ICC

Bulletin

Vol. 3 No. 3. July 1964

CONTENTS – *TABLE DES MATIERES*

The ICC BULLETIN, a bi-lingual publication (English and French) is published quarterly.

Subscription rate : $ 5.00 per annum (single copy $ 1.50)

Editorial Office: International Computation Centre, Casella postale No. 10053, Zona dell'E.U.R., Rome. Cable address: INTERCALCOLO.

On a family of Turing machines and the related programming language

by Corrado Böhm (1)

Summary – Two basic Turing machines λ, R (already introduced elsewhere) are defined for a one way infinite tape and any finite alphabet $\{c_0, c_1, \ldots, c_n\}$ where $n \geqslant 1$ and c_0 is the blank symbol of the tape. The family introduced is the least family of which λ, R are members and closed under the application of the *composition* or *product* rule (if A, B, are machines, so $B \circ A$, commonly written AB is too) and the *iteration* rule (if A is a machine so (A) is too, the minimum power of A by which the observed square in the final configuration is blank).

The main result is that, under the more or less usual definitions of computing a function value by the machine, every partial recursive function of n variables can be evaluated by a machine of the proposed family.

1. *A programming language equivalent to the Turing machines family*

Let us give an alphabet $C = \{ c_1, c_2, \ldots, c_n \}$ $(n \geqslant 1)$ and a Turing machine provided with an infinite tape to the left divided into squares of which only a finite number contains symbols of C, the remaining squares being blank, i. e. marked with the symbol c_0, and with a head which can read or write on one square at a time, and/or which is moveable on the tape.

It is well known (2) [5], [3], [7], [10] that such a machine is equivalent to a programme with a fixed number of atomic instructions.

In a previous paper [1] we showed that (at least in the case of $n = 1$, but as we shall see in a while the same thing is valid also for the general case of $n > 1$) an appropriate set is formed by the following three instructions:

R. Shift the head one square to the right, if any.

λ. Replace the scanned symbol c_i with c_{i+1} mod $(n + 1)$ and shift the head one square to the left.

$\uparrow$. If the scanned symbol is not c_0, then jump to the instruction marked with an arrow, otherwise go on with the next instruction (if any) immediately to the right.

Example.

$$S^+ \equiv \boxed{\rightarrow R \text{ } \lambda} \; [\lambda R]^n \lambda \; \boxed{\rightarrow \lambda R \; \boxed{\lambda R \; [\lambda R]^n \lambda \; \lambda} \; R \; \boxed{\rightarrow [\lambda R]^n \lambda}}$$

where $[H]^l$ is instead of $H H \ldots H$, l times, and $H^0 = \Phi$, the empty word.

---

(1) Work carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computation Centre (ICC) by the Italian Consiglio Nazionale delle ricerche (CNR) Research Group No 22 for 1963-64.

(2) The numbers in brackets refer to the bibliography.

Notes – 1.1) It has been shown in [1] that it is possible to normalize programmes in such a way that:

1.11) In a programme the first instruction performed, which is on the far left, must be λ and the last instruction performed, which is on the far right, must be R.

1.12) Any conditional jump always starts immediately after (and just to the right of) an instruction R and always reaches an instruction λ.

1.2) As it will be shown afterwards the converse conditional jump $\overset{\wedge}{+}$ (i.e. if the scanned symbol is c_0) can replace $\uparrow$.

In the following, both for printing and for formal reasons, we shall change the notation as follows.

First we append to each λ, reached by an arrow at least, an integer subscript $i = 1, 2, \ldots$ in such a way that in the programme there are not two λ with subscripts of the same value. Then, to each R which has an arrow on its right going to λ_i we append the subscript i and we suppress the arrow which now conveys no information.

If we repeat the procedure for every value of i, we shall finally obtain a word in the infinite alphabet

$$\Omega \equiv \{ \lambda, R \} \cup \Omega_\lambda \cup \Omega_R \qquad \text{where} \qquad \Omega_\lambda \equiv \{ \lambda_1, \lambda_2, \ldots \} \quad \Omega_R \equiv \{ R_1, R_2, \ldots \}$$

Example (the same as before but transformed)

$$\lambda_1 R [\lambda R]^n R_1 [\lambda R]^n \lambda\lambda_3 R_2 \lambda R [\lambda R]^n \lambda [\lambda R]^n \lambda R_3 \lambda R \lambda_2 R [\lambda R]^{n-1} \lambda [\lambda R]^n \lambda R_3$$

Let us suppose that among the atomic instructions there is also the jump $\overset{\wedge}{+}$, in the same way we can replace the arrows with subscripts, writing $\bar{R}_i$ instead of $R \overset{\wedge}{+}$, and extending Ω to

$$\Omega' \equiv \Omega \cup \Omega_{\bar{R}} = \Omega \cup \{ \bar{R}_1, \bar{R}_2, \ldots \}$$

A more formal definition of a programme in Ω (and/or Ω') could be the following:

Syntactic definition of a programme.

