This paper deals with logics of programs. The objective is to formalize a notion of program description, and to give both plausible (semantic) and effective (syntactic) criteria for the notion of truth of a description. A novel feature of this treatment is the development of the mathematics underlying Floyd-Hoare systems independently of such systems. Other directions that such research might take are also considered. This paper grew out of, and is intended to be usable as, class notes (27) for an introductory semantics course. The three sections of the paper are

1. A framework for the logic of programs.
Programs and their partial correctness theorems are treated as binary relations on states and formulae respectively. Truth-values are assigned to partial correctness assertions in a plausible (Tarskian) but not directly usable way.

2. Particular Programs.
Effective criteria for truth are established for some programs using the Tarskian criteria as a benchmark. This leads directly to a sound, complete, effective axiom system for the theories of these programs. The difficulties involved in finding such effective criteria for other programs are explored. The reader’s attention is drawn to Theorems 4, 16, 18 and 22-24, as worthy of mention even out of the context in which they now appear.

3. Variations and extensions of the framework.
Alternatives to binary relations for both programs and theories are speculated on, and their possible roles in semantics are considered. We discuss a hierarchy of varieties of programs and the importance of this hierarchy to the issues of definability and describability. Nodal logic is considered as a first-order alternative to Floyd-Hoare logic. We give an appropriate axiom system which is complete for loop-free programs and also puts conventional predicate calculus in a different light by lumping quantifiers with non-logical arguments rather than treating them as logical concepts.

Proofs of all theorems are relegated to an appendix.

1. A framework for the logic of programs.

1.1 Semantics: what a program is

In this paper we restrict our attention to programs that primarily manipulate and test their environment, in contrast say to the pure lambda calculus, whose semantics need not depend on the notion of a changing environment. Floyd-Hoare logic is aimed at the former kind of program, which does not readily lend itself to direct description using classical logic. Lambda calculus and pure LISP programs fare much better with classical logic. However, the manipulate-and-test paradigm dominates the programming milieu, and the popularity of the Floyd-Hoare method for dealing with this situation makes a foundational study of the method worthwhile.

The term semantics will connote for us the relation between word and object. Two such relations appear below, as formula and truth-value, and as concrete program and abstract program (cf. Scott (31)). When necessary we will refer to these respectively as s-semantics and E-semantics. This section deals with the latter, although we do not explicitly discuss concrete programs. (Section 3 (tota la) raises the possibility that the concrete/abstract dichotomy is too narrow a point of view for E-semantics.)

Binary Relations. We shall use binary relations for programs along lines proposed by Ellenberg and Elgot (13), de Bakker (9,10,11), and (with relations replaced by functions) Scott (9,31). We find it convenient to use them also for partial correctness theorems of programs.

We define a binary relation R from a set A (the domain of R) to a set B (the range of R) to be a subset of A x B (10) (as opposed to a function from A to B (13), which is not as convenient for our purposes). We further define

- \( R \subseteq A \times B \)
- \( R \cup S \subseteq A \times B \) for any sets, infinite union and intersection included
- \( (a,c) \in (b,c) \) (composition)
- \( (a,b) \in R \) (inverse)
- \( xRy \) for \( xa \in y \) for \( x \in A \)
- \( ax \in y \) for \( x \in B \)
- \( X \subseteq Y \) for \( \forall x (x \in X \rightarrow x \in Y) \)

Symbols. Central to the notion of environment is the symbol and its value (for interpretation, or denotation). We shall confine our attention to function symbols, predicate symbols, and logical connectives, interpreted respectively as functions, predicates, and either boolean functions or binary relations (see (2) below), all of fixed arity. We denote the collections of such symbols as \( J \), \( P \) and \( C \) respectively, and use subscripts to identify the collections of a given arity: thus \( J \) is the collection of binary function symbols, \( C \) will always include \( \land \), \( \lor \) and \( 3x \) for all \( x \in J \) (i.e., first-order quantification, though very little of what we prove changes if we permit \( 3x \) for all \( x \in J \), while \( P \) will always include \( = \). We let \( D \) denote the (single) domain for the functions and predicates.

We adopt the following notations.

- \( A : B \) the set of all functions from \( A \) to \( B \)
- \( f : A \rightarrow B \)
- \( A^k \) (Cartesian product of \( k \) a’s)

Expressions. Expressions are trees whose vertices are labelled with symbols such that:
- (i) each symbol’s arity equals the out-degree of the vertex it labels
- (ii) going from the root to a leaf, the sequence of symbol-types encountered forms a contiguous substring of \( \sigma \rightarrow \sigma^e \)

We use the following concepts and notations.

Formula An expression whose root’s label is in \( D \sigma \)
Term An expression whose root’s label is in \( \tau \)
\( E \) The set of expressions
\( E^r \) The set of formulae of \( E \)
\( E^t \) The set of terms of \( E \)
Ground Describes an expression containing no modalities.

Useful abbreviations and their expansions are:

- \( P \in D \)
- \( \neg (\Phi \land \Psi) \) (mutual, multands for \( \sigma, \tau, \pi, \xi \))
- \( \Phi \land \Psi \) (mutual, multands for false)
- \( E \) \( E_F \) \( E_{F_1} \) \( \ldots \) \( E_{F_k} \)
- \( \Phi_E \) \( \Phi_{E_1} \) \( \ldots \) \( \Phi_{E_k} \)
- \( 3 \) \( 3_{E_1} \) \( 3_{E_2} \) \( 3_{E_k} \)
- \( \langle 3_{E_1}, \ldots, 3_{E_k} \rangle \)
- \( \langle (\Phi_{E_1}), \ldots, (\Phi_{E_k}) \rangle \)

Interpretations. We now assign meaning to expressions, along the lines spelled out by Tarski (36). An interpretation \( \sigma \) (which for us will play the role of an environment) specifies for each symbol \( A \) the value \( A_\sigma \) in \( \sigma \). Given \( \sigma \), we can then infer the value in \( \sigma \) of an expression \( E = A_E \). The value will be written \( \Phi_{E_\sigma} \) (slightly generalizing the usual usage), and is defined by
1.2 Logic: how to describe a program

So far we have said what a program is, a E3-sentential concern. We now consider ways to talk about programs, a concern of logic.

Partial correctness assertions. A partial correctness assertion (pca) is an ordered pair of formulæ of $\mathcal{L}_1$, called respectively the antecedent and the consequent. Pca's were first studied carefully by Floyd [11], who called them verification conditions. They were later further popularized by Hoare [16]. Though they do not constitute the only possible description language, as we shall see in section 3, and also are top-sided in their ability to discuss termination (they can only discuss non-termination), they are nevertheless of considerable practical and theoretical interest. We shall refer to program-oriented logics whose language is $\mathcal{L}_2$ as Floyd-Hoare logics.

The meaning of a pca is defined as an extension to the Tarskian definitions (1) and (2) (t-semantics). We extend it so that it is defined not only on $\mathcal{L}_2$ but also on $U^{2_1}$, as follows:

$$\langle s,j \rangle \vdash (P,0) \iff (s,P) \vdash 0$$

That is, a transition satisfies a pca, or the pca is true of the transition, when the truth of the antecedent before the transition implies the truth of the consequent after. We refer to these two usages of $\vdash$ as unary and binary respectively; more generally, we distinguish conventional logics from Floyd-Hoare logics by calling them respectively unary and binary logics. Unary logics deal with static situations, binary logics with dynamic situations.

