MIT/LCS/TR-168

SEMANTICAL CONSIDERATIONS ON FLOYD-HOARE LOGIC

Vaughan R. Pratt

September 1976

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

SEMANTICAL CONSIDERATIONS ON FLOYD-HOARE LOGIC

Vaughan R. Pratt
Massachusetts Institute of Technology
Cambridge, MA 02139
August 1976

ABSTRACT

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 axiom 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 theories 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. Modal 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 assignments rather than treating them as logical concepts.

This research was supported by the National Science Foundation under contracts DCR74-12997 and MCS76-18461.

SEMANTICAL CONSIDERATIONS ON FLOYD-HOARE LOGIC

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 concrete program and abstract program (cf Scott [31]), and as formula and truth-value (cf Tarski [34]). When necessary we will refer to these respectively as $[\![\,]\!]$-semantics and $\models$-semantics. These reflect what we feel should be the two main concerns of theoretical semantics, namely abstract programs and their logics. This section (1.1) deals with the former, although we do not explicitly discuss concrete programs. (Section 3.1 raises the possibility that the concrete/abstract dichotomy is too narrow a point of view for $[\![\,]\!]$-semantics.) The role of section 1.1 is to provide a rigorous foundation for the remainder of the paper, which is concerned (except for section 3.1) with logics of programs.

Binary Relations. We shall use binary relations for programs along lines proposed by Eilenberg 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 theories 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 AxB [10] (as opposed to a function from 2^A to 2^B [13], which is not as convenient for our purposes). We further define:

aRb	$(a,b) \in R$
aRbSc	$aRb \wedge bSc$
R∪S, R∩S, R-S	as for any sets, infinite union and intersection included
R•S	$\{(a,c) \mid \exists b\,[aRbSc]\}$ (composition)

R^-	$\{(b,a) \mid aRb\}$ (converse)	
XRb	$\bigwedge_{a\in X} aRb$	for $X\subseteq A$
aRY	$\bigwedge_{b\in Y} aRb$	for $Y\subseteq B$
XR	$\{b \mid \exists a\in X[aRb]\}$	for $X\subseteq A$ (exception: $R = \models$)
RY	$\{a \mid \exists b\in Y[aRb]\}$	for $Y\subseteq B$
$X\models$	$\{P \mid X\models P\}$ (this is an exception to XR above)	

Symbols. Central to the notion of environment is the symbol and its value (or 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 $\mathfrak{F}$, $\mathfrak{P}$, and $\mathfrak{C}$ respectively, and use subscripts to identify the collections of a given arity; thus $\mathfrak{F}_2$ is the collection of binary function symbols. $\mathfrak{C}$ will always include $\wedge$, $\neg$ and $\exists x$ for all $x\in\mathfrak{F}_0$ (i.e. first-order quantification, though very little of what we prove changes if we permit $\exists x$ for all $x\in\mathfrak{F}$), while $\mathfrak{P}$ will always include $=$. We let D denote the (single) domain for the functions and predicates.

We adopt the following notations.

$A\to B$	the set of all functions from A to B ;
$f:A\to B$	$f \in A\to B$;
A^k	$AxAx\ldots xA$ (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 $\mathfrak{C}^*\mathfrak{P}\mathfrak{F}^*$.

We use the following concepts and notations.

Formula	An expression whose root's label is in $\mathfrak{C}\cup\mathfrak{P}$;
Term	An expression whose root's label is in $\mathfrak{F}$
$\mathfrak{L}$	The set of expressions;
$\mathfrak{L}_f$	The set of formulae of $\mathfrak{L}$;
$\mathfrak{L}_t$	The set of terms of $\mathfrak{L}$;
Ground	Describes an expression containing no modalities.

Useful abbreviations and their expansions are:

$P\vee Q$	$\neg(\neg P\wedge\neg Q)$	(mutatis mutandis for $\supset$, $\equiv$, $\forall x$)
true	$x=x$	(mutatis mutandis for false)
$\underline{E}$	$(E_1,\ldots,E_k)$	

$\underset{\sim}{E}=\underset{\sim}{F}$ $\quad E_1=F_1 \wedge \ldots \wedge E_k=F_k$

$\mathcal{I}\models\underset{\sim}{E}$ $\quad (\mathcal{I}\models E_1,\ldots,\mathcal{I}\models E_k)$

$\exists\underset{\sim}{s}$ $\quad \exists s_1 \exists s_2 \ldots \exists s_k$

$P(\underset{\sim}{E_i},\underset{\sim}{F_i})$

$(P(E_1,F_1),\ldots,P(E_k,F_k))$.

Interpretations. We now assign meaning to expressions, along the lines spelled out by Tarski [34]. An interpretation $\mathcal{I}$ (which for us will play the role of an environment) specifies for each symbol A the value $A_{\mathcal{I}}$ of A in $\mathcal{I}$. Given $\mathcal{I}$, we can then infer the value in $\mathcal{I}$ of an expression $E = A(\underset{\sim}{F})$. The value will be written $\mathcal{I}\models E$ (slightly generalizing the usual usage), and is defined by

$$\mathcal{I}\models A(\underset{\sim}{F}) \equiv A_{\mathcal{I}}(\mathcal{I}\models\underset{\sim}{F}) \quad . \qquad (1)$$

Note that the argument $\underset{\sim}{F}$ on the left becomes $\mathcal{I}\models\underset{\sim}{F}$ on the right regardless of what A is. Under this condition we say that A is referentially transparent [28].

The only exception to (1) is when A is a modality, which is a unary logical connective whose interpretation $\mathcal{A}$ (independent of $\mathcal{I}$) is a binary relation on interpretations. (Alternatively we could say that its interpretation $A_{\mathcal{I}}$ is a set of interpretations depending on $\mathcal{I}$, namely those accessible from $\mathcal{I}$ via $\mathcal{A}$, in which case $\mathcal{I}\mathcal{A}\mathcal{J}$ would be written $\mathcal{J} \in A_{\mathcal{I}}$.) The definition becomes

$$\mathcal{I}\models A(P) \equiv \bigvee_{\mathcal{I}\mathcal{A}\mathcal{J}} \mathcal{J}\models P \qquad (2)$$

This asserts the existence of an interpretation $\mathcal{J}$, accessible from $\mathcal{I}$ via $\mathcal{A}$, in which P is true. It is Kripke's [20] semantical interpretation of what is written $\Diamond$ by modal logicians. Such an A is not referentially transparent; we then say it is referentially opaque [28]. An immediate application is to the definition of modalities of the form $\exists x$ where $x \in \mathfrak{I}_0$ and $\exists x$ is interpreted as the equivalence relation relating pairs of interpretations differing only in their assignment to x. The section on programs as binary relations will suggest a further application.

Given a set X of interpretations we shall write $X\models P$ for $\bigwedge_{\mathcal{I}\in X} \mathcal{I}\models P$, $X\models$ for $\{P \mid X\models P\}$ (the theory of X) and $\models P$ for $\{\mathcal{I} \mid \mathcal{I}\models P\}$.

Programs as binary relations on states. We have thus far defined only conventional concepts from logic, using more or less conventional definitions. We now define a transition to be an ordered pair of states, where a state is defined to be an

interpretation. Intuitively a transition represents an initial and a final state. Following de Bakker [9,10,11], we define a program to be a set of transitions, i.e. a binary relation on states.

Note that this definition makes the interpretation of a modality a program. Given a program a we let $\langle a \rangle$ denote the modality whose interpretation is a, and abbreviate $\neg\langle a\rangle\neg$ to $[a]$, in imitation of the symbols of classical modal logic. A little thought reveals that $[a]P$ means essentially "after executing a, P holds," or more precisely, "every transition from this state leads to a state satisfying P," while $\langle a\rangle P$ means "there exists a transition from this state to a state satisfying P." In this light, another way of viewing our interpretation of $\exists x$ is as a program that non-deterministically assigns an arbitrary element of D to x.

