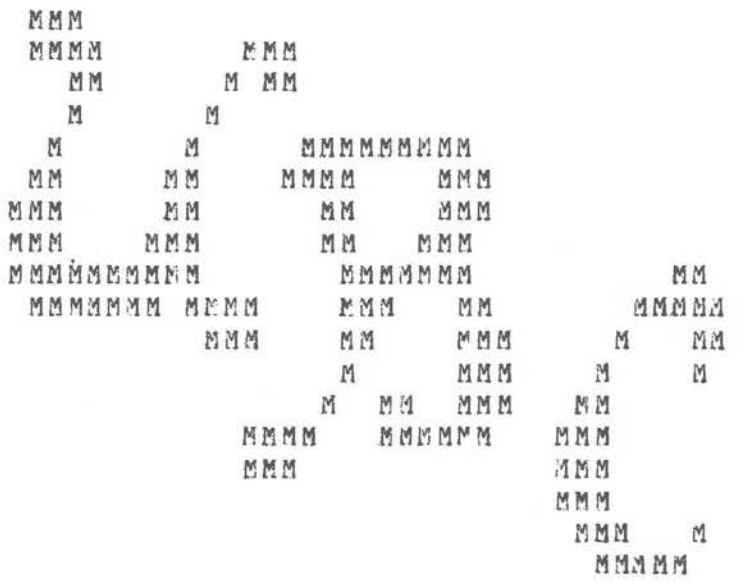


Two Canonical Forms for Programs

by

J. L. Baker

Tecnical Report 76-6

September 1976

Department of Computer Science
University of British Columbia
Vancouver, B. C.

TWO CANONICAL FORMS FOR PROGRAMS

J. L. BAKER
Department of Computer Science
University of British Columbia

ABSTRACT

Since theories of computation provide (among other things) formal framework for practical program minimization, and since a distinction between program syntax and semantics can be based on the possiblity of performing such minimization algorithmically, it is reasonable to formulate the theory of computation as a theory of machines controlled by programs which are in themselves purely syntactic.

Here the algebraic structure of programs in such a theory is presented. Two canonical forms are exhibited, characterizing respectively strong (syntactic) and weak (computational) equivalence of programs.

0. Introduction

In this paper, I expose the outlines of a theory of programmable machines along the lines suggested by Scott (1967), and develop fully the part of that theory dealing only with programs. As is usual and natural, I say two programs are equivalent if the sets of computations they control are equal. The "canonical forms" mentioned in the title are defined so that each canonical-form program is terminal, with respect to homomorphisms of programs, in the subalgebra of programs equivalent to it. There are two canonical forms because the definition of equivalence may be based on all computations or just on terminating computations.

Since my presentation is concrete (programs being based on finite directed graphs), it follows that these canonical-form programs are minimal in the sense that they contain as few instructions as possible. Since "program" is used here in a purely syntactic sense, the minimizing constructions I offer are substantially the same as those, first given more than fifteen years ago, for the minimization of uninterpreted program schemes. The purpose of the present paper is to clarify the systematic position of such constructions in the theory of computation. Their position in the theory of programmable machines outlined here is made explicit in the body of this paper. In the rest of these introductory remarks, I offer the reader grounds to accept that theory as a framework for a more general theory of computation.

One of the earliest objectives to arise in the use of programmable high-speed electronic digital computers was that their programs should occupy as little memory space as possible. In the early 1950's, when machine-language programs were prepared by hand, such minimization was a matter of ingenuity. It was, however, apparent that it should become merely a matter of skill-- that programmers should be able to use some sort of algebra of programs in a more or less routine way to achieve much or all of the minimization possible. It was intuitively clear what should comprise the <u>domain</u> of an algebra of programs, that is, what should be its constants and for what its variables should stand: namely, the atomic operations of computing--either the instructions in a machine's own

repertoire, or the operations performed by standard or arbitrary subroutines. It was likewise clear that the laws of composition of such an algebra should represent the sequencing, selection, and repetition of computing operations. What remained to be seen was the content of such an algebra--its laws, embodying useful manipulative techniques for passing from one expression to another in search of a minimal member of a family of equivalent programs.

In the U.S.S.R. during the period 1953:1965, a good deal of thought was given to the development of such an algebra of programs (Trakhtengerts, 1967). At the beginning of this period, Lyapunov propounded a notation for its formulas, which were called "logical schemes". Many workers then studied optimizing transformations of these schemes, mainly in restricted applications, to particular machine architectures or to the computational operations and sequences peculiar to linear algebra, for example. The problem of minimizing schemes while making no assumptions concerning the nature of their atomic operations--of minimizing completely uninterpreted schemes, that is--was resolved by Yanov (1960).

During the same period in the West, much less attention was paid to the problem of program size minimization, and essentially none to the development of an algebra of programs. This can be seen as part of the complex involving the more rapid expansion of computing in the West: The use of high-level languages (relative to machine language) required large-scale machines and programs for their translation. It also permitted the creation of many programs and of large programs, reinforcing the need for large machines. The existence of large-scale machines made program size minimization seem unimportant. The existence of many programs and of large programs discouraged efforts toward minimization as impossible to be done by hand, deeply difficult to be done automatically. Furthermore, the use of high-level languages eliminated the need for an algebra of programs as a direct aid to programming.

Nevertheless, a problem equivalent to the minimization of uninterpreted schemes, namely the state minimization of the "machines" of Moore (1956), did appear in the West (as well as in the U.S.S.R.), and was resolved by

Nerode (1958). A clear and refined presentation of this result was given by Rabin and Scott (1959), a watershed for automata theory. Its connection with Yanov's work was pointed out by Rutledge (1963).

We may fairly describe the problem to whose resolution I have just alluded as that of the minimization of a program using only syntactic information. Since 1960, there has been some research into weaker notions of program equivalence, taking into account such semantic information as independence of instructions within a program*, but theoretical studies have not offered any hope for automatic program minimization in such terms. In particular, Luckham, Park, and Patterson (1970) shows that equivalence is undecidable for program schemes with free variables. Similarly, there are no types of infinite-state automata for which the equivalence problem is known to be algorithmically solvable.

It seems to me that these results, identifying finite-state control structures and uninterpreted program schemes as the class of computational schemes with decidable equivalence, consititute the discovery of an invariant--so fundamental an insight that it should be represented in a theory of computation as an aspect of definitional structure. Usual formal presentations of automata theory do not respect this insight, nor does the "schematology" research following Luckham, Park, and Patterson (1970). The former fuses finite and infinite memory components in specifying transition functions, and the latter provides at least minimal semantics for at least some instructions. In contrast, the suggestion to represent computational schemes mathematically as programs for machines, made by Scott (1967) and followed here, does respect this insight.

One would hope that the systematic position of the purely syntactic would appear fairly clearly in more abstract presentations of the theory of computation. This hope is satisfied by Elgot (1975), where it is shown

---

* x:=0; y:=0 are independent, x:=0; x:=1 are not.

that any "sequacious function" (function computable in a highly abstract sense) admits a "normal description" (counterpart to "program" as used here). However, connection with minimization does not appear. On the other hand, minimization is the focus of Arbib and Manes (1974), which presents sufficient conditions for minimization of "machines" as defined in a very general, category-theoretic setting. It turns out that this notion of machine is general enough to include "program" as used here, and that the minimization presented here does specialize the Arbib-Manes construction* However, connection with decidable equivalence (effective minimization) does not appear. In the discussions of Goguen (1974) and Goguen, Thatcher, Wagner, and Wright (1975), syntax appears merely as the complete trivialization of semantics.

To my way of thinking, then, the consequences of (the universally assumed) finite programmatic control of computation are not adequately shown in any of the abstract presentation just mentioned, or in any others I have met. This is to be expected, considering that these are, properly speaking, theories of computable functions rather than of computation. The support such theories offer the development of a usable (not necessarily effective) algebra of programs is certainly less direct than that offered by a theory of programmable machines, and is, to my taste, too indirect altogether.

To summarize: An important motivation for theoretical studies of computation has been the need for an algebra of programs--the tangible specificiations on which practical computations are based. In itself, an algebra of programs must be, like any algebra, purely syntactic--a matter of form. It will be useful, of course, only if it accurately reflects the semantics of programs--the results of the computations they specify. Theoretical studies have succeded in the syntactic domain, even so far as to characterize its limits intrinsically. It is therefore reasonable to formulate a theory of computation in such a way that syntax and semantics appear as distinct non-trivial components.

---

* As suggested by its being called "Nerode realization" there.

Finally, I mention the pedagogical value of a programmable-machine approach to the theory of computation: Practical experience is likely to be the strongest reason for a student to seek the insights which theoretical study can offer. He will proceed most quickly and securely if the formulation he uses generalizes his experience fairly directly. My teaching, both of undergraduate and graduate students, has had much benefit from the use of a programmable-machine approach in presenting a variety of theoretical topics in a uniform and intuitively well-motivated manner. I confess, in fact, that I am seeking the reader's attention here not only for the sake of the present modest results, but also to introduce him in detail to a theory whose presentation I intend to continue in further papers.

The following notation is used here: if X is a set and R an equivalence relation in X, then X/R denotes the quotient set, and, for $x \in X$, $[x]_R$ denotes the R-equivalence class of x. Card X denotes the Cardinality of X, $X \setminus Y$ denotes the difference $\{x \in X | x \in Y\}$. $\square$ denotes the empty set.