For a set $a$ of transitions (i.e., a program), $\vdash (P,0)$ means that $(P,0)$ is true of every transition in the program as we then say $(P,0)$ is true of $a$, and that a satisfies $(P,0)$. Similarly, $\vdash a$ denotes the set of pca's true of $a$, which we shall call the partial correctness theory of $a$, abbreviated to $(a)$ (following Hoare [16], but using boldface to distinguish $\mathcal{L}_2$ from set brackets $\{\}$). Since $(a)$ is a set of pairs of formulæ we can treat it as a binary relation on $\mathcal{L}_2$ and write $$(s,P) \vdash (s,P)$$

One can think of $(P,0)$ as providing an upper bound on programs, in the sense that the programs satisfying $(P,0)$ are just the subsets of $\mathcal{L}_2$. In this role, $(P,0)$ can assert non-termination, but because $\vdash(P,0)$ for any $(P,0)$, it cannot assert termination.

The Duality Principle for Programs. In static logic there is a duality between true and false. In dynamic logic a similar duality obtains between forward and backward execution of programs. The (easily checked) Duality Principle for a program $a$ is

$$\lambda a \vdash a \iff \lambda a \vdash a$$

where $\lambda a$ is defined as $(a,\lambda a)$. Thus $\lambda a \vdash q$ is equivalent to $\lambda a \vdash q$. This principle can occasionally simplify discussion of forward execution by reducing it to backward execution or vice versa. The axiom of modus tollens in static logic ($a \vdash b \vdash c \vdash a$) can be thought of as the duality principle applied to the program $\lambda a$.

Weakest Antecedent and Strongest Consequences. We observed earlier that $(a)P$ could be interpreted as "after executing $a$, $P$ holds." It follows that $(\lambda a)(a)P$ holds. Moreover, no weaker antecedent than $(a)P$ will permit the consequent $P$; indeed, if $(a)P$ then $(a)\vdash (a)P$, so there exists $\lambda a \vdash (a)$ satisfying $(a)P$ such that $\vdash (a)P$. We call any formula logically equivalent to $(a)P$ a weakest antecedent (Dijkstra [121] of $P$ via $a$). This is summarized by

$$\vdash (\lambda a)(a)P \iff (a)P$$

By the duality principle (D), the above holds equally well for $(a)P$ but $(\lambda a)P$, and hence $(\lambda a)(\lambda a)P$, and hence $(\lambda a)(\lambda a)(a)P$, and so forth. By weakening $\vdash (a)P$ we call any formula logically equivalent to $(a)P$ a strongest consequent (Floyd [151] of $P$ via $a$). The dual of (D) is

$$\vdash (a)P \iff (a)P \vdash (a)P$$

This is summarized by

$$\vdash (a)P \iff (a)P \vdash (a)P$$

110

Thus:

$$\vdash (P \lor (a)0) \iff (P \lor (a)0)$$(a stronger statement of (D))
Though we have given syntactic characterizations of weakest antecedent and strongest consequent, these translate immediately into semantic characterizations by virtue of our having already specified the semantics of modalities. This approach is slightly more convenient to work with than defining the concepts directly in terms of interpretations, particularly since we need the concept of modality for other purposes.

Tidy Programs. Including the modality $<a>$ in $Q$ is not really critical, since the whole idea of Floyd-Hoare logic becomes superfluous (see Section 3.2 for details). We shall limit $Q$ to $\land$, $\lor$, and $\exists x$ for all $x \in Q$. In this case we may ask whether $Z$ contains any formula logically equivalent to $[a]P$.

If for a given program $a$, the answer to this question is yes for all $P \in a$, we say that $a$ is forward tidy. (Schwarz [32] uses the terminology "backward exactly connected"). The dual epiphil is forward tidy (Pratt [27] and Independently Schwarz [32]). Note that the concept of tidiness finds no application in de Bakker's and Fleerens' [11] $\mathcal{E}$-semantical treatment of partial correctness, where for an "anecdotal" $V$ they define the strongest "consequent" via $a$ to be $V_a$, which will always be defined. While elegant, this is not logic, that is, the concept of language does not appear; they are talking about a different though closely related problem.

A program that is either forward or backward tidy we shall call tidy: when it is both, we shall call it very tidy.

When $a$ is forward tidy it is convenient to have a function $aa : Z \rightarrow Z$, such that $aa$ takes $P$ to a stronger consequent of $P$ via $a$. We call $aa$ a forward tidiness function of $a$. If there exists a recursive $a$, we shall say that $a$ is recursively forward tidy. For convenience we will sometimes treat $aa$ as a binary relation, writing it as $a \rightarrow a$. A backward tidiness function of $b$ (if any) is written $\langle b \rangle$; $\langle cb \rangle$ will denote the converse of the relation corresponding to the function $\langle b \rangle$. The following facts formalize this.

\[ \exists P (a \rightarrow a) \land (a \rightarrow a) \lor (a \rightarrow a) \]  
\[ \forall P (a \rightarrow a) \land (a \rightarrow a) \lor (a \rightarrow a) \]  
\[ \exists P (a \rightarrow a) \land (a \rightarrow a) \lor (a \rightarrow a) \]  
\[ \forall P (a \rightarrow a) \land (a \rightarrow a) \lor (a \rightarrow a) \]

A program may have many tidiness functions, any one of which will serve our purposes. The following is useful.

Tidiness Duality Lemma (TDL). A program $a$ is forward tidy if and only if $\langle a \rangle$ is backward tidy.

The following lemma supplies one valuable role for tidiness; see Theorem 7 below for another equally valuable role.

Tidiness Characterization Lemma (TCL).

(a) Let $a$ be forward tidy. Then $\langle a \rangle = \langle a \rangle \rightarrow \langle 1 \rangle$.

(b) Let $b$ be backward tidy. Then $\langle b \rangle = \langle 0 \rangle \rightarrow \langle cb \rangle$.

With this lemma, to know a tidiness function of $a$ is to know the theory of $a$, or at least to reduce the problem to knowing the theory of $\langle 1 \rangle$ in the sense of Theorem 12 below.

We have now completely specified the notion of truth for pca's with respect to a given program. Note that the definition is plausible (what simpler rigorous yet direct definition of truth could there be?), but not accessible (evaluating the truth of $P(a) \rightarrow P(\langle a \rangle)$ directly from the definitions may bog down in the infinites of either the set $a$ of transitions (when checking $I_{P(\langle P(\lnot Q) \rightarrow P(\langle 1 \rangle)$ for each $t \in a$) or the universe $U$ of states (when evaluating $\exists x P(t)$)). Hence we would like to trade off plausibility for effectiveness, leading to an axiom system for $\alpha$ that is sound, complete and effective. This trade-off has its analog in unary logic. In both logics this gives rise to the need to distinguish truth and proof.

In general $\alpha$ is not accessible in the above sense; however, in some simple yet useful cases, $\alpha$ is quite accessible. The following section focuses on such special cases.

2. Particular programs

2.1 Basic Programs.

The Identity and Empty Programs. The empty relation $\emptyset$, containing no transitions, and the identity relation $1_{\emptyset}$, which we shall henceforth abbreviate to $1$, are two simple programs of particular interest. These are characterized by the properties $a \rightarrow P = P \lor a = a$ and $a \lor 1 = 1 \rightarrow a$. Thus they resemble - and in fact are - the additive and multiplicative identities respectively of a semi-ring (ring with no additive inverses) with addition operator $\lor$ and multiplications operator $\rightarrow$.

(All proofs in this section are relegated to an appendix.)

Theorem 1. \[ \langle \emptyset \rangle = Z \]

Theorem 2. \[ \langle 1 \rangle = \langle I_{\emptyset} \rangle \rightarrow \langle I_{\emptyset} \rangle \]

For convenience, rather than referring to $\emptyset$ when we want to talk about the set of valid formulae of unary logic we shall use $1$. Though this implies a restriction of $Z$, to implications, this is a trivial restriction in our case. (Recall Floyd's remark [15]: "One may say facetiously that the subject matter of formal logic is the study of the verifiable interpretations of the program consisting of the null statement."