Restrictions on interpretations. If every symbol were always to have the same value there could be but one state and hence but two programs, the identity program I and the empty program Φ. Useful assignment statements would not be possible. Conversely, if no restrictions (save those of arity and type) are placed on the possible values of symbols (as in pure predicate calculus), a wealth of programs is possible. We would then be studying uninterpreted program schemes. With the exception of Theorems 16-18 (and in some sense Theorems 4 and 5), our results are independent of where one lies in this spectrum. When we use familiar symbols (e.g. $\wedge$, $\neg$, $\exists x$, $=$, $<$, 0, 1, $+$, $-$, ...), these will always be assumed to have their standard interpretation (which in practice is a function of whether D is the natural numbers, integers, reals or whatever). The universe U of possible states is thus a function solely of D, $\Im$, Φ and whatever restrictions are in force on interpretations of symbols. When $U \models P$ we shall say that P is valid.

We distinguish between symbols with a single fixed standard interpretation, symbols whose interpretation can be changed by a program, and symbols in neither category, by calling them respectively standard symbols, assignables and labels. None of these distinctions are relevant to the statement or proof of most of the theorems of Section 2, but they are important in interpreting those theorems. For example, knowing that a symbol is a label means that we know it cannot change during program execution, and hence it can safely be used to name, say, input values in both the antecedent and the consequent of a partial correctness assertion.

1.2 Logic: how to describe a program

So far we have said what a program is, a $[\,]$-semantical

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 formulae of $\mathfrak{L}_f$, called respectively the antecedent and the consequent. Pca's were first studied carefully by Floyd [14], 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 lop-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 $\mathfrak{L}_f^2$ as Floyd-Hoare logics.

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

$$(\mathfrak{s},\mathfrak{f}) \models (P,Q) \quad \equiv \quad (\mathfrak{s} \models P \supset \mathfrak{f} \models Q) \tag{3}$$

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 $\models$ 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), $a \models (P,Q)$ means that (P,Q) is true of every transition in the program a; we then say (P,Q) is true of a, and that a satisfies (P,Q). Similarly, $a \models$ denotes the set of pca's true of a, which we shall call the partial correctness theory of a, abbreviated to $\mathbf{\{}a\mathbf{\}}$ (following Hoare [16], but using boldface to distinguish $\mathbf{\{}\ \mathbf{\}}$ from set brackets $\{\ \}$). Since $\mathbf{\{}a\mathbf{\}}$ is a set of pairs of formulae we can treat it as a binary relation on $\mathfrak{L}_f$ and write $P\,\mathbf{\{}a\mathbf{\}}\,Q$ for $a \models (P,Q)$.

One can think of (P,Q) as providing an upper bound on programs, in the sense that the programs satisfying (P,Q) are just the subsets of $\models (P,Q)$. In this role, (P,Q) can assert non-termination, but because $\mathfrak{P} \models (P,Q)$ for any (P,Q), 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

$$\{a^-\} = \neg\{a\}\neg \qquad (D)$$

where $\neg(P,Q)$ is defined as $(\neg P,\neg Q)$. Thus $P\{a^-\}Q$ is equivalent to $\neg Q\{a\}\neg P$. 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 ($a \supset b \equiv \neg b \supset \neg a$) can be thought of as the duality principle applied to the program I.

Weakest Antecedents and Strongest Consequents. We observed earlier that $[a]P$ could be interpreted as "after executing a, P holds." It follows that $([a]P)\{a\}P$ holds. Moreover, no weaker antecedent than $[a]P$ will permit the consequent P; indeed, if $\mathfrak{I} \not\models [a]P$ then $\mathfrak{I} \models \langle a\rangle\neg P$, so there exists $\mathfrak{J}$ satisfying $\mathfrak{I}a\mathfrak{J}$ such that $\mathfrak{J} \not\models P$. We call any formula logically equivalent to $[a]P$ a weakest antecedent (Dijkstra [12]) of P via a. This is summarized by

$$P\{a\}Q \quad \equiv \models (P \supset [a]Q) \qquad (W)$$

By the duality principle (D), all of the above holds equally well for $\neg P\{a^-\}(\neg[a]P)$, and hence for $P\{a\}(\langle a^-\rangle P)$. We call any formula logically equivalent to $\langle a^-\rangle P$ a strongest consequent (Floyd [15]) of P via a. The dual of (W) is

$$P\{a\}Q \quad \equiv \models (\langle a^-\rangle P \supset Q) \qquad (S)$$

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 $\langle a\rangle$ in $\mathcal{C}$ is not really cricket, since the whole idea of Floyd-Hoare logic becomes superfluous (see section 3.2 for details). We shall limit $\mathcal{C}$ to $\wedge$, $\neg$ and $\exists x$ for all $x \in \mathfrak{I}_0$. In this case we may ask whether $\mathcal{L}_f$ contains any formula logically equivalent to $[a]P$. If for a given program a the answer to this question is yes for all $P \in \mathcal{L}_f$, we say that a is backward tidy. (Schwarz [32] uses the terminology "backward exactly connected".) The dual epithet is forward tidy (Pratt [27] and independently Schwarz [32]). Note that the concept of tidiness finds no application in de Bakker's and Meertens' [11] $[\![\,]\!]$-semantical treatment of partial correctness, where for an "antecedent" $V \subseteq U$ they define the strongest "consequent" via a to be Va, 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 $a\Rightarrow : \mathcal{L}_f \to \mathcal{L}_f$, such that $a\Rightarrow$ takes P to a strongest consequent of P via a. We call $a\Rightarrow$ a forward tidiness function of a. If there exists a recursive $a\Rightarrow$ we shall say that a is recursively forward tidy. For convenience we will sometimes treat $a\Rightarrow$ as a binary relation, writing it as $(a\Rightarrow)$. A backward tidiness function of b (if any) is written $\Leftarrow b$; $(\Leftarrow b)$ will denote the converse of the relation corresponding to the function $\Leftarrow b$. The following facts formalize this.

$$\exists Q(P(a\Rightarrow)Q \wedge Q\equiv\langle a^-\rangle P) \text{ if a is forward tidy} \qquad (F\exists)$$
$$\forall Q(P(a\Rightarrow)Q \supset Q\equiv\langle a^-\rangle P) \qquad (F\forall)$$
$$\exists P(P(\Leftarrow b)Q \wedge P\equiv[b]Q) \text{ if b is backward tidy} \qquad (B\exists)$$
$$\forall P(P(\Leftarrow b)Q \supset P\equiv[b]Q) \qquad (B\forall)$$

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

Tidiness Duality Lemma (TDL): Program a is forward tidy if and only if a^- 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 $\{a\} = (a\Rightarrow)\circ\{|\}$.
(b) Let b be backward tidy. Then $\{b\} = \{|\}\circ(\Leftarrow b)$.

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 $\{|\}$ 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 this definition is plausible (what simpler rigorous yet direct definition of truth could there be?), but not accessible (evaluating the truth of $P\{a\}Q$ directly from the definitions may bog down in the infinities of either the set a of transitions (when checking $t\models(P,Q)$ for each $t \in a$) or the universe U of states (when evaluating $\exists x[P]$)). Hence we would like to trade off plausibility for effectiveness, leading to an axiom system for $\{a\}$ 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 $\{a\}$ is not accessible in the above sense; however, in some simple yet useful cases, $\{a\}$ 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 Φ, containing no transitions, and the identity relation I_U, which we shall henceforth abbreviate to I, are two simple programs of particular interest. These are characterized by the properties $a \cup \Phi = \Phi \cup a = a$, $a \cdot \Phi = \Phi \cdot a = \Phi$ and $a \cdot I = I \cdot a = a$. 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 $\cup$ and multiplication operator $\cdot$.

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

Theorem 1. $\{\Phi\} = \mathcal{L}_f^2$.

Theorem 2. $\{I\} = \{(P,Q) \mid U \models (P \supset Q)\}$.

For convenience, rather than referring to $U\models$ when we want to talk about the set of valid formulae of unary logic we shall use $\{I\}$. Though this implies a restriction of $\mathcal{L}_f$ to implications, this is a trivial restriction in our case. (Recall Floyd's remark [15] : "One might 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 $\mathcal{L}$, and denotes the program

$$\begin{aligned} [\![P]\!] &= I_U \cap (\models P) \qquad (T) \\ &= \{(\vartheta,\vartheta) \mid \vartheta \in U \wedge \vartheta \models P\} . \end{aligned}$$

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

Theorem 3. Let R be a test.
(a) $\langle [\![R]\!]^- \rangle P \equiv R \wedge P$. (Forward tidiness)
(b) $[[\![R]\!]]P \equiv R \supset P$. (Backward tidiness)

The Tidiness Characterization Lemma allows us to deduce the remainder of $\{\![R]\!\}$.