If A is a set, A^* denotes the set of strings (terminating sequences) in A. For $x \in A^*$, $|x|$ denotes the length of x (number of componentes). $(x]$ denotes the string obtained by deleting the first component, if any, of x, $[x)$ the string obtained by deleting the last component, if any, of x. $\langle\rangle$ denotes the empty string. $([\langle\rangle)=(\langle\rangle]=\langle\rangle$.)

"Function" here means "partial function". Specifically, $f:X \to Y$ means that f is a function (unambiguous relation) defined for some members of the set X and taking values in Y. dom $f = \{x \in X | f(x)$ is defined$\}$. Ran $f=\{y \in Y | y=f(x)$ for some $x \in X\}$. As usual, the barred arrow specifies a function by its action on an element. $x \mapsto y$ means (in the proper context) $y=f(x)$.

I am grateful for the financial assistance of the National Science foundation of the United States* and the National Research Council of Canada† in developing the material presented here, as well as for the patience and interest of my students at the Universities of Calgary and British Columbia.

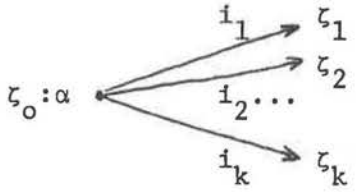
---

* Grant GJ-66, administered by Dr. Hellmut Golde.

† Grant A7882.

1. A theory of programmable devices

Informally, this theory concerns devices which execute programs. A program for a device $\mathfrak{D}$ is a finite directed graph with labeled edges and nodes. (Loops and multiple edges are permitted.) A typical node ζ_o has the appearance



Here, the label* α on ζ_o is one of a set of commands valid for $\mathfrak{D}$, and the labels i_1, i_2 ..., i_k on edges directed from ζ_o to ζ_1, ζ_2 ..., ζ_k are distinct elements of a set $\mathfrak{D}_V(\alpha)$, the valence of α. A computation by Π on $\mathfrak{D}$ is a sequence of pairs $<\zeta,m>$, where ζ is a node of Π and m is an element of $\mathfrak{D}_Q$, the memory set of $\mathfrak{D}$. Associated with each command α is a partial function $\mathfrak{D}_\alpha:\mathfrak{D}_Q \to \mathfrak{D}_Q \times \mathfrak{D}_V(\alpha)$. Pairs $<\zeta,m>$, $<\zeta',m'>$ can occur consecutively in a computation by Π on $\mathfrak{D}$ only if Π includes a node ζ_o as above, $\zeta=\zeta_o$, and $\mathfrak{D}_\alpha(m)=<m',i_j>$ and $\zeta'=\zeta_j$ for some j. Execution of the program step specified at ζ_o, then, comprises modification of the memory configuration of $\mathfrak{D}$ and (deterministic) selection of the next program step from those specified at ζ_1, ζ_2 ..., ζ_k.

The specification of a device $\mathfrak{D}$ also includes an input set $\mathfrak{D}_S$, and input function $\mathfrak{D}_I:\mathfrak{D}_S \to \mathfrak{D}_Q$, an output set $\mathfrak{D}_T$, and an output function $\mathfrak{D}_O:\mathfrak{D}_Q \to \mathfrak{D}_T$. A program node is terminal if it is not labelled with any command (and has no edges leading away from it). Each program Π has a specified start node Π_S. The

---

* Unfortunately, the sense in which the word "label" is used in the study of graphs conflicts with the sense in which it is used in the study of programming. To those who study graphs, the word denotes something that can occur more than once, like a particular opcode in a machine-language program. To those who study programming, the word denotes something which cannot meaningfully occur more than once, like (the name of) a particular node in a graph. I use "label" in the former sense in the first two paragraphs of this section, and avoid using it in the rest of the paper. It is good to think of the ζ_i as labels in the programming sense.

partial function computed by Π on $\mathfrak{D}$, $\mathfrak{D}_\Pi:\mathfrak{D}_S\to\mathfrak{D}_T$, is determined thus: For $x\in\mathfrak{D}_S$, if there is a computation by Π on $\mathfrak{D}$ starting with $\langle\Pi_S,\mathfrak{D}_I(x)\rangle$ and ending with some $\langle\zeta,m\rangle$ with ζ terminal, then $\mathfrak{D}_\Pi(x)=\mathfrak{D}_O(m)$. Otherwise, $\mathfrak{D}_\Pi$ is not defined at x.

The following is a more formal statement of the basic definitions for this theory.

1.01. A program Π comprises the following:

Π_Q , a finite set, the nodes;

$\Pi_S \in \Pi_Q$, the start node;

Π_A , a partial function with $\mathrm{dom}\Pi_A\subset\Pi_Q$, the action function;

Π_B , a partial function with $\mathrm{dom}\,\Pi_B\subset\mathrm{dom}\Pi_A\times U$ for some finite set U, and with $\mathrm{ran}\Pi_B\subset\Pi_Q$, the branching function.

It is also convenient to define

$\Pi_T=\Pi_Q\setminus\mathrm{dom}\Pi_A$, the terminal nodes;

$\Pi_C=\mathrm{ran}\Pi_A$, the commands;

$\Pi_V(\zeta)=\{i\,|\,\langle\zeta,i\rangle\in\mathrm{dom}\Pi_B\}$, the valence of ζ, defined for each $\zeta\in\Pi_Q$;

$\Pi_U=\cup\{\Pi_V(\zeta)\,|\,\zeta\in\Pi_Q\}$, the unified set of valences.

For example, supposing the node ζ_o illustrated in the first paragraph of this section to occur in a program Π, we have $\Pi_A(\zeta_o)=\alpha$ and $\Pi_B(\zeta_o,i_1)=\zeta_1$, etc.

The usual dots-and-arrows notation for directed graphs is convenient for specifying programs. For a node ζ in a program Π: to specify that $\Pi_A(\zeta)=\alpha$, write "$\zeta:\alpha$"(or just "α" if no reference to ζ is needed) near the dot representing ζ; to specify that ζ is a terminal node, write "ζ:" or nothing near its dot; to specify that $\Pi_B(\zeta,i)=\zeta'$, write "i" near the arrow representing the appropriate edge $\langle\zeta,\zeta'\rangle$. Designate Π_S by putting its dot at the head of an arrow with nothing at its tail. To avoid graphic inconvenience, use an arrow with "ζ", but no dot at its head to indicate that that arrow is to be taken as ending at the dot

for node ζ.

1.02. A device $\mathfrak{D}$ comprises the following:

$\mathfrak{D}_Q$, $\mathfrak{D}_S$, $\mathfrak{D}_T$, sets, the memory, input, and output sets;

$\mathfrak{D}_I: \mathfrak{D}_S \to \mathfrak{D}_Q$, $\mathfrak{D}_O: \mathfrak{D}_Q \to \mathfrak{D}_T$, partial functions, the input and output functions;

$\mathfrak{D}_C$, a set, the commands;

$\mathfrak{D}_G$, a partial function with $\operatorname{dom} \mathfrak{D}_G \subset \mathfrak{D}_C \times \mathfrak{D}_Q$ and $\operatorname{ran} \mathfrak{D}_G \subset \mathfrak{D}_Q \times U$ for some set U, the general interpretation.

it is also convenient to define, for each $\alpha \in \mathfrak{D}_C$,

$\mathfrak{D}_V(\alpha) = \{i \mid \mathfrak{D}_G(\alpha,m) = \langle m', i\rangle$ for some m, m'$\}$, the valence of α;

$\mathfrak{D}_\alpha: \mathfrak{D}_Q \to \mathfrak{D}_Q \times \mathfrak{D}_V(\alpha): m \mapsto \mathfrak{D}_G(\alpha,m)$, the interpretation of α.

1.03. If Π is a program and $\mathfrak{D}$ a device, then $\mathcal{C}(\Pi,\mathfrak{D})$, the set of computations by Π on $\mathfrak{D}$, is the set of sequences $\langle\zeta_o,m_o\rangle \ldots \langle\zeta_n,m_n\rangle$ in $\Pi_Q \times D_Q$ in which, for all $j \in \{1,2\ldots,n\}$, $\mathfrak{D}_{\Pi_A(\zeta_{j-1})}(m_{j-1}) = \langle m_j, i\rangle$ and $\Pi_B(\zeta_{j-1},i) = \zeta_j$ for some $i \in \Pi_V(\zeta_{j-1})$.

$\mathcal{C}_T(\Pi,\mathfrak{D})$, the set of terminating computations by Π on $\mathfrak{D}$, is the set of sequences $\langle\zeta_o, m_o\rangle \ldots \langle\zeta_n,m_n\rangle$ as above in which $\zeta_n \in \Pi_T$.

The length of a computation of the above form is n.

1.04. Lemma. If Π is a program, $\mathfrak{D}$ a device, and $\langle\zeta_o,m_o\rangle \in \Pi_Q \times \mathfrak{D}_Q$, then there is at most one sequence $\langle\zeta_o,m_o\rangle \ldots \langle\zeta_n,m_n\rangle$ in $\mathcal{C}_T(\Pi,\mathfrak{D})$.

1.05. If Π is a program and $\mathfrak{D}$ a device, then $\mathfrak{D}_\Pi$, the function computed by Π on $\mathfrak{D}$, is a partial function defined thus:

$$\mathfrak{D}_\Pi: \mathfrak{D}_S \to \mathfrak{D}_T: x \mapsto \mathfrak{D}_O(m),$$

where $\langle\Pi_S, \mathfrak{D}_I(x)\rangle \ldots \langle\zeta,m\rangle \in \mathcal{C}_T(\Pi,\mathfrak{D})$ (uniquely, by (1.04)).

