Computational Linguistics

D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

CORRADO BÖHM AND GIUSEPPE JACOPINI
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and R. That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Ianov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

This paper was presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2880.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computation Centre (ICC), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 22 for 1963–64.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object x of a set X, the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc. There are other boxes of predicative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $x \in X$ occurs. Examples of diagrams are: $\Sigma(\alpha, \beta, \gamma, a, b, c)$ [Figure 3] and $\Omega_5(\alpha, \beta, \gamma, \delta, \epsilon, a, b, c, d, e)$ [see Figure 4]. It is easy to see a difference between them. Inside the diagram Σ, some parts which may be considered as a diagram can be isolated in such a way that if $\Pi(a, b)$, $\Omega(\alpha, a)$, $\Delta(\alpha, a, b)$ denote, respectively, the diagrams of Figures 5–7, it is natural to write

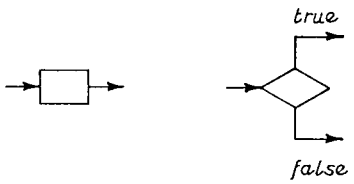
$$\Sigma(\alpha, \beta, \gamma, a, b, c) = \Omega(\alpha, \Delta(\beta, \Omega(\gamma, a), \Pi(b, c))).$$

Nothing of this kind can be done for what concerns Ω_5; the same happens for the entire infinite class of similar diagrams

$$\Omega_1[=\Omega], \Omega_2, \Omega_3, \cdots, \Omega_n, \cdots,$$

whose formation rule can be easily imagined.

Let us say that while Σ is decomposable according to subdiagrams Π, Ω and Δ, the diagrams of the type Ω_n are not decomposable. From the last consideration, which should be obvious to anyone who tries to isolate with a



FIGS. 1–2. Functional and predicative boxes

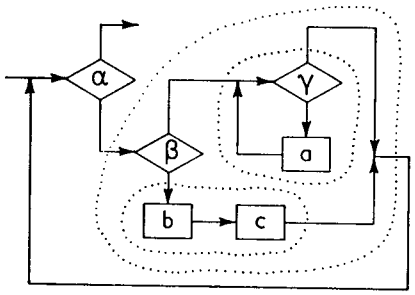


FIG. 3. Diagram of Σ

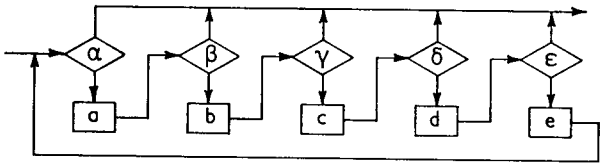
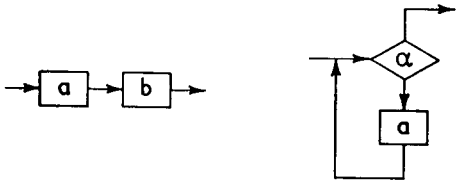
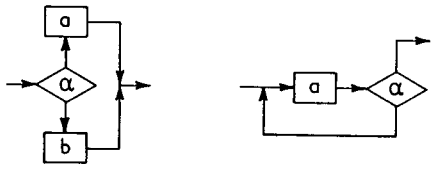


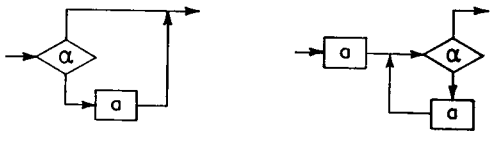
FIG. 4. Diagram of Ω_5



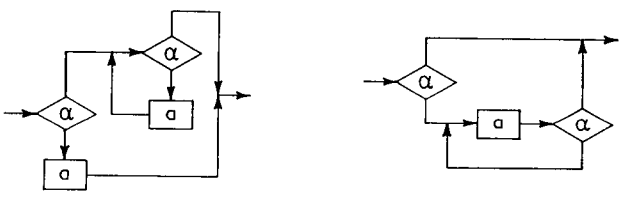
FIGS. 5–6. Diagrams of Π and Ω



FIGS. 7–8. Diagrams of Δ and Φ



FIGS. 9–10. Diagram of Λ and a diagram equivalent to Φ



FIGS. 11–12. Diagrams equivalent to Λ and Ω

broken line (as was done for Σ) a part of Ω_n provided with only one input and one output, it follows that:

It is not possible to decompose all flow diagrams into a finite number of given base diagrams.

