form *many* programs $\mathcal{P}$ with the same $\#(\mathcal{P})$ by simply adding any number of unlabelled statements '$Y \leftarrow Y$'. To prevent this, we *stipulate* that a program may not end with an unlabelled statement of the form '$Y \leftarrow Y$'. Let us denote by $\mathcal{G}$-PROG the set of all such programs. Then

$$\# : \mathcal{G}\text{-PROG} \to \mathcal{N}$$

is *injective* and even *bijective*. So the inverse of $\#$ is an *effective enumeration* of $\mathcal{G}$-PROG.

Now let $\mathcal{P}_n$ be the n-th program under the above GN, i.e. the program $\mathcal{P}$ with $\#(\mathcal{P}) = n$. Then

$$\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \cdots.$$

is an effective enumeration of $\mathcal{G}$-PROG.

EXERCISES:

1. Let $\mathcal{P}$ be the program

   ```
   if X ≠ 0 goto A
   Y++
   ```

   which computes the *zero* function. What is $\#(\mathcal{P})$?
2. What is $\mathcal{P}_0$? What is $\mathcal{P}_{99}$?
3. Show that every $\mathcal{G}$-computable function has *infinitely many* gn's, i.e. $\forall a\ \exists$ *infinitely many* b : $\varphi_a = \varphi_b$.

8 The Church-Turing Thesis

The *Church-Turing Thesis (CT)*, formulated in terms of $\mathcal{G}$-computability, states that *any function which is computable by any algorithm whatsoever, is computable by a $\mathcal{G}$-program.* This thesis was first formulated in the 1930's, independently by Church, using the formalism of the λ-calculus, and Turing, using the formalism of Turing machines.

Although CT cannot be *mathematically proven* since it uses the non-mathematical notion of "*algorithm*", its acceptance is based on three arguments. Firstly, there is the philosophical analysis of the notion of "algorithm", as done by Turing. Secondly, many attempted formalisms of the notion of "algorithm" have been found to be equivalent, for example: Turing machine computability, λ-computability, $\mathcal{G}$-computability, Pascal-computability, etc. Thirdly, no counterexample to CT has been found in over 50 years.

Clearly, by CT, $\mathcal{G}$-COMP $=$ EFF. Similarly, we can formulate a *relativised version of CT (Rel-CT)*, which says that $\mathcal{G}\text{-COMP}(\vec{g}) = \text{EFF}(\vec{g})$.

The collection [5] contains many of the famous pioneering papers on computability theory, including those of Church and Turing in which their respective versions of CT were first formulated and justified.

NOTE: Any theorem which requires CT in its proof will be marked with the superscript 'CT', and any proof which uses CT (even if not required) will also be so marked.

9 The Halting Problem; The Universal Function Theorem

9.1 Decidability

Let B and C be n-ary relations. We say that B is

- *primitive recursive (PR)* iff its characteristic function χ_B is;
- *$\mathcal{G}$-computable* or *recursive* iff χ_B is $\mathcal{G}$-computable;
- *decidable* or *effective* or *algorithmic* iff χ_B is.

Thus, B is *decidable* if there is an *algorithm* to test for membership of B. Similarly we can define *relativised* versions of the above notions for relations (i.e. *primitive recursive in $\vec{g}$*, *recursive in $\vec{g}$* and *decidable in $\vec{g}$*, respectively).

Theorem 9.1 *$B \cup C$, $B \cap C \in PR(B, C)$, and $\bar{B} \in PR(B)$. Hence if $B, C \in PR$, then so are $B \cup C$, $B \cap C$ and $\bar{B}$.*

Proof. Since $\chi_{B\cup C} = \chi_B \vee \chi_C$, $\chi_{B\cap C} = \chi_B \wedge \chi_C$, and $\chi_{\bar{B}} = \neg\chi_B$, the results follow from Theorem 6.2. □

Corollary 9.1 *$B \cup C$, $B \cap C$ and $\bar{B}$ are recursive in B, C. Hence if B, C are recursive, then so are $B \cup C$, $B \cap C$ and $\bar{B}$.*

Proof. By Corollary 5.4. □

NOTES:

1. Intuitively $B \cup C$ and $B \cap C$ are decidable in B, C, and $\bar{B}$ is decidable in B. Hence if B, C are decidable, then so are $B \cup C$, $B \cap C$ and $\bar{B}$.
2. Clearly, if B is recursive (in $\vec{g}$), then B is certainly decidable (in $\vec{g}$). By Rel-CT, also the converse is true, so that B is recursive (in $\vec{g}$) iff B is decidable ($\vec{g}$).

9.2 The halting problem

The *Halting Problem* is the relation

$$HP = \{(\mathcal{P}, x) | \mathcal{P} \text{ halts on } x\} \subseteq \mathcal{G}\text{-PROG} \times \mathcal{N}.$$

We say that the Halting Problem is *decidable* or *(effectively) solvable* if the above relation is decidable; in other words, if there is an *algorithm* which, when given a *$\mathcal{G}$-program* $\mathcal{P}$ and an *input* x, determines whether $\mathcal{P}$ eventually halts on x. The obvious question now is: Is *HP* decidable? In this section we answer the question using CT and the Gödel numbering of $\mathcal{G}$-PROG.

Let $\boldsymbol{Halt}(y, x)$ be the characteristic predicate of *HP*, i.e.

$$\boldsymbol{Halt}(y, x) = \begin{cases} 1 & \text{if } \mathcal{P}_y \text{ halts on } x \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 9.2 *$\boldsymbol{Halt}$ is not $\mathcal{G}$-computable.*

Proof. Suppose it is. Then there exists a macro for it:

$$\boxed{\boldsymbol{Halt}(\mathrm{V},\mathrm{U})}.$$

Consider the program $\mathcal{P}$:

$$\boxed{[A]\ \text{if } \boldsymbol{Halt}(X,X) \text{ goto } A}.$$

$$\Psi_{\mathcal{P}}(x) \simeq \begin{cases} \uparrow & \text{if } \boldsymbol{Halt}(x,x) \\ 0 & \text{otherwise.} \end{cases}$$

So for all x,

$$\Psi_{\mathcal{P}}(x)\downarrow \iff \neg \boldsymbol{Halt}(x,x). \tag{11}$$

Letting $p = \#(\mathcal{P})$, (11) yields, for all x,

$$\boldsymbol{Halt}(x,p) \iff \neg \boldsymbol{Halt}(x,x).$$

Finally, putting $x = p$, we obtain

$$\boldsymbol{Halt}(p,p) \iff \neg \boldsymbol{Halt}(p,p),$$

a contradiction. □
Note the use of *diagonalisation* or *self-application* in the proof above.

We now use CT to show the *unsolvability* or *undecidability* of *HP*.

Theorem^{CT} 9.3 *There is* no algorithm *which, when given a $\mathcal{G}$-program $\mathcal{P}$ and a number x, will determine if $\mathcal{P}$ halts on input x.*

Proof: Suppose there is such an algorithm. Then there is an algorithm which, given any y and x, determines if program $\mathcal{P}_y$ halts on input x. Hence by CT there is a $\mathcal{G}$-program which does the same, a contradiction to Theorem 9.2. □

EXERCISE:
(Another version of the unsolvability of HP)
Show that the *diagonal* set below is *not* decidable:

$$\{x | \boldsymbol{Halt}(x,x)\} = \{x | \phi_x(x)\downarrow\}.$$

9.3 The universal $\mathcal{G}$-program; UFT

Reiterating, we have a method (GN) for uniquely and effectively associating $\mathcal{G}$-programs with numbers. In this way we can code $\mathcal{G}$-programs so as to use them essentially as inputs to other $\mathcal{G}$-programs, or even to themselves. In the previous subsection we used this technique and CT to show that there is *no* algorithm by which we can determine whether a program $\mathcal{P}$ halts on an input x. In this section we use the Gödel numbering to prove another important but positive result.

Let $\varphi_y^{(n)}$ denote the n-ary function computed by program $\mathcal{P}_y$. Then

$$\varphi_0^{(n)}, \varphi_1^{(n)}, \varphi_2^{(n)}, \cdots$$

is an enumeration of $\mathcal{G}$-COMP$^{(n)}$, and y is the *gn* or *index* of $\varphi_y^{(n)}$. We define the ($(n+1)$-ary) *universal function* $\Phi^{(n)}$ for $\mathcal{G}$-COMP$^{(n)}$ by:

$$\Phi^{(n)}(x_1,\cdots,x_n,y) \simeq \varphi_y^{(n)}(x_1,\cdots,x_n).$$

NOTE: We often drop the superscript '(n)' from Φ and φ when $n = 1$.

The following is the *universal function theorem (UFT) for* $\mathcal{G}$-COMP:

Theorem 9.4 $\Phi^{(n)} \in \mathcal{G}$*-COMP*$^{(n+1)}$. *In fact, there is a* universal program $\mathcal{U}_n$ *for $\mathcal{G}$-COMP$^{(n)}$ which computes $\Phi^{(n)}$. That is, $\Psi_{\mathcal{U}_n}^{(n+1)} = \Phi^{(n)}$.*

Proof 1 (using CT): Consider the following algorithm:

"With inputs $x_1,\cdots,x_n,y$:
construct the program $\mathcal{P}_y$;
apply it to inputs $x_1,\cdots,x_n$."

This provides an effective method for computing $\Phi^{(n)}(\vec{x},y)$ for any $\vec{x},y$. Hence by CT, $\Phi^{(n)}$ is $\mathcal{G}$-computable.
Proof 2 (not using CT): We will actually *construct* $\mathcal{U}_n$, following [1], §4.3. First we make some general remarks on the construction of the program.

