A note from the typesetter

Care has been taken to preserve the original intent as much as possible but this document is not intended to be historical- rather only a more convenient setting to read Scott and de Bakker's unpublished work at IBM Vienna 1969. The original scan can be found at

https://ir.cwi.nl/pub/20373

de Bakker, J.W, & Scott, D. (1989). A theory of programs : an outline of joint work : IBM seminar Vienna, August 1969. In J.W Klop, J.J.C Meijer, & J.J.M.M Rutten (Eds.), J.W. de Bakker, 25 jaar semantiek (pp. 1–30). CWI.

A Theory of Programs

An Outline of Joint Work by

J.W. De Bakker and Dana Scott

IBM Seminar, Vienna, August 1969

1 Machines

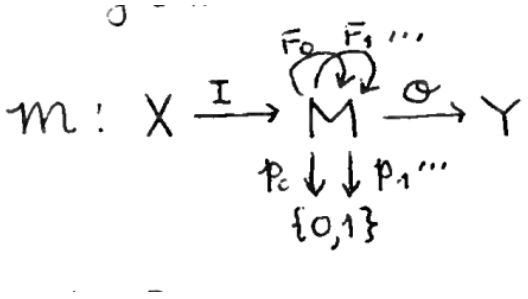
A machine is a structure

$$\mathscr{M} = (\mathcal{I}, \mathcal{O}, F_0, p_0, F_1, p_1, \ldots)$$

of *partial* functions for which there exist (uniquely determined) sets X, M, Y, such that

$$\begin{aligned}
&\mathcal{I} : X \to M \quad \textit{(the input function)}\\
&\mathcal{O} : M \to Y \quad \textit{(the output function)}\\
&F_i : M \to M \quad \textit{(the operations)}\\
&p_j : M \to \{0,1\} \quad \textit{(the tests)}.
\end{aligned}$$

As a diagram, we can write



2 Computations

A computation (from $x \in X$ to $y \in Y$) is a finite sequence

$$\xi_0, F_{i_0}, \xi_1, F_{i_1}, \xi_2, \ldots, \xi_{k-1}, F_{i_{k-1}}, \xi_k$$

whete each $\xi_\ell \in M$ and $\xi_{\ell+1} = F_{i_l}(\xi_\ell)$ for $\ell < k$ (and where $I(x) = \xi_0$ and $O(\xi_k) = y$).

3 Programs

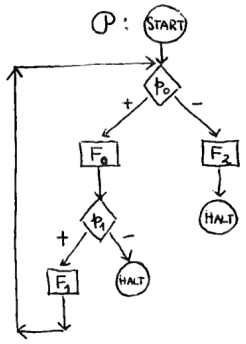
Informally, a program is a "method" of selecting *at most* one computation starting from each $x \in X$. Thus, a program $\mathscr{P}$ associates to the machine $\mathscr{M}$ a partial function

$$\mathscr{P}(\mathscr{M}) : X \to Y.$$

Programs must be defined "independently" of particular machines, and indeed we can identify the program with this mapping from machines to functions. A program as a mapping must satisfy some general conditions to be mentioned later. First we give some examples.

4 Flow Diagrams

It is intuitively clear that, given an arbitrary machine $\mathscr{M}$, this diagram allows us to generate for each $x \in X$ at most one computation, offered by following the "flow" of the diagram. We say that the diagram (a "syntactical" object) defines a program (a mathematical object).



5 Procedures

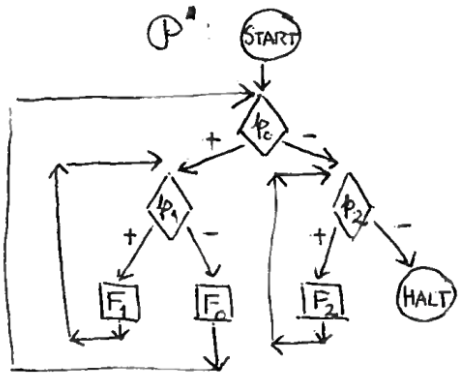
$$\mathscr{P} : \begin{cases} P_0 \Rightarrow (p_0 \to F_0; P_1, F_2) \\ P_1 \Rightarrow (p_1 \to F_1; P_0, E) \end{cases}$$

Here we have a system of procedure declarations (where by convention P_0 is the "principal" one). $(p \to P, P')$ is the conditional expression; $P; P'$ is composition (P followed by P'); and E is the symbol for the identity function. The above system defines the same program as the flow diagram [*in section 4*].