However, together with this decomposition, that could be called *strong*, another decomposition may be considered which is obtained by operating on a diagram equivalent to the one to be decomposed (that is, the diagram has to express the same transformation, whatever the meaning of the boxes contained in it may be). For instance, it may be observed that if we introduce $\Phi(\alpha, a)$ [as in Figure 8] and $\Lambda(\alpha, a)$ [as in Figure 9] the diagrams of Φ, Λ and Ω become, respectively, equivalent to Figures 10, 11 and 12.

Thus, the following decompositions may be accepted:

$$\Phi(\alpha, a) = \Pi(a, \Omega(\alpha, a))$$

$$\Lambda(\alpha, a) = \Delta(\alpha, \Omega(\alpha, a), a)$$

$$\Omega(\alpha, a) = \Lambda(\alpha, \Phi(\alpha, a)).$$

Nevertheless, it is to be reckoned that the above statement holds even with regard to the new wider concept of decomposability. In fact, it does not seem possible[1] for every Ω_n to find an equivalent diagram which does not contain, as a subprogram, another Ω_n or an Ω of higher order. For instance, note that

$$\Omega_3(\alpha, \beta, \gamma, a, b, c) = \Lambda(\alpha, \Pi(a, \Omega_3(\beta, \gamma, \alpha, b, c, a)))$$
$$= \Omega_6(\alpha, \beta, \gamma, \alpha, \beta, \gamma, a, b, c, a, b, c)$$

and similar formulas hold for all orders of Ω.

The proved unfeasibility is circumvented if a new predicate is added and if, among the elementary operations, some are assumed which either add one bit of information to the object of the computation or remove one from it. The extra bits have a stack structure (formally described below as nested ordered pairs) since it is sufficient to operate and/or take decisions only on the topmost bit.

Therefore, three new functional boxes denoted by T, F, K, and a new predicative box ω are introduced. The effect of the first two boxes is to transform the object x into the ordered pair (v, x) where v can have only the values **t** (true) or **f** (false); more precisely,

$$x \xrightarrow{T} (\mathbf{t}, x), \quad x \xrightarrow{F} (\mathbf{f}, x), \quad (\mathbf{t}, x) \xrightarrow{T} (\mathbf{t}, (\mathbf{t}, x))$$

and so on. Box K takes out from an ordered pair its second component

$$(v, x) \xrightarrow{K} x, \quad (\mathbf{t}, (\mathbf{f}, (\mathbf{t}, x))) \xrightarrow{K} (\mathbf{f}, (\mathbf{t}, x)).$$

The predicate ω is defined as

$$\omega[(v, x)] = \mathbf{t} \Leftrightarrow v = \mathbf{t},$$

i.e., the predicate ω is verified or not according to whether the first component of the pair is T or F; ω and K are defined only on a pair; on the contrary, all the boxes

[1] We did not, however, succeed in finding a plain and sufficiently rigorous proof of this.

$\alpha, \beta, \gamma, \cdots, a, b, c, \cdots$ operating on x are not defined on a pair. The following statement holds:

If a mapping $x \rightarrow x'$ is representable by any flow diagram containing $a, b, c, \cdots, \alpha, \beta, \gamma, \cdots$, it is also representable by a flow diagram decomposable into Π, Φ *and* Δ *and containing the same boxes which occurred in the initial diagrams, plus the boxes K, T, F and ω.*

That is to say, it is describable by a formula in $\Pi, \Phi, \Delta, a, b, c, \cdots, T, F, K, \alpha, \beta, \gamma, \cdots, \omega$.