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 $(F(\underset{\sim}{S}),T)$ of $\mathfrak{L}$, corresponding to the left and right sides of a conventional assignment statement. No loss of generality ensues from parsing the left-hand side as $F(\underset{\sim}{S})$; all expressions can be so written. When $\underset{\sim}{S}$ 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 as 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 $U\to U$, 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 λ-notation thus.

$$[\![F(\underset{\sim}{S})\leftarrow T]\!] = \lambda\vartheta.\lambda A.\text{if } A\neq F \text{ then } A_\vartheta \quad (A)$$
$$\text{else } \lambda\underset{\sim}{s}.\text{if } \underset{\sim}{s}\neq\vartheta\neq\underset{\sim}{S} \text{ then } A_\vartheta(\underset{\sim}{s})$$
$$\text{else } \vartheta\neq T \ .$$

(Note: $\underset{\sim}{s}$ and $\underset{\sim}{S}$ agree in arity.)

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

$[\![F(\underset{\sim}{S})\leftarrow T]\!]$: $U\to U$

ϑ: $\mathfrak{I}\to(D^*\to D)$ (at least in this case, with only terms involved)

$\vartheta\neq$: $\mathfrak{L}_t\to D$

It should be remarked that this is not meant as an interpretive definition of assignment in the sense that to execute an assignment one executes 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 our definition, though of course as in any definition the wording of the definition may be varied.

Before proceeding to a characterization of $\{\![F(\underset{\sim}{S})\leftarrow T]\!\}$ we introduce the notion of substitution, suitably generalized to handle arrays. We define $[T/F(\underset{\sim}{Z})]P$ (abbreviated as P'), the result of substituting the expression T for the subtrees in P with root F , as follows.

$F(\underset{\sim}{E})' = [\underset{\sim}{E}'/\underset{\sim}{Z}]T$ (S1)

$A(\underset{\sim}{E})' = A(\underset{\sim}{E}')$ non-modal $A \neq F$ (S2)

$(\exists xP)' = \exists y(([y/x]P)')$ new y (S3)

S1 performs the substitution for F ; its right side simplifies to T when F is zeroary. Clearly $\underset{\sim}{E}'$ is $(E_1', \ldots, E_k')$. Substitution of this for $\underset{\sim}{Z}$ in T means substitution of the whole tuple wherever $\underset{\sim}{Z}$ occurs in T . $\underset{\sim}{Z}$ is not a tuple of $\mathfrak{L}$ but rather a place-holder (for the arguments of F) that we shall employ below in instances of T .

S2 caters for referentially transparent symbols. The only modality we provide for is $\exists x$ as in S3, which is sufficient for our purposes. (It is easy to check that $\exists$ may be replaced by $\forall$ in S3 with no other modification since $\neg$ is referentially transparent and covered by S2.)

(When P contains assignment modalities (not the case in this paper), a difficulty arises in extending S3, namely that of generalizing renaming of bound variables. The reader interested in pursuing this further might consider the supposedly valid formula $X=0 \supset [X+1/Y][X \leftarrow X+2](Y=1 \wedge X=2)$, which is in fact not valid if the substitution is performed naively. This may be transformed to $X=0 \supset [X+1/Y][Z \leftarrow X+2](Y=1 \wedge Z=2)$ to avoid this problem, along the lines of S3, but is this desirable? The reason this is not a problem for $\exists X$ thought of as $\langle X \leftarrow RANDOM \rangle$ is that RANDOM is independent of X , so renaming it to $\langle Z \leftarrow RANDOM \rangle$, or for that matter renaming $\langle X \leftarrow 1 \rangle$ to $\langle Z \leftarrow 1 \rangle$, is not as distressing as renaming $\langle X \leftarrow X+2 \rangle$ to $\langle Z \leftarrow X+2 \rangle$. Clearly we cannot rename it to $\langle Z \leftarrow Z+2 \rangle$ without renaming other occurrences of X possibly outside the scope of the substitution, as in the example where we have $X=0$. It would appear that renaming to $\langle Z \leftarrow X+2 \rangle$ 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 $\mathfrak{I}_3$, namely IF-THEN-ELSE, 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 $\mathfrak{L}$) 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.

$G(\underset{\sim}{\text{IF R THEN E ELSE F}})$
$\quad \rightarrow$ IF R THEN $G(\underset{\sim}{E})$ ELSE $G(\underset{\sim}{F})$ for $G \in \exists \forall P$;

$E \rightarrow$ IF R THEN E ELSE E ; (to facilitate preceding rule)

IF R THEN S ELSE T $\rightarrow (R \wedge S) \vee (\neg R \wedge T)$ for $S,T \in \mathfrak{L}_f$.

IF lemma. Evaluating $\mathfrak{I} \models W$ 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

$$\mathfrak{I}\models(\text{IF } P \text{ THEN } S \text{ ELSE } T) = \text{if } \mathfrak{I}\models P \text{ then } \mathfrak{I}\models S \text{ else } \mathfrak{I}\models T.$$

The following reports joint work with R. Hale [27].

Theorem 4. Let $F(\underset{\sim}{S})\leftarrow T$ be an assignment.

(a) $\langle[F(\underset{\sim}{S})\leftarrow T]\rangle P \equiv \exists t_{\underset{\sim}{s}}[P' \wedge \underset{\sim}{s}=\underset{\sim}{S}' \wedge F(\underset{\sim}{s})=T']$
where $E' = [(\text{IF } \underset{\sim}{Z}=\underset{\sim}{s} \text{ THEN } t \text{ ELSE } F(\underset{\sim}{Z}))/F(\underset{\sim}{Z})]E$

(b) $[[F(\underset{\sim}{S})\leftarrow T]]P \equiv P''$
where $E'' = [(\text{IF } \underset{\sim}{Z}=\underset{\sim}{S} \text{ THEN } T \text{ ELSE } F(\underset{\sim}{Z}))/F(\underset{\sim}{Z})]E$.

When F is 0-ary, IF $\underset{\sim}{Z}=\underset{\sim}{s}$ THEN t ELSE $F(\underset{\sim}{Z})$ can be simplified to t, giving Floyd's [15] construction of the most general consequent of an assignment statement as a special case, and IF $\underset{\sim}{Z}=\underset{\sim}{S}$ THEN T ELSE $F(\underset{\sim}{Z})$ can be simplified to T, giving Hoare's [16] backward substitution rule for assignment as a special case, namely that

$$[X\leftarrow T]P \equiv [T/X]P .$$

(We shall often abbreviate $[[\,]]$ to $[\,]$.) Fortuitously $[X\leftarrow T]$ and $[T/X]$ are much alike, and we rely on $\leftarrow$ versus $/$ to disambiguate them. Actually, since they are equivalent, the only reason one would want to distinguish them is when one wants to stress that $[T/X]P$ is an abbreviation for something in $\mathfrak{L}$ while $[X\leftarrow T]P$ is an unabbreviated formula of modal logic. Thus $[X\leftarrow T]$ is semantic inasmuch as it has an interpretation under $\models$, while $[T/X]$ is syntactic in that it specifies a transformation on an expression.

Lambda-calculus adherents will note the obvious similarity between $[X\leftarrow T]P$ and $(\lambda X.P)T$; 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 $\lambda a(y).a(x)+1$, where due regard needs to be given to whether $x=y$.

Second-Order Assignment. We may call the above programs first-order assignments because individuals of 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, $F\leftarrow G$ would change the whole interpretation of F, not just its value at one point. The program $[[F\leftarrow G]]$ would then be

$$\lambda\vartheta.\lambda A.\text{if } A\neq F \text{ then } A_{\vartheta} \text{ else } G_{\vartheta} .$$

This notion of second-order assignment is not as general as it might be. For example, one might want to perform $F\leftarrow G\circ H$

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 $[F \leftarrow G \cdot H]$ is not tidy - it is backward tidy, by a variation on the argument in the proof of the following theorem, that $[F \leftarrow G]$ is backward tidy.)

Thinking of the first order quantifier $\exists x$ as $\langle x \leftarrow RANDOM \rangle$, we can think of the second-order quantifier $\exists f$ as the second-order assignment $\langle f \leftarrow RANDOMFUNCTION \rangle$. This illustrates just how close our use of "second-order" is to the conventional use.