It will be necessary to code not only programs, but also *states* by numbers. For example, if $\boldsymbol{dom}(\sigma) = \{Y, X_1, X_2, Z_1\}$, and $\sigma(Y) = 0$, $\sigma(X_1) = 2$, $\sigma(X_2) = 3$, $\sigma(Z_1) = 1$ (say), then $\#(\sigma) = [0,2,1,3] = p_1^0 \cdot p_2^2 \cdot p_3^1 \cdot p_4^3$. (Also for convenience we will use macros freely and ignore the rules for letters for variables and labels.)

For each $n > 0$, $\mathcal{U}_n$ *simulates* the computation of the program numbered X_{n+1} on the input variables $X_1,\cdots,X_n$. Suppose

$$\mathcal{P} = (I_1,\cdots,I_m).$$

Then

$$X_{n+1} = \#(\mathcal{P}) = [\#(I_1),\cdots,\#(I_m)] - 1.$$

The variables Z, S, and K store the sequence of instructions, the gn of the current state, and number of the instruction about to be executed, respectively. So

$$Z = [\#(I_1),\cdots,\#(I_m)],$$

S is initialised to $p_1^Y p_2^{X_1} p_3^{Z_1} p_4^{X_2} p_5^{Z_2}\cdots$, and K is initialised to 1. Note that the input variables X_1, $X_2,\cdots$ have *even* places in the effective enumeration of program variables (see §7.3), so the variables occupying the *odd* places assume the value 0 at the beginning of the program. Now, if at any stage

$$(Z)_K = \#(I_K) = \langle a, \langle b,c\rangle\rangle,$$

and we put

$$U = \boldsymbol{r}((Z)_K) = \langle b,c\rangle,$$

then, for the next instruction,

$$\begin{aligned} \boldsymbol{\ell}((Z)_K) &= a, && \text{is its label,} \\ \boldsymbol{\ell}(U) &= b, && \text{its type,} \\ \boldsymbol{r}(U) &= c, && \text{the variable involved.} \end{aligned}$$

The universal program $\mathcal{U}_n$ is then

$$
\boxed{\begin{array}{ll}
 & Z \leftarrow X_{n+1} + 1 \\
 & S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i} \\
 & K \leftarrow 1 \\
[C] & \text{if } K = \boldsymbol{Lt}(Z) + 1 \vee K = 0 \text{ goto } F \\
 & U \leftarrow \boldsymbol{r}((Z)_K) \\
 & P \leftarrow p_{\boldsymbol{r}(U)+1} \\
 & \text{if } \boldsymbol{\ell}(U) = 0 \text{ goto } N \\
 & \text{if } \boldsymbol{\ell}(U) = 1 \text{ goto } A \\
 & \text{if } \neg(P|S) \text{ goto } N \\
 & \text{if } \boldsymbol{\ell}(U) = 2 \text{ goto } M \\
 & K \leftarrow \boldsymbol{min}_{i \le \boldsymbol{Lt}(Z)}[\boldsymbol{\ell}((Z)_i) + 2 = \boldsymbol{\ell}(U)] \\
 & \text{goto } C \\
[M] & S \leftarrow \lfloor S/P \rfloor \\
 & \text{goto } N \\
[A] & S \leftarrow S \cdot P \\
[N] & K{+}{+} \\
 & \text{goto } C \\
[F] & Y \leftarrow (S)_1
\end{array}}
$$

□

9.4 The step-counter predicate

We consider the predicates

$\boldsymbol{stp}^{(n)}(\vec{x}, y, t)$
$\Leftrightarrow \mathcal{P}_y$, with inputs $\vec{x}$, halts in t or fewer steps
$\Leftrightarrow \exists$ a computation of $\mathcal{P}_y$, with inputs $\vec{x}$, of length $\le t + 1$.

Theorem 9.5 $\boldsymbol{stp}^{(n)} \in \mathcal{G}\text{-}COMP$.

Proof 1 (using CT): Use the algorithm

"Run $\mathcal{P}_y$ with inputs $\vec{x}$ up to t steps;
if it has halted,
then $\boldsymbol{stp}^{(n)}(\vec{x}, y, t) \leftarrow 1$
else $\boldsymbol{stp}^{(n)}(\vec{x}, y, t) \leftarrow 0$."

Proof 2 (not using CT): *Modify* the universal program to include a *step counter* Q, as follows. (Note that only two lines have been added (*), and one line changed (**)).

$$
\boxed{\begin{array}{lll}
 & Z \leftarrow X_{n+1} + 1 & \\
 & S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i} & \\
 & K \leftarrow 1 & \\
[C] & Q{+}{+} & (*) \\
 & \text{if } Q > X_{n+2} + 1 \text{ goto } E & (*) \\
 & \text{if } K = \boldsymbol{Lt}(Z) + 1 \vee K = 0 \text{ goto } F & \\
 & U \leftarrow \boldsymbol{r}((Z)_K) & \\
 & P \leftarrow p_{\boldsymbol{r}(U)+1} & \\
 & \text{if } \boldsymbol{\ell}(U) = 0 \text{ goto } N & \\
 & \text{if } \boldsymbol{\ell}(U) = 1 \text{ goto } A & \\
 & \text{if } \neg(P|S) \text{ goto } N & \\
 & \text{if } \boldsymbol{\ell}(U) = 2 \text{ goto } M & \\
 & K \leftarrow \boldsymbol{min}_{i \le \boldsymbol{Lt}(Z)}[\boldsymbol{\ell}((Z)_i) + 2 = \boldsymbol{\ell}(U)] & \\
 & \text{goto } C & \\
[M] & S \leftarrow \lfloor S/P \rfloor & \\
 & \text{goto } N & \\
[A] & S \leftarrow S \cdot P & \\
[N] & K{+}{+} & \\
 & \text{goto } C & \\
[F] & Y{+}{+} & (**)
\end{array}}
$$

□

NOTES:

1. The predicate

$$\boldsymbol{stp}_1^{(n)}(\vec{x}, y) \Leftrightarrow \text{“}\mathcal{P}_y, \text{ with inputs } \vec{x}, \text{ halts (at all)”}$$

is not $\mathcal{G}$-computable, since it is (essentially) HP.

2. Similarly, the predicate

$$\boldsymbol{stp}_2^{(n)}(\vec{x}, y) = \begin{cases} t + 1 & \text{if } \mathcal{P}_y \text{ halts on } \vec{x} \text{ in } t \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

is not $\mathcal{G}$-computable, since a $\mathcal{G}$-program for $\boldsymbol{stp}_2^{(n)}$ could easily provide a solution to HP.

3. We can prove a stronger result than Theorem 9.5:

Theorem 9.6 $\boldsymbol{stp}^{(n)} \in PR$.

Proof: Let

$$\boldsymbol{K}^{(n)}(\vec{x}, y, t)$$

be the *instruction counter* function, giving the number of the instruction to be read by $\mathcal{P}_y$, with inputs $\vec{x}$, at time $t + 1$, and

$$\boldsymbol{S}^{(n)}(\vec{x}, y, t)$$

giving the *state*, at time $t + 1$, when $\mathcal{P}_y$ has inputs $\vec{x}$.

We define $\boldsymbol{K}^{(n)}$ and $\boldsymbol{S}^{(n)}$ by primitive recursion on t. For the basis we let

$$\begin{array}{rcl} \boldsymbol{K}^{(n)}(\vec{x}, y, 0) & = & 1, \\ \text{and} \quad \boldsymbol{S}^{(n)}(\vec{x}, y, 0) & = & \prod_{i=1}^{n} p_{2i}^{x_i}. \end{array}$$

For the induction step we put

$$\begin{array}{ll} k = \boldsymbol{K}^{(n)}(\vec{x}, y, t), & s = \boldsymbol{S}^{(n)}(\vec{x}, y, t), \\ L = \boldsymbol{Lt}(y + 1), & u = \boldsymbol{r}((y + 1)_k), \\ b = \boldsymbol{\ell}(u), & c = \boldsymbol{r}(u), \\ p = p_{c+1}. & \end{array}$$

Then $\boldsymbol{K}^{(n)}(\vec{x},y,t+1) =$

$$\begin{cases} 0 & \text{if } k=0 \text{ or } k>L \\ k+1 & \text{if } (0\le k\le L)\wedge(b\le 2\vee p\not| s) \\ (\mu i<L)[\boldsymbol{\ell}(y+1)_i) = b\dot{-}2] & \text{otherwise,} \end{cases}$$

and $\boldsymbol{S}^{(n)}(\vec{x},y,t+1) =$

$$\begin{cases} s*p & \text{if } (0\le k\le L)\wedge(b=1) \\ \boldsymbol{quot}(s,p) & \text{if } (0\le k\le L)\wedge(b=2)\wedge p|s \\ s & \text{otherwise.} \end{cases}$$

By Theorem 7.2 $\boldsymbol{K}^{(n)}$, $\boldsymbol{S}^{(n)} \in PR$. Finally,

$$\boldsymbol{stp}^{(n)}(\vec{x},y,t) \Leftrightarrow \neg[0<\boldsymbol{K}^{(n)}(\vec{x},y,t)\le \boldsymbol{Lt}(y+1)].$$

□

We conclude this section by answering some of the questions concerning the properness of the "$\subseteq$" inclusions in the diagrams in §5. In particular, $\mathcal{G}$-COMP=EFF, by CT, and $\mathcal{G}$-COMP $\subset$ FN, since $\mathcal{G}$-COMP is *countable* $(\varphi_0,\varphi_1,\varphi_2,\cdots)$, and FN is *uncountable* by Cantor's theorem (Theorem 2.3*(a)*).

NOTE: By re-proving Cantor's Theorem in the present context, we can produce a *non-computable total function* f as follows. Define

$$f(n) = \begin{cases} \varphi_n(n)+1 & \text{if } \varphi_n(n)\downarrow \\ 0 & \text{if } \varphi_n(n)\uparrow. \end{cases}$$

Then $f \notin \mathcal{G}$-COMP, since (as we can easily see) for all n $f(n)\neq\varphi_n(n)$. (So f is a *witness* that $\mathcal{G}$-COMP $\subset$ FN.) Intuitively, f is not computable because the above definition by cases is *not effective*, owing to the undecidability of HP. Note the use of *diagonalisation* again here!