A programme is a word P in Ω (Ω') *such that for* $i = 1, 2, \ldots$:

1.3) for each occurrence of R_i (and/or $\bar{R}_i$) there is in P exactly one occurrence of λ_i.

1.4) for each occurrence of λ_i there is in P at least one occurrence of R_i (and/or $\bar{R}_i$).

Semantics. Let us interpret λ, R as functions partially defined on configurations having configurations as values. A *configuration* is defined as an ordered pair of words over $C \cup \{ c_0 \}$ the first one of which must not be of the type c_0^h, $h > 0$ (3) and represents the finite contents of the tape plus the indication

(3) If such an exclusion was not made, applying the first rule of (1.6) one would have $R^h L^h \neq I$ with L defined in the next note.

of the scanned symbol, which conventionally is the first symbol to the left of the second word. The i-th jump is conditioned by a predicate p, which is true if the scanned symbol is c_0 and false if it is different from c_0, otherwise undefined.

More precisely let x, y, be symbols to express variable words in the set Γ of all words over the alphabet $C \cup \{ c_0 \}$. Let us indicate with c (and c', c'') the variables in the set C (and $C \cup \{ c_0 \}$) and with xy the word resulting from the concatenation of the words x and y.

Then R, λ, p can be partially defined recursively for each configuration:

1.5) $R(x, \Phi)$ undefined — $\lambda(x, \Phi)$ undefined — $p(x, \Phi)$ undefined

1.6) $R(\Phi, c_0 y) = (\Phi, y)$ — $\lambda(\Phi, c_i y) = (\Phi, c_0 c_{i+1} y)$ — $p(x, c_0 y)$ *true*

1.7) $R(\Phi, c y) = (c, y)$ — $\lambda(x c', c_i y) = (x, c' c_{i+1} y)$ — $p(x, c y)$ *false*

1.8) $R(x c', c'' y) = (x c' c'', y)$

where $i = 0, 1, \ldots, n$ and the sum is *mod* $(n + 1)$.

Now let $\mathcal{Z} = (Q_z, m_z, q_0, q_m)$ a Turing machine over the alphabet $C \cup \{ c_0 \}$, where $Q_z = \{ q_0, q_1, \ldots, q_m \}$ is the set of internal states and where m_z is the table of $\mathcal{Z}$, i.e.

$$m_z(q_i, c_j) = (c_{ij}, t_{ij}, q_{ij})$$

$$q_i \in Q_z - \{ q_m \};\ c_j, c_{ij} \in C \cup \{ c_0 \};\ t_{ij} \in \{ R, I, L \};\ q_{ij} \in Q_z$$

I, L [4] being respectively the identity and left shifts, q_0 the initial state and q_m the final state of $\mathcal{Z}$.

For simplicity, let us identify c_j with j $(j = 0, 1, \ldots n)$.

Our preliminary result is:

To each Turing machine $\mathcal{Z}$ over $C \cup \{ c_0 \}$ corresponds a programme P_z over Ω (or over $\Omega' - \Omega_R$) in the sense that for each starting configuration (a, b), $\mathcal{Z}$ and P_z either do not stop at all or both stop in the same configuration $P_z(a, b) = (a', b')$.