Theorem 5. Let $F \leftarrow G$ be a second-order assignment. Then
$$[[F \leftarrow G]]P = [G/F]P$$
($[G/F]$ is a convenient abbreviation for $[G(\underline{Z})/F(\underline{Z})]$.)
Hence second-order assignment is backward tidy.

Open problem. Is $F \leftarrow G$ always forward tidy?

2.2 Loop-free Programs.

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. $\{a \cup b\} = \{a\} \cap \{b\}$.

Note the exact analog of this binary logic theorem in unary logic; in both logics, "the theory of the union (of two subsets of either $\mathcal{L}_f$ or $\mathcal{L}_f^2$) 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. $\{a \cdot b\} \supseteq \{a\} \cdot \{b\}$.

The $\supseteq$ cannot be strengthened to $=$ without knowing more about a and b . For example, let $V \subsetneq U$ have no $P \in \mathcal{L}_f$ satisfying $\vdash P = V$. Let $a = I_V = \{(\vartheta,\vartheta) \mid \vartheta \in V\}$ and let $b = I_{U-V}$, so that $a \cup b = I_U$ and $a \cap b = \Phi$. Then $a \cdot b = \Phi$, so $\{a \cdot b\}$ is vacuously $\mathcal{L}_f^2$, the set of all pca's, including $(\underline{true},\underline{false})$. But by the construction there can be no P simultaneously satisfying $\underline{true}\{a\}P$ and $P\{b\}\underline{false}$, so $(\underline{true},\underline{false})$ cannot be in $\{a\} \cdot \{b\}$, whence $\{a \cdot b\} \supset \{a\} \cdot \{b\}$.

Neither a nor b in this example is tidy, and in fact we can strengthen theorem 7A as follows.

Theorem 7. $\{a \cdot b\} = \{a\} \cdot \{b\}$ when a is forward tidy or b is backward tidy.

Theorem 8.
(a) If a,b are forward tidy, so are $a \cup b$ and $a \cdot b$;
(b) If a,b are backward tidy, so are $a \cup b$ and $a \cdot b$.

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 be 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 excepting forward tidiness for second order assignment).

So far we have considered only the programming constructs of tests (subsuming I and Φ), 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 "$[\![P]\!] \cdot a \cup [\![\neg P]\!] \cdot b$" , much as we considered $\forall x P$ to be an abbreviation for $\neg \exists x \neg P$. Similarly, we would regard the goto construct as a notation for describing flowchart programs textually, provided 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 $[\![\,]\!]$-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 $a\Rightarrow$ $(\Leftarrow a)$ is recursive. In the following we use "many-one reducibility" [30]: we say $X \leq_m Y$ when there exists a recursive function f such that $x \in X$ iff $f(x) \in Y$.

Theorem 10. $\{a \cup b\} \leq_m \{a\} \times \{b\}$ (Cartesian product).

Theorem 11.
(a) If a is forward recursively tidy, $\{a \cdot b\} \leq_m \{b\}$.
(b) If b is backward recursively tidy, $\{a \cdot b\} \leq_m \{a\}$.

Theorem 12. If a is recursively tidy, $\{a\} \leq_m \{I\}$.

Theorem 13. Instructions are recursively very tidy.

Theorem 14. If a,b are forward (backward) recursively tidy, so are $a \cup b$ and $a \cdot b$.

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, $\{a\} \leq_m \{I\}$. (Note that $\{I\}^n = \{I\} \times \{I\} \times \ldots \times \{I\} \leq_m \{I\}$, for any n, since the n questions about membership in $\{I\}$ can be rephrased as a single conjunction.)

This asserts that to decide whether (P,Q) 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 "theory of the underlying logic."

Axiom Systems. The above results are quite strong, promising recursive reductions to $\{I\}$. If we do not mind weakening this to recursive enumerability, we can write out a simple non-deterministic enumerator (or axiom system) for the pca's true of a given loop-free program.

A1. $\{I\}$. (i.e. we take all of $\{I\}$ as axioms.)
A2. $P\{a\}Q,\ P\{b\}Q \vdash P\{a \cup b\}Q$.
(This is equipollent with Hoare's $P\{a\}Q,\ P'\{b\}Q' \vdash P \wedge P'\{a \cup b\}Q \vee Q'$.)
A3. $P\{a\}Q,\ Q\{b\}R \vdash P\{a \cdot b\}R$.
A4. $Q\{P\}P \wedge Q$ (or $P \supset Q\{P\}Q$).
A5. $P\{F(\underset{\sim}{S}) \leftarrow T\}\exists y \underset{\sim}{s}[P' \wedge \underset{\sim}{s} = \underset{\sim}{S}' \wedge F(\underset{\sim}{s}) = T']$ (or $P''\{F(\underset{\sim}{S} \leftarrow T)\}P$) where E' and E'' are defined as in Theorem 4.
A6. $([G/F]Q)\{F \leftarrow G\}Q$ (second order assignment).

An issue we do not resolve here is whether the object

inside $\{\ \}$ is a program or a concrete representation of it. If the latter, then we also need a rule:

A≡. $P\{a\}Q \vdash P\{a'\}Q$ provided a and a' represent the same program.

This axiom system is a good approximation to the one proposed by Hoare [16]. Theorems 1 to 8 provide immediate confirmation of its soundness and completeness. Note the absence of Hoare's "Rules of Consequence" $P\supset Q,\ Q\{a\}R \vdash P\{a\}R$ and its backward dual $P\{a\}Q,\ Q\supset R \vdash P\{a\}R$. We achieve its effect by using $I\circ a = a\circ I = a$. Then Hoare's Rules of Consequence can be derived from $P\{I\}Q,\ Q\{a\}R \vdash P\{I\circ a\}R \vdash P\{a\}R$, and dually.

We draw the reader's attention to our efforts to separate "competence" from "performance" (cf [6]) in the above. Without mentioning axiom systems, we established some properties of theories of programs (competence) from which we could readily infer the "correctness" of a non-deterministic system (performance, in this case as realized by the given axiom system). We feel that such a separation has some merit, and would like to see it applied more frequently in all domains where the dichotomy makes sense, including everyday programming.

2.3 Regular Programs.

We now consider a larger class of programs by including transitive closure as an operation. The reflexive transitive closure a^* of a is the least x (with respect to $\subseteq$) satisfying $aU[UxUx\circ x = x$, which can be shown to be $U\{a^n \mid n\geq 0\}$, where $a^i = a\circ a\circ\ldots\circ a$ i times. We call the closure of the set of assignments and tests under U, $\circ$ and * the class of regular assignment-and-test programs. The connection with flowcharts is as for the loop-free case, except that the infinitely many paths that arise when loops are permitted are disposed of by using Kleene's transformation of such graphs into regular expressions. (Because we have union as one of our constructs, permitting non-deterministic programs, the obstacle raised by Ashcroft and Manna [3] for directly translating deterministic flowcharts into deterministic "structured" programs involving just assignment, composition, if-then-else and while-do does not arise here.)

We may summarize the results of this section as follows. Regular programs do not in general have as tractable theories as loop-free programs. Even when F is completely uninterpreted and $\{I\}$ is r.e., the innocent looking program $[X\leftarrow F(X)]^*$ does not have an r.e. theory. However, as a sort of consolation prize, invariance theories (sets of pca's of the form (P,P)) turn out to be well behaved with respect to * .

It is easy to find a regular program without an r.e. theory. Let $0 \in \mathfrak{I}_0$ and $+1,-1 \in \mathfrak{I}_1$ (successor and predecessor), all with their standard interpretations on the natural numbers. Let a be a program implementing Minsky's universal two-counter machine [26]. Then if $\{a\}$ were r.e., the halting problem could be solved by simultaneously running a and looking in $\{a\}$ for (P,false) where P says that the counters initially describe $\varphi_x(x)$. This capitalizes on the fact that though pca's cannot in general assert termination, they can assert non-termination.

When all function symbols are uninterpreted, $\{a\}$ as described above is still not r.e., though to prove this takes a little more care. The idea is to say enough in the antecedent P referred to above to constrain 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 16. Let $|\mathfrak{I}_0| \geq 4$, $|\mathfrak{I}_1| \geq 3$, $|\mathfrak{I}_3| \geq 1$, with $V \in \mathfrak{I}_0$, $F \in \mathfrak{I}_1$. Let the symbols of $\mathfrak{I}$ and $\mathcal{P}$ (excepting =) take on all possible interpretations in the universe U . Then $\{[\![V \leftarrow F(V)]\!]^*\}$ is not r.e., despite $\{I\}$ and $\{[\![V \leftarrow F(V)]\!]\}$ both being r.e.