Tests. A test is a formula $P$ of $L$, and denotes the program

\[ \langle P \rangle = 1_{\emptyset} \rightarrow \langle P \rangle \]

\[ = (\langle 1 \rangle \lor \langle \emptyset \rangle) \rightarrow (\langle 1 \rangle \lor \langle \emptyset \rangle) \]

Thus $\langle P \rangle$ will execute (without side effects) just when $P$ is true. (The $\langle 1 \rangle$ is borrowed from Scott [31].) Though we have reserved the word "test" for $P$ itself, we shall also refer to $\langle P \rangle$ as a test when the meaning is clear. Observing that $1 = \{true\}$ and $\emptyset = \{false\}$ allows us to subsume many theorems or axioms about $1$ and $\emptyset$ under those about tests.

Theorem 3. Let $R$ be a test.

(a) $\langle R \rangle \rightarrow \langle P \rangle$ if $R \rightarrow P$ (forward tidiness)

(b) $\langle R \rangle \rightarrow \langle P \rangle$ if $R \leftarrow P$ (backward tidiness)

The Tidiness Characterization Lemma allows us to deduce the remainder of $\langle R \rangle$.

A ground test has no modalities, and corresponds to the sorts of tests permitted in, say, ALGOL. Though the above theorem did not rely on tests being ground, when we come to exhibit particular programs to make a point or prove a theorem, we will restrict ourselves to ground tests.

Assignments. An assignment is a pair of terms $F(\Sigma, T)$ of $L$, corresponding to the left and right sides of a conventional assignment statement. No loss of generality ensues from parsing the left-hand side as $F(\Sigma)$; all expressions can be so written. When $\Sigma$ is a 0-tuple we have simple variable assignment; otherwise we have array assignment. Since the array arguments are not constrained to be integers, this encompasses the notion of record in, say, Pascal [35]. For Floyd-Hoare logic no distinction need be drawn between functions and arrays. The corresponding program is a function of type $U \rightarrow U$ which of course is just the special case of a binary relation on $U$ where each element of $U$ appears in the relation as the first component of a pair just once. It may be defined with the aid of $\lambda$-notation thus.

\[ EF(\Sigma, T) = \lambda x. \lambda a. \text{if A is true then } a \lambda x. \text{if } x \in \Sigma \text{ then } a \lambda b. \text{else } b \]

(Notes $\Sigma$ and $T$ agree in arity.)

It will help in following this definition to keep in mind the following types of functions

\[ EF(\Sigma, T) : U \rightarrow U \]

\[ \Sigma = \{ \text{int} \} \]

(At least in this case, only terms involved)

\[ \Sigma = \{ \text{int} \} \]

111
It should be remarked that this is not meant as an interpretive definition of assignment in the sense that to execute an assignment one has to use the body of the definition. Rather we are defining a mathematical object, which the body uniquely specifies given \( U \). No detail of this object may be changed without doing violence to the intent of the definition, though of course as in any definition the wording of the definition may be varied.

Before proceeding to a characterization of \( E \) we introduce the notion of substitution, suitably generalized to handle arrays. We define \( F(\{z_i\}P) \) (abreviated as \( P^1 \)), the result of substituting the expression \( T \) for the subtrixes in \( P \) with root \( F \), as follows:

\[
\begin{align*}
E &= E_1 / E_2 & (S1) \\
A &= A \land T & (S2) \\
(2P) &= \langle y \mid (y, T) \rangle & (S3)
\end{align*}
\]

\( S_1 \) performs the substitution for \( F \); its right side simplifies to \( T \) when \( F \) is zercary. Clearly \( E \) is \( (E_1, \ldots, E_n) \). Substitution of this for \( z_i \) in \( T \) means substitution of the whole tuple wherever \( z_i \) occurs in \( T \). \( z_i \) is not a tuple of \( T \) but rather a place-holder (for the arguments of \( F \)) that we shall employ below in instances of \( T \).

\( S_2 \) caters for referentially transparent symbols. The only modality we provide for is \( x \) as in \( S_3 \), which is sufficient for our purposes. (It is easy to check that \( S \) may be replaced by \( V \) in \( S_3 \) with no other modification since \( x \) is referentially transparent and covered by \( S_2 \).

When \( P \) contains assignment modalties (not the case in this paper), a difficulty arises in extending \( S_3 \), namely that of generalizing renaming of bound variables. The reader interested in pursuing this further might consider the supposedly valid formula \( x_0 \geq \langle x_1, y_1 \rangle \| x_2, y_2 \rangle \geq \langle x_3, y_3 \rangle \), which is in fact not valid if the substitution is performed naively. This may be transformed to \( x_0 \geq \langle x_1, y_1 \rangle \| x_2, y_2 \rangle \geq \langle x_3, y_3 \rangle \) to avoid this problem, along the lines of \( S_3 \), but is this desirable? The reason this is not a problem for \( \exists x \) thought of as \( \langle x \mid \text{Random} \rangle \) is that \( \text{Random} \) is independent of \( x \), so renaming it to \( \langle z \mid \text{Random} \rangle \), or for that matter renaming \( \langle x \mid \text{Is} \rangle \) to \( \langle z \mid \text{Is} \rangle \), is not as disturbing as renaming \( \langle x \mid \text{Is} \rangle \) to \( \langle z \mid \text{Is} \rangle \). Clearly we cannot rename it to \( \langle z \mid \text{Is} \rangle \) without renaming other occurrences of \( x \) possibly outside the scope of the substitution, as in the example where we have \( x \geq 0 \). It would appear that renaming to \( \langle z \mid \text{Is} \rangle \) is our only option. But then what happens in the case of array assignment? We would appreciate seeing a solution to this problem.

In addition to substitution we need a temporary addition to \( S_3 \), namely \( IF-THEN-ELSE-P \), a peculiar symbol taking a formula for its first argument and terms for its second and third arguments. It is removed (in order to yield an expression of \( \lambda \)) by the following transformations, which move it up the tree using the first two transformations until all its arguments are formulae, permitting application of the third transformation.

\[
G(IF\ R\ THEN\ E\ ELSE\ F) \\
\quad \quad = (IF\ R\ THEN\ G(E)\ ELSE\ G(F)) \quad \text{for } G \in \text{JUP} \\
\quad \quad = (IF\ R\ THEN\ E\ ELSE\ F) \quad \text{(to facilitate preceding rule)}
\]

IF-THEN-ELSE-P

When \( E \) is a formula containing IF-terms yields the same truth value whether \( IF \) is first removed from the above transformations or left in place and evaluated using

\[
\text{IF}(IF\ P\ THEN\ S\ ELSE\ T) = \text{if } IF\ P\ THEN\ S\ ELSE\ T
\]

The following reports joint work with R. Hale (271).

Theorem 4. Let \( F,G,T \) be an assignment.

1. \( F = G = T = F \) if \( F = G = T \).

The \( = \) cannot be strengthened to without knowing more about \( a \) and \( b \). For example, let \( ac \) have no \( P \) satisfying \( y = z \), satisfying \( P = V \). Let \( a = 1 \cdot y = \langle d, 3 \rangle | S \) and \( b = I \cdot y = \langle d, 3 \rangle | S \), then \( \text{let } ab = P \), so \( ca \) is vacuously \( \langle d, 3 \rangle | S \), the set of all \( p \)'s, including \((\text{true},\text{false})\). But by the construction there can be no \( P \) simultaneously satisfying \( \text{true}(x) \) and \( \text{false}(x) \), so \((\text{true},\text{false})\) cannot be in \((a+b) \), whence \((a+b) \neq (a+b) \).