Proof. Case Ω (jump on $p(x, c' y) =$ *false*, that is $c' \in C$)

For easier recognition of the jumps, we shall write

i, j instead of $2(n + 1)\ i + j$ and

$i, \underline{j}$ instead of $2(n + 1)\ i + (n + 1) + j$

It is a routine matter to check that for P_z we can choose the word

$$\lambda R_{0,n} \lambda R_{0,n} H_0 G_{00} \ldots G_{0n} H_1 G_{10} \ldots G_{1n} \ldots H_{m-1} G_{m-10} \ldots G_{m-1n} H_m$$

---

[4] $L = [\lambda R]^n \lambda$

where $H_i = \lambda_{i,n}\, R_{i,n-1}\, \lambda\, R_{i,n-1}\, \lambda_{i,n-1} \ldots \lambda\, R_{i,1}\, \lambda_{i,1}\, R\, \lambda\, R$ $(i = 0, 1, \ldots, m-1)$

$H_m = \lambda_{m,n}\, R\, \lambda_{m,n}\, R\, [\lambda\, R]^{n-2}$

$G_{ij} = \lambda_{i,j}\, R\, [\lambda\, R]^{c_{ij}-2}\, U_{t_{ij}}\, \lambda\, R_{q_{ij},n}\, \lambda\, R_{q_{ij},n}$ $(i = 0, 1, \ldots, m-1\ ;\ j = 0, 1, \ldots, n)$

being the difference mod $(n+1)$ and

$$U_R = R\ ,\quad U_L = [\lambda\, R]^n\, \lambda\ ,\quad U_I = \Phi$$

Case $\Omega' - \Omega_R$ (jump on $p\,(x, c'\,y) = true$, i.e. $c' = c_0$).
The programme P_z is simpler: if we write

$$i, j \text{ instead of } (n+1)\ i + j$$

it is again a routine matter to see that for P_z we can choose the word

$$\lambda\, R_{0,n}\, \lambda\, R_{0,n-1} \ldots \lambda\, R_{0,1}\, \lambda R\, H'_{00}\, H'_{01} \ldots H'_{0n}\, H'_{10} \ldots H'_{1n} \ldots H'_{m-1\,0} \ldots H'_{m-1\,n}\, H'_m$$

where

$$H'_{ij} = \lambda_{i,j}\, R\, [\lambda R]^{c_{ij}-1}\, U_{t_{ij}}\, \lambda R_{q_{ij},n}\, \lambda R_{q_{ij},n-1} \ldots \lambda R_{q_{ij},0} \quad \begin{bmatrix} i = 0, 1, \ldots, m-1 \\ j = 0, 1, \ldots, n \end{bmatrix}$$

and where

$$H'_m = \lambda_{m,n}\, R\, \lambda_{m,n-1}\, R \ldots \lambda_{m,1}\, R\, \lambda_{m,0}\, R\, [\lambda\, R]^n$$

with the same conventions as before.

2. *Choice of a sub-language* $\mathfrak{S}''$ *in* Ω' *and its properties.*

Let the set of all programmes in Ω' be $\mathfrak{S}'$, i. e. the set of all the words over Ω' which satisfy definitions 1.3) and 1.4).

$\mathfrak{S}'$ shares some inconveniences, though slighter, with the programming languages of present computers (also those with an automatic address allocation by means of stacks, etc.) (5).

If we want to combine new programmes by collating some prebuilt programmes, it is generally necessary to substitute some subscripts to avoid collisions that would let the word out of $\mathfrak{S}'$.

The reason for these disadvantages is to be found in the fact that programme formation rules, starting from the base alphabet, are context-depending, or, looking at the thing from another angle, in the fact that the base alphabet is infinite.

We want to extract from $\mathfrak{S}'$, by means of drastic limitations, a language $\mathfrak{S}'' \subset \mathfrak{S}'$ that is context-free and has the faculty of computing all the partial recursive functions.

---

(5) Remark that $\mathfrak{S}'$ can be generalized as, e.g., in Ianov [4] in order to describe programmes for any kind of existing computer.

Definition of $\mathfrak{S}''$:

A word P over Ω' is a programme in $\mathfrak{S}''$ if, besides satisfying conditions 1.3) and 1.4) (which guarantee that $P \in \mathfrak{S}'$) fulfills the following properties:

2.1) For each occurrence of λ_i there is not more than one occurrence of R_i or, alternatively, of $\bar{R}_i$.

2.2) The number of subscripts (jumps) is even $2N$, and their set is partitionable in pairs (i_j, k_j), $(j = 1, 2, \ldots, N)$ in such a way that the following pairs of words occur in P

$$\bar{R}_{i_j}\ \lambda_{k_j}\ ,\ R_{k_j}\ \lambda_{i_j}$$

from left to right in the order written.