Now,

$$\begin{array}{ccccccc} & & \mathcal{G}\text{-COMP} & = & \text{EFF} & \subset & \text{FN} \\ & & \cup & & \cup & & \cup \\ \text{PR} & \subseteq & \mathcal{G}\text{-TCOMP} & = & \text{TEFF} & \subset & \text{TFN} \end{array}$$

and, using Rel-CT,

$$\begin{array}{ccccccc} \text{PR}(\vec{g}) & \subseteq & \mathcal{G}\text{-COMP}(\vec{g}) & = & \text{EFF}(\vec{g}) & \subset & \text{FN} \\ \cup & & \cup & & \cup & & \cup \\ \text{TPR}(\vec{g}) & \subseteq & \mathcal{G}\text{-TCOMP}(\vec{g}) & = & \text{TEFF}(\vec{g}) & \subset & \text{TFN} \end{array}$$

10 Recursive Enumerability

10.1 Recursively enumerable relations

Let B be an n-ary relation on $\mathcal{N}$. We say that B is

- *recursively enumerable (r.e.)* or $\mathcal{G}$*-semicomputable* iff B is the domain of some $\mathcal{G}$-computable function, i.e. there exists a $\mathcal{G}$-computable function g such that $B = \boldsymbol{dom}(g) = \{\vec{x} | g(\vec{x})\downarrow\}$; and
- *semi-decidable* or *semi-effective* iff there is an algorithm which gives *positive information* (only) on membership of B, i.e. with input $\vec{x}$, the algorithm halts iff $\vec{x}\in B$.

NOTES:

1. By CT, B is r.e. iff B is semi-decidable.
2. If B is decidable, then B is certainly semi-decidable, since an algorithm which decides B can easily be modified to one which gives positive information only on B. (However, the converse is not true, as we will see!) The analogous result for $\mathcal{G}$-computable B is:

Theorem 10.1 *If B is recursive, then B is r.e.*

Proof: Since χ_B is $\mathcal{G}$-computable, there exists a macro which computes it. The program

$$\boxed{[A]\ \mathsf{if}\ \chi_B(X_1,\cdots,X_n)=0\ \mathsf{goto}\ A}$$

halts only on input $\vec{x}\in B$. □

Theorem 10.2 *B is recursive iff B and $\bar{B}$ are r.e.*

Proof: (⇒:) Suppose B is recursive. By Theorem 9.1, $\bar{B}$ is recursive, and the result follows from Theorem 10.1.

(⇐:) Suppose B and $\bar{B}$ are r.e. Say

$$\begin{array}{rl} & B = \boldsymbol{dom}(g),\ g \text{ computed by program } \mathcal{P}_p, \\ \text{and} & \bar{B} = \boldsymbol{dom}(h),\ h \text{ computed by program } \mathcal{P}_q. \end{array}$$

Intuitively, on any input $\vec{x}$, we *dovetail* executions of $\mathcal{P}_p$ and $\mathcal{P}_q$ until one of them halts. Note that, by Theorem 9.5, there is a macro for $\boldsymbol{stp}^{(n)}$. So the program

```
[A]  if stp^(n)(X⃗, p̄, T) goto C
     if stp^(n)(X⃗, q̄, T) goto E
     T++
     goto A
[C]  Y++
```

computes χ_B. □

Theorem 10.3 *If B, C are r.e., then so are $B\cap C$ and $B\cup C$.*

Proof: Suppose

$$\begin{array}{rl} & B = \boldsymbol{dom}(g),\ g \text{ computed by program } \mathcal{P}_p, \\ \text{and} & \bar{B} = \boldsymbol{dom}(h),\ h \text{ computed by program } \mathcal{P}_q. \end{array}$$

The program

```
Y ← g(X⃗)
Y ← h(X⃗)
```

halts for inputs in $\boldsymbol{dom}(g)\cap\boldsymbol{dom}(h) = B\cap C$.

On the other hand, dovetailing $\mathcal{P}_p$ and $\mathcal{P}_q$, the program

```
[A]  if stp^(n)(X⃗, p̄, T) goto E
     if stp^(n)(X⃗, q̄, T) goto E
     T++
     goto A
```

halts for inputs in $\boldsymbol{dom}(g) \cup \boldsymbol{dom}(h) = B \cup C$. $\square$

Intuitively, if B and C are semi-decidable, then so are $B \cap C$, and $B \cup C$.

Let REC and RE denote the classes of recursive sets and r.e. sets, respectively. Then, clearly,

$$\boxed{\text{PR} \subseteq \text{REC} \subseteq \text{RE} \subseteq \wp(\mathcal{N})}$$

We devote the rest of the section to the questions concerning the properness of the above "$\subseteq$" inclusions (except for the leftmost one, which will be answered later — §14, Exercise 3).

By Corollary 9.1, REC is closed under $\cup$, $\cap$ and $^-$ and RE is closed under $\cup$ and $\cap$. The obvious question now is: Is RE closed under $^-$? The answer to this question also resolves the question concerning the second "$\subseteq$" inclusion.

Let $W_n = \boldsymbol{dom}(\varphi_n)$. So for all x,

$$x \in W_n \Longleftrightarrow \varphi_n(x) \downarrow,$$

yielding an *effective enumeration* of RE:

$$W_0, W_1, W_2, \cdots$$

Now let $K = \{x | x \in W_x\}$. Then

$$x \in K \Longleftrightarrow x \in W_x \Longleftrightarrow \varphi_x(x) \downarrow . \tag{12}$$

Theorem 10.4 *K is r.e., but* not *recursive.*

Proof: K is the *domain* of the function $\lambda x \cdot \Phi(x, x)$, which, by Theorem 9.4, is $\mathcal{G}$-computable. So K is r.e. Suppose K is recursive. Then, by Theorem 10.2, $\bar{K}$ is r.e. Therefore for some n,

$$\bar{K} = W_n. \tag{13}$$

So for all x,

$$x \in W_n \overset{(13)}{\Longleftrightarrow} x \in \bar{K} \overset{(12)}{\Longleftrightarrow} x \notin W_x.$$

Putting $x = n$,

$$n \in W_n \Longleftrightarrow n \notin W_n,$$

a contradiction. $\square$

Corollary 10.1 *$\bar{K}$ is not r.e.*

Proof:

$$\begin{array}{rcll} \bar{K} \text{ r.e.} & \Rightarrow & K, \bar{K} \text{ r.e.} & \text{(Theorem 10.4)} \\ & \Rightarrow & K \text{ recursive} & \text{(Theorem 10.2).} \end{array}$$

This contradicts Theorem 10.2. $\square$

NOTES:

1. Note again the use of diagonalisation (or self-reference) in the proof of Theorem 10.4.
2. The non-recursiveness of K is just another formulation of the unsolvalility of HP (see §9.2, Exercise).
3. REC $\subset$ RE by Theorem 10.4, with witness K.
4. Similarly, RE $\subset \wp(N)$, by Corollary 10.1, with witness $\bar{K}$.
5. Alternatively, we can argue that RE $\subset \wp(\mathcal{N})$ because RE is *countable* by the enumeration $W_0, W_1, \cdots$ whereas $\wp(\mathcal{N})$ is *uncountable* by Cantor's theorem (Theorem 2.3(b)). Hence we have

$$\boxed{\text{PR} \subseteq \text{REC} \subset \text{RE} \subset \wp(\mathcal{N})}$$

EXERCISE: By re-proving Cantor's theorem in the present context, produce a witness that RE $\subset \wp(\mathcal{N})$. What is the connection between this witness and the one in Note 4?

10.2 Characterisation of recursively enumerable sets using CT

Although the theorems in this section do not depend on CT, we will give proofs using CT for simplicity (following [2]).

Theorem 10.5 *If f is total $\mathcal{G}$-computable, then $\boldsymbol{ran}(f)$ is r.e.*

Proof^{CT}: Suppose that f is total computable. The following *algorithm halts only* on inputs in $\boldsymbol{ran}(f)$:

> "With input x:
> compute (in turn) $f(0), f(1), f(2), \cdots$
>   until you find an i with $f(i) = x$;
> then halt."

By CT there is a $\mathcal{G}$-program corresponding to this algorithm. $\square$

Theorem 10.6 *If f is $\mathcal{G}$-computable, then $\boldsymbol{ran}(f)$ is r.e.*

Proof^{CT}: By *modifying* the algorithm in the proof of Theorem 10.5 as follows:

> "With input x:
> generate $\boldsymbol{ran}(f)$ by dovetailing (interleaving),
>   i.e. in stages:
> at *stage* n:
>   do n steps in the computation of
> $f(0), f(1), f(2), \cdots, f(n-1)$;
>   halt when you find an i with $f(i) = x$."

Again, by CT there is a $\mathcal{G}$-program corresponding to this algorithm. $\square$

Theorem 10.7 *If f is total $\mathcal{G}$-computable and strictly increasing, then $\boldsymbol{ran}(f)$ is recursive.*

Proof^{CT}: By *modifying* the algorithm in the proof of Theorem 10.5 as follows:

"With input x:
compute (in turn) $f(0), f(1), f(2), \cdots$
until you find an i such that $f(i) \geq x$;
if $f(i) = x$: output 1;
if $f(i) > x$: output 0."

□

The next two theorems can be considered a converse to Theorem 10.5.

Theorem 10.8 *If B is r.e. and $B \neq \emptyset$, then there exists a* total *$\mathcal{G}$-computable function f such that $B = \boldsymbol{ran}(f)$.*

Proof ^{CT}: Let g be $\mathcal{G}$-computable with $\boldsymbol{dom}(g) = B$. The following algorithm computes a total function f with $\boldsymbol{dom}(f) = B$:

"With input x:
generate list of elements of B by dovetailing:
at *stage* n:
do n steps in the computation of
$g(0), g(1), \cdots, g(n-1)$;
for all $i < n$ such that $g(i) \downarrow$ in $\leq n$ steps,
add i to list;
[Note: List is *infinite* (even if B is finite),
since it has *repetitions*.]
output element number x in the list."