The special case, namely that

\[
\langle x \mid T \rangle = \langle T \rangle
\]

We shall often abbreviate \( \langle x \mid T \rangle \) for \( \langle T \rangle \) when \( \langle x \mid T \rangle \) is much alike, and we rely on \( = \) versus \( \neq \) to disambiguate them. Actually, since they are equivalent, the only reason one would want to distinguish them is when one wants to stress that \( \langle T \rangle \) is an abbreviation for something in \( L \) since \( \langle x \mid T \rangle \) is an unabbreviated formula of modal logic. Thus \( \langle x \mid T \rangle \) is semantic inasmuch as it has an interpretation under \( F \), while \( \langle T \rangle \) is syntactic in that it specifies a transformation on an expression.

Lambda-calculus adherents will note the obvious similarity between \( \langle x \mid T \rangle \) and \( \langle x \mid \mathcal{P} T \rangle \); our above equivalence corresponds to the syntactic beta-reduction rule of the lambda-calculus. Our generalization to array assignment gives the appropriate rule for a lambda-calculus with arrays where single array elements can be bound, as in \( A(y) \mid A(x) = 1 \), where due regard needs to be given to whether \( x \neq y \).

Second-Order Assignment. We may call the above programs first-order assignments because individuals of \( \mathcal{D} \) are being "moved around." A second-order assignment might be a pair of function symbols of the same arity, and would permit wholesale assignment of a function to a function symbol of the same arity. Thus if \( F \) and \( G \) were both binary, \( F \cdot G \) would change the whole interpretation of \( F \), not just its value at one point. The program \( EF \cdot G \) would then be

\[
\lambda x.\lambda y.\text{if } F \neq \text{then } G \neq y
\]

This notion of second-order assignment is not as general as it might be. For example, one might want to perform \( F \cdot G = H \) where \( F,G,H \) are all unary. However, this would introduce higher type functionals (in this case composition) into the language, which would make matters more complicated than we are willing to allow here. (This is not to imply that \( EF \cdot G = H \) is not tidy -- it is backward tidy, by a variation on the argument in the proof of the following theorem, that \( EF \cdot G \) is backward tidy.)

Thinking of the first order quantifier \( \exists x \) as \( \langle x \mid \text{Random} \rangle \), we can think of the second-order quantifier \( \exists \) as \( \langle x \mid \text{Random} \rangle \). This illustrates just how close our use of "second-order" is to the conventional use.

Theorem 5. Let \( F \cdot G \) be a second-order assignment. Then \( EF \cdot G = H \) if \( (G \cdot F) \) is a convenient abbreviation for \( (G \cdot F) \).

Hence second-order assignment is backward tidy.

The problem is \( F \cdot G \) always forward tidy? No; this amounts to asking whether one second-order quantifier helps. In general it does not.

Union. We have already defined the union of two binary relations as being conventional set union, taking advantage of the representation of relations here as sets of transitions.

Theorem 6. \( \{a \cup b\} = \{a \cup b\} \).

The exact analog of this binary logic theorem in unary logic: in both logics, "the theory of the union of two subsets of either \( E \) or \( E \) is the intersection of the theories." In contrast, there is no analog of the following theorem in unary logic, in line with the idea that composition is a dynamic rather than a static operation.

Composition. Again, we have already defined the composition of two binary relations.

Theorem 7A. \( a \cdot b \geq a \cdot (a \cdot b) \).

The \( \geq \) cannot be strengthened to without knowing more about \( a \) and \( b \). For example, let \( a \) have no \( P \) satisfying \( y = z \), satisfying \( P = V \). Let \( a = V \cdot y = \langle d, 3 \rangle | S \) and \( b = I \cdot y = \langle d, 3 \rangle | S \), then \( \text{let } ab = P \), so \( a \cdot b \) is vacuously \( \langle d, 3 \rangle | S \), the set of all \( p \)'s, including \((\text{true},\text{false})\). But by the construction there can be no \( P \) simultaneously satisfying \( \text{true}(x) \) and \( \text{false}(x) \), so \((\text{true},\text{false})\) cannot be in \((a+b) \), whence \((a+b) \neq (a+b) \).
Neither a nor b in this example is tidy, and in fact we can strengthen theorem 7A as follows.

**Theorem 7.** \((a,b) = (a') \iff (b') = (b)\) when a is forward tidy or b is backward tidy.

**Theorem 8.**
(a) If a, b are forward tidy, then all a+b and a-b are forward tidy. 
(b) If a, b are backward tidy, then all a+b and a-b are backward tidy.

**Loop-free programs.** The significance of union and composition is that, together with tests and assignments, they allow us to synthesize the abstract programs that correspond to loop-free flowcharts. The correspondence between the two may be formalized as follows. Define a flowchart to be a directed graph with edges labelled with tests and assignments (cf. (17)), and having a start vertex and a set of final vertices. Take the corresponding binary relation to the union, over all paths p from the start vertex to a final vertex, of the composition of the sequence of instructions along p. In the case of loop-free flowcharts, i.e., directed acyclic graphs, there can only be finitely many such paths, so such an abstract program can be synthesized from tests, assignments, finite union and composition. The foregoing theorems then tell us:

**Corollary 9.** All loop-free assignment-and-test programs are very tidy (possibly exceeding forward tidiness for second order assignment).

So far we have considered only the programming constructs of tests (subsuming I and P), assignments, finite union and composition. We could proceed to consider further constructs such as if-then-else along the same lines. However, our preference in this case is to consider "if P then a else b" to be an abbreviation for "P * a U (¬P) * b", much as we considered X*P to be an abbreviation for "¬X * P". Similarly, we would regard the goto construct as a notation for describing flowchart programs textually, providing this gave rise to acyclic flowcharts, allowing us to further translate the flowchart into a program involving only tests, assignments, finite union and composition. We discuss the case when the goto gives rise to a loop later, under the heading of regular programs.

If one wanted to be more formal one might distinguish translational semantics from I3-semantics, classifying our definition of if-then-else as being of the former kind. The economies of description possible with such translational definitions do not need stressing.

**Recursiveness.** Tidiness by itself does not guarantee usability of the tidiness functions. We say that a is forward (backward) recursively tidy when a++ (a--) is recursive. In the following we use "many-one reducibility" (301); we say \(X \leq_m f (x)\) when there exists a recursive function f such that x \(\in X\) iff \(f(x) \in Y\).

**Theorem 10.** \((a,b) \leq_m (a',b')\) (Cartesian product).

**Theorem 11.**
(a) If a is forward recursively tidy, \((a,b) \leq_m (a',b')\).
(b) If b is backward recursively tidy, \((a,b) \leq_m (a',b')\).

**Theorem 12.** If a is recursively tidy, \((a) \leq_m (a')\).

**Theorem 13.** Instructions are recursively very tidy.

**Theorem 14.** If a, b are forward (backward) recursively tidy, so are a+b and a-b.

By themselves these theorems are somewhat dull. Taken together, however, they yield the following interesting result, used to advantage in King's thesis (18), using backward tidiness.

**Corollary 15.** If a is a loop-free assignment-and-test program, \(\leq_m (a')\). (Note that \((E) = (E(x_1) \ldots x_n) \leq_m (E')\), for any n, since the n questions about membership in \((E)\) can be rephrased as a single conjunction.)

This asserts that to decide whether \((P, D)\) is true of a, it suffices to ask whether a given first-order predicate calculus formula holds.

It follows that the theory of programs without loops is no less tractable than the classic "theory of the underlying logic."
little more care. The idea is to say enough in the antecedent $P$ referred to above to contain the domain to have a substructure isomorphic to the natural numbers with $0$ and successor.

The fact that $a$ is a universal program plays an important role in these proofs. Thus the following theorem is of considerable interest.

**Theorem 15.** Let $|S| \geq 4$, $|T| \geq 3$, $|F| \geq 1$, with $V \subseteq S$, $F \subseteq T$. Let the symbols of $S$ and $P$ (except $=$) take on all possible interpretations in the universe $U$. Then $\text{EFV}\neg(\neg\text{EFV})$ is not r.e., despite \( \text{EFV} \) and $\text{EFV}\neg(\neg\text{EFV})$ both being r.e.