2.3) Given any two distinct values of j, say q and r, let us consider in P the (geometrical) intervals

$$I_q = \bar{R}_{i_q} \ldots \lambda_{i_q}\ ,\ I_r = \bar{R}_{i_r} \ldots \lambda_{i_r}$$

Such intervals are in the following relation

$$\text{either } I_q \subset I_r \text{ or } I_r \subset I_q \text{ or } I_r \cap I_q \text{ is empty.}$$

Notes – 2.11) The condition 2.1) is not semantically restrictive since it is possible to put an arbitrary number of identity functions into a programme so as always to satisfy such a condition.

2.21) The condition 2.2) is very much restrictive since it reduces all jumps into decisions about entering or not a cyclic procedure.

2.31) The condition 2.3) sets other two restrictions:

a) It is only possible to enter a cycle from its first instruction. There is the same limitation in the FORTRAN language (DO statement).

b) It is only possible to go out of a cycle from its last instruction.

The essential advantage of the $\mathfrak{S}''$ language is that it is context-free. In fact if we replace (see property 2.2)

$$\bar{R}_{i_j}\ \lambda_{k_j} \quad \text{with } R\,(\,\lambda$$

$$R_{k_j}\ \lambda_{i_j} \quad \text{with } R\,)\,\lambda \qquad \text{for each } j = 1, 2, \ldots, N$$

it is easy to see that the information conveyed is the same since the property 2.3 is shared by every system of brackets.

To sum up, we showed that the finite alphabet $\Omega'' = \{\lambda, R, (,)\}$ is adequate for the $\mathfrak{S}''$ language, which can be defined by the following formation rules (6):

λ, R are words of $\mathfrak{S}''$ (axiom of atoms);
if α, β are words of $\mathfrak{S}''$ *so is* $\alpha\,\beta$ (composition rule);
if α is a word of $\mathfrak{S}''$ *so is* (α) (iteration rule).

(6) From now on we shall be free of the normalization rules 1.11 and 1.12 which were used only to simplify the notation within $\mathfrak{S}'$.

Semantics

Every $P \in \mathfrak{S}''$ is interpreted as partially defined function on the set $\mathcal{C}$ of the configurations into itself. λ, R have the known interpretation. If α, β are functions and (a, b) is any configuration, to apply $\alpha\,\beta$ to (a, b) means to apply the function γ defined as follows:

$$\gamma\ (a,\ b) = \beta\ (\alpha\ (a,\ b))$$

i.e. $\alpha\,\beta$ is the composition of α and β.

Finally, if α is a function and we indicate with α^n the composition of α with itself n times ($\alpha^0 = I$, identity function) is it easy to see that when (α) is applied to a configuration (a, b), it has the interpretation $\alpha^n\ (a, b)$ where it is

$$n = \mu\ \nu\ \{\, p\ (\alpha^\nu\ (a, b))\ \}\,.$$

From now on, for simplicity we shall identify the language $\mathfrak{S}'$ with the family of all Turing machines (and also with the family of corresponding functions from $\mathcal{C}$ into $\mathcal{C}$) and for each machine $\mathcal{Z}$ we shall consider the definition set $\mathcal{C}_z$ of the initial configurations with the property that every other configuration is defined (7).

Obviously the following properties hold:

2.4 *Right monotony* If $\mathcal{Z} \in \mathfrak{S}'$, $(\gamma_1\ ,\ \gamma_2) \in \mathcal{C}_z$, $\mathcal{Z}\ (\gamma_1\ ,\ \gamma_2) = (\delta_1\ ,\ \delta_2)$ then for every word x: $\mathcal{Z}\ (\gamma_1\ ,\ \gamma_2\, x) = (\delta_1\ ,\ \delta_2\, x)$

2.5 *Invertibility* If to the previous hypotheses we add: $\mathcal{Z}$ is a word over $\{\lambda, R\}$, then another word $\mathcal{Z}^{-1}$ exists over $\{\lambda,\ R\}$ such that

$$\mathcal{Z}^{-1}\ (\delta_1,\ \delta_2) = (\gamma_1,\ \gamma_2) \qquad \text{i.e. } \mathcal{Z}\ \mathcal{Z}^{-1} = I$$

Note – The above stated holds also for $\mathfrak{S}''$. Of course not all machines having right inverse are of the type mentioned. Let us consider for instance the machine H (8) which does nothing if the head scans initially a symbol different from c_0 and otherwise goes to the left until it scans the first c_0.