6 While Statements

$$\mathscr{P}' : (p_0 * (p_1 * F_1); F_0); (p_2 * F_2)$$

Read "$(p * P)$" as "<u>while</u> p <u>do</u> P". The same program can be defined by flow diagrams or by procedures



$$\mathscr{P}' : \begin{cases} P_0 \Rightarrow (p_0 \to P_1; P_0, P_2) \\ P_1 \Rightarrow (p_1 \to F_1; P_1, F_0) \\ P_2 \Rightarrow (p_2 \to F_2; P_2, E) \end{cases}$$

7 Equivalences

Two programs $\mathscr{P}$ and $\mathscr{P}'$ are <u>equivalent</u> *on* a machine $\mathscr{M}$ iff $\mathscr{P}(\mathscr{M}) = \mathscr{P}'(\mathscr{M})$. They are (<u>strongly</u>) <u>equivalent</u> iff they are equivalent on *all* machines; that is iff $\mathscr{P} = \mathscr{P}'$. Two flow diagrams (systems of procedures, <u>while</u> systements) are equivalent iff they define equivalent programs.

Theorem 1. *Every program defined by a <u>while</u> statement is (effectively) equivalent to one defined by a flow diagram, but there is a flow diagram that defines a program <u>not</u> defined by any <u>while</u> statement.*

Theorem 2. *(the same for flow diagrams and procedures)*

Theorem 3. *It is (effectively) decidable whether two flow diagrams are equivalent*

Theorem 4. *It is (effectively) decidable whether a system of procedures defines the null program Ω.*

Problems: Is it (effectively) decidable whether two procedures are equivalent? Is it decidable whether a procedure is equivalent to some flow diagram? A flow diagram to a <u>while</u> statement?

8 General Properties

We write $\mathscr{M} \subseteq \mathscr{M}'$ iff $\mathscr{M}$ and $\mathscr{M}'$ have the same sets X, M, Y and have operations and tests $F_i \subseteq F_i'$ and $p_i \subseteq p_i'$ for all i, j (that is, F_i' is consistent with, but more defined than F_i).

(I). $\mathscr{M} \subseteq \mathscr{M}'$ implies $\mathscr{P}(\mathscr{M}) \subseteq \mathscr{P}(\mathscr{M}')$
(Property (I) can be generalized by defining morphisms $\varphi : \mathscr{M} \to \mathscr{M}'$ between machines. See below.)

(II). If for $\mathscr{M}_n \subseteq \mathscr{M}_{n+1}$ for $n = 0, 1, 2, \ldots$ then

$$\mathscr{P}\left(\bigcup_{n=0}^{\infty} \mathscr{M}_n\right) = \bigcup_{n=0}^{\infty} \mathscr{P}(\mathscr{M}_n)$$

(Property (II) can be generalized to directed unions (and, no doubt, to direct limits)).

Write $\mathscr{M}^{(n,m)}$ for the result of modifying $\mathscr{M}$ by replacing F_i and p_j by the totally undefined functions for $i \geq n$ and $j \geq m$.

(III). For every program $\mathscr{P}$ there exists n, m such that for all machines $\mathscr{M}$

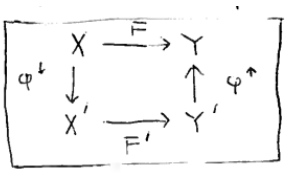
$$\mathscr{P}(\mathscr{M}) = \mathscr{P}(\mathscr{M}^{(n,m)})$$

The above are defined for procedures, and (suitably generalized) ought to be taken as "axiomatic" for the notion of program.

9 Categories

Let $\mathbb{F}$ be the caregory whose objects are *partial functions* $F : X \to Y$ and whose morphisms $\varphi : F \to F'$ are pairs $\varphi = (\varphi^{\downarrow}, \varphi^{\uparrow})$ where

$$\varphi^{\downarrow} : X \to X'$$
$$\varphi^{\uparrow} : Y' \to Y$$
$$F \subseteq \varphi^{\downarrow}; F'; \varphi^{\uparrow}$$

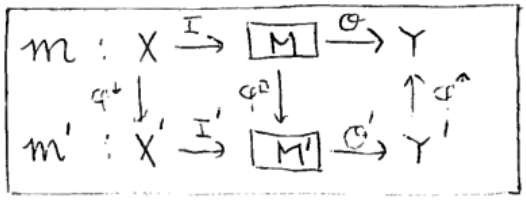


(Here, ";" denotes composition of relations so that $(F; G)(x) = G(F(x))$ in the case of functions).