□

Theorem 10.9 *If B is r.e. and* infinite, *then there exists a* total 1-1 *$\mathcal{G}$-computable function f such that $B = \boldsymbol{ran}(f)$.*

Proof ^{CT}: EXERCISE. □

By combining the above results, we get:

Theorem 10.10 *(a) Suppose $B \neq \emptyset$. Then B is r.e. iff B is the range of a* total *$\mathcal{G}$-computable function.*
(b) B is r.e. iff B is the range of a $\mathcal{G}$-computable function.

Proof: *(a)* From Theorems 10.5 and 10.8.
(b) From Theorems 10.6 and 10.8, and since $\emptyset$ is r.e., being the domain and the range of $\lambda x \cdot \uparrow$. □

NOTE: This theorem gives the justification for the terminology "recursively enumerable". (Compare Theorem 2.2 and Notes 1 and 2 following it.)

EXERCISES:

1. Prove Theorem 10.9.
2. Prove: Suppose $B \neq \emptyset$. Then B is r.e. iff B is the range of a *1-1* $\mathcal{G}$-computable function.

11 Enumerability of Total Computable Functions

In §9.3 we defined an ($n+1$-ary) ($\mathcal{G}$-computable) universal function for $\mathcal{G}$-COMP$^{(n)}$ in terms of an enumeration $\varphi_0^{(n)}, \varphi_1^{(n)}, \cdots$ of $\mathcal{G}$-COMP$^{(n)}$. In this section we show that this *cannot* be done for $\mathcal{G}$-TCOMP$^{(n)}$ (even when $n = 1$). It is for this reason that we consider (*partial*) $\mathcal{G}$-computable functions as *more fundamental* than *total* $\mathcal{G}$-computable functions.

For any binary function F and $n \in \mathcal{N}$, let

$$F_n =_{\mathrm{df}} \lambda x \cdot F(n, x).$$

We now investigate whether the UFT holds for $\mathcal{G}$-TCOMP$^{(1)}$, i.e. whether there is a *universal function* $F \in \mathcal{G}$-TCOMP$^{(2)}$, for which the sequence

$$F_0, F_1, F_2, \cdots \tag{14}$$

enumerates all of $\mathcal{G}$-TCOMP$^{(1)}$. (Note that there *is* a UFT for $\mathcal{G}$-COMP, by Theorem 9.4.)

Theorem 11.1 *If $F \in \mathcal{G}$-TCOMP$^{(2)}$, then*
(a) for all n, $F_n \in \mathcal{G}$-TCOMP$^{(1)}$, but
(b) we can find a function $h \in \mathcal{G}$-TCOMP$^{(1)}$ which is outside *the enumeration (14), i.e. for all n, $F_n \neq h$.*

Proof: *(a)* Clear.
(b) Define $h(x) = F(x, x) + 1$. □

Corollary 11.1 *There exists no UFT for $\mathcal{G}$-TCOMP.*

NOTES:

1. Note the use of diagonalisation in the proof of Theorem 11.1.
2. By CT this theorem says: Given any effective enumeration of some class of total computable functions, we can "diagonalise out" to obtain a total computable function outside the class!
3. Thus, although $\mathcal{G}$-TCOMP is *enumerable* by classical reasoning (being a subset of the *enumerable* set $\mathcal{G}$-COMP), it is (by CT) *not effectively enumerable*! (See also Exercise 3 below.)
4. Why can the method of "diagonalising out" not be used to contradict the UFT for $\mathcal{G}$-COMP? Because the definition $h(x) \simeq \varphi_x(x)+1$ does *not* imply that for all y, $\varphi_y \neq h$. For suppose $h = \varphi_n$. Then the equation

$$\varphi_n(n) \simeq h(n) \simeq \varphi_n(n) + 1$$

just means that $\varphi_n(n) \uparrow$.

EXERCISES:

1. Let $\mathcal{G}$-COMP-PRED be the class of $\mathcal{G}$-computable predicates, i.e. the *total* functions $P : \mathcal{N} \to \mathbf{2}$. Is there a UFT for $\mathcal{G}$-COMP-PRED?
2. (a) Let PR-DERIV be the set of all PR-derivations. Show how (by Gödel numbering or otherwise) to give an *effective enumeration* of PR-DERIV, and hence (as a sublist) an effective enumeration of the set PR-DERIV$^{(1)}$ of PR-derivations of unary functions. This induces an *effective enumeration* $f_0, f_1, f_2, \cdots$ of PR$^{(1)}$.
   (b) Let F be the binary *universal function* for PR$^{(1)}$ under the enumeration in (a), i.e. for all m and n, $F(m, n) = f_m(n)$. Clearly F is effective, and hence in $\mathcal{G}$-TCOMP, by CT. But is F *primitive recursive*? More generally, is there a UFT for PR at all?

3. Show that the set $\{y|\varphi_y \text{ is total}\}$ is not r.e. (Hint: Otherwise there would be a UFT for $\mathcal{G}$-TCOMP).

12 μ-Primitive Recursive Functions

The main result of this section is the equivalence of the class of *μ-primitive recursive functions* and the class of $\mathcal{G}$-computable functions.

We inductively define the class μPR of μ-primitive recursive functions. This is the least class of functions which

1. contains the *initial functions* $\boldsymbol{S}$, $\boldsymbol{Z}$ and $\boldsymbol{U}_i^n$;
2. is closed under *composition* and *primitive recursion*; and
3. is closed under the (unbounded) *μ-operator*, i.e. if $g \in \mu\mathrm{PR}^{(n+1)}$ and

$$f(\vec{x}) \simeq \mu y[g(\vec{x}, y) \simeq 0], \tag{15}$$

then $f \in \mu\mathrm{PR}^{(n)}$;

where $\mu\mathrm{PR}^{(n)}$ is the class of μPR functions of arity n. (The μ-operator was introduced in §6.5.)

NOTES:

1. Without clause (3), the definition yields the class PR. The effect of clause (3) is to include *partial functions*. For example, if $g = \lambda\vec{x}, y \cdot 1$, then f is the totally undefined function.
2. Note the *constructive* or *computational* meaning of μ: Suppose, for example, that in (15), for some given $\vec{x}$,

$$g(\vec{x}, 0) = 1,\ g(\vec{x}, 1) = 1,\ g(\vec{x}, 2)\uparrow,\ g(\vec{x}, 3) = 0.$$

Then $f(\vec{x})\uparrow$, since in the computation of $g(\vec{x}, y)$ for $y = 0, 1, 2, \cdots$, we never reach $y = 3$.

3. Each μPR function has an associated μPR-*derivation*, which is similar to a PR-derivation, but with the extra possibility of obtaining a function from a previous function in the derivation by applying the μ-operator.

Proposition 12.1 *In (15), $f \in \mathcal{G}$-COMP(g). Hence if $g \in \mathcal{G}$-COMP, then so is f. In other words, $\mathcal{G}$-COMP is* closed *under the μ-operator.*

Proof: The following $\mathcal{G}$-program with an oracle (or macro) for g, computes f:

```
[A]  Z ← g(X⃗, Y)
     if Z = 0 goto E
     Y++
     goto A
```

□

Next we give two celebrated results, essentially due to Kleene (using a different formalism and terminology — see [3], Part III).

Theorem 12.1 (Normal Form Theorem for $\mathcal{G}$-COMP) *For all n, there exists a PR $(n+2)$-ary predicate $\boldsymbol{T}^{(n)}$, and a PR function $\boldsymbol{U}$, such that for all e and $\vec{x}$,*

$$\varphi_e^{(n)}(\vec{x}) \simeq \boldsymbol{U}(\mu y \boldsymbol{T}^{(n)}(e, \vec{x}, y)). \tag{16}$$

Proof: A *computation number* (gn of a computation) has the form

$$y = p_1^{e_1} p_2^{e_2} \cdots p_\ell^{e_\ell}$$

where for $1 \le t \le \ell$, e_t is a *snapshot* at time t, i.e.

$$e_t = \langle k_t, s_t \rangle$$

$$\begin{aligned} \text{where} \quad & k_t = \boldsymbol{K}^{(n)}(e, \vec{x}, t \dot{-} 1), \\ \text{and} \quad & s_t = \boldsymbol{S}^{(n)}(e, \vec{x}, t \dot{-} 1), \end{aligned}$$

as defined in §9.4.

We define $\boldsymbol{T}^{(n)}(e, \vec{x}, y)$ as the predicate

"y is the computation number when $\mathcal{P}_e$ has input $\vec{x}$:" In symbols, putting $L_e = \boldsymbol{Lt}(e+1)$ and $L_y = \boldsymbol{Lt}(y)$:

$$\begin{aligned} &(\forall t \le L_y)[(y)_{t+1} = \langle \boldsymbol{K}^{(n)}(e, \vec{x}, t), \boldsymbol{S}^{(n)}(e, \vec{x}, t) \rangle] \\ &\wedge (\forall t < L_y)[(1 \le \boldsymbol{K}^{(n)}(e, \vec{x}, t) \le L_e) \\ &\wedge \neg(1 \le \boldsymbol{K}^{(n)}(e, \vec{x}, L_y) \le L_e)]. \end{aligned}$$

We define $\boldsymbol{U}(y)$ as the value of the output variable at the final state in computation y. In symbols:

$$\boldsymbol{U}(y) = (\boldsymbol{r}((y)_{\boldsymbol{Lt}(y)}))_1.$$

It is clear that $\boldsymbol{T}^{(n)}$, $\boldsymbol{U} \in$ PR, and that (16) holds. □

Theorem 12.2 *$\mu PR = \mathcal{G}$-COMP.*

Proof: We will show that

$$f \text{ is } \mu\text{PR} \Leftrightarrow f \text{ is } \mathcal{G}\text{-computable.}$$

($\Rightarrow$:) This is obvious from CT. However, a proof without CT exists, and serves as confirmation for CT. We will effectively associate, with each μPR-*derivation* of a function f, a $\mathcal{G}$-program for f by *CV induction* on the *length of the derivation*. (Compare proof of Lemma 5.2.) If the last step in the derivation is an *initial function*, or formed by *composition* or *primitive recursion*, use Proposition 5.2. If the last step is an application of the *μ-operator* (the new case), use Proposition 12.1.