It can be represented by

$$H \equiv r'\ \lambda\ R_1\ \lambda_2\ R\ r''\ \lambda\ r'\ \lambda\ R_2\ \lambda_1\ R\ r'$$

where

$$r' = [\lambda\ R]^n \qquad r'' = [\lambda\ R]^{n-1}$$

Evidently H^{-1} exists and may be represented by

$$H^{-1} \equiv r'\ \lambda\ R_1\ \lambda_2\ R\ r'\ R_2\ \lambda_1\ R\ r'$$

Though we have not characterised, from a functional point of view, either $\mathfrak{S}'$ or $\mathfrak{S}''$, nevertheless from the next statement we can deduce that H^{-1} is not representable within $\mathfrak{S}''$.

---

(7) In other words the programmes of $\mathfrak{S}'$ avoid the head to exceed the right end of the tape during the computation.

(8) I am indebted to Dr. G. Jacopini for this example and for a first formulation of the next theorem.

2.6 If $\mathcal{Z}_2$ has brackets, $\mathcal{Z}_1$ does not exist even in $\mathfrak{S}'$, such that $\mathcal{Z}_1\ \mathcal{Z}_2 = I$.
In fact, by absurdum, suppose we have

$$(2.61) \qquad \mathcal{Z}_1\ \mathcal{Z}_2'\ (\mathcal{Z}_2'')\ \mathcal{Z}_2''' = I$$

where $\mathcal{Z}_2'$ and $\mathcal{Z}_2'''$, possibly empty, are words over $\{\lambda, R\}$.
From (2.61) we deduce because of the associative property of the composition and the invertibility of $\mathcal{Z}_2'''$

$$\mathcal{Z}_2'''\ \mathcal{Z}_1\ \mathcal{Z}_2'\ (\mathcal{Z}_2'') = I$$

which is absurd, since the final configuration, which, for I, coincides with the initial one, is arbitrary and not of the type $(x,\ c_0\ y)$ as for every programme which ends with brackets. Applying 2.6 to the case $\mathcal{Z}_1 = H$ we see that the conclusion is false and therefore the premise too is false, i.e. if $\mathcal{Z}_2 = H^{-1}$ exists in $\mathfrak{S}''$, must not have brackets; but this cannot be true because, e.g., the number of instructions executed with such a programme is fixed, while in the case of H^{-1} this number depends on the initial configuration. Therefore H^{-1} is not representable within $\mathfrak{S}''$.

More generally we can say that all the programmes invertible in $\mathfrak{S}'$ which are not words in $\{\lambda, R\}$ are exluded from $\mathfrak{S}''$.

3. *Computing partial recursive functions within $\mathfrak{S}''$*

Let us now state the main result of this paper:

For every partial recursive function f *of* m $\geqslant 0$ *variables a programme* $P_f \in \mathfrak{S}''$ *exists which computes it.*

This result cannot altogether be a surprise since a theorem, syntactically (but not semantically) analog, was stated by J. Robinson [6] for the recursive functions of one variable.

We were led to such a result by trying to simplify and to normalize the flow diagrams of the programmes for Turing machines and/or usual computers. In our opinion, an interesting point lies in the fact that the proving formulas are essentially independent of the number n of symbols in C. (9) This was feasible by adopting a uniform method (10) of representation of the integers through digits of the base $\{1, 2, \ldots, n\}$ $(n \geqslant 1)$ according to which every integer $\nu > 0$ has a uniquely determined representation of the type

$$\nu_h\ \nu_{h-1} \ldots \nu_0 \qquad 1 \leqslant \nu_i \leqslant n$$

where

$$\nu = \sum_{0}^{h} \nu_i\ n^i \qquad (11)$$

---

(9) Really only the sub-programmes r, r' depend on n.
(10) In the case $n = 2$ our method coincides with Smullyan's [8] p. 10.
(11) In such a representation the decimal digit set becomes

$$\{1, 2, \ldots, 9, \mathrm{t}\} \quad \text{and} \quad 10 \leftrightarrow \mathrm{t}\ ,\ 100 \leftrightarrow 9\,\mathrm{t}\ ,\ 1010 \leftrightarrow 9\,\mathrm{t}\,\mathrm{t}, \ldots$$

As a consequence, the number 0 is represented by the empty word Φ. Before proving 3., let us define the meaning of « computing », and a useful notion of partial recursive function as follows:

I) A programme P_f *computes* the function f of n variables if, denoting by $a_1, a_2, \ldots, a_m$ the words over C corresponding to the values of $x_1, x_2 \ldots x_m$ and by $f(a_1, \ldots, a_m)$ the word corresponding to the value of the function, we have

$$P_f(\Phi, \square\, a_1 \square\, a_2 \square \ldots \square\, a_m \square) = (\Phi, \square\, f(a_1, a_2, \ldots, a_m) \square)$$

where $\square$ is the blank symbol c_0.