Let $\mathbb{M}$ be the category whoste objects are *machines* and whose morphisms $\varphi : \mathscr{M} \to \mathscr{M}'$ are triples $\varphi = (\varphi^{\downarrow}, \overline{\varphi^{\square}, \varphi^{\uparrow}})$ where $\varphi^{\downarrow} : X \to X'$ and $\overline{\varphi^{\square} : M \to M'}$ and $\varphi^{\uparrow} : Y' \to Y$ and where for all i, j:

$$\mathcal{I} \subseteq \varphi^{\downarrow}; \mathcal{I}'; (\varphi^{\square})^{-1}$$
$$\mathcal{O} \subseteq \varphi^{\square}; \mathcal{O}'; (\varphi^{\uparrow})^{-1}$$
$$F_i \subseteq \varphi^{\square}; F_i'; (\varphi^{\square})^{-1}$$
$$p_j \subseteq \varphi^{\square}; p_j'$$

(Here, "$^{-1}$" denotes converse of a relation)



Let $\mathscr{P}$ be a program and let $\varphi : \mathscr{M} \to \mathscr{M}'$. Define $\mathscr{P}(\varphi) = (\varphi^{\downarrow}, \varphi^{\uparrow})$ where $\varphi = (\varphi^{\uparrow}, \varphi^{\square}, \varphi^{\downarrow})$.

Theorem 5. *If $\varphi : \mathscr{M} \to \mathscr{M}'$ in the category $\mathbb{M}$, then $\mathscr{P}(\varphi) : \mathscr{P}(\mathscr{M}) \to \mathscr{P}(\mathscr{M}')$ in the category $\mathbb{F}$.*

This is at least clear for programs defined by procedures; it should be taken as "axiomatic" in general. Thus it follows that $\mathscr{P} : \mathbb{M} \to \mathbb{F}$ is a *functor* between categories. This is the proper generalization of property (I) above as a function $\mathscr{P}$ ought to be rather "continuous" in some suitable sense.

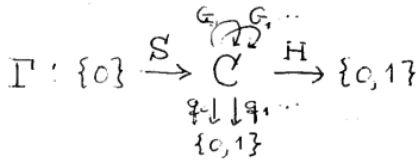
If $\mathscr{M} : X \xrightarrow{\mathcal{I}} \boxed{M} \xrightarrow{\mathcal{O}} Y$, define $[\mathscr{M}] : M \xrightarrow{E} \boxed{M} \xrightarrow{E} M$ to be the machine with the same operations and tests but with $\mathcal{I}, \mathcal{O}, X, Y$ replaced with E, E, M, M where E is the identity on M. Then we have the result

$$(\mathcal{I}, E, \mathcal{O}) : \mathscr{M} \to [\mathscr{M}] \quad \text{in } \mathbb{M}.$$

It was the "obvious correctness" of this relationship that motivated the above definitions. The categories $\mathbb{M}$ and $\mathbb{F}$ require much more study, however, before their usefulness can be determined.

10 Control Devices

By control device we shall understand a "Boolean" machine:



11 "Abstract" programs

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ and let $\{0,1\}^\infty$ be the set of all partially defined infinite sequences of 0's and 1's. If $\mathscr{M}$ is a machine and $\xi \in M$, define

$$p(\xi) = (p_0(\xi), p_1(\xi), \ldots, p_n(\xi), \ldots) \in \{0,1\}^\infty$$

An "abstract" program $\mathscr{P}^\Gamma_{f,g}$ is defined by giving a control device Γ (generally fixed so that a definite class of programs is considered) and two functions

$$f, g : \{0,1\}^\infty \times \{0,1\}^\infty \to \mathbb{N}$$

such that the selected computations of $\mathscr{P}^\Gamma_{f,g}$ on a machine $\mathscr{M}$ (in the notation of page 1) are those for which there exist (uniquely determined) computations

$$\gamma_0, G_{j_0}, \gamma_1, G_{j_1}, \ldots, \gamma_{k-1}, G_{j_{k-1}}, \gamma_k$$

on the machine Γ where for $\ell \leq k$

$$\begin{gathered} i_\ell = f(p(\xi_\ell), q(\gamma_e ll)) \\ j_\ell = g(p(\xi_\ell), q(\gamma_e ll)) \\ \gamma_0 = \mathcal{S}(0), \mathcal{H}(\gamma_\ell) = 0, \quad \text{for } \ell < k \\ \mathcal{H}(\gamma_k) = 1 \end{gathered}$$

Thus $\mathcal{S}$ and $\mathcal{H}$ control the start and halt and f and g tell where to look for the next operation to execute. We need Γ as a "memory" to keep track of where we are in the intermediate stages of "reading" the text of the program definition. Assuming that f and g are "finitely given" (i.e. depend on a fixed bounded number of coordinates of $p(\xi)$ and $q(\xi)$), we can then prove that $\mathscr{P}^\Gamma_{f,g}$ is a functor with basic properties (I), (II), (III), (we need some slight consistency conditions on f and g).