($\Leftarrow$:) By Theorem 12.1. □

NOTES:

4. As with PR-derivations (see §11, Exercise 2) we can give an *effective enumeration* of the set μPR-DERIV of μPR-derivations, and hence an effective enumeration of μPR. The proof of Theorem 12.2 actually gives *effective maps* between μPR-DERIV and $\mathcal{G}$-PROG (PR in their gn's, in fact), thus providing us with a *second* effective enumeration of $\mathcal{G}$-COMP (=μPR). (The first was induced by the Gödel numbering of $\mathcal{G}$-PROG — see §7.3.)
5. Theorems 12.1 and 12.2 together show that any μPR (or equivalently, $\mathcal{G}$-computable) function has a μPR-derivation in which the μ-operator is used only once!

6. There is also a *relativised* notion of μ-primitive recursiveness, and a relativised version of Theorem 12.2:
$$\mu\mathrm{PR}(\vec{g}) = \mathcal{G}\text{-COMP}(\vec{g}). \tag{17}$$

EXERCISE: Define the class $\mu\mathrm{PR}(\vec{g})$, and outline a proof for (17).

13 'loop' Programs

13.1 Definition

Up to now our development of computability theory was done in terms of the $\mathcal{G}$ programming language. We have asserted (in §8) the equivalence of this notion with many other notions of computability, and proved (in §12) its equivalence to μ-primitive recursiveness. In this section, and the next, we turn to other simple programming languages, and investigate whether the corresponding notions of computability are equivalent to $\mathcal{G}$-computability or not.

First we consider the *programming language* $\mathcal{L}$ (for "loop"), with the *instructions*

$$\begin{array}{l} V \leftarrow 0 \\ V \leftarrow W \\ V{+}{+} \\ \left\{\begin{array}{l} \mathsf{loop}\ V \\ \quad\vdots \\ \mathsf{end} \end{array}\right. \\ \mathsf{skip} \end{array}$$

and define an $\mathcal{L}$-program as a finite sequence of instructions such that the 'loop' and 'end' instructions occur in matching pairs.

Comparing $\mathcal{L}$ with $\mathcal{G}$, we find that

- '$V \leftarrow W$' and '$V \leftarrow 0$' are primitive instructions in $\mathcal{L}$, but not in $\mathcal{G}$ (not an important difference);
- '$V--$' is primitive in $\mathcal{G}$ but not in $\mathcal{L}$ (also not important);
- $\mathcal{L}$ has *loops* instead of *labels* and *branches* (this is the important difference!).

To complete our description of the $\mathcal{L}$-language, we give the precise meaning of the *loop segment*

$$\left\{\begin{array}{l} \mathsf{loop}\ V \\ \quad \mathcal{P} \quad \}\ \text{block} \\ \mathsf{end} \end{array}\right.$$

Suppose that, when we read the 'loop' instruction, the value of V is v. Then the block $\mathcal{P}$ of instructions is executed v times — even if the value of V is changed in $\mathcal{P}$. This means that $\mathcal{L}$-programs always *halt*!

NOTE: The convention with respect to *input, output* and *auxiliary* variables is the same as before; i.e. all variables other than the *input* variables are initialised to 0.

EXAMPLES: $\mathcal{L}$-programs for *addition* and *multiplication*, respectively, are

$$\boxed{\begin{array}{l} Y \leftarrow X_1 \\ \mathsf{loop}\ X_2 \\ \quad Y{+}{+} \\ \mathsf{end} \end{array}} \quad \text{and} \quad \boxed{\begin{array}{l} \mathsf{loop}\ X_1 \\ \quad \mathsf{loop}\ X_2 \\ \quad\quad Y{+}{+} \\ \quad \mathsf{end} \\ \mathsf{end} \end{array}}\,.$$

13.2 Relationship to other notions of computability

Let $\mathcal{L}$-COMP be the class of functions computable by $\mathcal{L}$-programs.

Proposition 13.1 *$\mathcal{L}$-COMP $\subseteq$ $\mathcal{G}$-TCOMP.*

Proof: Firstly, all $\mathcal{L}$-computable functions are total. Secondly, all $\mathcal{L}$-computable functions are $\mathcal{G}$-computable by the following translation $\mathcal{Q} \mapsto \mathcal{Q}'$ of $\mathcal{L}$-programs into $\mathcal{G}$-programs (by CV induction on the lengths of programs $\mathcal{Q}$): $\boxed{V{+}{+}}$ and $\boxed{\mathsf{skip}}$ are translated to themselves, and we have $\mathcal{G}$-macros for $\boxed{V \leftarrow 0}$ and $\boxed{V \leftarrow W}$. Finally,

$$\boxed{\begin{array}{l} \mathsf{loop}\ V \\ \quad \mathcal{Q} \\ \mathsf{end} \end{array}}$$

can be translated into

$$\boxed{\begin{array}{ll} & Z \leftarrow V \\ [A] & \mathsf{if}\ Z = 0\ \mathsf{goto}\ E \\ & \mathcal{Q}' \\ & Z-- \\ & \mathsf{goto}\ A \end{array}}$$

where Z is a *new* (auxiliary) variable. □

NOTE: We can easily define a *GN*, and hence an *effective enumeration*, of $\mathcal{L}$-programs:

$$\mathcal{Q}_0, \mathcal{Q}_1, \mathcal{Q}_2, \cdots$$

Let F_e be the unary function computed by $\mathcal{Q}_e$. Then

$$F_0, F_1, F_2, \cdots$$

is an *enumeration* of $\mathcal{L}$-COMP$^{(1)}$. Let

$$F(e, x) = F_e(x). \tag{18}$$

Then F is total $\mathcal{G}$-computable, by CT. Hence by Theorem 11.1,

$$\mathcal{L}\text{-COMP} \subset \mathcal{G}\text{-TCOMP} \tag{19}$$

with witness $\lambda x \cdot (F(x,x) + 1)$ (or F itself).

The rest of this section is devoted to showing that

$$\mathcal{L}\text{-COMP} = \mathrm{PR}.$$

Lemma 13.1 *PR $\subseteq$ $\mathcal{L}$-COMP.*

Proof: Suppose $f \in$ PR. We find an *$\mathcal{L}$-program* or *macro* for f by *CV induction* on the length of a PR-derivation for f. We must consider the following cases:

- The initial functions, i.e. the *zero*, *projection* and *successor* functions are computed by $\boxed{Y \leftarrow 0}$, $\boxed{Y \leftarrow X_i}$ and $\boxed{\begin{array}{l} Y \leftarrow X \\ Y{+}{+} \end{array}}$, respectively.
- The $\mathcal{G}$-program for *composition* in the proof of Theorem 4.2 is also an $\mathcal{L}$-program.
- To obtain an $\mathcal{L}$-program for *primitive recursion with parameters* we must modify the method for Theorem 4.4. Assuming $\mathcal{L}$-macros for g and h, f is computed by

$$\boxed{\begin{array}{l} Y \leftarrow g(X_1, \cdots, X_n) \\ \mathsf{loop}\ X_{n+1} \\ \quad Y \leftarrow h(X_1, \cdots, X_n, Z, Y) \\ \quad Z{+}{+} \\ \mathsf{end} \end{array}}$$

The case of primitive recursion without parameters is similar. □

In order to prove the converse of Lemma 13.1, we require certain definitions and intermediate results.

Let $\mathcal{L}_n$ be the class of $\mathcal{L}$-programs with loop-end pairs nested to the depth of *at most* n, and $\mathcal{L}_n$-COMP the class of functions computed by $\mathcal{L}_n$-programs.

EXAMPLE: The program for *addition* is in $\mathcal{L}_1$, and for *multiplication* is in $\mathcal{L}_2$ (see previous example).

These definitions suggest a *hierarchy* of *$\mathcal{L}$-programs*

$$\mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2 \subset \cdots, \ \mathcal{L} = \cup_{n=0}^{\infty} \mathcal{L}_n$$

and a *hierarchy* of *$\mathcal{L}$-computable functions*

$$\begin{array}{c} \mathcal{L}_0\text{-COMP} \subseteq \mathcal{L}_1\text{-COMP} \subseteq \mathcal{L}_2\text{-COMP} \subseteq \cdots, \\ \mathcal{L}\text{-COMP} = \cup_n \mathcal{L}_n\text{-COMP}. \end{array}$$

Let us assume for now that

- programs (or blocks) contain only auxiliary variable $Z_1, Z_2, \cdots$, and
- a block within a loop ('loop $V \cdots$ end') does not contain the *loop variable* V. There is no loss of generality, since

$$\boxed{\begin{array}{l} \mathsf{loop}\ V \\ \quad \mathcal{P} \\ \mathsf{end} \end{array}} \cong \boxed{\begin{array}{l} W \leftarrow V \\ \mathsf{loop}\ W \\ \quad \mathcal{P} \\ \mathsf{end} \end{array}}$$

where W is a new auxiliary variable (and '$\cong$' denotes semantic equivalence of programs).