II) The set of *partial recursive functions* is obtained by applying to the basic functions

$O^{(m)}$ zero function of m variables $O^m(x_1, \ldots, x_m) = 0 \quad m \geqslant 0$

S^+ successor $S^+(x) = x + 1$

S^- predecessor $S^-(x) = x - 1$, not defined for $x = 0$

$U_i^{(m)}$ selection function $U_i^{(m)}(x_1, \ldots, x_m) = x_i \quad (i = 1, 2, \ldots, m)$

a finite number of times either the following schemas:

Composition If $f^{(l)}$, $g_i^{(m)}$ are partial recursive (where $m \geqslant 0$, $l > 0$, $i = 1, 2, \ldots, l$) also $h^{(m)}$ defined by $h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_l(x_1, \ldots, x_m))$ is partial recursive.

Recursion. If $f^{(m+1)}$, $g^{(m+2)}$, $q^{(m+1)}$, p (of one variable) are partial recursive also $h^{(m+1)}$ defined by

$$h(y, x_1, \ldots, x_m) = \begin{cases} f(y, x_1, \ldots, x_m) \text{ if } q(y, x_1, \ldots, x_m) = 0 \\ g(h(p(y), x_1, \ldots, x_m), p(y), x_1, \ldots, x_m) \text{ otherwise} \end{cases}$$

is partial recursive.

Note – We preferred to adopt the above stated recursion schema instead of the more known schemas of primitive recursion and minimalization to keep to the usual way of programming. Nevertheless it is easy to see that the minimalization schema can be obtained writing $f^{(m+1)} = U_1^{(m+1)}$, $g^{(m+2)} = U_1^{(m+2)}$, $p = S^+$ which implies

$$h'(x_1 \ldots, x_m) \equiv h(0, x_1, \ldots, x_m) = \mu y \{ q(y, x_1, \ldots, x_m) = 0 \}$$

moreover the primitive recursion schema can be obtained writing $q = U_1^{(m+1)}$, $p = S^-$.

Proof of 3.

We need now to define some sub-programmes which are useful for the proof. First of all let us write

$$r \equiv \lambda R \ , \ r' \equiv [\lambda R]^n$$

and remember that

$$r\, r' = r'\, r = I \ , \ r'\, \lambda = L$$

in a way that

$$r(\Phi, c_i) = (\Phi, c_{i+1}) \ , \ r'(\Phi, c_i) = (\Phi, c_{i-1})$$

Let us introduce the routines T^- and T^+

$$T^- \equiv (r)\ R^2\ ((r'\ L\ r\ R)\ R)\ L \quad , \quad T^+ \equiv (r)\ L^2\ ((r'\ R\ r\ L)\ L)\ R$$

It is easy to verify that if x is any word over C and $c \in C \cup \{\Box\}$ we have

$$T^-\ (\Phi,\ c \Box x \Box) = (x,\ \Box c \Box)$$
$$T^+\ (x \Box,\ c) = (\Phi,\ \Box x \Box)$$

with the help of T^-, T^+ we can build the doubling routine

$$W \equiv L T^- L_\Box L T^- L\ (L_\Box^2\ r\ R_\Box^2\ R\ r\ L^2\ r'\ (L_\Box^2\ R\ r\ R_\Box^2\ R\ r\ L^2 r')\ L)\ R\ T^+\ R_\Box R\ T^+$$

where is $\quad L_\Box \equiv L\ (L) \quad R_\Box \equiv R\ (R)$ and

whose effect is

$$W\ (\Phi,\ \Box x \Box) = (\Phi,\ \Box x \Box x \Box)$$

More generally we build W_m (being $W_1 \equiv W$)

$$W_m \equiv L T^{-m} L_\Box^m L\ T^{-m} L\ (L_\Box^{m+1}\ r\ R_\Box^{m+1}\ R\ r\ L^2\ r'\ (L_\Box^{m+1}\ R\ r\ R_\Box^{m+1}\ R\ r\ L^2 r')\ L) \cdot$$
$$\cdot\ R\ T^{+m}\ R_\Box^m\ R\ T^{+m}$$

whose effect is

$$W_m\ (\Phi, \Box x_1 \Box x_2 \Box \ldots \Box x_m \Box) = (\Phi, \Box x_m \Box x_1 \Box x_2 \Box \ldots \Box x_m \Box)$$