NOTE. A binary switch is the most natural interpretation of the added bit v. It is to be observed, however, that in certain cases if the object x can be given the property of a list, any extension of the set X becomes superfluous. For example, suppose the object of the computation is any integer x. Operations T, F, K may be defined in a purely arithmetic way:

$$x \xrightarrow{T} 2x + 1, \quad x \xrightarrow{F} 2x, \quad x \xrightarrow{K} \left[\frac{x}{2}\right]$$

and the oddity predicate may be chosen for ω. The added or canceled bit v emerges only if x is thought of as written in the binary notation system and if the actions of T, F, K, respectively, are interpreted as appending a one or a zero to the far right or to erase the rightmost digit.

To prove this statement, observe that any flow diagram may be included in one of the three types: I (Figure 13), II (Figure 14), III (Figure 15), where, inside the section lines, one must imagine a part of the diagram, in whatever way built, that is called $\mathfrak{A}$ or $\mathfrak{B}$ (not a subdiagram). The branches marked 1 and 2 may not always both be present; nevertheless, from every section line at least one branch must start.

As for the diagrams of types I and II, if the diagrams in Figures 16–17, are called A and B,[2] respectively, I turns into Figure 20 and may be written

$$\Pi(\Pi(T, \Phi(\omega, \Pi(\Pi(K, a), A))), K)$$

and II turns into Figure 21, which may be written

$$\Pi(\Pi(T, \Phi(\omega, \Pi(K, \Delta(\alpha, A, B)))), K).$$

The case of the diagram of type III (Figure 15) may be dealt with as case II by substituting Figure 22, where $\mathfrak{C}'$ indicates that subpart of $\mathfrak{C}$ accessible from the upper entrance, and $\mathfrak{C}''$ that part accessible from the lower entrance.

If it is assumed that A and B are, by inductive hypothesis,[3] representable in Π, Φ and Δ, then the statement is demonstrated.

It is thus proved possible to completely describe a program by means of a formula containing the names of diagrams Φ, Π and Δ. It can also be observed that Ω, Π and Δ could be chosen, since the reader has seen (see

[2] If one of the branches 1 or 2 is missing, A will be simply Figure 18*a* or 18*b*, and similarly for B. If the diagram is of the type of Figure 19 where $V\{\in\}(T, F)$, it will be simply translated into $\Pi(V, A^*)$ where A^* is the whole subdiagram represented by $\mathfrak{A}$.

[3] The induction really operates on the number $3N + M$, where M is the number of boxes T and F in the diagram and N is the number of all boxes of any other kind (predicates included).