Now consider a block $\mathcal{P}$ with $\boldsymbol{var}(\mathcal{P}) \subseteq \vec{Z} \equiv Z_1, \cdots, Z_n$. We think of $\mathcal{P}$ as *transforming* the values of $\vec{Z}$ by

$$\begin{array}{rl} & \vec{Z} \longleftarrow (f_1(\vec{Z}), \cdots, f_n(\vec{Z})) \\ \text{or} & \vec{Z} \longleftarrow \vec{f}(\vec{Z}), \end{array} \tag{20}$$

for certain n-ary functions $\vec{f} = f_1, \cdots, f_n$. We also say that $\mathcal{P}$ *defines the transformation (20) on* $\vec{Z}$. Consider now a loop segment

$$\mathcal{Q} \equiv \boxed{\begin{array}{l} \mathsf{loop}\ V \\ \quad \mathcal{P} \\ \mathsf{end} \end{array}}$$

with $V \not\equiv Z_1, \cdots, Z_n$. Then $\boldsymbol{var}(\mathcal{Q}) \subseteq \{\vec{Z}, V\}$, and $\mathcal{Q}$ transforms the values of these variables by

$$\begin{array}{l} \vec{Z} \leftarrow \vec{g}(\vec{Z}, V) \\ V \leftarrow V \end{array} \tag{21}$$

for certain $(n+1)$-ary functions $\vec{g} = g_1, \cdots, g_n$ (since, by assumption, the value of the loop variable V does not change with the execution of $\mathcal{Q}$). What is the relationship between $\vec{f}$ in (20) and $\vec{g}$ in (21)? Note that $g_i(\vec{z}, v)$ is the final value of z_i after v iterations of block $\mathcal{P}$, assuming that v is the initial value of V.

Lemma 13.2 *(With the above notation:)* $\vec{g} \in PR(\vec{f})$.

Proof: We have

$$\begin{array}{rcl} g_i(\vec{z}, 0) & = & z_i \\ g_i(\vec{z}, t+1) & = & f_i(g_1(\vec{z}, t), \cdots, g_n(\vec{z}, t)). \end{array}$$

So $\vec{g}$ is defined from $\vec{f}$ by *simultaneous primitive recursion*. The result follows from Theorem 7.2 (generalised to n functions). □

Lemma 13.3 *Suppose that $\mathcal{P}$ is an $\mathcal{L}$-program with $\boldsymbol{var}(\mathcal{P}) \subseteq \vec{Z} \equiv Z_1, \cdots, Z_n$, and that $\mathcal{P}$ defines the transformation $\vec{Z} \leftarrow \vec{f}(\vec{Z})$, with $\vec{f} = f_1, \cdots, f_n$. Then* $\vec{f} \in$ PR.

Proof: Since $\mathcal{P}$ is an $\mathcal{L}$-program, $\mathcal{P} \in \mathcal{L}_n$, for some n. We show that if $\mathcal{P} \in \mathcal{L}_n$ then $\vec{f} \in$ PR, by *induction* on n:

- **Basis:** $n = 0$. $\mathcal{P}$ has *no* loop-end pair, and consists only of the instructions

$$\begin{array}{l} Z_i \leftarrow 0, \\ Z_i \leftarrow Z_j, \\ Z_i{+}{+}. \end{array}$$

So we must have

$$\begin{array}{rrcl} & f_i(\vec{Z}) & = & Z_j + k, \\ \text{or} & f_i(\vec{Z}) & = & k, \end{array}$$

for $i = 1, \cdots, n$, some j and some k. Therefore $\vec{f} \in$ PR.

- **Induction step:** Suppose the result holds for $n = k$. Let $\mathcal{P} \in \mathcal{L}_{k+1}$. Then $\mathcal{P}$ is of the form

```
Q_0
loop V_1
  P_1
end
Q_1
loop V_2
  P_2
end
Q_2
  ⋮
Q_{r-1}
loop V_r
  P_r
end
Q_r
```

  where $\mathcal{Q}_i, \mathcal{P}_i \in \mathcal{L}_k$. By the *induction hypothesis*, the *transformations* defined by these are all in PR. By Lemma 13.2, the transformation defined by $\boxed{\begin{array}{l}\mathsf{loop}\ V_i\\ \quad\mathcal{P}_i\\ \mathsf{end}\end{array}}$ is in PR, and the result follows from the closure of PR under *composition*. □

We are now ready to prove the converse of Lemma 13.1:

Lemma 13.4 *$\mathcal{L}$-COMP $\subseteq$ PR.*

Proof: Suppose the k-ary function h is computable by the $\mathcal{L}$-program $\mathcal{P}$, containing the variables $Z_1, \cdots, Z_\ell, X_1, \cdots, X_k, Y$. Put

$$\mathcal{P} \equiv \mathcal{P}(Z_1, \cdots, Z_\ell, X_1, \cdots, X_k, Y).$$

Let

$$\mathcal{Q} \equiv \mathcal{P}(Z_1, \cdots, Z_\ell, Z_{\ell+1}, \cdots, Z_{\ell+k}, Z_{\ell+k+1})$$

and suppose $\mathcal{Q}$ defines a transformation

$$\vec{Z} \leftarrow \vec{f}(\vec{Z})$$

with $\vec{Z} \equiv Z_1, \cdots, Z_{\ell+k+1}$ and $\vec{f} = f_1, \cdots, f_{\ell+k+1}$. By Lemma 13.3, $\vec{f} \in$ PR. Also

$$h(x_1, \cdots, x_k) = f_{\ell+k+1}(\underbrace{0, \cdots, 0}_{\ell \text{ times}}, x_1, \cdots, x_k, 0)$$

Therefore $h \in$ PR. □

Finally,

Theorem 13.1 *$\mathcal{L}$-COMP = PR*

Proof: By Lemmas 13.1 and 13.4. □

NOTE: Again, there is a *relativised* notion of 'loop' computability, and a relativised version of Theorem 13.1:

$$\mathcal{L}\text{-COMP}(\vec{g}) = \text{PR}(\vec{g}) \tag{22}$$

Corollary 13.1 *PR $\subset \mathcal{G}$-TCOMP.*

Proof: By (19) and Theorem 13.1. □

13.3 Ackermann's function

As we have seen, the function F in (18) is $\mathcal{G}$-computable, but not PR. We conclude this section with a *more interesting* and "natural" witness that PR $\subset \mathcal{G}$-TCOMP. To set the stage, consider the hierarchy of PR definitions of well-known functions:

$$\begin{array}{rclrcl} x+0 &=& x, & x+\mathbf{S}y &=& \mathbf{S}(x+y)\\ x*0 &=& 0, & x*\mathbf{S}y &=& x+(x*y)\\ x\uparrow 0 &=& 1, & x\uparrow\mathbf{S}y &=& x*(x\uparrow y)\\ x\uparrow\uparrow 0 &=& 1, & x\uparrow\uparrow\mathbf{S}y &=& x\uparrow(x\uparrow\uparrow y)\\ &&&&\vdots&\end{array}$$

NOTE 1: The *hyperexponential*

$$x\uparrow\uparrow y = x^{\cdot^{\cdot^{\cdot^{x}}}} \Big\}\,(y \text{ times})$$

increases very rapidly with y.[1]

We systematise the above sequence of constructions by putting

$$f_1 = +,\ f_2 = *,\ f_3 = \uparrow,\ f_4 = \uparrow\uparrow, \cdots$$

and defining

$$\begin{cases} f_0(x,y) &=& \mathbf{S}y\\ f_{n+1}(x,0) &=& \begin{cases} x \text{ if } n=0\\ 0 \text{ if } n=1\\ 1 \text{ if } n>1\end{cases}\\ f_{n+1}(x,\mathbf{S}y) &=& f_n(x, f_{n+1}(x,y)).\end{cases}$$

NOTES:

2. For all n, $f_n \in$ PR (by induction on n).
3. It is also easy to see that $f_n \in \mathcal{L}_n$-COMP (again by induction on n).
4. However, we can show that $f_{n+1} \notin \mathcal{L}_n$-COMP, since it "increases too rapidly"! (See [1], Chapter 13, for a proof for a related hierarchy.)

Now let $\boldsymbol{A}(z,x,y) = f_z(x,y)$. This is (a version of) *Ackermann's function.*

NOTES:

5. The function $\boldsymbol{A}$ is defined by *double recursion* (on the first and third arguments):

$$\begin{cases} \boldsymbol{A}(0,x,y) &=& \boldsymbol{S}y\\ \boldsymbol{A}(\boldsymbol{S}z,x,0) &=& \begin{cases} x \text{ if } n=0\\ 0 \text{ if } n=1\\ 1 \text{ if } n>1\end{cases}\\ \boldsymbol{A}(\boldsymbol{S}z,x,\boldsymbol{S}y) &=& \boldsymbol{A}(z,x,\boldsymbol{A}(\boldsymbol{S}z,x,y)).\end{cases}$$

6. $\boldsymbol{A}$ is $\mathcal{G}$-computable (for example, by CT).
7. However, $\boldsymbol{A} \notin$ PR! For suppose

$$\boldsymbol{A} \in \text{PR} = \mathcal{L}\text{-COMP} = \cup_n \mathcal{L}_n\text{-COMP}$$

[1] For example, $3 \uparrow\uparrow 4$ is much larger than 10^{80}, Eddington's estimate of the number of electrons in the universe.

Then for some n, $\boldsymbol{A} \in \mathcal{L}_n$-COMP. So

$$f_{n+1} = \lambda x, y \cdot \boldsymbol{A}(n+1, x, y) \in \mathcal{L}_n\text{-COMP},$$

a contradiction to Note 4.

EXERCISES:

1. Define the class $\mathcal{L}$-COMP$(\vec{g})$, and outline a proof for (22).
2. **(Tail recursion)** Suppose f is defined from g and h by the equations

$$\begin{cases} f(x,0) & = & g(x) \\ f(x,n+1) & = & f(h(x,n),n). \end{cases}$$

Show that $f \in \mathcal{L}$-COMP(g,h) and (hence) $f \in$ PR(g,h). Note that in the "recursive call" (the expression on the right hand side of the second equation), f is on the "outside" — this is characteristic of tail recursion. Also the parameter *changes* (from x to $h(x,n)$), so that these equations (as they stand) do *not* form an instance of definition by primitive recursion.