To illustrate the above definitions, we may consider the following:

1.06. Example. Specify a device IXP by: $IXP_Q=\{a,b,\dashv\}^*\times\{a\}^*$, $IXP_S=\{a,b\}^*$, $IXP_T=\{0\}$, $IXP_I: x \mapsto \langle x\dashv,\langle\rangle\rangle (x\in\{a,b\}^*)$, $IXP_O: \langle\langle\rangle,\langle\rangle\rangle \mapsto 0$ (undefined otherwise), $IXP_C=\{\delta, P\leftarrow a, \leftarrow P\}$, and IXP_G is given in the following table, which also exhibits IXP_α and $IXP_V(\alpha)$ for $\alpha\in IXP_C$:

α	$IXP_G(\alpha,\langle x,y\rangle) = IXP_\alpha(x,y)$	$IXP_V(\alpha)$
δ	$\langle\langle(x],y\rangle,c\rangle$ if $x=c(x],c\in\{a,b,\dashv\}$ (undefined if $x=\langle\rangle$)	$\{a,b,\dashv\}$
$P\leftarrow a$	$\langle\langle x,ay\rangle,0\rangle$	$\{0\}$
$\leftarrow P$	$\langle\langle x,(y]\rangle,a\rangle$ if $y=a(y]$ (undefined if $y=\langle\rangle$)	$\{a\}$

The name IXP is intended to suggest that this device is the product* of a (one-way) input device and a pushdown-store device. The first component of an element of its memory set is the unconsumed portion of an input string, and the second component is the pushdown store, top at left. IXP interprets commands thus: δ - consume one input symbol and branch accordingly (this is a traditional finite-state-acceptor state transition); $P\leftarrow a$ - push an "a" onto the stack; $\leftarrow P$ - pop one stack symbol. The input function of IXP provides an endmark $\dashv$ for the input and initializes the stack to $\langle\rangle$.

Since the output function of IXP provides no information, the only possible interest in functions computable on IXP concerns their domains: For each program Π, $dom(IXP_\Pi)$, the set of inputs for which Π halts on IXP, is a language over $\{a,b\}$. Thus, programs for IXP are the counterparts in this theory of the traditional deterministic one-way pushdown acceptors, In particular, $\{dom(IXP)_\Pi \mid \Pi$ is a program$\}$ is exactly the set of languages over $\{a,b\}$ which are

---

* The notion of product of devices is not formally developed here. It has been presented in a preliminary way in Baker (1975).

deterministically acceptable by traditional one-way-input pushdown-store machines.

Figure 1 exhibits a program Δ such that $dom(IXP_\Delta)=\{a^ib^i|i\varepsilon\{0,1,2,...\}\}$. In terms of definition 1.01, we have $\Delta_Q=\{1,2,3,4,5,6,7\}$; $\Delta_S=2$; $\Delta_A(1)=P\leftarrow a$, $\Delta_A(2)=\delta$, etc.; $\Delta_B(1,0)=2$, $\Delta_B(2,a)=1$, $\Delta_B(2,\dashv)=6$, etc.; $\Delta_T=\{6,7\}$; $\Delta_C=IXP_C$; $\Delta_V(1)=\{0\}$, $\Delta_V(2)=\{a,b,\dashv\}...$, $\Delta_V(4)=\{b,\dashv\}...$, $\Delta_V(6)=\Delta_V(7)=\square$

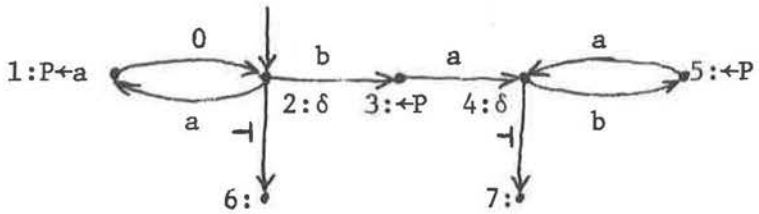


Figure 1. A program Δ such that $IXP_\Delta = \{a^ib^i|i\varepsilon\{0,1,2,...\}\}$.

(i)	(ii)	(iii)	(iv)
<2,<aabb⊣,<>>>	<2,<aab⊣,<>>>	<2,<abb⊣,<>>>	<2,<aba⊣,<>>>
<1,<abb⊣,<>>>	<1,<ab⊣,<>>>	<1,<bb⊣,<>>>	<1,<ba⊣,<>>>
<2,<abb⊣,a>>	<2,<ab⊣,a>>	<2,<bb⊣,a>>	<2,<ba⊣,a>>
<1,<bb⊣,a>>	<1,<b⊣,a>>	<3,<b⊣,a>>	<3,<a⊣,a>>
<2,<bb⊣,aa>>	<2,<b⊣,aa>>	<4,<b⊣,<>>>	<4,<a⊣,<>>>
<3,<b⊣,aa>>	<3,<⊣,aa>>	<5,<⊣,<>>>	
<4,<b⊣,a>>	<4,<⊣,a>>		
<5,<⊣,a>>	<7,<<>,a>>		
<4,<⊣,<>>>			
<7,<<>,<>>>			

Figure 2. Four elements of $\mathcal{C}(\Delta,IXP)$

Figure 2 exhibits four elements of $\mathcal{C}(\Delta,IXP)$. Each begins with $<\Delta_S,IXP_I(x)>$ for some $x\varepsilon IXP_S$, and each is maximal in the sense that it ends with a configuration $<\zeta_{j-1},m_{j-1}>$ for which there is no $<\zeta_j,m_j>$ satisfying the conditions in (1.03). (i) is terminating and, since $<<>,<>>\varepsilon dom(IXP_0)$, shows aabb ε dom IXP_Δ. (ii) is also terminating, but $<<>,a>\notin dom(IXP_0)$. (iii) is maximal

because $\langle \dashv, \langle\rangle \rangle \notin \mathrm{dom}(IXP_{\leftarrow P}) = \mathrm{dom}(IXP_{\Delta_A(5)})$. (iv) is maximal because $IXP_{\Delta_A(4)}(a\dashv, \langle\rangle) = IXP_{\delta}(a\dashv, \langle\rangle) = \langle\langle \dashv, \langle\rangle\rangle, a\rangle$ and $\langle 4, a\rangle \notin \mathrm{dom}\Delta_B$. (ii,iii,iv) thus show $\{aab, abb, aba\} \cap \mathrm{dom} IXP_{\Delta'} = \square$.

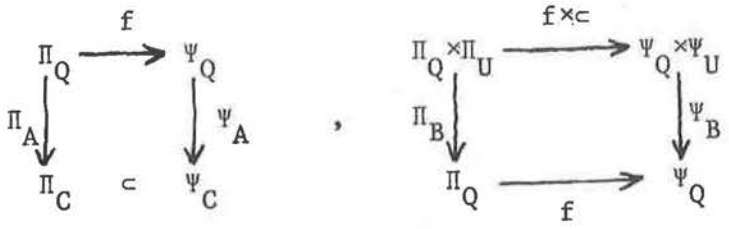
2. Homomorphism of programs, strong equivalence, first canonical form.

It is natural to explore the consequences of the above definitions by a study of homomorphisms - functions which preserve the structure of programs or machines. In the present paper, we study homomorphisms of programs only. The following definition seems the natural one:

2.01. If Π and Ψ are programs, then a function $f:\Pi_Q \to \Psi_Q$ is a homomorphism if and only if

(i) $\text{dom } f=\Pi_Q$;
(ii) $f(\Pi_S)=\Psi_S$;
(iii) $\Psi_A(f(\zeta))=\Pi_A(\zeta)$ for $\zeta\varepsilon\Pi_Q$;
(iv) $\Psi_B(f(\zeta),i)=f(\Pi_B(\zeta,i))$ for $\zeta\varepsilon\Pi_Q$, $i\varepsilon\Pi_U\cup\Psi_U$.

The equations in (iii,iv) are to be understood thus: either both sides are defined and the equation holds, or neither side is defined. Thus (iii) implies $f(\Pi_T)\subset\Psi_T$, and (iv) implies $\Psi_V(f(\zeta))=\Pi_V(\zeta)$ for all $\zeta\varepsilon\Pi_Q$. It is also implied that $\Pi_C\subset\Psi_C$, $\Pi_U\subset\Psi_U$. (iii,iv) may equivalently be specified by requiring that the following diagrams commute, where it is understood that the composites must be equal as partial functions.



The notion of program structure which underlies this definition is very restrictive. In fact, a program's individuality, that of it which is not preserved by homomorphism, may be characterized in a word as its redundancy. Consider, for instance, the following:

2.02. Example. Let Θ be the program of figure 3.

Define $g: \Delta_Q \to \Theta_Q : 5 \mapsto 4$, $i \mapsto i$ if $i \neq 5$, where Δ is the program of example 1.06. Then g is a homomorphism exhibiting the redundancy of {4,5} in Δ.

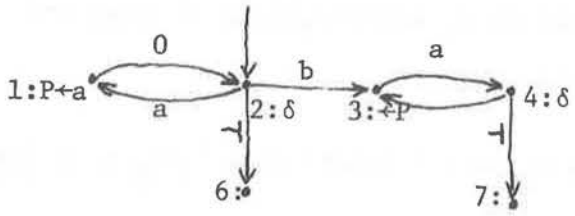
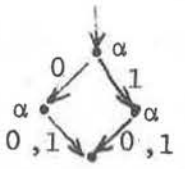
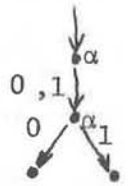


Figure 3. A program Θ, homomorphic image of Δ (figure 1.).