Conjecture: There are a few more "nice" properties (like (I), (II), (III)) such that any functor $\mathscr{P} : \mathbb{M} \to \mathbb{F}$ having these properties is of the form $\mathscr{P}^\Gamma_{f,g}$.

Note: in this abstract setting is is convenient to make the harmless convention that on all machines $\mathscr{M}$ we have $F_0 = E$, the identity on M.

12 Examples

The "abstract" version of the flow diagram uses the control device where

$$\begin{aligned}
\mathcal{C} &= \{-1, 0, 1, 2, \ldots\} \\
\mathcal{S}(0) &= 0 \\
\mathcal{G}_j(\gamma) &= j - 1 \\
q_j(\gamma) &= \begin{cases} 1 & \gamma = j \\ 0 & \text{otherwise} \end{cases} \\
H(\gamma) &= \begin{cases} 1 & \gamma = -1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

The "abstract" version of the procedure uses the more general control device where

$$\begin{aligned}
\mathcal{C} &= \{0, 1, 2, \ldots\}^* = \{\sigma_0, \sigma_1, \sigma_2, \ldots\} \\
\mathcal{S}(0) &= 0 \\
\mathcal{G}_j(n\gamma) &= \sigma_j\gamma \\
q_j(n\gamma) &= \begin{cases} 1 & n = j \\ 0 & \text{otherwise} \end{cases} \\
H(\gamma) &= \begin{cases} 1 & \gamma = \Lambda \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

where Λ is the null sequence and the σ_n represent some (recursive!) enumeration of the finite sequences of integers. (In other words, the control of a procedure computation is in general a push-down store).

13 Deductions

For the time being we restrict attention to procedures and ask how it can be established where two of them are *equivalent*. In one sense the question is answered because we have given a completely precise definition of the program defined by a procedure with the aid of a certin control device. That answer is not too helpful, because no simple "methods" of proof for proving equivalence are provided by the bare definition. Two different (though related) deductive systems are presented below which might be called the "algebraic" and the "second-order relational" theories. The algebraic method is more efficient for

proving equivalences; while the relational method is better for problems of correctness. It is not known whether an algebraic theory can be complete- because if it is, and if it's theorems are recursively enumerable, then we would have a recursive decision method for equivalence. The relational theory is complete- because we use second-order logic, but the theorems are *not* enumerable.

14 The Algebraic Theory

14.1 Language

Lower-case letters are Boolean variables (mostly we use p, q, r). Upper-case letters are predicate variables, except we use E and Γ as constants. Compound terms are constructed from upper-case letters by these three operations

$$(p \to \tau, \sigma) \qquad \tau; \sigma \qquad \mu X[\tau]$$

where in place of p we can have any Boolean and in place of X any procedure variable. The first is the conditional expression, the second, a composition; and the third a variable binding operator μ whose meaning is explained below. Atomic formulas are either equations $\tau = \sigma$ or inclusions $\tau \subseteq \sigma$. Lists $\Phi_0, \Phi_1, \ldots, \Phi_{n-1}$ of atomic forumulas are used as short-hand for the conjugation $[\Phi_0, \Phi_1, \ldots, \Phi_{n-1}]$. Theorems are of the form of implications $\Phi \vdash \Psi$ between lists. We use for simplicity in these notes the usual notation $\tau(X, Y)$, $\Phi(X, Z)$, $\Psi(X, \tau(X))$ to indicate (roughly!) free variables and results of substitutions.

14.2 Validity

Consider an implication $\Phi \vdash \Psi$. Suppose the free variables are $p_0, p_1, p_2, \ldots$ and $F_0, F_1, F_2, \ldots$. The implication is valid just in case for all sets M and all systems $F_i : M \to M$ and $p_j : M \to \{0, 1\}$ of partial functions and predicates on M, if Φ is *true* for these then so is Ψ. Now a list is true iff all it's terms are. An atomic $\tau = \sigma$ is true iff τ and σ denote the same function on M into M. An atomic $\tau \subseteq \sigma$ is true iff τ denotes a function included in that denoted by σ. Thus, given the value of the free variables we need still only define what is the function denoted by a term. E denotes the identity function. Ω denotes the empty ("undefined") function. Conditionals and compositions denote functions obtained from the denotations of the parts in the usual way. The special term $\mu X[\tau(X)]$ denotes the least function G such that $\tau(G) \subseteq G$ ("least" in the sense of the partial ordering $\subseteq$). We shall see below why it always exists, and why it is of interest in connection with procedures.