Corollary 17. When $\{I\}$ is r.e., $[\![V \leftarrow F(V)]\!]^*$ is not recursively tidy.

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

The proof of Theorem 16 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 $\leq \in \mathcal{P}$ then $\{[\![X \leftarrow X+1]\!]^*\}$ 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 $\{a\} \cap I_{\mathcal{L}_f}$, the pca's (P,P) that express invariance.

Theorem 19. $(\mathcal{P}) = (I) = I_{\mathcal{L}_f}$.

Theorem 20. $(a \cup b) = (a) \cap (b)$.

Theorem 21. $(a \circ b) \supseteq (a) \circ (b) = (a) \cap (b)$.

This inequality $\supseteq$ cannot be strengthened to = even if we make a and b tidy or make a=b , as witnessed by

$a = b = [X \leftarrow F(X)]$ where F is uninterpreted. For let $Q = P(X) \wedge \forall y [P(y) \equiv \neg P(F(y))]$. Then $Q\langle a \circ a \rangle Q$ but not $Q\langle a \rangle Q$. Compare this with the way tidiness came to the rescue in Theorem 7. An amusing consequence of Theorem 21 is:

Corollary 22. For a given program a, the structure $(\{\langle a^n \rangle \mid n \geq 0\}, \subseteq)$ is a homomorph of the natural number division lattice $(N, |)$, with $\langle a \rangle$ as the least element and $\langle I \rangle$ as the greatest. Further, when $a = [X \leftarrow F(X)]$ with F uninterpreted, the homomorphism becomes an isomorphism.

Considering that invariance theories fare less well with $\circ$ 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. $\langle a^* \rangle = \langle a \rangle$

We can now add to our axiom system:

A7: $P[a]P \vdash P[a^*]P$.

We note in passing that an apparent limitation of the method of proving flowchart programs correct by labelling between-instruction points in the flowchart with assertions is that the only assertions one can make about loops are invariance assertions (in contrast, say, to being able to write $P[a^*]Q$ in Hoare's notation). (We are again thinking of flowcharts as state transition diagrams, i.e. as directed graphs with edges labelled with instructions.) Theorem 23 strikes an optimistic note of sorts by seeming to claim completeness given this limitation on what one can claim about loops. This completeness is unfortunately a mirage, since the limitation is a mirage; one can in fact make other than invariance assertions about loops by the device of having ϵ-transitions (edges labelled with the identity program I) leading to and from the loop. This however does not change the fact that Floyd's induction rule for flowchart programs [15] cannot be stronger than our A7.

Cook [7] has recently found a situation where Corollary 15 can be extended to regular programs. The following theorem distills a key idea in Cook's proof.

Theorem 24. (Star Interpolation Theorem). Let a^* be tidy, with $P[a^*]R$. Then there exists Q satisfying $P \supset Q \supset R$ and $Q[a]Q$. (An equivalent statement of the theorem is that if a^* is tidy, $[a^*] = [I] \circ \langle a \rangle \circ [I]$.)

(We like the name "interpolation theorem" for this theorem because of its vague resemblance to the celebrated Craig

Interpolation Lemma [8], which states more or less that if $P \supset R$ is valid then there exists Q such that $P \supset Q \supset R$ is valid and Q contains only predicate symbols common to both P and R.)

The significance of this theorem is that to prove $P\{a^*\}R$ it suffices to prove $Q\{a\}Q$ for the Q whose existence is guaranteed by the interpolation theorem, then infer $Q\{a^*\}Q$ (e.g. by our A7), and then use $P \supset Q \supset R$ and Hoare's Rules of Consequence (or our A≡). Cook has shown that when $\{|\}$ is sufficiently "expressive", as is the case when $0,1,+,\times \in \mathfrak{Z}$ and have their standard interpretations, then all regular programs are tidy, allowing the Interpolation Theorem to be applied. (Cook actually showed this for what one might call "context-free" programs, namely the class of programs with recursion, which translates in our case into the closure of the regular programs under the operation of taking fixed points of those first order functions on programs definable by first-order lambda abstraction.)

In the following we need the notion of <u>enumeration reducibility</u> [30], written $A \leq_e B$, which roughly speaking means that given an enumeration of B, A can be effectively enumerated. Thus if $A \leq_e B$ and B is r.e. then A is r.e..

<u>Corollary 25</u>. When all regular programs are tidy, $\{a\} \leq_e \{|\}$.

<u>Corollary 26</u>. Under the conditions of Theorem 16, if $\{|\}$ is r.e. then $[\![V \leftarrow F(V)]\!]^*$ is not tidy.

We remark in passing that programs such as operating systems that are intended to run forever can be handled quite elegantly using *. At first this seems impossible since a program that never terminates is semantically equivalent to the empty program, for which all pca's hold. Indeed, when we translate the program

<u>while</u> <u>true</u> <u>do</u> a

into

$([\![\text{true}]\!] \circ a)^* \circ [\![\text{false}]\!]$

we immediately observe $[\![\text{false}]\!] = \Phi$ and $x \circ \Phi = \Phi$. However, if we simply remove the offending "$\circ[\![\text{false}]\!]$", we are left with a program that simplifies to a^*. Then $P\{a^*\}Q$ tells us that if at some time during the running of the program (e.g. at start-up time) P held, then after every execution of a, no matter how long this continues, Q will hold. Thus although we were unable to use the theory of the original program, it being Φ, the theory of a closely related program furnished us with precisely the information we required. This is a good example of how Floyd-Hoare logic can be more useful than might at first appear.

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 formulae). 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 "I/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

(i)	Grammars	(Permits finite programs)
(ii)	Languages	(Permits sophisticated control)
(iii)	*-ary relations	(Permits parallelism)
(iv)	Multiweighted binary relations	(Preserves complexity information)
(v)	Weighted binary relations	(Ditto)
(vi)	Binary relations	(Preserves I/O information)
(vii)	States	(Preserves termination information)

This hierarchy is not intended as some hard-and-fast structure, but rather as some interesting points in the partial ordering (by information content) of varieties of programs. The following is also not meant to be so much prescriptive as descriptive, and we will often use "might be" in place of "is."

Let us begin with grammars. To motivate this, we can start with the following program for computing factorial(X).

A:=1; while X>0 do begin A:=XxA; X:=X-1 end.

This program serves to control a processor that emits a string of instructions. As such it serves the same function as a grammar. While this program may not look much like a grammar, if we rewrite it as a regular expression with alphabet Σ (the set of tests and assignments of section 2), we might have

$$A{\leftarrow}1;(X>0;\ A{\leftarrow}XxA;\ X{\leftarrow}X-1)^*;\ X\leq 0$$

as the regular expression generating all possible execution

sequences, where we have written ; for concatenation. Another way to generate this set is with a finite-state transition diagram, which would be a flowchart program of sorts, though with the usual roles of edges and vertices interchanged. See R. Karp's Ph.D. thesis [17] for an early example of this state-transition style of flowchart. Context-free grammars can of course be used for parameter-less recursion [10,11].

From grammars we move to the languages they generate. The usual operations of union, concatenation and Kleene closure apply here. Otherwise there is little to say about them at this point.

To get from languages to *-ary relations we need Elgot's [14] notion of fusion product. Let R,S be two binary relations. Take their fusion product to be $\{(a,b,c) \mid aRbSc\}$. The result is a 3-ary relation. Fusion product generalizes to relations of arbitrary arity. We define a *-ary relation to be a set of k-tuples (for various $k \geq 1$) over some domain, call it U since our application is to the domain of states. Let R and S be two *-ary relations. Then their fusion product $R \cdot S$ is $\{(a,b,\ldots,c,d,e,\ldots,f,g) \mid (a,b,\ldots,c,d) \in R \wedge (d,e,\ldots,f,g) \in S\}$. The union of *-ary relations is defined in the obvious way. This system is a semi-ring (ring with no additive inverses) [2,4] with addition operator $\cup$ and multiplication operator $\cdot$. The reflexive transitive closure of R is defined, as for any semi-ring, as the (necessarily unique) least fixpoint of $\lambda x.R \cup I \cup x \cup (x \cdot x)$. (Here x is least when $x \cup f = f$ for any fixpoint f.) We call the elements of a *-ary relation a path, each k-tuple being a path of length $k-1$. This generalizes the notion of transition used earlier in that the intermediate states are recorded as well as the initial and final states.