**Corollary 17.** When $\text{EFV}$ is r.e., $\text{EFV}\neg(\neg\text{EFV})$ is not recursively tidy.

(Alter Theorem 24 we will be able to strengthen this by dropping "recursively").

The proof of Theorem 15 appears to take advantage of the fact that $F$ is uninterpreted, by allowing us to say "if $F$ were interpreted as a single-stepper for a universal machine, then..." The following lends credence to that view.

**Theorem 18.** If $s \in P$ then $\text{EFV}\neg(\neg\text{EFV})$ is recursively very tidy.

**Invariance Theories.** A sense in which $*$ is tractable can be found in the invariance theory of $a$, written $(a)$, which is $(s|n)|L$, the pca's $(P, P)$ that express invariance.

**Theorem 19.** $(s|n) = \text{EFV}(s|n)$.

**Theorem 20.** $(a \cdot b) = (a) \cdot (b)$.

**Theorem 21.** $(a \cdot b) = (a) \cdot (b)$.

This inequality $\geq$ cannot be strengthened to $=$ even if we make $a$ and $b$ tidy or make $a \cdot b$, as witnessed by $a = b \cdot \text{EFV}(\text{EFV})$ where $F$ is uninterpreted. For let $Q = P(X) \cdot V(Y) = P\cdot P(Y)$. Then $Q = (a \cdot b)$ but not $(a \cdot b)$.

Compare this with the way tidy came to the rescue in Theorem 7. An amusing consequence of Theorem 21 is:

**Corollary 22.** For a given program $a$, the structure $\langle (\text{EFV}^*)_{|\text{AOD}}, \text{S} \rangle$ is a homomorphism of the natural number division lattice $(\mathbb{N}, |)$, with $(a)$ as the least element and $(1)$ as the greatest. Further, when $a = \text{EFV}(\text{EFV})$ with $F$ uninterpreted, the homomorphism becomes an isomorphism.

Considering that invariance theories fare less well with $*$ than do full theories (as per Theorem 7), we should not be too surprised to find in view of Theorem 16 that invariance theories run into difficulties with $*$ as well. This however is not the case.

**Theorem 23.** $(a)^* = (a)$.

We can now add to our axioms:

\[ P(a)^* \vdash P(a)^* \]

\[ P(a)^* \vdash P(a)^* \text{ (comm, hom)} \]

**3. Extensions.**

The theory of sections 1 and 2 is based on quite simplistic notions of program (binary relation on states) and theory (binary relation on formulæ). Dealing with other program constructs than union, composition and reflexive transitive closure may not always be possible in this framework. We explore this in section 3.1 as the definability problem. For example, the notions of concurrent process, block structure, and call-by-name, seem not to be definable for binary-relation programs. We broaden the usual notion of "mathematical semantics" as "1/O semantics" to embrace a variety of notions of "abstract program." In section 3.2 we look at one approach to the problem of extending the descriptive adequacy of Floyd–Hoare logics, which is handicapped by its ability to be only an upper bound (with respect to inclusion) on programs.

**3.1 A Program Hierarchy.**

In this section we cease to identify programs with binary relations on states, for we will be considering a hierarchy of 7 kinds of programs. In order of decreasing information, this hierarchy is
In the context function, only one weight is associated with each transition. Thus the context function corresponding to the multilateral binary relation \( R \) is a function from \( U^2 \) to \( \mathbb{N}(c) \) which maps \((a,b)\) to the least \( n \) such that \((a,n,b)\) is in \( R \). Let \( q = (a,b) \) be such a pair. The intuition is that this function gives the fastest way of getting from \( a \) to \( b \).

By composing the context function with the function that maps \( w \) to 0 and everything else to 1, we get a function from \( U^2 \) to \( (0,1) \) that can be considered to be a binary relation in the usual way. We have reached the binary relations that we used in Sections 1 and 2.

Finally, by projecting a binary relation onto its first coordinate, we get the domain of that relation, namely those states that lead to a final state. This supplies enough information to discuss transition without getting specific as to what state the program terminates in.

There is an interesting trade-off here between definability and describability. As one moves down the hierarchy, programs become more describable, but operations on programs become less definable. The reason Floyd-Hoare theories describe type (vii) programs easily is because these are so low in the hierarchy. A theory of termination applied to type (vii) programs is even easier; the set of initial states that lead to a final state can be described with formulae in \( L \), with truth defined via unary \( h \) as usual. On the other hand, there are almost no proposals in the literature for languages suitable for describing programs of types (ii)-(v), other than in the trivial sense in which they describe the information in the program preserved in the transition to level (vii). An exception is Kroeger’s (21) notion of “thickness,” capturing running time; this appears explicitly in his modal language, but no formal semantics analogous to (2) of our section 1.1 or (3) in 1.2 is given in (21), and it is not clear to us how to construct such a semantics based on our level (v). This level is of particular interest because it incorporates the minimum information needed to describe the running time complexity of a program.

In considering definability we will start with determinism and totality, then turn to other operations. The notions of determinism and totality depend on which kind of program one is discussing. For example, if we are discussing level (vii) programs, then a deterministic program would be a function. We call this \( \text{G-determinism} \) to correspond to level (vii), or more mnemonically and independently of our particular hierarchy, TD-determinism (for input/output). At level (vii) determinism is not definable. A reasonable definition of a \( \text{G-deterministic} \) (level i), G for Grammar) program written as a flowchart (directed graph) might be that it satisfies:

(a) All final nodes are leaves (i.e., have out-degree 0),
(b) If there exist distinct edges \((u,v)\), \((v,w)\) then they are labelled with tests not simultaneously satisfiable.

Alternatively one could refine the same condition in terms of domains of the instructions labelling edges:

(a) If \( u \) is final, all edges \((u,v)\) have empty domains;
(b) distinct edges \((w,u)\), \((v,w)\) have disjoint domains.

To define \( \text{Z-determinism} \) (for L-determinism), it helps to know the notion of the prefix tree of a language. For \( L \subseteq X \), let \( w \) be the word \((w,z) \) of the immediate prefixes of \( L \), and let \( s = \emptyset \) be the least \( L \) satisfying LULwL = L, the prefixes of \( L \). Then the prefix tree of \( L \) is the directed graph \((T,L) \) = \((s,\emptyset) \) wzhwzLz). (Recall that graphs are presented as \((V,E)\) where \( V \) is the vertex set and \( E \) the edge set.) Consider the edge \((y,v)\) for \( (z) \) to be labelled \( w \) if \( \text{call } (z) \) to be labelled \( w \) if \( (z) \) in \( \text{L} \) final. Clearly all leaves are final, but the converse does not necessarily obtain.

A program represented as a language has such a prefix tree, which is the non-deterministic non-total analogue of decision trees (25). Such a tree can be executed by starting at the root (guaranteed to exist since every language has an empty and following a path along which no tests evaluate false. Halting is permitted only at final nodes. Since we have produced \( L \) (a possibly infinite) state transition diagram that generates \( L \), we have an object with which we can apply whatever definition we used for \( \text{G-determinism} \) to this graph. Hence we can say that a
program is \( L \)-deterministic just when the prefix tree of the language representation of the program is \( G \)-deterministic.

