2. Programs Which Compute Functions

The basis for our study of *computable functions* is the programming language $\mathcal{G}$ (for “goto”; it is called $\mathcal{S}$ in [DW83]).

2.1 Syntax and Informal Semantics of $\mathcal{G}$

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

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

and also

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

We also write

— $V, W, V', \ldots$ for any variable (*metavariables for variables*),

— $L, L_1, \ldots$ for any label (*metavariables for labels*),

— omit the subscript 1, e.g. ‘X’ means X_1, and ‘A’ means A_1.

Statements $S, \ldots$ have one of the following 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 *finite list* of instructions, possibly the *empty* list $\langle\rangle$.

The ***informal semantics*** of $\mathcal{G}$-programs are clarified by the following assumptions. (The formal semantics are given later, in §2.3.)

- Aux. 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.
- If '... goto L' occurs, with more than 1 occurrence of label L, go to the first occurrence.
- Execution of a program *halts* if it has *either*

  — executed its last instruction, *or*

  — executed an instruction '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".

Notation.

(a) Variables can only take values in $\mathbb{N}$, i.e., they have type nat.

(b) We indicate the *value* of a variable by its lower case equivalent, e.g., x_1 denotes the *value* of X_1.

(c) More generally, lower case letters $x_1, x_2, \ldots, k, m, n, r, \ldots, u, v, \ldots$ will denote *natural numbers* (elements of $\mathbb{N}$).

Under the above informal semantics, it is clear that

each $\mathcal{G}$-program computes a function on $\mathbb{N}$.

(This will be formalised later, in §3.1.) This function is, in general, ***partial***, since for some input values the program may *diverge*.

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, with expansions

```
    Z++
    if Z ≠ 0 goto L
```

and

```
[L] V--
    if V ≠ 0 goto L
```

respectively.

Note. 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,
- replacing 'E' by the label for the statement immediately following the macro, if such a statement exists.

This is discussed more systematically in §3.2.

2.2 Examples of $\mathcal{G}$-programs.

- **Identity function** $\lambda x \cdot x$

— First attempt:

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

This is wrong since for input 0 it gives output 1 instead of 0.

— Second attempt:

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

But now the value of the input variable X is destroyed!

— 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 get the assignment macro $\boxed{V \leftarrow W}$:

```
V ← 0
Above program with X and Y
replaced by W and V
```

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

```
     Y ← X1
     Z ← X2
[B]  if Z ≠ 0 goto A
     goto E
[A]  Z−−
     Y++
     goto B
```

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

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

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

Note that the two statements $(*)$ 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.)

EXERCISES: 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 $\lambda x_1, x_2 \cdot (x_1 \leq x_2)$.

2.3 Formal Semantics for $\mathcal{G}$

We introduce the following concepts:

- $\boldsymbol{var}(S)$ 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 $\mathbb{N}$. We denote states by σ,τ, ... , e.g., $\sigma = \{(X, 3), (Y, 2), (Z, 4)\}$.

- σ is a *state of a 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 $W \in \boldsymbol{dom}(\tau)$,

$$\tau(W) = \begin{cases} \sigma(W) & \text{if } W \not\equiv V \\ m & \text{if } W \equiv V. \end{cases}$$

  (**Note**: Here and elsewhere, '$\equiv$' denotes *syntactic identity*.)

- For a program $\mathcal{P}$, $|\mathcal{P}|$ is the ***length*** of $\mathcal{P}$, i.e., the number of instructions in $\mathcal{P}$, and $(\mathcal{P})_i$ is the i-th instruction of $\mathcal{P}$, for $1 \le i \le |\mathcal{P}|$.

- A ***snapshot*** or ***instantaneous description*** of $\mathcal{P}$, with $|\mathcal{P}| = \ell$, is a pair $s = (i, \sigma)$ where $1 \le i \le \ell + 1$ and σ is a *state* of $\mathcal{P}$. Intuitively, σ is the state just before the execution of $(\mathcal{P})_i$ if $1 \le i \le \ell$, 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., $i \leq |\mathcal{P}| = \ell$, 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
skip $j = i + 1$ 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 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, $s_2, \ldots, s_k$ of snapshots such that $s_1 = (1, \sigma_1)$ and for $i = 1, \ldots, k-1$, 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, $s_2, \ldots$ of snapshots such that $s_1 = (1, \sigma_1)$ and for $i = 1, 2, \ldots$ 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.