The map from languages to *-ary relations is defined with the help of the function $[\![\ ]\!]$, defined in section 1 for tests and assignments (perhaps varied for tests so that it maps P to $\vdash P$). Extend $[\![\ ]\!]$ to strings by letting it map concatenation to fusion product; thus if $a \in \Sigma^*$ and $|a| = n$, $[\![a]\!]$ will be an $(n+1)$-ary relation. If $|a| = 0$, take $[\![a]\!]$ to be U, the set of all states, a unary relation. Extend $[\![\ ]\!]$ to sets of strings completely additively, so that for any set of strings L, finite or infinite, $[\![\cup L]\!] = \cup [\![L]\!]$. This completes the definition of $[\![\ ]\!]$. A useful theorem is that $[\![\ ]\!]$ takes Kleene closure to transitive closure, which follows from the complete additivity of $[\![\ ]\!]$.

We now throw away the names of the intermediates states in the paths and consider just the path lengths. Thus the $(n+1)$-tuple $(a,\ldots,b)$ becomes the triple (a,n,b). We call a set of such triples multiweighted binary relations; each transition (a,b) has a set of weights; each such weight w

corresponds to an element (a,w,b) in the multiweighted relation. The intuition is that the weights represent the costs of the possible transitions from a to b. In the variation where $[\![P]\!]$ is taken to be $\vDash P$, only assignments enter into this accounting.

In the cost function, only one weight is associated with each transition. Thus the cost function corresponding to the multiweighted binary relation R is a function from U^2 to $N \cup \{\infty\}$ which maps (a,b) to the least n such that (a,n,b) is in R, or to ∞ if there is no such n. The intuition is that this function gives the fastest way of getting from a to b.

By composing the cost function with the function that maps ∞ 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 termination 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 (vi) 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 $\mathcal{L}_f$, with truth defined via unary $\vDash$ as usual. On the other hand, there are almost no proposals in the literature for languages suitable for describing programs of types (i)-(v), other than in the trivial sense in which they describe the information in the program preserved in the transition to level (vi). 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 (vi) programs, then a deterministic program would be a

function. We call this G-determinism to correspond to level (vi), or more mnemonically and independently of our particular hierarchy, IO-determinism (for input/output). At level (vii) determinism is not definable. A reasonable definition of a 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), and

(b) if there exist distinct edges (w,u), (w,v) then they are labelled with tests not simultaneously satisfiable.

Alternatively one could frame 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), (w,v) have disjoint domains.

To define 2-determinism (or L-determinism), it helps to have the notion of the prefix tree of a language. For $L \subseteq \Sigma^*$, let πL be $\{w \in \Sigma^* \mid \exists a \in \Sigma [wa \in L]\}$, the immediate prefixes of L, and let $\pi^* L$ be the least L' satisfying

$L \cup L' \cup \pi L' = L'$,

the prefixes of L. Then the prefix tree of L is the directed graph $T(L) = (\pi^* L, \{(\pi w, w) \mid w \in \pi^* L\})$. (Recall that graphs are presented as (V,E) where V is the vertex set and E the edge set.) Consider the edge (w,wa) (for $a \in \Sigma$) to be labelled a. Call those vertices of T(L) that are in 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 [29]. Such a tree can be executed by starting at the root (guaranteed to exist when the language is non-empty) and following a path along which no tests evaluate to false. Halting is permitted only at final nodes. Since we have produced from L a (possibly infinite) state transition diagram that generates L, we have an object to which we can apply whatever definition we used for 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. For $3 \leq k \leq 6$, k-totality seems best 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 $\{(w,u_1), \ldots (w,u_k)\}$ whose labels are tests such that $P_1 \vee \ldots \vee P_k$ is satisfiable. Alternatively, we could simply require that for every non-final vertex w there exists a set of edges $\{(w,u_1), \ldots (w,u_k)\}$ the union of whose domains is U.

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 in 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 "fastest(a)" which computes a 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 nondeterministic system, then the fastest should be chosen. This operation is not definable beyond type (iv). 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 (ii). Recursion is definable at level (vi) [10]. 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 (i), e.g. indexed grammars [1]), then the concept of block structure is not definable beyond type (ii). Call-by-value can be captured at level (ii) by combining block structure with assignment, but call-by-reference seems to call for either a very complex language (i.e. at level (i) 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 "thunks," but it too seems not definable beyond level (ii). (It 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 i to type j programs to a class C of type i programs is an isomorphism, we call C an <u>i→j-preserving</u> class. A program in such a class contains no information that 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 precedes every basic instruction by "T←T+1" where T is a new variable. This transformation yields a program (i) whose type 5 version is in a 5→6-preserving class, and (ii) whose type 6 version is identical to the type 6 version of the unmodified program to within the effect on T. 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; 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 return 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 S5 may be used to discuss correctness and termination simultaneously. Considering that S5 logics are those whose modalities have equivalence relations for their interpretations, we may infer that either Burstall was on the right track but had not developed the idea to the point where S5 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. Schwarz [33] has developed Burstall's work further, with a definite commitment to S5. Kroeger [21] has also proposed a modal approach to the logic of programs, in considerably more detail than Burstall, and with a concern for $\models$-semantics equal to ours. A major difference between our approach and Kroeger's is that where we regard programs as (interpretations of) modalities (unary logical connectives), Kroeger regards them as propositional variables, and has only one (program-independent) modality. Both systems represent interesting applications of modal logic, though the connection of ours with conventional first-order predicate calculus is more readily established through our program-oriented semantics of $\exists x$.

Recall from section 1.1 the interpretations of $[a]P$ and $\langle a\rangle P$. Under these interpretations the following formulae are visibly valid:

$[X \leftarrow 1]X=1$
$\langle X \leftarrow 1\rangle \underline{true}$
$[X>0]X>0$
$Y>0 \supset [X>0]Y>0$
$X=0 \supset \langle X=0\rangle \underline{true}$
$\langle c^*\rangle \underline{true}$
$X\geq 0 \supset \langle (X \leftarrow X-1)^*\rangle X=0$

These particular valid formulae generalize in some

obvious ways, which we can call axioms.

Logical Axioms
All tautologies of Propositional Calculus.
$[a](P \supset Q) \supset ([a]P \supset [a]Q)$.

Logical Inference Rules
$P,\ P \supset Q \vdash Q$.
$P \vdash [a]P$ (subsumes $P \vdash \forall x P$).

Some theorems that follow from these axioms are:

$[a](P \wedge Q) \equiv [a]P \wedge [a]Q$.
$\langle a \rangle (P \vee Q) \equiv \langle a \rangle P \vee \langle a \rangle Q$.
$\langle a \rangle (P \wedge Q) \supset (\langle a \rangle P \wedge \langle a \rangle Q)$.
$[a]P \supset (\langle a \rangle Q \supset \langle a \rangle (P \wedge Q))$.
$[a](P \supset Q) \supset (\langle a \rangle P \supset \langle a \rangle Q)$.

Axioms for Basic Programs
$\forall x P \supset [T/x]P$ (any $T \in \mathfrak{L}_t$) Universal Axiom.
$P \supset \forall x P$ unless $x \in P$ ∀ Frame Axiom.
where $x \in A(\underset{\sim}{B}) \equiv A \neq \exists x \wedge (A = x \vee x \in \underset{\sim}{B})$ (free occurrences).
$[P]Q \equiv P \supset Q$ Test Axiom.
$[F(\underset{\sim}{S}) \leftarrow T]P \equiv [\text{IF } \underset{\sim}{Z} = \underset{\sim}{S} \text{ THEN } T \text{ ELSE } F(\underset{\sim}{Z})/F(\underset{\sim}{Z})]P$ Assignment Axiom.
(Here IF-THEN-ELSE is removed as in Theorem 4.)

The two quantification axioms assert that "x←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 $\exists x$ denotes a particular program (x←RANDOM), we prefer to think of axioms involving $\exists x$ as non-logical.