It is clear that homomorphism preserves computations - that is, that if $f:\Pi_Q \to \Psi_Q$ is a homomorphism, then $\{<f(\zeta_o),m_o> \ldots <f(\zeta_n), m_n> \mid <\zeta_o,m_o> \ldots <\zeta_n,m_n> \in \mathcal{C}(\Pi,\mathfrak{B})\} \subset \mathcal{C}(\Psi,\mathfrak{B})$ for any device $\mathfrak{B}$, and in particular $\mathfrak{B}_\Psi = \mathfrak{B}_\Pi$. The converse is not true, however. Consider:

2.03. Example. $\Pi =$  , $\Psi =$ 

There is no homomorphism $\Pi \to \Psi$ or $\Psi \to \Pi$, but, for any device $\mathfrak{D}$, $\mathfrak{D}_\Pi = \mathfrak{D}_\Psi = \mathfrak{D}_\cap \circ \phi \circ \phi \circ \mathfrak{D}_I$, where $\mathfrak{D}_\alpha : m \mapsto <\phi(m), i>$.

By these considerations, we are led to seek a notion of structural equivalence of programs - stronger than homomorphism - which will be logically equivalent to the correspondence of computations suggested by (2.03). The following development is natural:

2.04. If Π is a program, define

the extended branching function $\Pi_{\overline{B}}: \Pi_Q \times \Pi_U^* \to \Pi_Q$ recursively thus: $\Pi_{\overline{B}}(\zeta,<>)=\zeta$; $\Pi_{\overline{B}}(\zeta,xi)=\Pi_B(\Pi_{\overline{B}}(\zeta,x),i)$ for $i \in \Pi_U$.

Define the extended action function $\Pi_A: \Pi_Q \times \Pi_U^* \to \Pi_C^*$ recursively thus: $\Pi_{\overline{A}}(\zeta,<>)=<>$; $\Pi_{\overline{A}}(\zeta,xi)=\Pi_{\overline{A}}(\zeta,x)\Pi_A(\Pi_{\overline{B}}(\zeta,x))$ if $i \in \Pi_V(\Pi_{\overline{B}}(\zeta,x))$.

Corollary. If Π is a program, then, for all $\zeta\varepsilon\Pi_Q, x, y\varepsilon\Pi_U^*$,

(i) $\Pi_{\bar{B}}(\zeta,xy)=\Pi_{\bar{B}}(\Pi_{\bar{B}}(\zeta,x),y)$;

(ii) $\Pi_{\bar{A}}(\zeta,xy) =\Pi_{\bar{A}}(\zeta,x)\Pi_{\bar{A}}(\Pi_{\bar{B}}(\zeta,x),y)$. Also,

(iii) $\text{dom}\ \Pi_{\bar{A}}=\text{dom}\Pi_{\bar{B}}$.

2.05. If Π and Ψ are programs, then Π is strongly equivalent to Ψ if and only if, for all $x\varepsilon U^*$, $\Pi_A(\Pi_{\bar{B}}(\Pi_S,x))=\Psi_A(\Psi_{\bar{B}}(\Psi_S,x))$ and $\Pi_{\bar{B}}(\Pi_S,x)\varepsilon\Pi_T \Leftrightarrow \Psi_{\bar{B}}(\Psi_S,x)\varepsilon\Psi_T$, where $U=\Pi_U\cup\Pi_U$.

Corollary. If Π and Ψ are strongly equivalent programs, then $\{x|\langle\Pi_S,x\rangle\ \varepsilon\text{dom}\Pi_{\bar{B}}\}=\{x|\langle\Psi_S,x\rangle\ \varepsilon\text{dom}\Psi_{\bar{B}}\}$. Also, strong equivalence of programs is an equivalence relation.

We verify that strong equivalence follows from homomorphism:

2.06. Theorem. If Π and Ψ are programs and there is a homomorphism $f: \Pi_Q\to\Psi_Q$, then Π is strongly equivalent to Ψ.

Proof: Let $x\varepsilon(\Pi_U\cup\Psi_U)^*, y\varepsilon(\Pi_U\cup\Psi_U\cup\{\dashv\})^*$.

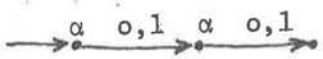
By the definitions involved (2.01 and 2.04) and an easy induction, we have:

$$\begin{aligned}\Psi_A(\Psi_{\bar{B}}(\Psi_S,x)) &=\Psi_A(\Psi_{\bar{B}}(f(\Pi_S),x))\\ &=\Psi_A(f(\Pi_{\bar{B}}(\Pi_S,x)))\\ &=\Pi_A(\Pi_{\bar{B}}(\Pi_S,x)).\end{aligned}$$

As already remarked, (2.03) is a counter-example to the converse of (2.06).

Another apporach to the question of structural equivalence of programs appears upon consideration that a program, as defined here, can be taken to be the transition diagram for a Moore-type finite state machine with output (the commands serving as outputs). As such, it is subject to the Nerode optimization process (NOp), the result of applying which to a program Π is a canonical form $\tilde{\Pi}$. It turns out, not unexpectedly, that these canonical forms have the property that programs Π and Ψ are strongly equivalent if and only if

$\tilde{\Pi}=\tilde{\Psi}$. For example, the programs of (2,03) have the common canonical form



A parenthetical remark is in order here. The traditional definition of a species of machine is informally motivated by a diagram like Figure 4.

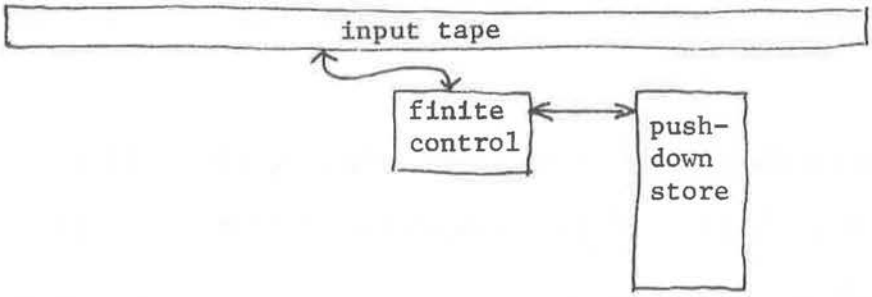


Figure 4. Informal motivation for traditional definition of deterministic pushdown acceptor.

The "finite control" of machines of a traditionally-defined species is supposed somehow to be a finite-state machine, so, one would expect, subject to NOp. Unfortunately, traditional definitions disable any easy "factoring out" of non-finite parts such as would be necessary to apply NOp directly. A virtue of the formulation presented in this paper (this aspect of it due directly to Scott, 1967) is that is specifies such a factorization at the outset. In our view, the "finite control" is a program. The rest of the structure, including any non-finite part, is represented by a device. The resulting applicability of NOp is not itself of any great significance, but it is symptomatic of a clarity and intuitive appeal inherent in our view.

We turn now to the construction of the canonical form obtained by NOp, and to verification of its relation to strong equivalence.

2.07. If Π is a program, define the canonical equivalence induced by Π, a binary relation Π_E in Π_U^*, by $x\Pi_E y$ if and only if

$$\Pi_A(\Pi_{\bar{B}}(\Pi_S, xz))=\Pi_A(\Pi_{\bar{B}}(\Pi_S, yz)) \text{ and } \Pi_{\bar{B}}(\Pi_S, xz)\varepsilon\Pi_T \Leftrightarrow \Pi_{\bar{B}}(\Pi_S, yz)\varepsilon\Pi_T \text{ for all } z\varepsilon\Pi_U^*.$$

Corollary. If Π is a program, then Π_E is an equivalence relation and satisfies

(i) If $x\Pi_E y$ and $z\varepsilon\Pi_U^*$, then $xz\Pi_E yz$ (Π_E is right-invariant);
(ii) Π_U^*/Π_E is finite;
(iii) If $x\Pi_E y$, then $\Pi_V(\Pi_{\bar{B}}(\Pi_S,x))=\Pi_V(\Pi_{\bar{B}}(\Pi_S,y))$;
(iv) If $x\Pi_E y$, then $\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,x),z)=\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,y),z)$ for all $z\varepsilon\Pi_U^*$.

2.08. If Π is a program, then the first canonical form of Π is a program $\tilde{\Pi}$ specified as follows (writing $[x]$ for $[x]_{\Pi_E}\varepsilon\Pi_U^*/\Pi_E$):

$\tilde{\Pi}_Q=\{[x]\,|\,<\Pi_S,x>\varepsilon\mathrm{dom}\Pi_{\bar{B}}\}$.
$\tilde{\Pi}_S=[<>]$.
$\tilde{\Pi}_A:[x]\mapsto \Pi_A(\Pi_{\bar{B}}(\Pi_S,x))$.
$\tilde{\Pi}_B:<[x],i>\mapsto [xi]$ if $i\varepsilon\Pi_V(\Pi_{\bar{B}}(\Pi_S,x))$.

By the corollary to (2.07), $\tilde{\Pi}$ is well-defined.