14 'while' Programs

The third programming language that we consider, is the $\mathcal{W}$ programming language which is similar to $\mathcal{L}$, except that instead of the loop–end instruction, it has the instruction

```
while V ≠ 0 do
    ⋮
end.
```

We also need '$V--$' as a primitive instruction (for technical reasons). It is clear that, in contrast to $\mathcal{L}$-programs, $\mathcal{W}$-programs can diverge. It is therefore necessary to clarify the relationship between the function classes $\mathcal{W}$-COMP, $\mathcal{L}$-COMP and $\mathcal{G}$-COMP.

Lemma 14.1 *$\mathcal{L}$-COMP $\subseteq$ $\mathcal{W}$-COMP.*

Proof: $\mathcal{L}$-programs $\mathcal{P}$ can be translated into $\mathcal{W}$-programs $\mathcal{P}'$ by CV induction on the length of $\mathcal{P}$, using

loop V $\quad\mathcal{Q}$ end	$\mapsto$	$Z \leftarrow V$ while $Z \neq 0$ do $\quad\mathcal{Q}'$ $\quad Z--$ end

where Z is a new variable. □

Lemma 14.2 *$\mathcal{W}$-COMP $\subseteq$ $\mathcal{G}$-COMP.*

Proof: $\mathcal{W}$-programs $\mathcal{P}$ can be translated into $\mathcal{G}$-programs $\mathcal{P}'$, using

while $V \neq 0$ do $\quad\mathcal{Q}$ end	$\mapsto$	$[A]$ if $V = 0$ goto E $\quad\mathcal{Q}'$ $\quad$goto A	□

For the converse direction, we must show how to eliminate 'goto' instructions:

Lemma 14.3 *$\mathcal{G}$-COMP $\subseteq$ $\mathcal{W}$-COMP.*

Proof: A direct translation of $\mathcal{G}$-programs to $\mathcal{W}$-programs (by CV induction on the lengths of $\mathcal{G}$-programs) is very hard. Instead, we show that any $\mathcal{G}$-computable function is $\mathcal{W}$-computable, using the normal form theorem for $\mathcal{G}$-COMP (Theorem 12.1). Let $f \in \mathcal{G}$-COMP, say $f = \varphi_e^{(n)}$. Then

$$f(\vec{x}) \simeq \varphi_e^{(n)}(\vec{x}) \simeq \boldsymbol{U}(\mu y \boldsymbol{T}^{(n)}(e, \vec{x}, y)).$$

Let $\bar{\boldsymbol{T}}^{(n)} = \neg \boldsymbol{T}^{(n)}$. Since $\boldsymbol{T}^{(n)} \in$ PR, so are $\bar{\boldsymbol{T}}^{(n)}$ and $\boldsymbol{U}$. Therefore $\bar{\boldsymbol{T}}^{(n)}$ and $\boldsymbol{U}$ are $\mathcal{L}$-computable, and by Lemma 14.1, also $\mathcal{W}$-computable. So a $\mathcal{W}$-program for f is

$Z \leftarrow 0$ $V \leftarrow \bar{\boldsymbol{T}}^{(n)}(\bar{e}, \vec{X}, Z)$ while $V \neq 0$ do $\quad Z{+}{+}$ $\quad V \leftarrow \bar{\boldsymbol{T}}^{(n)}(\bar{e}, \vec{X}, Z)$ end $Y \leftarrow \boldsymbol{U}(Z)$

where Z and V are new variables. □

Corollary 14.1

$$\mathcal{W}\text{-COMP} = \mathcal{G}\text{-COMP}(= \mu\text{PR}).$$

PROOF: From Lemmas 14.2 and 14.3. □

NOTES:

1. This provides further confirmation for CT!
2. Again, there is a *relativised* notion of 'while' computability, and a relativised version of Corollary 14.1:

$$\mathcal{W}\text{-COMP}(\vec{g}) = \mathcal{G}\text{-COMP}(\vec{g}).$$

This brings us to our final display, in which all the questions about proper inclusions, raised in the previous pages, have been answered:

$$\begin{array}{ccccccccc} & & & & \text{COMP} & \overset{CT}{=} & \text{EFF} & \subset & \text{FN} \\ & & & & \cup & & \cup & & \cup \\ \text{PR} & = & \mathcal{L}\text{-COMP} & \subset & \text{TCOMP} & \overset{CT}{=} & \text{TEFF} & \subset & \text{TFN} \end{array}$$

where 'COMP' means *any one of* $\mathcal{G}$-COMP, $\mathcal{W}$-COMP and μPR, and 'TCOMP' means *any one of* $\mathcal{G}$-TCOMP, $\mathcal{W}$-TCOMP and TμPR (= the class of *total* μPR functions).

EXERCISES:

1. Let $\mathcal{WC}$ be the programming language for 'while' and the *conditional* instruction, i.e. the language

$\mathcal{W}$ together with the construct

```
if V = 0
  then
    P_1
  else
    P_2
fi.
```

Prove or disprove: $\mathcal{WC}$-COMP = $\mathcal{W}$-COMP. Do *not* use CT.

2. Show that Ackermann's function is $\mathcal{WC}$-computable. (Write a program for Ackermann's function in $\mathcal{WC}$.)
3. Show that for sets: PR $\subset$ REC. (Hint: Give an effective enumeration of PR sets.)

15 The S^n_m Theorem

In the previous sections we defined various notions of computability, and investigated their interrelationship. In the remaining three sections, we will study some interesting properties of the indexing (or Gödel numbering) of $\mathcal{G}$-computable functions.

NOTES:

1. From now on, we will write "computable" for "$\mathcal{G}$-computable", and "COMP" for the class "$\mathcal{G}$-COMP".
2. Although our indexing of computable functions is induced by our GN of the programming language $\mathcal{G}$ (and so depends on a particular GN of a particular programming language), it can be shown that the results below (S^n_m theorem, fixed point and recursion theorems, and Rice's theorem) hold under very general assumptions on the indexing of computable functions.

The main result of this section, the S^n_m theorem of Kleene (also known as the *parameter theorem*), is very useful for manipulating indices of functions, and is one of the main tools in the proof of the recursion theorem (§16).

Theorem 15.1 (S^n_m Theorem) *For all $m, n > 0$, there is an $(n+1)$-ary function $S^n_m \in PR$ such that for all $u_1, \cdots, u_n, x_1, \cdots, x_m, y$*

$$\varphi_y^{(m+n)}(\vec{x}, \vec{u}) \simeq \varphi^{(m)}_{S^n_m(y,\vec{u})}(\vec{x}).$$

For some intuition on what this theorem says, let $m = n = 1$. Then there exists a binary PR function $S = S^1_1$ such that for all x, u, y,

$$\varphi_y^{(2)}(x, u) = \varphi_{S(y,u)}(x).$$

We may think of $\varphi_y^{(2)}$ for *fixed* y, u as a unary function $\lambda x \cdot \varphi_y^{(2)}(x, u)$. This function is $\mathcal{G}$-computable, with gn z (say). So for all x,

$$\varphi_z(x) \simeq \varphi_y^{(2)}(x, u).$$

The theorem then says that z depends *primitive recursively* on y and u, i.e.

$$z = S(y, u) \text{ for } S \in \text{PR}.$$

Proof: By induction on n:

- **Basis**: $n = 1$. We want a PR function S^1_m such that for $\vec{x} \equiv x_1, \cdots, x_m$,

$$\varphi_y^{(m+1)}(\vec{x}, u) \simeq \varphi^{(m)}_{S^1_m(y,u)}(\vec{x}).$$

Let $\mathcal{P}_y$ be a progam for $\varphi_y^{(m+1)}$. For fixed y, u we now want a program $\mathcal{Q}$ for computing $\lambda\vec{x} \cdot \varphi_y^{(m+1)}(\vec{x}, u)$. We can think of $\mathcal{Q}$ as consisting of two parts:

$$\begin{array}{ll} \mathcal{Q}_1: & \text{initialise } X_{m+1} \text{ to } u, \\ \mathcal{Q}_2: & \text{then execute } \mathcal{P}_y. \end{array}$$

Clearly, we can take

$$\mathcal{Q}_1 \equiv \boxed{\left.\begin{array}{c} X_{m+1}{+}{+} \\ \vdots \\ X_{m+1}{+}{+} \end{array}\right\} u \text{ times}}.$$

Now the gn of instruction '$X_{m+1}{+}{+}$' is

$$\langle 0, \langle 1, 2m+1 \rangle\rangle = 16m + 10.$$

So

$$\begin{array}{rcl} \#(\mathcal{Q}_1) & = & (\prod_{i=1}^u p_i)^{16m+10} \dot{-} 1 \\ & = & q_1(u) \text{ (say)} \\ \text{and} \quad \#(\mathcal{Q}_2) & = & y, \end{array}$$

where $q_1 \in$ PR. Therefore

$$\begin{array}{rcl} \#(\mathcal{Q}) & = & \boldsymbol{concat}(q_1(u)+1, y+1) \dot{-} 1 \\ & = & S^1_m(y, u), \end{array}$$

where $S^1_m \in$ PR (by Proposition 7.5).