Theorem 6. *The set of valid implications is not recursively enumerable (sorry!)*

Problem: Is the set of $\vdash \Psi$ recursively enumerable (and hence recursive)?

14.3 Axioms and Rules

The axioms and rules for conjugations and rules are well known. As for ;, $\subseteq$, E, Ω we give the axioms of a partially ordered semigroup with unit and zero.

$$\begin{aligned} \vdash (p \to X, Y); Z &= (p \to X; Z, Y; Z) \\ \vdash (p \to X, X) &\subseteq X \\ X \subseteq X', Y \subseteq Y' \vdash (p \to X, Y) &\subseteq (p \to X', Y') \\ (p \to X, \Omega) \subseteq Z, (p \to \Omega, Y) \subseteq Z \vdash (p \to X, Y) &\subseteq Z \end{aligned}$$

(Maybe some others are required?? This point about conditionals is not too clean and needs more study.) For μ we have

$$Y = \mu X[\tau(X)] \vdash \tau(Y) \subseteq Y \quad (\mu - \text{Axiom})$$

$$\frac{\Phi \vdash \Psi(\mu X[\tau(X)])}{\Phi \vdash \Psi(\Omega) \qquad \Phi, \Psi(X) \vdash \Psi(\tau(X))} \quad (\mu - \text{Rule})$$

(in the rule X is not free in Φ).

14.4 Application

Consider the following system of procedures

$$\mathscr{P}'' = \begin{cases} P_0 \Rightarrow \tau_0(P_0, P_1) \\ P_1 \Rightarrow \tau_1(P_1, P_2) \\ P_2 \Rightarrow \tau_2(P_2, P_0) \end{cases}$$

where the τ_i are terms with the P_i and possibly other free variables. The P_i variables, of course, play a special role, and the above "rewrite" rules mean that the P_i should be computed as the "least" functions that result from replacing procedure "call" by the corresponding "body". Hence,

$$P_2 = \mu Z[\tau_2(Z, P_0)],$$

and then

$$P_1 = \mu Y[\tau_1(Y, \mu Z[\tau_2(Z, P_0)])],$$

and finally

$$P_0 = \mu X[\tau_0(X, \mu Y[\tau_1(Y, \mu Z[\tau_2(Z, P_0)])])].$$

Thus the whole program can be defined by the algebraic expression on the right hand side. Proving equations between expressions, then, is proving equivalence of programs.

14.5 Justification

With a combination of results proved within the system and semantics outside the theory, we will see that the axioms are valid and that the rules preserve validity. Later we give some examples of particular equivalence proofs.

1. $X \subseteq X' \vdash \tau(X) \subset \tau(X')$
   Proof: The theorem must first be generalized to any number of variables and then proved by induction on the complexity of the term τ. The cases of conditionals and composition are already assumed as (obviously valid) axioms. For the μ-operator we do a representative special case. Thus assume $\sigma(X, Y)$ monotone in both variables, and consider $\tau(X)$ to be $\mu X[\sigma(X, Y)]$. By the first μ-axiom

   $$\vdash \sigma(X', \tau(X')) \subseteq \tau(X')$$

   hence by assumption of σ

   $$X \subseteq X' \vdash \sigma(X', \tau(X')) \subseteq \tau(X') \subseteq \tau(X').$$

   We can again apply the monotonicity of the term σ to derive:

   $$X \subseteq X', Y \subseteq \tau(X') \vdash \sigma(X, Y) \subseteq \tau(X').$$

   Note that $X \subseteq X' \vdash \Omega \subseteq \tau(X')$ is trivial; thus by the rule for the μ-operator we have

   $$X \subseteq X' \vdash \mu Y[\sigma(X, Y)] \subseteq \tau(X'),$$

   which is the desired result for τ.