Corollary. If Π is a program, then, for all $x,y\varepsilon\Pi_U^*$ (again writing $[x]$ for $[x]_{\Pi_E}$),

(i) $\tilde{\Pi}_{\bar{B}}([x],y)=[xy]$ if $[xy]\varepsilon\tilde{\Pi}_Q$, undefined otherwise;
(ii) $\tilde{\Pi}_{\bar{A}}([x],y)=\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,x),y)$. Also,
(iii) $\tilde{\Pi}_T=\begin{cases}\square \text{ if, for all } x\varepsilon\Pi_U^*,\ \Pi_{\bar{B}}(\Pi_S,x)\notin\Pi_T\\ \{\{x\,|\,\Pi_{\bar{B}}(\Pi_S,x)\varepsilon\Pi_T\}\} \text{ otherwise.}\end{cases}$

2.09. Examples. With respect to (2.03), we have $\tilde{\Pi}=\tilde{\Psi}=$

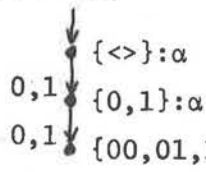


Figure 3 exhibits the first canonical form of Δ of (1.06) and Θ of (2.02)

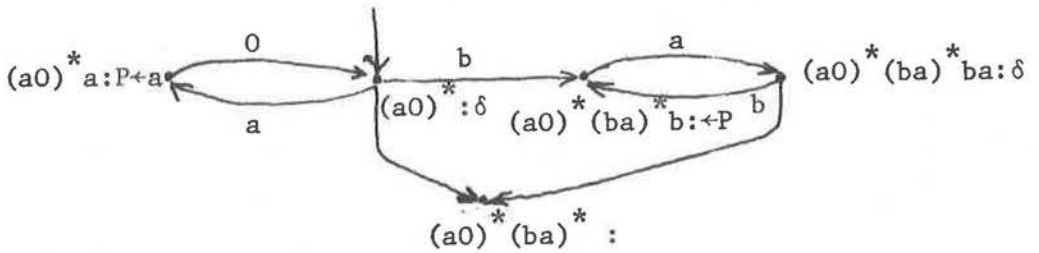


Figure 3. $\tilde{\Delta}=\tilde{\Theta}$.

2.10. Theorem. If Π and Ψ are programs, then Π is strongly equivalent to Ψ if and only if $\tilde{\Pi}=\tilde{\Psi}$

Proof: Suppose Π strongly equivalent to Ψ. Then clearly $x\Pi_E y$ if and only if $x\Psi_E y$, for all $x,y\in(\Pi_U\cup\Psi_U)^*$. By the corollary to (2.05), then, $\Pi_E=\Psi_E$ and $\tilde{\Pi}_Q=\tilde{\Psi}_Q$. It follows that $\tilde{\Pi}=\tilde{\Psi}$.

Conversely, suppose $\Pi=\Psi$, and let $x\in(\Pi_U\cup\Psi_U)^*$. For $\Xi=\Pi$ or Ψ, $\Xi_{\bar{B}}(\Xi_S,x)$ is defined only if $x\in\Xi_U^*$, and in that case $\Xi_A(\Xi_{\bar{B}}(\Xi_S,x))=\tilde{\Xi}_A(\tilde{\Xi}_{\bar{B}}(\tilde{\Xi}_S,x))$ and $\Xi_{\bar{B}}(\Xi_S,x)\in\Xi_T \Leftrightarrow \tilde{\Xi}_{\bar{B}}(\tilde{\Xi}_S,x)\in\tilde{\Xi}_T$.

It follows that Π is strongly equivalent to Ψ.

Corollary. If Π and Ψ are programs and there is a homomorphism $f:\Pi_Q\to\Psi_Q$, then $\tilde{\Pi}=\tilde{\Psi}$.

To complete section 2, we establish two important properties of first canonical form, and exhibit algorithms realizing NOp in terms of our formulation.

2.11. Lemma. If Π is a program and Π' is defined thus:

$\Pi'_Q=\{\Pi_B(\Pi_S,x)\mid x\in\Pi_U^*\}$,

$\Pi'_S=\Pi_S$, and

Π'_A,Π'_B are the restrictions of Π_A to Π'_Q, Π_B to $\Pi'_Q\times\Pi_U$, respectively;

then Π' is a program and there is a unique homomorphism $f:\Pi'_Q\to\tilde{\Pi}_Q$ with ran $f=\tilde{\Pi}_Q$.

Proof: If $x,y\in\Pi_U^*$ and $\Pi_{\bar{B}}(\Pi_S,x)=\Pi_{\bar{B}}(\Pi_S,y)$, then $x\Pi_E y$ by definition of Π_E. Therefore $f:\Pi_{\bar{B}}(\Pi_S,x)\mapsto[x]_{\Pi_E}$ is well-defined as a function. Clearly Π', f are as required. If $g:\Pi'_Q\to\tilde{\Pi}_Q$ is any homomorphism, then by induction on $|x|$,

$g(\Pi_B(\Pi_{\bar{S}},x))=\tilde{\Pi}_B(\tilde{\Pi}_{\bar{S}},x)=[x]_{\Pi_E}$. Thus f is unique.

2.12. Theorem. If Π is a program, then $\tilde{(\tilde{\Pi})}=\tilde{\Pi}$.

Proof: Define Π' as in (2.11). Notice that $g:\Pi'_Q\to\Pi_Q:\zeta\mapsto\zeta$ is a homomorphism. Applying the corollary of (2.10) twice, we have $\tilde{(\tilde{\Pi})}=\tilde{(\Pi')}=\tilde{\Pi}$.

Corollary. If Π is a program, then Π is strongly equivalent to $\tilde{\Pi}$.

2.13. Theorem. If Π is a program and Ψ is a program strongly equivalent to Π, then card $\Psi_Q \geq$ card $\tilde{\Pi}_Q$.

Proof: By (2.10), $\tilde{\Psi}=\tilde{\Pi}$. Obtain Ψ' from Ψ as in (2.11). Then card $\Psi_Q \geq$ card $\Psi'_Q \geq$ card $\tilde{\Psi}_Q$=card $\tilde{\Pi}_Q$.

2.14. Algorithm. Given a program Π, to determine the set $\{\Pi_{\bar{B}}(\Pi_S,x)\,|\,x\in\Pi_U^*\}$.

Let $Q_o=\{\Pi_S\}$. For j=1,2,..., compute $Q_j=Q_{j-1}\cup\{\Pi_B(\zeta,i)\,|\,\zeta\in Q_{j-1}$ and $i\in\Pi_V(\zeta)\}$. Stop when $Q_j=Q_{j-1}$. Q_j is then the desired set.

2.15. Algorithm. Given a program Π, to determine the relation $\{<\zeta,\eta>\,|\,\Pi_A(\Pi_{\bar{B}}(\zeta,x))=\Pi_A(\Pi_{\bar{B}}(\eta,x))$ and $\Pi_{\bar{B}}(\zeta,x)\in\Pi_T \Leftrightarrow \Pi_{\bar{B}}(\eta,x)\in\Pi_T$ for all $x\in\Pi_U^*\}$ in Π_Q.

Let $R_o=\{<\zeta,\eta>\,|\,\Pi_A(\zeta)=\Pi_A(\eta)$ and $\Pi_V(\zeta)=\Pi_V(\eta)\}$. For j=1,2,..., compute $R_j=\{<\zeta,\eta>\,|\,\zeta R_{j-1}\eta$ and $\Pi_B(\zeta,i)R_{j-1}\Pi_B(\eta,i)$ for all $i\in\Pi_V(\zeta)\}$.

2.16. If Π and Ψ are programs, then $f:\Pi_Q\to\Psi_Q$ is a homomorphism, then f is an isomorphism and Π and Ψ are isomorphic if and only if f is 1-1 and ran f=Ψ_Q.

2.17. Algorithm. Given a program Π, to obtain a program Π' isomorphic to $\tilde{\Pi}$.

Apply (2.14) to Π to obtain $\Pi''_Q=\{\Pi_{\bar{B}}(\Pi_S,x)\,|\,x\in\Pi_U^*\}$. Let $\Pi''_S=\Pi_S$, and let Π''_A,Π''_B be the restrictions of Π_A to Π''_Q, Π_B to $\Pi''_Q\times\Pi_U$, respectively.

Apply (2.15) to Π'', obtaining a relation R.

Let $\Pi'_Q=\Pi''_Q/R,\Pi'_S=[\Pi_S]_R,\Pi'_A:[\zeta]_R\mapsto\Pi_A(\zeta),\Pi'_B:<[\zeta]_R,i>\mapsto[\Pi_B(\zeta,i)]_R$ for $i\in\Pi_V(\zeta)$.

2.18. Algorithm. Given a program Π, to obtain regular expressions for the sets $\{x\,|\,\Pi_{\bar{B}}(\zeta,x)=\zeta'\}$ for all $\zeta,\zeta'\in\Pi_Q$.

Let $\Pi_Q=\{\zeta_1,\zeta_2,\ldots,\zeta_n\}$ with card Π_Q=n.

For all p,q=1,2..., n, set $X_{pq}^{o}=\{i\,|\,\Pi_B(\zeta_p,i)=\zeta_q\}\cup\{<>\,|\,p=q\}$.

For $r=1,2\ldots,n$, compute $X^r_{pq}=X^{r-1}_{pq}\cup X^{r-1}_{pr}(X^{r-1}_{rr})^*X^{r-1}_{rq}$ for all $p,q=1,2\ldots,n$.

For all $p,q=1,2\ldots,n$, $X^n_{pq}=\{x\,|\,\Pi_{\underline{B}}(\zeta_p,x)=\zeta_q\}$, as required.