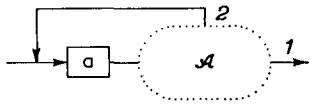


FIG. 13. Structure of a type I diagram

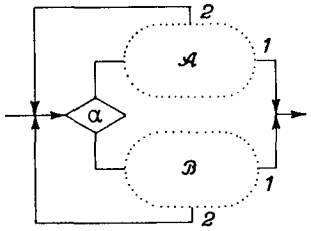


FIG. 14. Structure of a type II diagram

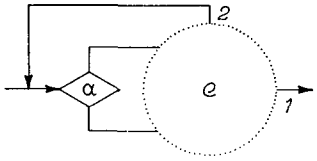


FIG. 15. Structure of a type III diagram

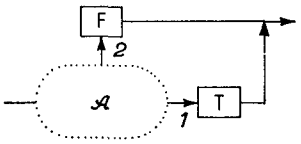


FIG. 16. A-diagram

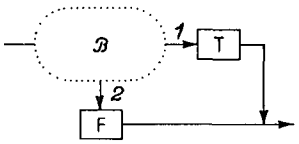


FIG. 17. B-diagram

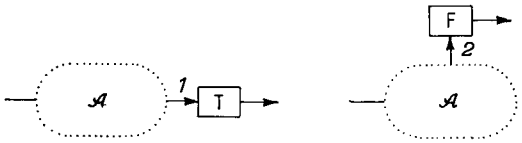


FIG. 18*a–b*. Two special cases of the A-diagram

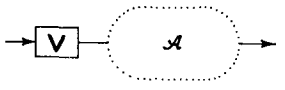


FIG. 19. Diagram reducible to $\Pi(V, A^*)$

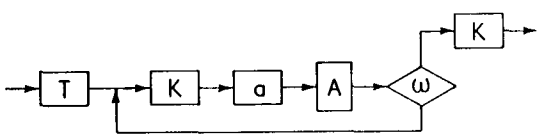


FIG. 20. Normalization of a type I diagram

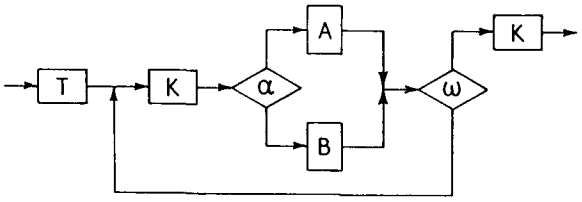


FIG. 21. Normalization of a type II diagram

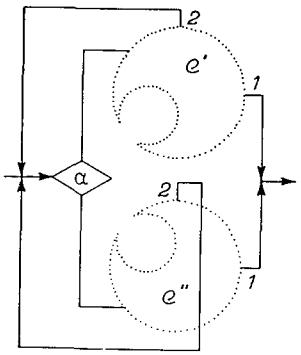


FIG. 22. Normalization of a type III diagram

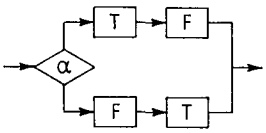


FIG. 23. The diagram $\underline{\alpha}$

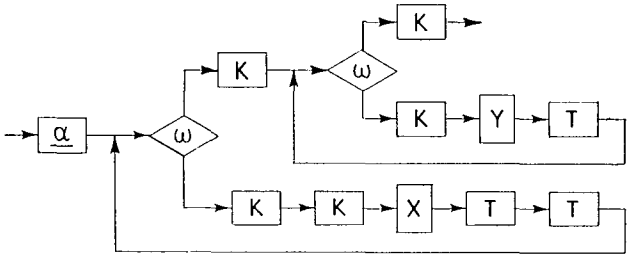


FIG. 24. Transformation of $\Delta(\alpha, X, Y)$

formula and Figure 10) that Φ can be expressed using Ω and Π. Moreover, it is observed that the predicate ω occurs only as the first argument of Φ (or, if desired, of Ω) and all the others as arguments of Δ: $\Phi(\omega, X)$ and $\Delta(\alpha, X, Y)$, etc.

Now let us define for every predicative box $\alpha, \beta, \cdots$ a new functional box $\underline{\alpha}, \underline{\beta}, \cdots$ with the following meaning:

$$\underline{\alpha} \equiv \Delta(\alpha, \Pi(T, F), \Pi(F, T)) \qquad \text{(Figure 23)}$$

This simplifies the language. In fact, any $\Delta(\alpha, X, Y)$ can be replaced by (see also Figure 24):

$$\Pi(\underline{\alpha}, \Pi(\Omega(\omega, \Pi(K, \Pi(K, \Pi(X, \Pi(T, T))))), \Pi(K, \Pi(\Omega(\omega, \Pi(K, \Pi(Y, T))), K)))).$$

Then we can simply write:

$$\Pi(X, Y) \equiv XY,$$

$$\Pi(\Pi(X, Y), Z) = \Pi(X, \Pi(Y, Z)) \equiv XYZ,$$

owing to the obvious associativity of Π. We may also write:[4]

$$\Omega(\omega, X) \equiv (X).$$

[4] The same notation is followed here as in [8].