Totality is definable at all levels. Extending our notion of \( k \)-determinism in the obvious way, \( 7 \)-totality (or \( D \)-totality) simply means that the domain is all of the universe \( U_k \). For \( k \)-totality seems to be the least defined as \( D \)-totality, whereas \( G \)-totality should be a syntactic notion that for a flowchart would say that for every non-final vertex \( w \) there should exist either an assignment edge \( (w, u) \), or a set of edges \( (w, u_1, \ldots, u_l) \) whose labels are tests such that \( \text{PR}_k \ldots \text{PR}_k \) is satisfiable. Alternatively, we could simply require that for every non-final vertex \( w \) there exists a set of edges \( (w, u_1, \ldots, u_l) \) whose union of whose domains is \( k \). Both these definitions are clearly stronger than \( D \)-totality. For \( L \)-totality we can do as we did for \( L \)-determinism, namely apply the definition of \( G \)-totality to the prefix tree of the language representation of the program.

We now consider which other operations on programs are definable at a given level. The less information a program, the fewer operations that can be defined. The operation of union is ubiquitous, applying to all types of programs. Composition applies to all but the last. An interesting programming language construct I have not seen proposed before is \( \text{"fastest(a)"} \) which computes a path by the fastest possible method, in the sense that if there is more than one way to execute the program \( a \), as there may be in a \( D \)-non-deterministic system, then the fastest should be chosen. This operation is not definable beyond type \( 4 \). An operation useful in operating systems is that of merge (or shuffle), which forms all possible order-preserving merges of the strings of its two arguments. This operation does not seem to be definable beyond type \( 3 \). Recursion is definable at level \( \omega \) \((\omega)\). In a program with recursion and block structure, if each new activation of a variable is regarded as in fact being a new variable (calling for a more sophisticated grammar than a context-free one if the definition is to be performed at level \( \omega \)), then the concept of block structure is not definable beyond type \( 3 \). Call-by-value can be captured at level \( \omega \) by combining block structure with assignment, but call-by-reference seems to call for either a very complex language (i.e., at level \( \omega \) a very powerful grammar) or for a different kind of assignment from the one we have been using, one that can interpret references. Once call-by-reference is provided for, call-by-name can be handled in imitation of the classic method of "thanks," but it too seems not definable beyond level \( \omega \). (I should be pointed out that "is definable" means roughly "makes sense," and does not at present have a better defined meaning.)

When the restriction of the homomorphism from type \( 1 \) to type \( j \) programs to a class \( C \) of type \( i \) programs is an isomorphism, we call \( C \) an \( i \)-preserving class. A program in some class \( C \) has no information which cannot be reconstructed from its type \( j \) counterpart, at least for the purpose of distinguishing it from other programs in \( C \). Knuth [19] (problem 1.2.1-13) describes a transformation on programs that preserves every basic instruction by labeling every new variable. This transformation yields a program \( P \) whose type 5 version is in a 5-6-preserving class, and \( P \) whose type 6 version is identical to the type 6 version of the unmodified program to within the effect on \( P \). The importance of this transformation is that in the transformed program the timing information is not lost in the transition from type 5 to type 6. Hence a pca, which ostensibly only describes type 6 programs, can in effect describe type 5 programs. Since pca's only supply upper bounds on programs, this method requires some independent guarantee of termination. Luckham and Suzuki [22] develop this idea further, and it appears that this guarantee has to come in the interpretation of the pca. They treat this as an application of the "law of the excluded middle.

### 3.2 Modal Logic

In this section we will look briefly at an alternative to Floyd-Hoare logic for describing programs, namely modal logic, a significant advantage of which is that it allows one to talk about correctness and termination in the same first-order language. As might be guessed from section 1, we now need to refer to our convention that programs are binary relations.

Part of this work was done jointly with R. Moore in 1974 [27]. A similar proposal has been briefly sketched by Burstall [5], who suggests that the classical modal logic 55 may be used to discuss correctness and termination simultaneously. Considering that 55 logics are those whose modalities have equivalence relations for their interpretations, we may infer that Burstall was on the right track but had not developed the idea to the point where it too could be seen to be inappropriate, or had a considerably different idea from us of how modal logic was to be applied to the problem. Schrecker [33] has developed Burstall's work further, with a definite commitment to 55. Kroeger [21] has also proposed a modal approach to the logic of programs, in considerably more detail than Burstall, and with a concern for \( k \)-semantics equal to ours. A major difference between our approach and Kroeger's is that we regard programs as interpretations of unary logical connectives, Kroeger regards them as propositional variables. Kroeger's objective however is to reason "naturally" about programs, whereas ours is more general and more elegant than Kroeger's, it would appear that both of us have met our respective (different) goals using different approaches.

Recall from section 1.1 the interpretations of \( aP \) and \( \langle a \rangle aP \). Under these interpretations the following formulate are visibly valid:

- \( (X \times Y, 1) \), \( (X \times Y, 0) \)
- \( (X \times Y, 0) \)
- \( (\langle x \rangle X, 0) \)
- \( (\langle x \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)
- \( (\langle - \rangle X, 0) \)

These particular valid formulas generalize in some obvious ways, which we can call axioms.

**Logical Axioms**

All tautologies of Propositional Calculus.

\( (aP) \Rightarrow (\langle a \rangle aP) \)

**Logical Inference Rules**

- \( P, P \vdash \vdash Q \)
- \( P \vdash \vdash (aP) \) (subsumes \( P \vdash \vdash aP \)).

Some theorems that follow from these axioms are:

- \( (aP) \Rightarrow (aP) \)
- \( (aP) \Rightarrow (aP) \)
- \( (aP) \Rightarrow (aP) \)
- \( (aP) \Rightarrow (aP) \)
- \( (aP) \Rightarrow (aP) \)
- \( (aP) \Rightarrow (aP) \)

**Axioms for Basic Programs**

\( VP \Rightarrow (\langle 0 \rangle P_{\text{TP}}) \) (any \( x \) \& \( \xi \))

**Universal Axiom.**

- \( P \Rightarrow \forall x \text{ SP} \) unless \( x \text{ TP} \) and \( y \text{ TP} \)

**V Frame Axiom.**

- \( \forall x \text{ SP} \Rightarrow (\langle x \rangle \text{ SP} \langle x \rangle \text{ SP}) \)

- \( (aP) \Rightarrow (\langle a \rangle aP \langle a \rangle aP) \)

- \( (\langle a \rangle aP) \Rightarrow (\langle a \rangle aP) \)

**Test Axiom.**

\( (\langle F \rangle \text{ TP}) \Rightarrow (\langle 1 \rangle F \text{ TP}) \) if \( F \text{ TP} \)

**IF Frame Axiom.**

- \( (\langle 1 \rangle F \text{ TP}) \Rightarrow (\langle F \rangle \text{ TP} \langle F \rangle \text{ TP}) \) Assignment Axiom.

**IF Frame-THEN ELSE is removed as in Theorem 4.1.**

The two quantification axioms assert that \( x \text{ RANDOM} \) can change the value of \( x \) to anything, and that nothing but \( x \) gets changed. Note the departure from conventional logic, where both these axioms would be regarded as logical axioms. Because particular programs are non-logical for us in the same sense that the particular function denoted by \( + \) is considered non-logical in conventional logic, and because \( 3x \) denotes a particular program \( x \text{ RANDOM} \), we prefer to think of axioms involving \( 3x \) as non-logical.

The logical axiom \( (aP) \Rightarrow (\langle a \rangle aP \langle a \rangle aP) \) and the non-logical \( V \)-Frame Axiom are combined in Mendelson's [25] system \( K \) of \( x \text{ SP} \Rightarrow (\langle x \rangle \text{ SP} \langle x \rangle \text{ SP}) \). Despite the elegance of such a compression, we feel there is some intrinsic merit in our separation.

Sample theorems that follow from these axioms are:

- **Tests**
  - \( (P \text{ SP}) \)
  - **Theorem of Intent.**
  - \( P \Rightarrow P \text{ TP} \text{ TP} \text{ TP} \text{ TP} \)
  - **Theorem of Performance.**
Manna and Pnueli (24) have proved that this program halted, claiming that such a proof by Floyd’s method of demonstrating termination [15], namely showing that traversing any loop decreased some well-founded quantity, would be very complicated. They proposed another approach. Our modal logic approach supplies yet another first-order approach with the added advantage that it has an elegant semantical basis.