- **Induction step**: Suppose the result holds for $n = k$. Then

$$\begin{array}{l} \varphi_y^{(m+k+1)}(\vec{x}, u_1, \cdots, u_{k+1}) \\ \simeq \varphi^{(m+k)}_{S^1_{m+k}(y,u_{k+1})}(\vec{x}, u_1, \cdots, u_k) \\ \simeq \varphi^{(m)}_{S^k_m(S^1_{m+k}(y,u_{k+1}),u_1,\cdots,u_k)}(\vec{x}). \end{array}$$

By defining

$$\begin{array}{l} S^{k+1}_m(y, u_1, \cdots, u_{k+1}) \\ =_{\text{df}} S^k_m(S^1_{m+k}(y, u_{k+1}), u_1, \cdots, u_k) \end{array}$$

the result follows. □

NOTE: In the universal function theorem (Theorem 9.4) and the S^n_m theorem we have two powerful tools for forming new computable functions from old:

- The UFT states that $\varphi_y^{(n)}(\vec{x})$ is a computable function of y and $\vec{x}$ *together*, i.e. it provides a way of moving arguments *up* from the index.
  EXAMPLE: $\varphi^{(2)}_{\varphi_z(y)}(x, \varphi_{\varphi_u(x)}(z))$ is a computable function of u, x, y, z.
- The S^n_m theorem makes it possible to move arguments *down* to the index while preserving primitive recursiveness.
  EXAMPLE: Suppose f is a 5-ary computable function of x, y, z, u, v. Then the arguments y, u, v (say) can be moved down to the index, i.e.
  $$f(x, y, z, u, v) \simeq \varphi_{g(y,u,v)}(x, z)$$
  for some $g \in$ PR.
- These two tools can be used "simultaneously". EXAMPLE: We can show that there is a function $g \in$ PR such that for all u and v, $\varphi_u \circ \varphi_v = \varphi_{g(u,v)}$. Indeed, for some computable function f and some PR function g,
  $$\begin{array}{rcll} \varphi_u(\varphi_v(x)) & \simeq & f(u, v, x), & \text{(by UFT)} \\ & \simeq & \varphi_{g(u,v)}(x), & \text{(by } S^n_m\text{)}. \end{array}$$

16 The Recursion Theorem

The following theorem, due to Kleene, is a powerful tool in computability theory. Its proof uses the S^n_m theorem, and involves a dazzling use of diagonalisation.

Theorem 16.1 (Recursion Theorem) *Let g be an $(m+1)$-ary computable function. Then there is some e such that for all $\vec{x}$,*

$$\varphi_e(\vec{x}) \simeq g(\vec{x}, e).$$

Proof: For all $v, \vec{x}$ there is some d such that

$$\begin{array}{rcll} g(\vec{x}, S^1_m(v, v)) & \simeq & \varphi_d^{m+1}(\vec{x}, v), & \\ & \simeq & \varphi^{(m)}_{S^1_m(d,v)}(\vec{x}) & \text{(by } S^n_m\text{)} \end{array}$$

Putting $v = d$ and $e = S^1_m(d, d)$, we obtain

$$g(\vec{x}, e) \simeq \varphi_e(x). \ \square$$

A useful alternative version of the recursion theorem is the following:

Corollary 16.1 (Fixed Point Theorem) *Let f be a total computable function. Then there is some e such that*

$$\varphi_e = \varphi_{f(e)}.$$

Proof: Let

$$g(z, x) \simeq \varphi_{f(z)}(x).$$

Then g is computable by the universal function theorem. Therefore by the recursion theorem there is some e such that for all x,

$$\varphi_e(x) \simeq g(e, x) \simeq \varphi_{f(e)}(x). \ \square$$

EXAMPLES:

1. There is some e such that for all x, $\varphi_e(x) = e$, i.e. there is a program which gives its own gn as output! This is the basic idea behind "self-reproducing programs" and viruses.
   Proof: Let $f = \lambda z, \vec{x} \cdot z \in$ COMP. By the recursion theorem there is some e such that for all x,
   $$\varphi_e(x) \simeq f(e, x) = e. \ \square$$
2. More generally: Take *any* total computable unary function g, for example $g(x) = x^x$. Then there is some e such that for all x,
   $$\varphi_e(x) = g(e) = e^e.$$

EXERCISE: Prove the result stated in Example 2 above.

17 Rice's Theorem

One of many interesting applications of the recursion theorem is in the proof of the following result, which we will use to give many simple examples of non-recursive sets.

We define the '$\sim$' relation on $\mathcal{N}$ by

$$x \sim y =_{\text{df}} \varphi_x = \varphi_y.$$

Proposition 17.1 *The relation '$\sim$' is an* equivalence relation *on $\mathcal{N}$. Hence it partitions $\mathcal{N}$ into* equivalence classes.

Note that the fixed point theorem says that for every total computable function f there is some e such that $f(e) \sim e$.

A set $A \subseteq \mathcal{N}$ is called an *index set* iff A is closed under '$\sim$', i.e. $\forall x, y\ (x \in A \wedge x \sim y \Rightarrow y \in A)$. Now given sets $A \subseteq \mathcal{N}$ and $F \subseteq$ COMP, let

$$\begin{array}{rcl} \mathbb{F}(A) & =_{\text{df}} & \{\varphi_x | x \in A\} \ \subseteq \ \text{COMP}, \\ \mathbb{I}(F) & =_{\text{df}} & \{x \in \mathcal{N} | \varphi_x \in F\} \ \subseteq \ \mathcal{N}. \end{array}$$

So $\mathbb{I}(F)$ is the set of indices of functions in F. The two operations $\mathbb{F}$ and $\mathbb{I}$ are *almost inverse* to each other, in the following sense.

Proposition 17.2
(a) For any $F \subseteq$ COMP, $\mathbb{F}(\mathbb{I}(F)) = F$.
(b) For any $A \subseteq \mathcal{N}$, $\mathbb{I}(\mathbb{F}(A)) = \{y | \exists x \in A(x \sim y)\}$, i.e. the closure *of A under '$\sim$'. Hence $\mathbb{I}(\mathbb{F}(A)) \supseteq A$, with equality iff A is an index set.*

Corollary 17.1 *A subset of $\mathcal{N}$ is an index set iff it is the set of indices of some set of computable functions.*

EXAMPLES OF INDEX SETS:

1. $\mathcal{N}$,

2. $\emptyset$,
3. $[a]$, $[a] =_{\mathrm{df}} \{b | b \sim a\}$, the '$\sim$'-equivalence class of a, for any $a \in \mathcal{N}$,
4. Any *union* of index sets.

Theorem 17.1 (Rice) *The only recursive index sets are $\mathcal{N}$ and $\emptyset$.*

Proof (J. Case): Suppose that

$$A \text{ is an index set,} \tag{23}$$

$$\emptyset \subset A \subset \mathcal{N}, \text{ and} \tag{24}$$

$$A \text{ is recursive.} \tag{25}$$

We will now get a contradiction from (23), (24) and (25). By (24), choose

$$a \in A, b \notin A, \tag{26}$$

and define

$$f(z,x) \simeq \begin{cases} \varphi_b(x) & \text{if } z \in A \\ \varphi_a(x) & \text{if } z \notin A. \end{cases}$$

Then f is computable, since A is recursive by (25). By the recursion theorem, there is some e such that

$$\varphi_e(x) \simeq f(e,x) \simeq \begin{cases} \varphi_b(x) & \text{if } e \in A \\ \varphi_a(x) & \text{if } e \notin A. \end{cases}$$

We consider the two possibilities:

$$\begin{array}{rl} & e \in A \Rightarrow \varphi_e = \varphi_b \Rightarrow e \sim b \overset{(23)}{\Rightarrow} b \in A, \\ \text{or} & e \notin A \Rightarrow \varphi_e = \varphi_a \Rightarrow e \sim a \overset{(23)}{\Rightarrow} a \notin A. \end{array}$$

Both possibilities lead to a contradiction to (26). □

Corollary 17.2 *The following sets are* not *recursive:*

(a) $[a]$, *for any* $a \in \mathcal{N}$,
(b) $\{z | \varphi_z \text{ total}\}$,
(c) $\{z | \varphi_z \text{ constant}\}$,
(d) $\{z | \varphi_z$ *defined on at most* finitely *many args.*$\}$,
(e) $\{z | \varphi_z \text{ increasing}\}$,

$\vdots$

NOTE: By CT, Corollary 17.2*(b)* says that there is *no effective* method to decide, given any $\mathcal{G}$-program, whether it defines a *total function.* (This is related to the unsolvability of HP.) In fact, by §11, Exercise 3, this problem is not even *semi-decidable*! This shows that the notion of *computable partial function* (or *partial* algorithm) is *more fundamental* than the notion of *computable total function* (or *total* algorithm).

EXERCISES:

1. Prove Proposition 17.2 and Corollary 17.1.
2. (A uniform version of §7.3, Exercise 3): Show that there is a binary function $f \in \mathrm{PR}$ such that for all y, $\lambda n \cdot f(y,n)$ is 1-1, and for all y and n, $f(y,n) \sim y$.
3. Show that for every *total computable* f, there is a *primitive recursive* g such that for all x, $g(x) \sim f(x)$.
4. Is the relation '$\sim$' recursive?
5. Let $f(x) =$ "the least y such that $y \sim x$". (Note that f is total.) Is f computable?

Acknowledgements: This paper is based on class notes for the course given by the first author at WOFACS'92. We wish to thank Chris Brink for the invitation to give this course, as well as the participants at the workshop who pointed out errors and provided useful feedback, particularly Willem Labuschagne and Matt Clarke. We must also acknowledge the strong influence of the excellent textbook [1], which the first author has used a number of times for an advanced undergraduate course in computability theory. In particular, the universal and step-counter programs displayed in §9 were taken from Chapter 4 of [1].

The preparation of this paper was supported by funding from the Foundation for Research Development, through the FACCS-Lab of the University of Cape Town, and by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

1. M. Davis and E. Weyuker. *Computability, Complexity and Languages.* Academic Press, New York, 1983.
2. H. Rogers, Jr. *Theory of Recursive functions and Effective Computability.* McGraw-Hill, New York, 1967. (Chapters 1, 2, and 5).
3. S. C. Kleene. *Introduction to Metamathematics.* North-Holland, New York, 1952. (Part III).
4. D. van Dalen, H. C. Doets and H. de Swart. *Sets: Naive, Axiomatic and Applied.* Pergamon Press, Oxford, 1978.
5. M. Davis (ed.) *The Undecidable.* Raven Press, New York, 1965.