   Discussion. We proved monotonicity by the axioms and rules for μ, but this proof also helps (in part) to establish the validity of these principles. In our calculus the expressions repsresent monotonic operations on partial functions. It is well-known that such operations have minimal fixed points. Speaking informally:

   $$\begin{aligned} \mu X[\tau(X)] &= \bigcap \{X : \tau(X) \subseteq X\} \\ &= \bigcup_{n=0}^{\infty} \tau^n(\Omega) \end{aligned}$$

   where $\tau^n(\Omega) = \overbrace{\tau(\tau(\ldots\tau(}^{n \text{ times}}\Omega)\ldots))$. The second equation, which justifies the special case of the rule used in the proof of (1), is correct because the operation is also continuous in this sense (speaking outside the theory)

   $$\tau\left(\bigcup_{n=0}^{\infty} X_n\right) = \bigcup_{n=0}^{\infty} \tau(X_n)$$

whenever $X_0 \subseteq X_1 \subseteq \cdots \subseteq X_n \subseteq \cdots$. This is clear for conditionals and compositions, but again must be established for μ's. Once conitnuity is understood, the validity of the full rule for μ (which we may call the induction rule) follows easily.

2. Fixed Point Properties

$$Y = \mu X[\tau(X)] \vdash \tau(Y) = Y \quad \text{and}$$
$$\tau(Y) \subseteq Y \vdash \mu X[\tau(X)] \subseteq Y$$

The proof uses monotonicity and induction as in the proof of (1).

3. Systems of Fixed Points

$$X = \mu X[\tau(X, \mu Y[\sigma(X, Y)])],$$
$$Y = \mu Y[\sigma(X, Y)],$$
$$\tau(X', Y') \subseteq X', \sigma(X', Y') \subseteq Y' \vdash$$
$$\tau(X, Y) \subseteq X \subseteq X', \sigma(X, Y) \subseteq Y \subseteq Y'$$

Proof: "Assume" the hypothesis. Then $\tau(X, \mu Y[\sigma(X, Y)]) \subseteq X$ and so $\tau(X, Y) \subseteq X$. Also $\sigma(X, Y) \subseteq Y$. Let $Y'' = \mu Y[\sigma(X', Y)]$. Then $Y'' \subseteq Y'$ and so $\tau(X', Y'') \subseteq X'$. But then $X \subseteq X'$ and so $\sigma(X, Y') \subseteq Y'$. Finally $Y \subseteq Y'$.

Discussion. The above theorems on minimal fixed points can be extended to systems with any number of procedure declarations.

14.6 Examples:

We define while as $(p * F) = \mu X[(p \to F; X, E)]$

(a) $(p * F); G = \mu Y[(p \to F; Y, G)]$. *Proof:* By definition $(p * F) = (p \to F; (p * F), E)$. Hence $(p * F); G = (p \to F; (p * F); G, G)$. Therefore, $\mu Y[(p \to F; Y, G)] \subseteq (p * F); G$. To prove the opposite inclusion, first let $Y = \mu Y[(p \to F; Y, G)]$. By induction it is enough to show $\Omega; G \subseteq Y$ (obvious!) and $X; G \subseteq Y \vdash (p \to F; X, E); G \subseteq Y$. So assume $X; G \subseteq Y$. But $(p \to F; Y, G) \subseteq Y$. Thus $(p \to F; X, E); G = (p \to F; X; G, G) \subseteq Y$.

(b) $p * (p * F) = p * (F; p * F)$ *Proof:* Let L be the left-hand side and R be the

right. Then

$$\begin{aligned} L &= (p \to (p * F); L, E) \\ &= (p \to (p \to F; (p * F), E); L, E) \\ &= (p \to F; (p * F); L, E) \end{aligned}$$

thus $R \subseteq L$. Next

$$\begin{aligned} R &= (p \to F; (p * F); R, E) \\ &= (p \to (p \to F; (p * F), E); R, E) \\ &= (p \to (p * F); R, E) \end{aligned}$$

thus $L \subseteq R$ and $L = R$ follows.

(c) $(p * F); (p \to G, H) = (p * F); H$

(d) $(p * F); (p * G) = p * F$

(e) $p * (p * F) = p * F$

all of these follow from (i) and the method shown for (ii).

15 The Relational Theory

Functions are, after all, relations. Thus it must be possible to axiomatize the functions defined by program expressions, hopefully without the minute detail of the definitions containing control devices which are closer to the ideas of the implementation. We do this for procedures.

15.1 Language

We use a standard second-order predicate calculus with equality and notaional conventions corresponding to our language of procedure declarations. In particular we employ the following style of variables and non-logical constants:

- Individual variables $\xi, \eta, \zeta, \xi', \eta', \ldots$
- 1-place predicate constants $p, \bar{p}, q, \bar{q}, \pi, \bar{\pi}, \ldots$
- Binary relation variables R, S, T, X, Y, Z
- Binary relation constants E, Ω, P_0, P_1, P_2, $\ldots$

- Relational operations $(R; S)$, $(p \to R, S)$
- Atomic Formulas: equations plus

$$R \subseteq S \qquad p(\xi) \qquad \xi R \eta$$

(where in place of R and S we can have relational terms and in place of p the other $\bar{p}, q, \bar{q}, \ldots$ and in place of ξ, η any individual variables.)