If we permit program modalities in tests, we are in effect allowing behavior conditional on "what might have been," that is, on properties of hypothetical worlds accessed by programs that leave behind no side effects after the test. This gives us a quite simple foundation for the semantics of languages like PLAN[ER] and CONNECT[ER], where such exploratory tests are possible.

4. Appendix

All theorems are re-stated here and proved if necessary.

Tidiness Duality Lemma (TDL): Program a is forward tidy if and only if a' is backward tidy.

\[ \text{Proof.} \] a forward tidy \[ \iff \forall P: a = a' \iff \forall P: a' = a \]

Tidiness Characterization Lemma (TCL): (a) Let a be forward tidy. Then \[ a = (a \rightarrow \sim E) \]
(b) Let b be backward tidy. Then \[ b = (\sim E \rightarrow b) \].

\[ \text{Proof.} \] TDL.

Theorem 1. \( \forall = \exists \).

\[ \text{Proof.} \] \( \forall P \cup 0, \exists \) is true vacuously.

Theorem 2. \( \exists = (P \cup 0) \cup (P \cup 0) \).

\[ \text{Proof.} \] \( \exists = (P \cup 0) \cup (P \cup 0) \) (by \( I \) and (11)), \( (P \cup 0) \cup (P \cup 0) \) (def. of 1), \( (P \cup 0) \cup (P \cup 0) \) (by \( I \)).

Theorem 3. Let \( R \) be a test.
(a) \( \sim E \rightarrow R \rightarrow P \) \( R \rightarrow P \). (Forward tidiness)
(b) \( (R) = R \). (Backward tidiness)

\[ \text{Proof.} \] (a) \( \sim E \rightarrow R \rightarrow P \) \( P \rightarrow R \).
(b) \( (R) = R \) (using (a), and \( (R) = \sim E \).

IF Lemma. Evaluating \( \sim R \) when \( W \) is a formula containing IF-terms yields the same truth value whether IF is first removed by the above transformations or left in place and evaluated using

\[ \sim (IF P \land THEN S ELSE T) = \sim (IF (P \land S \rightarrow T)) = \sim R \]

\[ \text{Proof.} \] Straightforward. (Use induction on depth of IF-terms.)
Theorem 4.a. Let $F \in T$ be an assignment.

(a) $\Delta F(S; T)=3$ if and only if $\exists y(\exists' \wedge \exists'' \wedge F(y)=T')$

where $E' = (\{F \in T \wedge y \exists' \exists'' \wedge F(y)=T\})$

(b) $\Delta F(S; T)=1$ if and only if $\exists z(\exists' \exists'' \wedge F(z)=T'')$

where $E'' = \{F \in T \wedge z \exists' \exists'' \wedge F(z)=T''\}$

Proof. We first prove the aging lemma.

Aging Lemma. Suppose $A_1 = A_2$ for all symbols $A \in F$, and $F_1(y) = F_2(y)$ for all $y \exists y'$. Then $\exists A(B) = \exists A(B)'$. Assume $\exists B = \exists B'$.

Proof. By induction on the height of $A(B)'.

Case (i). $A \in F$.

Subcase (a). $\exists B = x_3$.

$\exists F(B) = F \in T$ if and only if $x_3 \in S$.

$\exists F(B)' = F \in T$ if and only if $x_3 \in S$.

Subcase (b). $\exists B = x_4$.

$\exists F(B) = F \in T$ if and only if $x_4 \in S$.

$\exists F(B)' = F \in T$ if and only if $x_4 \in S$.

Case (ii). $F \in T$.

$\exists F(B) = F \in T$ if and only if $x \in S$.

$\exists F(B)' = F \in T$ if and only if $x \in S$.

Case (iii). Other $A$.

$\exists A(B) = A \in F$.

$\exists A(B)' = A \in F$.

Aging corollary 1. If $\exists F(T; T)$, $\exists F(S; T) = x_3$ and $\exists S = x_4$, then $\exists A(B) = \exists A(B)'$.

Aging corollary 2. Given $\exists$, if $\exists$ is $\lambda x_1 \exists x_2$ then $\exists x_1 \exists x_2$.

Then $\exists A(B) = \exists A(B)'$.

Transition existence lemma. Given $\exists$, if $\exists$ is as in the previous corollary, $\exists = \exists S$, and $\exists F(S; T)$ = $\exists T'$, then $\exists F(T; T)$.

Proof. By aging corollary 2 we have $\exists S = \exists S'$, and $\exists F(S; T)$ = $\exists T'$. Then $\exists F(T; T) = \lambda x_1 \exists x_2$ if $\exists A$ then $A_2$ else $\lambda x_1 \exists x_2$ if $\exists A$ then $A_2$ else $\exists T$.

Then $\exists F(S; T) = \lambda x_1 \exists x_2$ if $\exists A$ then $A_2$ else $\exists T$.

$\lambda x_1 \exists x_2$ if $\exists A$ then $A_2$ else $\exists T$.

$\lambda x_1 \exists x_2$ if $\exists A$ then $A_2$ else $\exists T$.

The preceding lemmas make it straightforward to verify each step.

Case (ii). $A \in F$.

$\exists F(B) = F \in T$ if and only if $x_3 \in S$.

$\exists F(B)' = F \in T$ if and only if $x_3 \in S$.

Case (iii). Other $A$.

$\exists A(B) = A \in F$.

$\exists A(B)' = A \in F$.

Theorem 5. Let $F \in G$ be a second-order assignment. Then $\Delta [F \in G]$ is a conjunction.

Proof. Essentially the same as for Theorem 4(b).
Theorem 8. If \( a, b \) are forward tidy, so are \( a \cup b \) and \( a \cup b \).

Proof. By Theorem 7, \( a \cup b \) is tidy if \( a \succ b \) and \( a \succ b \) are tidy. If \( a \succ b \) is tidy, then by Theorem 7, \( a \cup b \) is tidy if \( a \cup b \) is tidy.

Corollary 9. All loop-free assignment-and-test programs are very tidy (possibly excepting forward tidiness for second-order assignment).

Proof. Use induction on the height of a program, together with Theorems 1–8.

Theorem 10. \((a \cup b) \leq \min(a \times b)\) (Cartesian product).

Proof. \((a \cup b) \leq \min(a \times b)\) (Th 6)

Theorem 11.
(a) If \( a \) is forward recursively tidy, \((a \cup b) \leq \min(b)\).
(b) If \( b \) is backward recursively tidy, \((a \cup b) \leq \min(a)\).

Proof. \((a \cup b) = \min(a \times b)\) (Th 7)

Theorem 12. If \( a \) is recursively tidy, \((a) \leq \min(1)\).

Proof.
(a) If \( a \) is forward recursively tidy, \((a) = \min(1)\)

(b) When \( a \) is backward recursively tidy, \((a) \leq \min(1)\)

Theorem 13. Instructions are recursively very tidy.

Theorem 14. The strongest consequents and weakest antecedents given by Theorems 3 and 4 are easily calculated.

Theorem 15. If \( a, b \) are forward (backward) recursively tidy, so are \( a \cup b \) and \( a \cup b \).

Proof. In all four cases of Theorem 8, the desired weakest antecedents and strongest consequents are easily calculated.

Corollary 15. If \( a \) is a loop-free assignment-and-test program, \((a) \leq \min(1)\).

(Not that \( E[3]^n = (1)(1)(1) \cdots (1) \leq \min(1) \), for any \( n \), since the \( n \) questions about membership in \( \{1\} \) can be rephrased as a single conjunction.)