The logical axiom $[a](P \supset Q) \supset ([a]P \supset [a]Q)$ and the non-logical ∀ Frame Axiom are combined in Mendelsohn's [25] system K as $\forall x(P \supset Q) \supset (P \supset \forall x Q)$ unless $x \in P$. 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
$[P]P$ Theorem of Intent.
$Q \supset [P]Q$ Theorem of Invariance.
$P \supset \langle P \rangle \underline{\text{true}}$ Theorem of Performance.

Assignments
$\underset{\sim}{s} = \underset{\sim}{S} \wedge t = T \supset [F(\underset{\sim}{S}) \leftarrow T]F(\underset{\sim}{s}) = t \quad t \neq F \neq s_i$ Theorem of Intent.

$(P \wedge y=F(\underset{\sim}{S}) \wedge \underset{\sim}{s}=\underset{\sim}{S}) \supset$
$[F(\underset{\sim}{S})\leftarrow T][IF\ \underset{\sim}{Z}=\underset{\sim}{s}\ THEN\ y\ ELSE\ F(\underset{\sim}{Z})/F(\underset{\sim}{Z})]P$
Theorem of Invariance.

$<F(\underset{\sim}{S})\leftarrow T>\underline{true}$ Theorem of Performance.

The reader familiar with predicate calculus will recognize in the logical axioms and rules, together with the two quantification axioms, a sound complete axiom system for the pure predicate calculus, which we can regard as a language for talking about "assignment" programs of the form $x\leftarrow RANDOM$. This prompts the question, is the axiom system we have given sound and complete when $\mathcal{L}$ is extended to include test and assignment modalities? This is easily answered in the affirmative, simply because the axioms for assignments and tests involve a direct equivalence with a formula not involving the command, unlike the axioms for quantifiers. The absence of such an equivalence for $\exists x$ considerably complicates the completeness proof; fortunately for us, this difficult problem was solved long ago. With such an equivalence, we know that the left side of the equivalence is provable if and only if the right side is. Since the right side does not involve assignment or test modalities, it is provable if and only if it is valid, since our axiomatization of the pure predicate calculus is sound and complete. Finally, the right side is valid if and only if the left side is, by Theorems 3 and 4. Hence for any test or assignment a , $[a]P$ is provable if and only if it is valid.

We now expand the system to include finite union and composition. The following are obvious corollaries of Theorems 6 and 7.

$[a \cup b]P \equiv [a]P \wedge [b]P$ Union Axiom.
$[a \cdot b]P \equiv [a][b]P$ Composition Axiom.

All of the above axioms have already been established as theorems in Section 2. If a is some loop-free program, the axioms "specify" a series of transformations of $[a]P$ that terminates with a formula of $\mathcal{L}_f$. This says much the same as Corollary 15. It also allows us to prove, by induction on the height of programs, that these axioms keep the system sound and complete even when $\mathcal{L}$ is augmented with modalities involving $\cup$ and $\cdot$.

To deal with $*$, we have:

$<a^n>P \supset <a^*>P$ Axioms of Intent.
$P \supset [a]P \vdash P \supset [a^*]P$ Rule of Invariance.
$[N+1/N]P \supset <a>P \vdash P \supset <a^*>[0/N]P$ Rule of Performance.

In the Axioms of Intent for $*$, n is a meta-variable giving one axiom per natural number. In the Rule of Performance,

N is in $\Im_0$, and we require 0 in $\Im_0$ and +1 (successor) in $\Im_1$. Besides = in the assignment axiom for non-zero arities, this is the only rule (or axiom) requiring non-logical symbols, and then only when $N \in P$.

A word of caution is in order here about replacing $\vdash$ by $\supset$ in the Rule of Invariance for * . The meaning of $(P \supset [a]P) \supset (P \supset [a^*]P)$ is that for any state $\mathcal{I}$, if $P \supset [a]P$ holds in $\mathcal{I}$ then so does $P \supset [a^*]P$. A counter-example to this would be when P is $X<10$, a is $[\![X \leftarrow X+1]\!]$ and $\mathcal{I}$ satisfies $X=0$. "Running" a once in this state will certainly preserve P , but running it ten or more times will not. A similar warning holds for the Rule of Performance for * , even if we rephrase it as the rule $\forall n [P(n+1) \supset \langle a \rangle P(n)] \vdash P(T) \supset \langle a^* \rangle P(0)$. In this case one counter-example would be to make $\mathcal{I}$ satisfy $X=2 \wedge Y=1$, and to take P(n) to be X=n and a to be $X \leftarrow X-Y \circ Y \leftarrow 0$. Then in $\mathcal{I}$ the antecedent holds, but after running a once, X can no longer decrease, and will thereafter remain stuck at 1 .

To see these rules in action, we may show with their help that the following program halts when $X \geq 0$ initially.

$$
\begin{array}{l}
Y \leftarrow 0 \circ (X \neq 0 \circ \\
\qquad (X \neq 0 \circ X \leftarrow X-1 \circ Y \leftarrow Y+1)^* \circ X=0 \circ \\
\qquad Y \leftarrow Y-1 \circ \\
\qquad (Y \neq 0 \circ Y \leftarrow Y-1 \circ X \leftarrow X+1)^* \circ Y=0)^* \circ \\
X=0
\end{array}
$$

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 PLANNER and CONNIVER, 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.

Proof. a forward tidy $\equiv \forall P \exists Q (Q \equiv [a]P)$
$\equiv \forall P \exists Q (Q \equiv \neg\langle a\rangle\neg P)$
$\equiv \forall P \exists Q (Q \equiv \langle a\rangle P)$ $\quad \neg\neg P \equiv P$
$\equiv \forall P \exists Q (Q \equiv \langle a^{--}\rangle P)$
$\equiv$ a^- backward tidy. ∎

Tidiness Characterization Lemma (TCL):
(a) Let a be forward tidy. Then $\{a\} = (a\Rightarrow)\cdot\{I\}$.
(b) Let b be backward tidy. Then $\{b\} = \{I\}\cdot(\Leftarrow b)$.

Proof.
(b) $P\{b\}R \equiv P \supset [b]R$
$\supset \forall Q (Q \equiv [b]R \supset P \supset Q)$
$\supset \exists Q (P\{I\}Q \wedge Q(\Leftarrow b)R)$ (B∃), Th 2
$\equiv P(\{I\}\cdot(\Leftarrow b))R$

$\exists Q (P\{I\}Q \wedge Q(\Leftarrow b)R) \supset \exists Q (P \supset Q \wedge Q \equiv [b]R)$ (B∀), Th 2
$\supset P \supset [b]R$

Hence $P\{b\}R \equiv P(\{I\}\cdot(\Leftarrow b))R$

(a) $\{a\} = \neg\{a\}^-$ (D)
$= \neg(\{I\}\cdot(\Leftarrow a^-)^-$ (b), (TDL)
$= \neg((\Leftarrow a^-)^-\cdot\{I\}^-)$
$= \neg(\Leftarrow a^-)^- \cdot \neg\{I\}^-$
$= (a\Rightarrow)\cdot\{I\}$ (D) ∎

Theorem 1. $\{\mathcal{P}\} = \mathcal{L}_f^2$.

Proof. $\mathcal{P} \models (P,Q)$ is true vacuously. ∎

Theorem 2. $\{I\} = \{(P,Q) \mid U \models (P \supset Q)\}$.

Proof. $\{I\} = \{(P,Q) \mid (\vartheta,\vartheta') \in I \supset (\vartheta \models P \supset \vartheta' \models Q)\}$ (by (F))
$= \{(P,Q) \mid \vartheta \in U \supset (\vartheta \models P \supset \vartheta \models Q)\}$ (def. of I)
$= \{(P,Q) \mid \vartheta \in U \supset \vartheta \models (P \supset Q)\}$ (by (1))
$= \{(P,Q) \mid U \models (P \supset Q)\}$ ∎

Theorem 3. Let R be a test.
(a) $\langle[\![R]\!]^-\rangle P \equiv R \wedge P$. (Forward tidiness)
(b) $[[\![R]\!]]P \equiv R \supset P$. (Backward tidiness)

Proof
(a) It suffices to prove that $\vartheta \models \langle[\![R]\!]^-\rangle P \equiv \vartheta \models (R \wedge P)$.
$\vartheta \models \langle[\![R]\!]^-\rangle P \equiv \bigvee_{\vartheta'[\![R]\!]\vartheta} \vartheta' \models P$
$\equiv \vartheta \models R \wedge \vartheta \models P$
$\equiv \vartheta \models (R \wedge P)$

(b) $[[\![R]\!]]P \equiv \neg\langle[\![R]\!]\rangle\neg P$
$\equiv \neg(R \wedge \neg P)$ (using (a), and $[\![R]\!] = [\![R]\!]^-$)
$\equiv R \supset P$ ∎

IF lemma. Evaluating $\mathcal{I}\models W$ 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

$$\mathcal{I}\models(\text{IF } P \text{ THEN } S \text{ ELSE } T) = \text{if } \mathcal{I}\models P \text{ then } \mathcal{I}\models S \text{ else } \mathcal{I}\models T.$$

Proof. Straightforward. (Use induction on depth of IF-terms.) ∎

Theorem 4. Let $F(\underset{\sim}{S})\leftarrow T$ be an assignment.