15.2 Validity and Deduction

This is the usual notion from second-order logic; we do, however, have a few non-logical axioms to suit the application to procedures. Remember that the valid formulas are not recursively enumerable in second-order logic.

15.3 General Axioms

We assume the following definitions and axioms

(1) $\neg\exists\xi\ [p(\xi) \wedge \bar{p}(\xi)]$ (sim for $q, \pi, \ldots$)

(2) $\forall\xi, \eta\ [\xi E \eta \leftrightarrow \xi = \eta]$

(3) $\neg\exists\xi, \eta\ [\xi \Omega \eta]$

(4) $\forall\xi, \eta\ [\xi(R; S)\eta \leftrightarrow \exists\zeta\ [\xi R \zeta \wedge \zeta S \eta]]$

(5) $\forall\xi, \eta\ [\xi(p \to R, S)\eta \leftrightarrow [[p(\xi) \wedge \xi R \eta] \vee [\bar{p}(\xi) \wedge \xi S \eta]]]$

(6) $R \subseteq S \leftrightarrow \forall\xi, \eta\ [\xi R \eta \to \xi S \eta]$

(where R, S should be universally quantified).

The meaning of the axioms is clear except maybe for (1). Here the pair $p, \bar{p}$ is to represent a partial predicate (by convention all predicates in logic are total). The p is the true part and the $\bar{p}$ is the false part. They must be disjoint – but that is the only requirement. We do not need any similar tricks for partial functions since they are just relations in a straight-forward way.

15.4 Special Axioms

Consider a program $\mathscr{P}$ defined by a system of declarations

$$\mathscr{P} = \begin{cases} P_0 \Rightarrow \tau_0(P_0, \ldots, P_n) \\ \quad \vdots \\ P_n \Rightarrow \tau_n(P_0, \ldots, P_n) \end{cases}$$

Corresponding to this system we have

$$(i_{\mathscr{P}}) \qquad [\tau_0(P_0,\ldots,P_n) \subseteq P_0 \wedge \tau_1(P_0,\ldots,P_n) \subseteq P_1 \wedge \ldots \wedge \tau_n(P_0,\ldots,P_n) \subseteq P_n]$$

$$(ii_{\mathscr{P}}) \qquad \forall R \left[\bigwedge_{i=0}^{n} [\tau_i(R_0,\ldots,R_n) \subseteq R_i] \rightarrow \bigwedge_{i=0}^{n} [P_i \subseteq R_i] \right]$$

15.5 Justification

Given a system of procedures, we have mereley assumed – in second-order langauge – that P_i are the least relations where $\tau_i(P_0,\ldots,P_n) \subseteq P_i, i = 0,1,\ldots,n$. This was the same idea as the algebraic theory – except we do not have the μ-operator. We could introduce it, and then all the algebraic principles could be proved from the second-order axioms.

Remark: Given that the free variables of the procedures are $F_0, F_1, F_2, \ldots$ (our usual convention) we might want to assume that they are partial functions (our usual convention). We should have them, as general axioms,

$$\forall \xi, \eta, \eta' \; [\xi F_i \eta \wedge \xi F_i \eta' \rightarrow \eta = \eta']$$

These axioms do not seem to make too much difference, however.

15.6 Applications

Suppose $\mathscr{P}$ and $\mathscr{P}'$ are programs defined by procedures (where, say, in the second system we use constants $P_0', P_1', \ldots$). The equivalence problem is to deduce, therefore, the statement $P_0 = P_0'$ from all the axioms combined. The use of second-order deductions does not seem, however, any more convenient than the algebraic method for such equivalence. The point of the second-order system lies, rather, in the fact that more and different kind of problems can be expressed within it.

15.7 While Statements

For the sake of illustration we restruct our attention to while statements. These are special procedures, and we can introduce them by an additional operation on our relations into the language: $(p * R)$. Our axioms $(i_{\mathscr{P}})$ and $(ii_{\mathscr{P}})$ then become

$$(i_*) \qquad (p \rightarrow F; (p * F), E) \subseteq (p * F)$$

$$(ii_*) \qquad \forall R; [(p \rightarrow F; R, E) \subseteq R \rightarrow (p * F) \subseteq R]$$

(where F is to be taken as variable)

15.8 Example

Axiom (ii_*) seems to be a special case of our algebraic induction axiom – but it is much stronger in view of the quantifier $\forall R$. The reason is that we may specialize R to any relation, not just those that can be defined by terms using the operations we have given a notation for. As an illustration we prove

$$(p \to F; R, G) \subseteq R \to (p * F); G \subseteq R$$

(compare example (i) *in section 14*).