To sum up: every flow diagram where the operations $a, b, c, \cdots$ and the predicates $\alpha, \beta, \gamma \cdots$ occur can be written by means of a string where symbols of operations $a, b, c, \cdots, \underline{\alpha}, \underline{\beta}, \underline{\gamma}, \cdots, T, F, K$ and parentheses (,) appear. For example:

$$\Pi(a, b) = ab$$

$$(^5)\quad \Omega(\alpha, a) = \underline{\alpha}K(Ka\underline{\alpha}K)K$$

$$(^5)\quad \Phi(\alpha, a) = F(Ka\underline{\alpha}K)K$$

$$\Delta(\alpha, a, b) = \underline{\alpha}(KKaTT)K(KbT)K$$

$$(^5)\quad \Lambda(\alpha, a) = \underline{\alpha}K(KaT)K$$

$$(^5)\quad \Omega_2(\alpha, \beta, a, b) = F(K\underline{\alpha}(KTT)K(Ka\underline{\beta}(KTT)K(KbFT))K)K.$$

More abstractly, the main result can be summarized as follows. Let

X be a set of objects x

Ψ a set of unary predicates $\alpha, \beta, \cdots$ defined in X

O a set of mappings $a, b, \cdots$ from X to X

$\mathfrak{D}(\Psi, \mathrm{O})$ the class of all mappings from X to X describable by means of flow diagrams containing boxes belonging to $\Psi \cup \mathrm{O}$.

Y the set of objects y defined by induction as follows:

$$\begin{cases} X \subset Y \\ y \in Y \Rightarrow (\mathbf{t}, y) \in Y, (\mathbf{f}, y) \in Y \end{cases} \qquad (1)$$

ω a predicate, defined in Y (at least on $Y - X$) by

$$\begin{cases} \omega(\mathbf{t}, x) = \mathbf{t} \\ \omega(\mathbf{f}, x) = \mathbf{f} \end{cases}$$

T, F two mappings defined on Y by

$$T[y] = (\mathbf{t}, y)$$

$$F[y] = (\mathbf{f}, y)$$

K a mapping defined in Y by

$$K[(\mathbf{t}, y)] = K[(\mathbf{f}, y)] = y$$

$\underline{\Psi}$ a set of mappings $\underline{\alpha}, \underline{\beta}, \cdots$ defined on X, with values in Y as follows:

$$\underline{\alpha}[x] = \neg\alpha[x], (\alpha[x], x) \qquad (2)$$

etc.

Now, given a set Z of objects z, a set Q of mappings from Z to Z, and *one* unary predicate π defined in Z, let us recursively define for every $q \in Q$ a new mapping $^{\pi}(q)$, written simply (q) if no misunderstanding occurs, as

[5] These formulas have not been obtained using the general method as described. The application of that method would make the formula even more cumbersome.

follows:

$$\begin{cases} \pi[z] \to (q)[z] = z \\ \sim\pi[z] \to (q)[z] = (q)[q[z]]. \end{cases}$$

For every $q_1, q_2 \in Q$, let us call q_1q_2 the mapping defined by $q_1q_2[z] = q_2[q_1[z]]$. Let us call $\mathcal{E}(\pi, Q)$ the class of mappings from Z to Z defined by induction as follows:

$$\begin{cases} Q \subset \mathcal{E}(\pi, Q) \\ q \in \mathcal{E}(\pi, Q) \Rightarrow (q) \in \mathcal{E}(\pi, Q) \\ q_1 \in \mathcal{E}(\pi, Q), q_2 \in \mathcal{E}(\pi, Q) \Rightarrow q_1q_2 \in \mathcal{E}(\pi, Q). \end{cases}$$

Note the following useful properties of $\mathcal{E}$:

$$Q_1 \subset Q_2 \to \mathcal{E}(\pi, Q_1) \subset \mathcal{E}(\pi, Q_2) \tag{3}$$

$$Q_2 \subset \mathcal{E}(\pi, O_1) \to \mathcal{E}(\pi, O_1 \cup Q_2) = \mathcal{E}(\pi, Q_1). \tag{4}$$

The meaning of the last statement can easily be rewritten:

$$\mathfrak{D}(\Psi, O) \subset \mathcal{E}(\omega, O \cup \Psi \cup \{T, F, K\}). \tag{5}$$