2.19. Algorithm. Given a program Π, to obtain $\tilde{\Pi}$, the elements of $\tilde{\Pi}_Q$ specified as regular expressions.

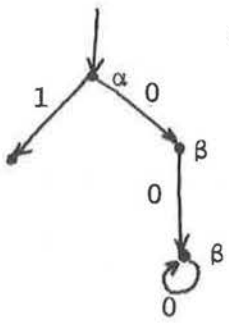
Apply (2.17) to obtain Π'. Apply (2.18) to Π'. $\tilde{\Pi}$ is determined by the isomorphism $\zeta \mapsto \{x\,|\,\Pi'_{\underline{B}}(\Pi'_S,x)=\zeta\}$ from Π'_Q to $\tilde{\Pi}_Q$.

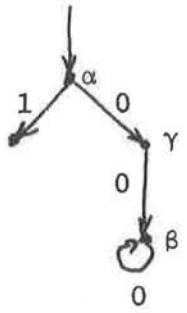
Proof that the above algorithms are as claimed is completely straightforward. (2.18) is a special case of algorithm 5.5 of Aho, Hopcroft, and Ullman(1974).

3. Weak equivalence of programs, some semantic considerations, second canonical form.

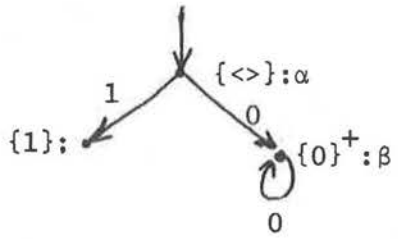
We have seen that the transformation $\Pi \mapsto \tilde{\Pi}$ is a program optimization, eliminating redundancy. It may reasonably be objected, however, that it is incomplete.

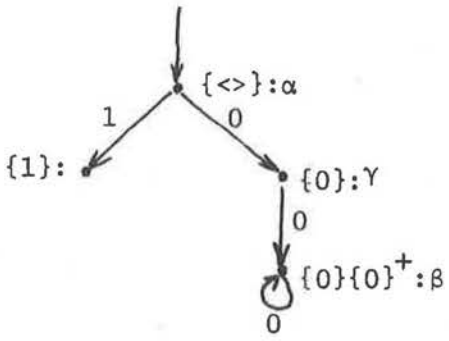
Consider:

3.01. Example. $\Pi =$ 

, $\Psi =$ 

.

We have $\tilde{\Pi} =$ 

, $\tilde{\Psi} =$ 

$\tilde{\Pi} \neq \tilde{\Psi}$ although the terminating computations by Π and Ψ are identical on any device. Furthermore, both $\tilde{\Pi}$ and $\tilde{\Psi}$ retain features that are redundant with respect to terminating computations: $\xrightarrow{\alpha}\bullet\xrightarrow{1}$ would seem a more suitable canonical form.

In defense of first canonical form, it can be said that its simplicity of definition and its close connection with classical finite-state-machine theory make it attractive. A more important defense is the fact that many real-world programs depend on interruption rather than termination for transfer of control away from them, so that non-terminating computations should not be overlooked in forming a general model for programs. In particular, recall that an operating

system is organized around a program which never terminates. Thus, consideration of examples like (3.01) should include interpretations such as

α - "interrupts disabled?"
β - "wait (for interrupt)"
γ - "set flag".

Under this interpretation, Π is non-optimal, Ψ is optimal, and the distinction between Π and Ψ is significant - all in accord with the notion of first canonical form.

However, most programming is better modeled if distinctions based only on the non-terminating computations of programs are not considered significant. Accordingly, we seek a second canonical form, based upon the following notion of equivalence.

3.02. If Π and Ψ are programs, then Π is weakly equivalent to Ψ if and only if $\mathfrak{D}_\Pi = \mathfrak{D}_\Psi$ for all devices $\mathfrak{D}$.

Corollary. Weak equivalence of programs is an equivalence relation.

It is certainly to be expected that strong equivalence implies weak, and it is so. Proof of this statement and others concerning weak equivalence is facilitated by considering a device F which traverses programs and reports commands encountered. Such a device, which we will now define, may be thought of as providing a free semantic interpretation of its programs - "free" because commands remain uninterpreted, "semantic" because the range of a function F_Π computable on F is the set of sequences of commands executable by Π.

3.03. A system of commands is an ordered pair $\langle C,v\rangle$, where C is a set (the commands) and v is a function with $\mathrm{dom}v = C$. (For $\alpha \varepsilon C, v(\alpha)$ is the valence of α.)

3.04. If $\langle C,v\rangle$ is a system of commands, then $\mathrm{Free}^{(C,v)}$ is a device

specified thus (writing F for $\text{Free}^{(C,v)}$):

Let $U=\cup\{v(\alpha)\mid\alpha\in C\}$.
$F_Q=U^*\times C^*$, $F_S=U^*$, $F_T=C^*$
$F_I:x\mapsto\langle x,\langle\rangle\rangle$, $F_0:\langle\langle\rangle,y\rangle\mapsto y$ (undefined on $\langle x,y\rangle$ if $x\neq\langle\rangle$);
$F_C=C$;
$F_\alpha:\langle ix,y\rangle\mapsto\langle\langle x,y\alpha\rangle,i\rangle$ if and only if $i\in v(\alpha)$, for all $\alpha\in C$. (thus $F_V=v$.)

Corollary. If $\langle C,v\rangle$ is a system of commands and Π is a program, then $\langle\zeta,x,y\rangle\ldots\langle\zeta',x',y'\rangle\in\mathcal{C}(\Pi,\text{Free}^{(C,v)})$ if and only if there is some x'' such that $x=x''x'$, $\zeta'=\Pi_{\overline{B}}(\zeta,x'')$, and $y'=y\Pi_{\overline{A}}(\zeta,x'')$.

In particular, $\text{Free}^{(C,v)}{}_\Pi(x)=y$ if and only if $\Pi_{\overline{B}}(\Pi_S,x)\in\Pi_T$ and $y=\Pi_{\overline{A}}(\Pi_S,x)$.

3.05. Theorem. If Π and Ψ are programs, then Π is weakly equivalent to Ψ if and only if $\text{Free}^{(C,v)}{}_\Pi=\text{Free}^{(C,v)}{}_\Psi$ where $C=\Pi_C\cup\Psi_C$ and $v:\alpha\mapsto(\cup\{\Pi_V(\zeta)\mid\Pi_A(\zeta)=\alpha\})\cup(\cup\{\Psi_V(\zeta)\mid\Psi_A(\zeta)=\alpha\})$.

Proof: If Π is weakly equivalent to Ψ, then $\text{Free}^{(C,v)}{}_\Pi=\text{Free}^{(C,v)}{}_\Psi$ by definition of weak equivalence.

Conversely, suppose $F_\Pi=F_\Psi$, $F=\text{Free}^{(C,v)}$, and let $\mathfrak{D}$ be any device. If $x\in\text{dom}\mathfrak{D}_\Pi$, then there are sequences $\langle\zeta_0,m_0\rangle\ldots\langle\zeta_n,m_n\rangle\in\mathcal{C}_T(\Pi,\mathfrak{D})$, $i_1\ldots i_n$, and $\alpha_1\ldots\alpha_n$ such that $\zeta_0=\Pi_S$, $m_0=\mathfrak{D}_I(x)$, $m_n\in\text{dom}\mathfrak{D}_0$, $\zeta_n=\Pi_{\overline{B}}(\Pi_S,i_1\ldots i_n)$, $\Pi_{\overline{A}}(\Pi_S,i_1\ldots i_n)=\alpha_1\ldots\alpha_n$, and $\mathfrak{D}_{\alpha_j}(m_{j-1})=\langle m_j,i_j\rangle$ for all $j\in\{1\ldots,n\}$. By the corollary to (3.04), $F_\Pi(i_1\ldots i_n)=\alpha_1\ldots\alpha_n$, so $F_\Psi(i_1\ldots i_n)=\alpha_1\ldots\alpha_n$. Again by the corollary to (3.04), there is a sequence $\eta_0\ldots\eta_n$ such that $\eta_0=\Psi_S$, $\eta_n\in\Psi_T$, and $\Psi_A(\eta_{j-1})=\alpha_j$ and $\Psi_B(\eta_{j-1},i_j)=\eta_j$ for all $j\in\{1\ldots,n\}$. It follows that $\langle\eta_0,m_0\rangle\ldots\langle\eta_n,m_n\rangle\in\mathcal{C}_T(\Psi,\mathfrak{D})$, so $\mathfrak{D}_\Psi(x)=\mathfrak{D}_\Pi(x)$. The same argument applies if $x\in\text{dom}\mathfrak{D}_\Psi$.

3.06. Theorem. If Ψ is a program and Π is a program strongly equivalent to Ψ, then Π is weakly equivalent to Ψ.

Proof: Suppose Π is strongly equivalent to Ψ. Then, by (2.10), $\tilde{\Pi}=\tilde{\Psi}$. By the

corollary to (2.08), if $x\varepsilon(\Pi_U \cup \Psi_U)^*$, then $\Pi_{\bar{B}}(\Pi_S,x)\varepsilon\Pi_T$ if and only if $\Psi_{\bar{B}}(\Psi_S,x)\varepsilon\Psi_T$, and $\Pi_{\bar{A}}(\Pi_S,x)=\Psi_{\bar{A}}(\Psi_S,x)$. By the corollary to (3.04), $\text{Free}^{(C,v)}_{\Pi}=\text{Free}^{(C,v)}_{\Psi}$, where $\langle C,v\rangle$ is as in (3.05). By (3.05), Π is weakly equivalent to Ψ.