Last of all we need two cancellation routines

$$K_L \equiv R\ ((r)\ R) \qquad K_R \equiv R_\Box\ R\ (T^+\ R_\Box\ R)\ T^+$$

whose effect is

$$K_L\ (\Phi,\ \Box x \Box y \Box) = (\Phi,\ \Box y \Box)$$
$$K_R\ (\Phi,\ \Box x \Box y \Box) = (\Phi,\ \Box x \Box)$$

Now we can write the programmes for the recursive functions. We can easily verify that the following programmes satisfy our requests.

$$O^{(m)} \equiv K_L^m\ L$$
$$U_i^{(m)} \equiv K_L^{i-1}\ K_R^{m-i}$$
$$S^+ \equiv R_\Box\ L\ (r\ (L_\Box\ L^2\ r\ R^2)\ r\ L)\ r\ L\ ((r)\ R\ (r)\ R\ (r))$$
$$S^- \equiv R_\Box\ L\ (r'\ (L_\Box)\ r'\ L)\ R\ r$$

Composition. Let $F, G_1, \ldots, G_l$ be the programmes for $f^{(n)}, g_1^{(m)}, \ldots, g_l^{(m)}$. Then the programmes for $h^{(m)}$ is

$$H \equiv W_m^m\, G_l\, W_{m+1}^m\, G_{l-1} \ldots W_{m+l-1}^m\, G_1\, F\, K_R^m$$

Recursion. Let F, G, Q, P be the programmes for the functions $f^{(n)}$, $g^{(m+2)}$, $q^{(m+1)}$, p. Then the programme for $h^{(m+1)}$ is

$$H \equiv L\, W_{m+2}^{m+1}\, Q\, R\, (((r)\, R)\, W_{m+2}^{m+1}\, P\, W_{m+2}\, S^+\, W_{m+2}^{m+1}\, Q\, R)\, W_{m+2}^{m+1}\, F\, R_\square R \cdot (L\, L_\square\, K\, G\, R_\square\, R)\, L\, L_\square\, K_R^{m+1}$$

This last formula ends the proof.

Notes – The proof method is more or less standard. It has many analogies with those of Turing [9], Davis [2], Hermes [3]. Then we leave the task of checking the formulas to the reader. It is sufficient to check the configurations at the beginning (or end) of every iteration and starting an inductive reasoning there. Some programming tricks were used to find the programmes W, $S+$ and $S-$. For these last three programmes we took advantage of the right monotony property 2.4). We reserve a deeper study of the language $\mathcal{S}''$ for a later paper.

BIBLIOGRAPHY

[1] Böhm, C. – Jacopini, G.: « Nuove tecniche di programmazione semplificanti la sintesi di macchine universali di Turing », Rend. Acc. Naz. Lincei, serie VIII, Vol. 32 fasc. 6, giugno 1962, pp. 913-922.

[2] Davis, M.: « Computability and Unsolvability », Mc Graw-Hill, New York 1958.

[3] Hermes, H.: « Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit », Springer Verlag, Berlin, 1961.

[4] Ianov, Yu. I.: « On the equivalence and transformation of program schemes », Doklady Akad. Nauk S.S.S.R., 113, pp. 39-42, (1957), (russian).

[5] Lee, C. Y.: « Automata and finite automata », Bell Syst. Techn. Journal, 39 (1960), Sept., pp. 1267-1295.

[6] Robinson, J.: « General recursive functions », Proc. Amer. Math. Soc., I, pp. 703-718, (1950).

[7] Shepherdson, J. C. – Sturgis, H. E.: « Computability of recursive functions », J. of the Ass. Comp. Mach., Vol. 10 (1963), pp. 217-255.

[8] Smullyan, R. M.: « Theory of Formal Systems », Annals of Mathematics Studies, Number 47, Princeton, 1961.

[9] Turing, A. M.: « On computable numbers with an application to the Entscheidungsproblem », Proc. Lond. Math. Soc. $\{2\}$, 42 (1936-7), pp. 230-265; addendum and corrigendum, 43 (1937), pp. 544-546.

[10] Wang, H.: « A variant to Turing's theory of computing machines », J. ACM 4 (1957), pp. 63-92.