3. Applications to the Theory of Turing Machines

In a previous paper [8], a programming language $\mathcal{P}'$ was introduced which described, in a sense specified in that paper, the family $\mathfrak{B}'$ of Turing machines for a (leftward) infinite tape and any finite alphabet $\{c_1, c_2, \cdots, c_n\} \cup \{\square\}$, where $n \geqq 1$, $\square$ is the symbol for the blank square on the tape. Using the notation of Section 2 (see Note),

$$\mathfrak{B}' \equiv \mathfrak{D}(\{\alpha\}, \{\lambda, R\}) \tag{6}$$

where

- α is the unary predicate true iff the square actually scanned (by the Turing machine head) is blank (i.e. contains $\square$);
- λ is the operation of replacing the scanned symbol c_i with c_{i+1} ($c_0 \equiv c_{n+1} \equiv \square$) and shifting the head one square to the left;
- R is the operation of shifting the head one square, if any, to the right.

Briefly, α is a predicate, λ and R are partially defined functions[6] in the set X of tape configurations. By "tape configuration" of a Turing machine is meant the content of the tape plus the indication of the square being scanned by the machine head.

Example. If the configuration (at a certain time) is

$$x \equiv \cdots \square\square c_1\underline{c_n}c_n,$$

then

$$\alpha[x] \equiv \mathbf{f}, \quad \lambda[x] \equiv \cdots \square\square\underline{c_1}\square c_n, \quad R[x] \equiv \cdots \square\square c_1c_n\underline{c_n}$$

where the underscore indicates the scanned square. In [8] a language $\mathcal{P}''$ (describing a proper subfamily of Turing machines) has been shown. It was defined as follows.

(*i*) $\lambda, R \in \mathcal{P}''$ (Axiom of Atomic Operations)

(*ii*) $q_1, q_2 \in \mathcal{P}''$ implies $q_1q_2 \in \mathcal{P}''$ (Composition Rule)

(*iii*) $q \in \mathcal{P}''$ implies $(q) \in \mathcal{P}''$ (Iteration Rule)

(*iv*) Only the expressions that can be derived from (*i*), (*ii*) and (*iii*) belong to $\mathcal{P}''$.

Interpreting q_1, q_2 as functions from X to X, q_1q_2 can be interpreted as the composition $q_2 \circ q_1$, i.e.

$$q_1q_2[x] \equiv q_2[q_1[x]] \qquad x \in X$$

and (q) can be interpreted as the composition of q with itself n times: $q \circ \cdots \circ q \equiv q^n$, i.e. $q^n[x] \equiv q[\cdots[q[x]]\cdots]$ where $q^0[x] = x$ and $n = \mu\nu\{\alpha[q^\nu[x]] = \mathbf{t}\}$, $\nu \geqq 0$, i.e. (q) is the minimum power of q (if it exists) such that the scanned square, in the final configuration, is $\square$.

From the point of view of this paper, the set $\mathfrak{B}''$ of the configuration mappings described by $\mathcal{P}''$ is

$$\mathfrak{B}'' \equiv \mathcal{E}(\alpha, \{\lambda, R\}). \tag{7}$$

The drawbacks of $\mathcal{P}''$ as opposed to $\mathcal{P}'$ are that not all Turing machines may be *directly* described by means of $\mathcal{P}''$. For instance, it was proved in [8] that the operation H^{-1} (performed by the machine, which does nothing if the scanned symbol is different from $\square$, and otherwise goes to the right until the first $\square$ is scanned) cannot be described in $\mathcal{P}''$ ($H^{-1} \notin \mathfrak{B}''$).

Nevertheless, the most surprising property of $\mathcal{P}''$ is that, according to the commonest definition of "computing" a function by a Turing machine, every partial recursive function f in $m \geqq 0$ variables can be evaluated by a program $P_f \in \mathcal{P}''$ (see [8]).

Although this last property enables us to build a one–one mapping (via a gödelization of the Turing machines) of $\mathcal{P}'$ in $\mathcal{P}''$, it is here preferred to find a more direct correspondence between Turing machines, without any reference to partial recursive functions. To every Turing machine M, let us associate the machine M^* whose initial (and final) tape configuration is obtained by interspersing a blank square between every two contiguous squares of the tape of M. During the computation, these auxiliary squares are used to record, from right to left, the values v of the switch stack.