Proof: With individual variables the conclusion requires

$$\forall \xi, \eta, \gamma\ [\xi(p * F)\eta \wedge \eta G\gamma \to \xi R\gamma]$$

equivalently

$$\forall \xi, \eta\ [\xi(p * F)\eta \to \forall \gamma\ [\eta G\gamma \to \xi R\gamma]].$$

This sugggests we introduce the relation S such that (and we use second-order logic to know the S exists)

$$\forall \xi, \eta\ [\xi S\eta \to \forall \gamma\ [\eta G\gamma \to \xi R\gamma]].$$

Now the problem is to show

$$(p * F) \subseteq S.$$

In view of the axiom (ii_*) it is sufficient to prove:

$$(p \to F; S, E) \subseteq S.$$

This requires two cases:

(a) $p(\xi) \wedge \xi(F; S)\eta \to \xi S\eta$

(b) $\bar{p}(\xi) \to \xi S\xi$

For (a) we assume $p(\xi)$ and $\xi(F; S)\eta$ and $\gamma S\eta$. We want to show that $\xi S\eta$. So assume in addition $\eta G\gamma'$ and show $\xi R\gamma'$. But we know by hypothesis $(p \to F; R, G) \subseteq R$. Also, since $\eta G\gamma'$ holds and $\gamma S\eta$, we have $\gamma R\gamma'$ hence $\xi(F; S)\gamma'$. But then $\xi R\gamma'$ follows at once. Case (b) is even easier.

Remark From the work of Hoare we can find a simplification of axiom (ii_*); namely, it can be replaced by a combination of these two axioms:

(i_*) $\quad \forall \xi, \eta\ [\xi(p * F)\eta \to \bar{p}(\eta)]$

(ii_*) $\quad \forall u\ [\forall \xi, \eta\ [p(\xi) \wedge p(\xi) \wedge \xi F\eta \wedge u(\eta)] \to [\forall \xi, \eta\ [u(\xi) \wedge \xi(p * F)\eta \to u(\eta)]]$

In particular, $(ii''_*$ reads more like an arithmetic induction axiom (here, u is a unary predicate variable and this second-order form with $\forall u$ is equivalent to the earlier form with $\forall R$. A similar simplification of the axiom $(ii_{\mathscr{P}})$ for procedures is not yet apparent.

15.9 Correctness

In order to prove programs "correct", Floyd, and later Hoare, have been working with the idea of taking (a part of) a program (relation) P and thinking of desirable properties u and v such that if you enter P in u, you exit P in v. Hoare writes (more or less)

$$u\{P\}v$$

In our second-order logical notation we would write more fully

$$\forall \xi, \eta \, [u(\xi) \wedge \xi P \eta \rightarrow v(\eta)]$$

This point of view has essential advantages:

1. It is part of a well-known logical system
2. $u\{P\}v$ becomes an ordinary proposition that can be negated, etc, etc
3. Floyd's "logical" rules become obvious
4. Floyd's incorrect existential rule is avoided

16 Conclusions

Starting from an intuitively correct idea of a machine, we explained and developed a theory of programming concepts (mainly procedures). This work could be extended by investigating more powerful control devices but that is probably not a good idea at this level of abstraction. What is needed is a more refined model of a machine. In the work above we have treated each "state vector" $\xi \in M$ as a whole – and it is remarkable how many sensible things there are say even so. But in "real life" a vector ξ has components, and these are generally modified more or less independently during computation. This idea should be introduced into the model; then in programs we will want to use assignment statements to modify the coordinates. This means that operations on M, the various F_i and p_j, are being analyized instead of being treated as "wholes". Along with these problems, we will also want to treat the scope of "variables". One way to do this is to make the components of the state vector into push-down stores. But there are so many problems of "reference" that we might want to use Strachey's method of L- and R-values and LUP's. If we can do this, we will then want to isolate general properties of programs (defined already the and of the machine model) in order to organize our deductions more clearly (the so-called "axiomatic" model). At one level of abstraction this has all been illustrated above. To really carry out the proposal for the "real life" situation is a big, big "program". The outlines seem clean, however, and we should be able to do it in a way that actually refines the present theory and keeps all the "abstract" results intact.