Theorem 16. Let \( |I_0| \geq 4 \), \( |J_1| \geq 3 \), \( |J_2| \geq 1 \), with \( Y \leq T_0 \), \( F \leq T_1 \). Let the symbols of \( \mathcal{J} \) and \( \mathcal{P} \) (excluding \( \mathcal{E} \)) take on all possible interpretations in the universe \( \mathcal{U} \). Then \( \mathcal{E}(Y+Y+Y+Y) \) is not r.e., despite \( \mathcal{E}(1) \) and \( \mathcal{E}(Y+Y+Y) \) both being r.e.

Proof. The idea is to make \( Y \) encode the contents of the two registers and the "program-counter" of a universal register machine \( p \) (presented as a directed graph, one edge–traversal of \( p \) corresponding to one application of \( F \) to \( Y \) ). The basic instructions labelling the edges of the graph will be \( X=K, X \leftarrow Y, 0=K, 1=Y, X \rightarrow Y, Y \rightarrow X \). (See Tinkin's (26) for a description of such a machine.) To define the program counter, we number the vertices of \( p \) with distinct natural numbers; the choice of numbers is unimportant. Let \( p \)'s start vertex be numbered \( s \) and final vertex \( f \). We assume without loss of generality that leaving each vertex \( v \) of \( p \) is either an assignment edge or a pair of edges labelled with complementary assignments \( X=0, X=0 \) or \( X=0, X=0 \). If necessary, add edges labelled \( X=0 \) and \( X \rightarrow 0 \) from \( f \) to \( s \). Now \( p \) may run for ever, and halting will be defined by reaching state \( f \), where it then is forced to stay. It is important to that control goes next be completely specified for every vertex, otherwise \( F \) may take \( Y \) to a value that damages our theorem. Another property we shall require of \( p \) is that it never attempt to decrement a zero register, which is easily arranged. We shall also require that when \( p \) has made up its mind to enter the final state, it sets \( Y \) and \( X \) to \( 0 \) first.

The 3-ary function symbol \( C \) is used to encode \( Y \) and the program counter. The following is the only property \( C \) needs to work reliably as an encoder.

\[ V_X V_Y (C(x)) = C(Y) \Rightarrow V_X Y \]

Call this sentence \( P_C \). It says that encoding is 1-1, i.e. does not lose information.

We also want to say that \( 0, U \) and \( D \) are supposed to behave similarly to standard 0, successor and predecessor. We let \( P_N \) denote

\[ V_X (U(x)) \neq 0 \land D(U(x)) = x \]

We now force \( F \) to execute one step of \( p \). We let \( P_F \) denote the and of a set of sentences, one per edge of \( p \), whose elements are defined by the following table, where \( i, j \) denote the numbers labelling the start and end of the corresponding edge.

\[ \text{Instruction on edge } (i, j) \quad \text{Corresponding sentence} \]

X=K 0 Y=F(K;C(I0),Y, U(01)) = C(U(x),Y, U(01))
X=K 1 Y=F(C(U(x),Y, U(01)) = C(X, Y, U(01))
X=K 0 Y=F(C(10),Y, U(01)) = C(I0, Y, U(01))
X=K 1 Y=F(C(U(x),Y, U(01)) = C(U(x), Y, U(01))

and similarly for \( Y \).
Claim 1. Given any interpretation \( I \) satisfying \( P_N \), in which all symbols save \( F \) are assigned interpretations, let \( N \) denote \( \text{let } \theta \text{ be an arbitrary variable } \theta \text{ of type } \text{let}(F \text{ and } \mathcal{M}) \text{ and let } M \text{ denote } \text{let}(\theta \text{ and } \mathcal{M}) \text{ labels a vertex of any } \mathcal{P} \). Thus \( \mathcal{N} \) is that subset of \( \mathcal{D} \) reachable via \( U_g \) from \( U_g \), and \( \mathcal{M} \) is that subset of \( \mathcal{D} \) corresponding to the vertices of the flowchart. Then the above table consistently and completely determines \( F_\mathcal{N} \) for \( x, y \in \mathcal{N} \) and \( z \in \mathcal{M} \), except when \( \mathcal{X} = \mathcal{O} \), in which case \( \mathcal{X} \) is determined when \( x = \mathcal{X} \), and similarly for \( y, z \). \( \mathcal{X} = \mathcal{O} \) is the necessary unique natural number satisfying \( \mathcal{X} \square \mathcal{O} = 0 \).

Proof. Completeness follows from the fact that every vertex labelled "I" has either an assignment leaving it, or a pair of tests. In the former case \( F_\mathcal{N} \), for \( x, y \in \mathcal{N} \), is completely specified except for the decrement instructions. In the latter case, \( F_\mathcal{N} \), for \( x, y \in \mathcal{N} \), is specified, as is \( F_\mathcal{N} \), for \( x, y \in \mathcal{N} \), accounting for all elements of \( \mathcal{N} \). Consistency follows from \( P \) and \( P_N \) which together ensure that each of the above equations specifies \( F_\mathcal{O} \) at a different element of \( \mathcal{O} \).

Claim 2. If \( x, y \in \mathcal{N} \) and \( z \in \mathcal{M} \) then \( F_\mathcal{N} \), for \( x \in \mathcal{N} \), is \( C_{\mathcal{N}} \), for \( x \in \mathcal{N} \), where \( p, q \) is started with \( x, y \in \mathcal{N} \), and \( \mathcal{X} = z \) respectively, then running \( p \) for one step yields \( a \) in \( x \) and \( b \) in \( y \), with control at \( x \leq x \) in \( \mathcal{O} \).

Proof. Straightforward.

Now assume that \( (EY + FV) \) is r.e. Then we can decide whether \( p \) started with \( x \) in \( x \), \( U_0 \) in \( \mathcal{D} \) will ever halt, thereby solving the halting problem for this universal machine, a contradiction [26]. To decide whether \( p \) halts, run \( p \) and at the same time enumerate \( (EY + FV) \) looking for \( (F_\mathcal{N} \) in \( \mathcal{O} \), \( 0, 0, 0 \) \) where \( P_i \) is \( V = C_{\mathcal{O} \mathcal{N}} \), \( 0, 0, 0 \) \).

The crucial observation is that we will find this pce if and only if \( p \) does not halt. For certainly if we find out if \( p \) then by Lemma 2 the machine cannot get into the state \( (0, 0, 1) \). Conversely, if the machine cannot get into this state, then by Lemmas 1 and 2 \( V = C_{\mathcal{O} \mathcal{N}} \), \( 0, 0, 0 \) \). This completes the proof of Theorem 16.

Corollary 17. When \( (1) \) is r.e., \((EY + FV)\) is not recursively tidy.

Proof. Suppose \( (EY + FV) \) be recursively tidy. Then \( (EY + FV) \), by Th. 11.16, \( m \) \( (1) \), but this would imply that \( (EY + FV) \) is r.e., contradicting Theorem 16.

Theorem 16. If \( x < P \), then \( (EY + FV) \) is r.e., recursively tidy.

Proof. \( (EY + FV) \), \( (1) \) or \( (1) \).

Proof. Straightforward.

Theorem 20. \( (a + b) = (a) + (b) \).

Proof. Straightforward.

Theorem 21. \( (a + b) \geq (a) + (b) \).

Proof. Straightforward.

Corollary 22. For a given program \( a \), the structure \( ((a) \square) \), \( c \) is a homomorphism of the natural number division lattice \( (N_+ \mid \leq) \), with \( (a) \) as the least element and \( (c) \) as the greatest. Further, when \( a = EY + FV \) with \( F \) interpreted, the homomorphism becomes an isomorphism.

References


(27) Pratt, V.R. Semantics of Programming Languages. Lecture notes for 6.832, Fall 1974, M.I.T.