More precisely, for every configuration $x \equiv \cdots\square u_1 \cdots u_{K-1}\underline{u_K}u_{K+1}\cdots u_m$ where $u_i \in \{\square, c_1, \cdots, c_n\}$, let us call x^* the configuration

$$x^* \equiv \cdots \square\square\square u_1 \cdots \square u_{K-1}\square u_K\square\underline{u_{K+1}} \cdots \square u_m.$$

If M designates the Turing machine which when applied[7] to configuration b gives e as the final configuration, i.e. if $M[b] = e$, then M^* is a machine such that $M^*[b^*] = e^*$.

We want to prove: $M \in \mathfrak{B}', \Rightarrow M^* \in \mathfrak{B}''$.

Taking advantage of the theorem (5), we may write

$$\mathfrak{B}' \subset \mathcal{E}(\omega, \{\lambda, R, \underline{\alpha}, T, F, K\}). \tag{8}$$

Following the definition (1) of Y, the mapping $x \to x^*$

[6] For more details, see [8, 9].

[7] For simplicity, as in (6), Turing machines and configuration mappings will be identified.

is now extended to a mapping $y \to y^*$ as follows:

$$\text{if } y^* \equiv \cdots \square u_{K-1} \square \underline{u}_K \cdots \Rightarrow$$
$$(\mathbf{t},\ y)^* \equiv \cdots \square \underline{u}_{K-1} \square u_K \cdots, \quad (9)$$
$$(\mathbf{f},\ y)^* \equiv \cdots \square \underline{u}_{K-1} c_1 u_K \cdots.$$

Obviously,

$$M \in \mathfrak{B} \Rightarrow M \in \mathcal{E}(\omega, \{\lambda, R, \underline{\alpha}, T, F, K\})$$

and therefore

$$M^* \in \mathfrak{B}'^* \Rightarrow M^* \in \mathcal{E}(\omega^*, \{\lambda^*, R^*, \underline{\alpha}^*, T^*, F^*, K^*\}).$$

It is only necessary to prove that

$$\mathcal{E}(\omega^*, \{\lambda^*, R^*, \underline{\alpha}^*, T^*, F^*, K^*\}) \subset \mathcal{E}(\alpha, \{\lambda, R\}).$$

First, observe that for every machine $Z^* \in \mathcal{E}(\omega^*, \{\cdots\})$,

$$^{\omega^*}(Z^*) \equiv R\ ^{\alpha}(LZ^*R)L,$$

where $L \equiv [\lambda R]^n\lambda$ is the operation of shifting the head one square to the left, has been proved; therefore,

$$\mathcal{E}(\omega^*, \{\cdots\}) \subset \mathcal{E}(\alpha, \{\cdots\} \cup \{\lambda, R\}).$$

Secondly, it can be easily checked that

$$\{\lambda^*, R^*, T^*, F^*, K^*\} \subset \mathcal{E}(\alpha, \{\lambda, R\}).$$

In fact,

$$\lambda^* = \lambda L, \qquad R^* = R^2, \qquad T^* = L^2,$$
$$F^* = L\lambda, \qquad K^* = R(\lambda R)R.$$

According to (4), it has been proved that

$$\mathcal{E}(\omega^*, \{\cdots\}) \subset \mathcal{E}(\alpha, \{\underline{\alpha}^*, \lambda, R\}).$$

Thirdly,

$$\underline{\alpha}^* \in \mathcal{E}(\alpha, \{\lambda, R\}).$$

From formula (2) and the convention (9) it must follow that:

$$\underline{\alpha}^*[\cdots \square u_2 \square u_1 \square \underline{\square} \cdots] = \cdots \square \underline{u_2} c_1 u_1 \square \square \cdots \quad (10)$$

$$\underline{\alpha}^*[\cdots \square u_2 \square u_1 \square \underline{c} \cdots] = \cdots \square \underline{u_2} \square u_1 c_1 c \cdots \quad (11)$$

where $u_1, u_2 \in \{\square, c_1, \cdots, c_n\}$ and $c \in \{c_1 \cdots, c_n\}$.

