3. $\mathcal{G}$-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 must

(1) show it is stable, i.e., not dependent on slight changes in the definition of $\mathcal{G}$; and

(2) link this with more traditional characterisations of computability.

These will both be done later in the course.

3.1 $\mathcal{G}$-computability

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

- For any $n > 0$ and any n numbers $x_1, \ldots, x_n$, consider a ***computation*** s_1, $s_2, \ldots$ for $\mathcal{P}$ with ***initial snapshot*** $s_1 = (1, \sigma_1)$, where $\sigma_1 : \boldsymbol{var}(\mathcal{P}) \to \mathbb{N}$ is defined by

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

  — 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 $f(x_1, \ldots, x_n) \downarrow y$.

  — Case 2: This computation is *infinite*.

  Then $f(x_1, \ldots, x_n) \uparrow$.

- 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 on p. 2-4 yields the following:

$$\psi_{\mathcal{P}}^{(2)}(x_1, x_2) = x_1 + x_2$$
$$\psi_{\mathcal{P}}^{(1)}(x_1) = x_1$$
$$\psi_{\mathcal{P}}^{(3)}(x_1, x_2, x_3) = x_1 + x_2$$

- 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

$$P : \mathbb{N}^n \to \mathbb{2}.$$

From the $\mathcal{G}$-programs in §2.2 and §2.3 it follows that the functions $\lambda x \cdot 0$, $\lambda x \cdot x$, $\lambda x \cdot \uparrow$, $\lambda x, y \cdot (x + y)$, $\lambda x, y \cdot (x * y)$, and $\lambda x, y \cdot (x \dot{-} y)$ are $\mathcal{G}$-computable.

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

Clearly, the following inclusions hold:

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

The questions as to whether the above "$\subseteq$" inclusions are proper, i.e., whether *all* functions (or all total functions) are computable, must still 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.

3.2 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 $\boxed{W \leftarrow f(V_1, V_2, \ldots, V_n)}$ for f derived from $\mathcal{P}$ as follows:

1. Assume

   — $\boldsymbol{var}(\mathcal{P}) \subseteq \{X_1, \ldots, X_n, Z_1, \ldots, Z_k, Y\}$,

   — $\boldsymbol{lab}(\mathcal{P}) \subseteq \{E, A_1, \ldots, A_l\}$,

   — 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 put

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

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

   $$\mathcal{P}_m \equiv \mathcal{P}(Z_m, Z_{m+1}, \ldots, Z_{m+n}, Z_{m+n+1}, \ldots, \\ Z_{m+n+k}, E_m, A_{m+1}, \ldots, A_{m+l}).$$

3. Then let the macro $\boxed{W \leftarrow f(V_1, \ldots, V_n)}$ 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 $i \in \{1, \ldots, n\}$, and
- if $f(v_1, \ldots, v_n) \uparrow$, then the macro for f will *not terminate* if it is entered in state σ such that $\sigma(V_i) = v_i$, $i = 1, \ldots, n$. (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

$$\boxed{\textsf{if } P(V_1, \ldots, V_n) \textsf{ goto } L}$$

where P is *any computable predicate*. The appropriate macro expansion is

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

Example. If we want to use the statement $\boxed{\textsf{if } V = 0 \textsf{ goto } L}$, we have to verify that the following 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

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

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

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

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

In this way, the notion of $\mathcal{G}$-***computability*** 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 on p. 3-3 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 questions as to the properness of the "$\subseteq$" inclusions must still be answered.

Lemma 3.1.

(*a*) $\mathcal{G}$-COMP $\subseteq \mathcal{G}$-COMP$(\vec{g})$

(*b*) $\mathcal{G}$-COMP $= \mathcal{G}$-COMP$(\langle\rangle)$

(*c*) *If* $\vec{g} \subseteq \vec{h}$, *then* $\mathcal{G}$-COMP$(\vec{g}) \subseteq \mathcal{G}$-COMP$(\vec{h})$.

Proof: Clear from the definition. $\square$

Theorem 3.2 (Transitivity).

(*a*) *If* $f \in \mathcal{G}$-COMP$(\vec{g})$ *and* $g_1, \ldots, g_k \in \mathcal{G}$-COMP, *then* $f \in \mathcal{G}$-COMP. *More generally:*

(*b*) *If* $f \in \mathcal{G}$-COMP$(\vec{g})$ *and* $\vec{g} \in \mathcal{G}$-COMP$(\vec{h})$, *then* $f \in \mathcal{G}$-COMP$(\vec{h})$,

(*c*) *If* $f \in \mathcal{G}$-COMP$(\vec{g}, \vec{h})$ *and* $\vec{g} \in \mathcal{G}$-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 $(i = 1, \ldots, k)$ in the (relative) $\mathcal{G}$-program for f.
(*b*), (*c*): Similarly. $\square$

3.4 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 §4.1, while this section, building on our theory of relative computability, contains two methods for forming new computable functions from old:

(a) Composition

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

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

where $\vec{x} \equiv x_1, \ldots, x_n$, and "$\simeq$" means that the lhs of (1) is defined iff the rhs of (1) is, in which case they are equal. So $f(\vec{x}) \downarrow y$ (say) $\iff$

$$\exists z_1, \ldots, z_k \ \left[h_1(\vec{x}) \downarrow z_1 \wedge \cdots \wedge h_k(\vec{x}) \downarrow z_k \wedge g(\vec{z}) \downarrow y\right].$$

Lemma 3.3. *In (1), if g and $\vec{h}$ are total, then so is f.*

Proof: Similar to Lemma 1.2(a). $\square$

Lemma 3.4. *In (1), f is $\mathcal{G}$-computable in g and $\vec{h}$. Hence if $g, h_1, \ldots, h_k$ are $\mathcal{G}$-computable, then so is f.*

Proof: Using oracles for $g, h_1, \ldots, h_k$, we can construct a (relative) $\mathcal{G}$-program for f:

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

The second part of the statement follows from Theorem 3.2(a). $\square$

(b) Primitive Recursion

A unary function f, defined by

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

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

Lemma 3.5. *For any $k \in \mathbb{N}$, the constant function $\lambda\vec{x}\cdot k$ is $\mathcal{G}$-computable.*

Proof: For $k = 0$, either the empty program or the program $\boxed{\mathsf{skip}}$ computes the function. For $k > 0$, the following program can be used:

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

$\square$

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

Lemma 3.6. *In* (2), *if h is total then so is f.*

Proof: Exercise. $\square$

Lemma 3.7. *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:

	$Y \leftarrow k$
$[A]$	if $X = 0$ goto E
	$Y \leftarrow h(Z, Y)$
	$Z{++}$
	$X{--}$
	goto A

As before, the second part of the statement follows from Thm 3.2(a). $\square$

This 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

$$\begin{cases} f(\vec{x}, 0) & \simeq \; g(\vec{x}) \\ f(\vec{x}, t+1) & \simeq \; h(\vec{x}, t, f(\vec{x}, t)) \end{cases} \tag{3}$$

with *parameters* $\vec{x} \equiv x_1, \ldots, x_n$ (where g and h have arities n and $n+2$ respectively), is said to be ***defined from g and h by primitive recursion (with parameters)***.

Lemma 3.8. *In* (3), *if g and h are total, then so is f.*

Proof: Exercise. $\square$

Lemma 3.9. *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:

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

$\square$

3.5 Effective calculability

A (partial) 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 must be answered.