Corollary 1. If Π and Ψ are programs and there is a homomorphism $f:\Pi_Q \to \Psi_Q$, then Π is weakly equivalent to Ψ. (By (2.06,3.06))

Corollary 2. If Π is a program, then Π is weakly equivalent to $\tilde{\Pi}$. (By the corollary to (2.12) and (3.06))

We turn now to the construction of the second canonical form itself. Its relation to weak equivalence will be exactly analogous to that of the first canonical form to strong equivalence.

3.07. If Π is a program, define the second canonical equivalence induced by Π, a binary relation Π_F in Π_U^*, by $x\Pi_F y$ if and only if: either $\Pi_{\bar{B}}(\Pi_S,xz)\notin\Pi_T$ and

$$\Pi_{\bar{B}}(\Pi_S,yz)\notin\Pi_T \text{ or } \Pi_{\bar{B}}(\Pi_S,xz)\varepsilon\Pi_T \text{ and } \Pi_{\bar{B}}(\Pi_S,yz)\varepsilon\Pi_T \text{ and}$$
$$\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,x),z)=\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,y),z), \text{ for all } z\varepsilon\Pi_U^*.$$

Corollary. If Π is a program, then Π is an equivalence relation and satisfies

(i) if $x\Pi_F y$ and $z\varepsilon\Pi_U^*$, then $xz\Pi_F yz$ (Π_F is right-invariant);
(ii) Π_U^*/Π_F is finite;
(iii) If $x\Pi_F y$ and $\Pi_{\bar{B}}(\Pi_S,xz)\varepsilon\Pi_T$ for some $z\varepsilon\Pi_U^*$, then $\Pi_A(\Pi_{\bar{B}}(\Pi_S,x))=\Pi_A(\Pi_{\bar{B}}(\Pi_S,y))$;
(iv) If $x\Pi_E y$, then $x\Pi_F y$.

3.08. If Π is a program such that $\{x\,|\,\Pi_{\bar{B}}(\Pi_S,x)\varepsilon\Pi_T\}\neq\Box$, then the second canonical form of Π is a program $\hat{\Pi}$ specified as follows (writing $[x]$ for $[x]_{\Pi_F}\varepsilon\Pi_U^*/\Pi_F$):

$\hat{\Pi}_Q=\{[x]\,|\,\Pi_{\bar{B}}(\Pi_S,xy)\varepsilon\Pi_T \text{ for some } y\varepsilon\Pi_U^*\}$.
$\hat{\Pi}_S=[\langle\rangle]$.
$\hat{\Pi}_A:[x]\mapsto \Pi_A(\Pi_{\bar{B}}(\Pi_S,x))$.
$\hat{\Pi}_B:\langle[x],i\rangle\mapsto[xi]$ if $\Pi_{\bar{B}}(\Pi_S,xiy)\varepsilon\Pi_T$ for some $y\varepsilon\Pi_U^*$.

By the corollary to (3.07), $\hat{\Pi}$ is well-defined.

Note that $\hat{\Pi}$ is not defined if and only if $\{x \mid \Pi_{\bar{B}}(\Pi_S,x)\in\Pi_T\}=\Box$.

Corollary. If Π is a program and $\hat{\Pi}$ is defined, then, for all $x,y\in\Pi_U^*$ (again writing $[x]$ for $[x]_{\Pi_F}$),

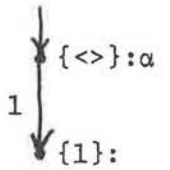
(i) $\hat{\Pi}_{\bar{B}}([x],y)=[xy]$ if $[xy]\in\hat{\Pi}_Q$, undefined otherwise;

(ii) $\hat{\Pi}_{\bar{A}}([x],y)=\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,x),y)$ if $[xy]\in\hat{\Pi}_Q$, undefined otherwise.

Also,

(iii) $\hat{\Pi}_T=\tilde{\Pi}_T=\{\{x \mid \Pi_{\bar{B}}(\Pi_S,x)\in\Pi_T\}\}$.

3.09. Example. With respect to (3.01), we have $\hat{\Pi}=\hat{\Psi}=$



3.10. Lemma. If Π is a program such that $\hat{\Pi}$ is defined, then Π is weakly equivalent to $\hat{\Pi}$.

Proof: Let $C=\Pi_C$, $v:\alpha\mapsto \cup\{\Pi_V(\zeta) \mid \Pi_A(\zeta)=\alpha\}$. Then $\text{Free}^{(C,v)}_{\Pi}(x)=y$

$\Leftrightarrow \Pi_{\bar{B}}(\Pi_S,x)\in\Pi_T$ and $y=\Pi_{\bar{A}}(\Pi_S,x)$ (By cor. to (3.04))

$\Leftrightarrow \hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,x)\in\hat{\Pi}_T$ and $y=\hat{\Pi}_{\bar{A}}(\hat{\Pi}_S,x)$ (By cor. to (3.08))

$\Leftrightarrow \text{Free}^{(C,v)}_{\hat{\Pi}}(x)=y$ (By cor. to (3.04)).

By (3.05), Π is weakly equivalent to $\hat{\Pi}$.

3.11. Theorem. If Π and Ψ are programs, then Π is weakly equivalent to Ψ if and only if $\hat{\Pi}=\hat{\Psi}$ or both $\hat{\Pi}$ and $\hat{\Psi}$ are undefined.

Proof: If $\hat{\Pi}=\hat{\Psi}$, then Π is weakly equivalent to Ψ by (3.10) and the transitivity of weak equivalence. If both $\hat{\Pi}$ and $\hat{\Psi}$ are undefined, then $\mathfrak{D}_\Pi=\mathfrak{D}_\Psi=\Box$ for any device $\mathfrak{D}$.

Conversely, suppose Π is weakly equivalent to Ψ, and let $\langle C,v\rangle$ be as in (3.05). By (3.05), $\text{Free}^{(C,v)}_{\Pi}=\text{Free}^{(C,v)}_{\Psi}$. For $\Xi=\Pi$ or Ψ, we have $\Xi_{\bar{B}}(\Xi_S,x)\in\Xi_T$ if and only if $x\in\text{dom Free}^{(C,v)}_{\Xi}$ by the corollary to (3.04). Thus $\hat{\Pi}$ and $\hat{\Psi}$

are either both defined or both undefined. By the same argument, $[x]_{\Pi_F} \varepsilon \hat{\Pi}_Q$ if and only if $[x]_{\Psi_F} \varepsilon \hat{\Psi}_Q$. Again for $\Xi=\Pi$ or Ψ, if $\Xi_{\bar{B}}(\Xi_S,xz)\varepsilon\Xi_T$, then $\text{Free}^{(C,v)}_{\Xi}(xz)=\Xi_{\bar{A}}(\Xi_S,x)\Xi_{\bar{A}}(\Xi_{\bar{B}}(\Xi_S,x),z)$ by the corollaries to (2.04,3.04). Considering the final substring of length $|z|$ of this, we see that $\hat{\Pi}_Q=\hat{\Psi}_Q$. It follows that $\hat{\Pi}=\hat{\Psi}$ if both are defined.

Corollary 1. If Π and Ψ are programs and $\tilde{\Pi}=\tilde{\Psi}$, then $\hat{\Pi}=\hat{\Psi}$ or both $\hat{\Pi}$ and $\hat{\Psi}$ are undefined. (By (2.10,3.06,3.11))

Corollary 2. If Π and Ψ are programs and there is a homomorphism $f:\Pi_Q \to \Psi_Q$, then $\hat{\Pi}=\hat{\Psi}$ or both $\hat{\Pi}$ and $\hat{\Psi}$ are undefined. (By the corollary to (2.10) and corollary 1)

To complete section 3, we establish three important properties of second canonical form, and exhibit algorithms for its construction.

3.12. Theorem. If Π is a program such that $\hat{\Pi}$ is defined, then $\widehat{(\hat{\Pi})}=\widehat{(\tilde{\Pi})}=\widetilde{(\hat{\Pi})}=\hat{\Pi}$. If $\hat{\Pi}$ is undefined, then so is $\widehat{(\tilde{\Pi})}$.

Proof: If $\hat{\Pi}$ is defined, then $\widehat{(\hat{\Pi})}=\hat{\Pi}$ by (3.10,3.11) and $\widehat{(\tilde{\Pi})}=\square$ by (2.12) and corollary 1 of (3.11). If $\hat{\Pi}$ is undefined, then $\tilde{\Pi}_T=\square$ by the corollary to 2.08, so $\widehat{(\tilde{\Pi})}$ is undefined.

It remains to show that $\widetilde{(\hat{\Pi})}=\hat{\Pi}$, assuming $\hat{\Pi}$ defined. This follows directly from the fact that $\hat{\Pi}_Q=\widetilde{(\hat{\Pi})}_Q$, which we will now prove. By (i) of the corollary to (3.08), $\langle\hat{\Pi}_S,x\rangle\varepsilon\text{dom}\hat{\Pi}_{\bar{B}}$ if and only if $[x]_{\Pi_F}\varepsilon\hat{\Pi}_Q$. Thus $\widetilde{(\hat{\Pi})}_Q=\{[x]^{\wedge}_{\Pi_E} \mid [x]_{\Pi_F}\varepsilon\hat{\Pi}_Q\}$, so it is only necessary to show that $[x]^{\wedge}_{\Pi_E}=[x]_{\Pi_F}$ if $[x]_{\Pi_F}\varepsilon\hat{\Pi}_Q$. Suppose $[x]_{\Pi_F}\varepsilon\hat{\Pi}_Q$. As already noted, $\langle\hat{\Pi}_S,x\rangle\varepsilon$ dom $\hat{\Pi}_{\bar{B}}$, so $x\varepsilon\hat{\Pi}^*_U$ and $[x]^{\wedge}_{\Pi_E}$ is defined.