In order to implement $\underline{\alpha}^*$, the program $L^3\lambda$ [which meets conditions (10)] must be merged with $L\lambda L^2$ [which meets (11)]. A solution[8] can be written in the form

$$\underline{\alpha}^* \equiv L^3\lambda R^4(X)L^5(Y)R,$$

which obviously satisfies (10). The program (X) can be chosen mainly to copy the symbol c on the first free blank square; the program (Y), to execute the inverse operation, i.e.,

$$(X)[\cdots \square u_2 c_1 u_1 \square \underline{c} \cdots] = \cdots \underline{c} u_2 \square u_1 c_1 \square \cdots$$
$$(Y)[\cdots\ c u_2 \square u_1 c_1 \square \cdots] = \cdots \square \underline{u_2} \square u_1 c_1 c \cdots$$

[8] The authors are indebted to the referee for this solution, which is shorter and more elegant than theirs.

It is not difficult to test that the choice

$$(X) \equiv (r'L(\lambda R)\lambda L(\lambda R)L^2\lambda R^6),$$
$$(Y) \equiv (r'R^5\lambda L^4),$$

where $r' \equiv [\lambda R]^n$, gives the desired solution.

RECEIVED NOVEMBER, 1965

REFERENCES

1. PETER, R. Graphschemata und rekursive Funktionen. *Dialectica 12* (1958), 373–393.
2. GORN, S. Specification languages for mechanical languages and their processors. *Comm. ACM. 4*, (Dec. 1961), 532–542.
3. HERMES, H. *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit.* Springer Verlag, Berlin, 1961.
4. CIAMPA, S. Un'applicazione della teoria dei grafi. *Atti del Convegno Nazionale di Logica, Torino 5–7* (April 1961), 73–80.
5. RIGUET, J. Programmation et théorie des catégories. Proc. ICC Symp. *Symbolic Languages in Data Processing*, Gordon and Breach, New York, 1962, 83–98.
6. IANOV, YU, I. On the equivalence and transformation of program schemes. *Dokl. Akad. Nauk SSSR 113*, (1957), 39–42. (Russian).
7. ASSER, G. Functional algorithms and graph schema. *Z. Math. Logik u. Grundlagen Math., 7*, (1961), 20–27.
8. BÖHM, C. On a family of Turing machines and the related programming language. *ICC Bull. 3*, (July 1964), 187–194.
9. BÖHM, C., JACOPINI, G. Nuove tecniche di programmazione semplificanti la sintesi di macchine universali di Turing. *Rend. Acc. Naz. Lincei {8}, 32*, (June 1962), 913–922.

LETTERS—cont'd from page 323

On 0 and *O*

EDITOR:

In reading the letter by Mr. Turner [On the Confusion Between "0" and "*O*", *Comm. ACM 9*, 1 (Jan. 1966), 35], I notice that his redefinition of the ALGOL ⟨identifier⟩ fails to allow those such as "012ABC", which contain a digit immediately after the zero. It seems that such a combination of characters will pose no major recognition problems, and should be allowed, providing only that it contains a letter somewhere, other than the initial "0", which may logically be taken as a letter or a digit.

I therefore offer an addition to Mr. Turner's redefinition:

⟨identifier⟩ ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩⟨digit⟩|
0⟨identifier⟩| 0⟨unsigned integer⟩⟨letter⟩

I personally have avoided most of the confusion between the two characters by attempting never to use either character in mnemonic symbols unless it has some mnemonic significance and it is difficult to take it to be the other. There are, however, some situations which are beyond the control of the programmer, but with which we are made to live; two of these are the FORTRAN internal functions, MAX0F and MIN0F. For situations such as these, Mr. Turner's solution seems somewhat appropriate.

MICHAEL L. PERSHING
Department of Computer Science
University of Illinois
Urbana, Illinois 61803