$[x]^{\wedge}_{\Pi_E}\subset[x]_{\Pi_F}$: Suppose $x\hat{\Pi}_E y$ and let $z\varepsilon\Pi^*_U$. By the corollary to (3.08), $\Pi_{\bar{B}}(\Pi_S,xz)\varepsilon\Pi_T \Leftrightarrow [xz]_{\Pi_F}\varepsilon\hat{\Pi}_T \Leftrightarrow \hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,xz)\varepsilon\hat{\Pi}_T$. Likewise for y. But by (iv) of the corollary to (2.07), $\hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,xz)\varepsilon\hat{\Pi}_T$ if and only if $\hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,yz)\varepsilon\hat{\Pi}_T$. Thus $\Pi_{\bar{B}}(\Pi_S,xz)\varepsilon\Pi_T$

if and only if $\Pi_{\bar{B}}(\Pi_S,yz)\varepsilon\Pi_T$. If this condition holds, then, again by the corollary to (3.08) $\Pi_{\bar{A}}(\Pi_{\bar{B}}(\Pi_S,x),z)=\hat{\Pi}_{\bar{A}}([x]_{\Pi_F},z)=\hat{\Pi}_{\bar{A}}(\hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,x),z)$ and likewise for y. By definition of $\hat{\Pi}_E$, it follows that $x\Pi_F y$.

$[x]_{\Pi_F}\subset[x]_{\hat{\Pi}_E}$: Suppose $x\Pi_F y$. By (i) of the corollary to (3.08), $\hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,x)=[x]_{\Pi_F}=[y]_{\Pi_F}=\hat{\Pi}_{\bar{B}}(\hat{\Pi}_S,y)$. It follows that $x\hat{\Pi}_E y$.

3.13. Lemma. If Π is a program such that $\hat{\Pi}$ is defined, and Π' is defined thus:

$\Pi'_Q=\{\Pi_{\bar{B}}(\Pi_S,x)\mid\Pi_{\bar{B}}(\Pi_S,xy)\varepsilon\Pi_T$ for some $y\varepsilon\Pi_U^*\}$,

$\Pi'_S=\Pi_S$, and

Π'_A,Π'_B are the restrictions of Π_A to Π'_Q, Π_B to $\Pi'_Q\times\Pi_U$, respectively;

then Π' is a program and there is a unique homomorphism $f:\Pi'_Q\to\hat{\Pi}_Q$ with ran $f=\hat{\Pi}_Q$.

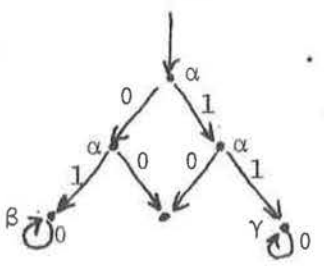
Proof: If $x,y\varepsilon\Pi_U^*$ and $\Pi_{\bar{B}}(\Pi_S,x)=\Pi_{\bar{B}}(\Pi_S,y)$, then $x\Pi_F y$ by definition of Π_F. Therefore $f:\Pi_{\bar{B}}(\Pi_S,x)\mapsto[x]_{\Pi_F}$ is well-defined as a function. Clearly Π',f are as required and f is unique.

3.14. Lemma. If Π is a program such that $\hat{\Pi}$ is defined and Π' is obtained as in (3.13), then $\tilde{\Pi}'=\hat{\Pi}$.

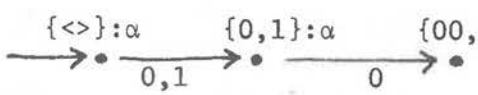
Proof: By (3.13) and the corollary to (2.10), $\tilde{\Pi}'=(\tilde{\hat{\Pi}})$. By (3.12), $(\tilde{\hat{\Pi}})=\hat{\Pi}$.

3.15. Theorem. If Π is a program such that, whenever $\langle\Pi_S,x\rangle\varepsilon\mathrm{dom}\Pi_{\bar{B}}$, there is $y\varepsilon\Pi_U^*$ such that $\Pi_{\bar{B}}(\Pi_S,xy)\varepsilon\Pi_T$, then $\hat{\Pi}=\tilde{\Pi}$.

Proof: To see that $\hat{\Pi}$ is defined, consider $x=\langle\rangle$ in the hypothesis. Under these hypotheses, the programs Π' of (2.11,3.13) are equal. By (2.11) and the corollary to (2.10), $\tilde{\Pi}=(\tilde{\Pi})$. By (3.14) $\tilde{\tilde{\Pi}}=\hat{\Pi}$.

3.16. Example. $\Pi=$ 

. $\tilde{\Pi}$ is isomorphic to Π, but

$\hat{\Pi}=$ 

Thus $\hat{\Pi}$ cannot in general be obtained from $\tilde{\Pi}$ by deleting "useless" features.

3.17. Theorem. If Π is a program such that $\hat{\Pi}$ is defined, and Ψ is a program weakly equivalent to Π, then card $\Psi_Q \geq \text{card}\hat{\Pi}_Q$.

Proof: By (3.11), $\hat{\Psi}=\hat{\Pi}$. Obtain Ψ' from Ψ as in (3.13). Then $\text{card}\Psi_Q \geq \text{card}\Psi'_Q \geq \text{card}\hat{\Psi}_Q = \text{card}\hat{\Pi}_Q$.

3.18. Algorithm. Given a program Π, to determine the set $\{\zeta\varepsilon\Pi_Q \mid \Pi_{\bar{B}}(\zeta,x)\varepsilon\Pi_T$ for some $x\varepsilon\Pi_U^*\}$.

Let $Q_o=\Pi_T$. For $j=1,2,\ldots$, compute $Q_j=Q_{j-1}\cup\{\zeta \mid \Pi_B(\zeta,i)\varepsilon Q_{j-1}$ for some $i\varepsilon\Pi_V(\zeta)\}$. Stop when $Q_j=Q_{j-1}$. Q_j is then the desired set.

3.19. Algorithm. Given a program Π, to determine if $\hat{\Pi}$ is defined, and, if so, to obtain $\hat{\Pi}$, the elements of $\hat{\Pi}_Q$ specified as regular expressions.

Apply (2.14) to obtain $M_1=\{\Pi_{\bar{B}}(\Pi_S,x) \mid x\varepsilon\Pi_U^*\}$.

Apply (3.18) to obtain $M_2=\{\zeta\varepsilon\Pi_Q \mid \Pi_{\bar{B}}(\zeta,x)\varepsilon\Pi_T$ for some $x\varepsilon\Pi_U^*\}$.

If $M_1\cap M_2=\square$, then $\hat{\Pi}$ is undefined.

Otherwise, Let: $\Pi'_Q=M_1\cap M_2$; $\Pi'_S=\Pi_S$; Π'_A, Π'_B be the restrictions of Π_A to Π'_Q, Π_B to $\Pi'_Q\times\Pi_U$, respectively.

Apply (2.19) to Π'. The result $(\tilde{\Pi'})$ is $\hat{\Pi}$. (By (3.14))

REFERENCES

Aho, A.V., J.E. Hopcroft, and J.D., Ullman. (1974) "The design and analysis of computer algorithms". Addison-Wesley, Reading, Mass.

Arbib,M.A. and E.G. Manes (1974), Machines in a category: an expository introduction. Siam Review. 16, 163:192.

Baker, J.L. (1975) Factorization of Scott-style automata. Category Theory Applied to Computation and Control,(1974 Symposium). Springer Lecture Notes in Computer Science. 25, 99:105.

Elgot, C.C. (1975) Monadic computation and iterative algebraic theories. Proceedings of Logic Colloquium (Bristol 1973). North-Holland. 175:230.

Goguen, J.A., Jr. (1974) On homomorphisms, correctness, termination, unfoldments and equivalence of flow diagram programs. Journal of Computer and System Sciences. 8, 333:365.

Goguen, J.A., Jr., J.W. Thatcher, E.G. Wagner, and J.B. Wright (1975) Initial Algebra Semantics, IBM Research Report RC5243.

Luckham, D.C., D.M.R. Park, and M.S. Patterson (1970). On formalised computer programs. Journal of Computer and Systems Sciences. 4, 220:249

Moore, E.F. (1956) Gedanken-experiments on sequential machines. Automata Studies. Princeton. Annals of mathematics studies no. 34. 129:153.

Nerode, A. (1958) Linear automaton transformations. Proc. Am. Math. Soc. 9, 541:544.

Rabin, M.O. and D. Scott (1959) Finite automata and their decision problems. IBM Journal of Research and Development. 3, 114:125.

Rutledge, J.D. (1964) On Ianov's program schemata. J.ACM. 11, 1:9.

Scott, D. (1967) Some definitonal suggestions for automata theory. Journal of Computer and System Sciences. 1, 187:212.

Trakhtengerts, E.A. (1967) Mathematical methods for the minimization of production monitoring and control programs (Survey). Automation and Remote Control. 1967 no. 8, 1181:1203.

Yanov, Yu. I. (1960) The Logical Schemes of Algorithms. Problems of Cybernetics (Pergamon). 1, 82:140.