(a) $\langle\!\langle F(\underset{\sim}{S})\leftarrow T\rangle\!\rangle P \equiv \exists y_{\underset{\sim}{s}}[P' \wedge \underset{\sim}{s}=\underset{\sim}{S}' \wedge F(\underset{\sim}{s})=T']$
where $E' = [(\text{IF } \underset{\sim}{Z}=\underset{\sim}{s} \text{ THEN } y \text{ ELSE } F(\underset{\sim}{Z}))/F(\underset{\sim}{Z})]E$

(b) $[\![F(\underset{\sim}{S})\leftarrow T]\!]P \equiv P''$
where $E'' = [(\text{IF } \underset{\sim}{Z}=\underset{\sim}{S} \text{ THEN } T \text{ ELSE } F(\underset{\sim}{Z}))/F(\underset{\sim}{Z})]E$.

Proof.

(a) We first prove the aging lemma.

Aging Lemma. Suppose $A_{\mathcal{I}} = A_{\mathcal{J}}$ for all symbols $A\neq F$, and $F_{\mathcal{I}}(\underset{\sim}{\chi}) = F_{\mathcal{J}}(\underset{\sim}{\chi})$ for all $\underset{\sim}{\chi}\neq\underset{\sim}{s}_{\mathcal{I}}$, and $F_{\mathcal{I}}(\underset{\sim}{s}_{\mathcal{I}}) = x_{\mathcal{I}}$. Then $\mathcal{I}\models A(\underset{\sim}{B}) = \mathcal{J}\models A(\underset{\sim}{B})'$.

Proof. By induction on the height of $A(\underset{\sim}{B})$. Assume $\mathcal{I}\models\underset{\sim}{B} = \mathcal{J}\models\underset{\sim}{B}'$.

Case (i). $A=F$.

Subcase (a), $\mathcal{I}\models\underset{\sim}{B} \neq \underset{\sim}{s}_{\mathcal{I}}$.

$\mathcal{I}\models F(\underset{\sim}{B}) = F_{\mathcal{I}}(\mathcal{I}\models\underset{\sim}{B})$
$= F_{\mathcal{J}}(\mathcal{J}\models\underset{\sim}{B}')$ ($\mathcal{I}\models\underset{\sim}{B}\neq\underset{\sim}{s}_{\mathcal{I}}$, ind. hyp.)
$= \mathcal{J}\models F(\underset{\sim}{B}')$
$= \mathcal{J}\models$ (IF $\underset{\sim}{B}'\neq\underset{\sim}{s}$ THEN $F(\underset{\sim}{B}')$ ELSE x)
($\mathcal{J}\models\underset{\sim}{B}' = \mathcal{I}\models\underset{\sim}{B}\neq\underset{\sim}{s}_{\mathcal{I}}$, IF lemma)
$= \mathcal{J}\models F(\underset{\sim}{B})'$ (def. of ')

Subcase (b), $\mathcal{I}\models\underset{\sim}{B} = \underset{\sim}{s}_{\mathcal{I}}$:

$\mathcal{I}\models F(\underset{\sim}{B}) = F_{\mathcal{I}}(\mathcal{I}\models\underset{\sim}{B})$
$= F_{\mathcal{I}}(\underset{\sim}{s}_{\mathcal{I}})$ (given)
$= x_{\mathcal{I}}$ (given)
$= \mathcal{J}\models$ (IF $\underset{\sim}{B}'\neq\underset{\sim}{s}$ THEN $F(\underset{\sim}{B}')$ ELSE x)
($\mathcal{J}\models\underset{\sim}{B}' = \mathcal{I}\models\underset{\sim}{B} = \underset{\sim}{s}_{\mathcal{I}}$, IF lemma)
$= \mathcal{J}\models F(\underset{\sim}{B})'$ (def. of ').

Case (ii), $A=\exists x$.
Cf. case (ii) of 4(b).

Case (iii), Other A.

$\mathcal{I}\models A(\underset{\sim}{B}) = A_{\mathcal{I}}(\mathcal{I}\models\underset{\sim}{B})$
$= A_{\mathcal{J}}(\mathcal{J}\models\underset{\sim}{B}')$ ($A_{\mathcal{I}}=A_{\mathcal{J}}$, ind. hyp.)
$= \mathcal{J}\models A(\underset{\sim}{B}')$
$= \mathcal{J}\models A(\underset{\sim}{B})'$ (def. of ') ∎

Aging corollary 1. If $\mathcal{S}[\![F(\underset{\sim}{S})\leftarrow T]\!]\mathcal{J}$, $\mathcal{S}\models F(\underset{\sim}{S}) = x_{\mathcal{J}}$ and $\mathcal{S}\models \underset{\sim}{S} = \underset{\sim}{s}_{\mathcal{J}}$, then $\mathcal{S}\models A(\underset{\sim}{B}) = \mathcal{J}\models A(\underset{\sim}{B})'$.

Aging corollary 2. Given $\mathcal{J}$, if $\mathcal{S}$ is

$$\lambda B.\text{if } B\neq F \text{ then } B_{\mathcal{J}} \text{ else } \lambda\underset{\sim}{x}.\text{if } \underset{\sim}{x}\neq\underset{\sim}{s}_{\mathcal{J}} \text{ then } F_{\mathcal{J}}(\underset{\sim}{x}) \text{ else } x_{\mathcal{J}}$$

then $\mathcal{S}\models A(\underset{\sim}{B}) = \mathcal{J}\models A(\underset{\sim}{B})'$.

Transition existence lemma. Given $\mathcal{J}$, if $\mathcal{S}$ is as in the previous corollary, $\underset{\sim}{s}_{\mathcal{J}} = \mathcal{J}\models\underset{\sim}{S}'$, and $F_{\mathcal{J}}(\underset{\sim}{s}_{\mathcal{J}}) = \mathcal{J}\models T'$, then $\mathcal{S}[\![F(\underset{\sim}{S})\leftarrow T]\!]\mathcal{J}$. (That is, constructing $\mathcal{S}$ from $\mathcal{J}$ in this way guarantees a transition from $\mathcal{S}$ to $\mathcal{J}$ via $[\![F(\underset{\sim}{S})\leftarrow T]\!]$.)

Proof. By aging corollary 2 we have $\underset{\sim}{s}_{\mathcal{J}} = \mathcal{S}\models\underset{\sim}{S}$, and $F_{\mathcal{J}}(\mathcal{S}\models\underset{\sim}{S}) = \mathcal{S}\models T$.

Then
$$\begin{aligned}
\mathcal{S}[\![F(\underset{\sim}{S})\leftarrow T]\!] &= \lambda A.\text{if } A\neq F \text{ then } A_{\mathcal{J}} \text{ else } \lambda\underset{\sim}{x}.\text{if } \underset{\sim}{x}\neq\mathcal{S}\models\underset{\sim}{S} \text{ then } F_{\mathcal{J}}(\underset{\sim}{x}) \text{ else } \mathcal{S}\models T\\
&= \lambda A.\text{if } A\neq F \text{ then } A_{\mathcal{J}} \text{ else } \lambda\underset{\sim}{x}.\text{if } \underset{\sim}{x}\neq\mathcal{S}\models\underset{\sim}{S} \text{ then } F_{\mathcal{J}}(\underset{\sim}{x}) \text{ else } F_{\mathcal{J}}(\mathcal{S}\models\underset{\sim}{S})\\
&\quad \text{(def. of } \mathcal{S}\text{ , } \underset{\sim}{s}_{\mathcal{J}} = \mathcal{S}\models\underset{\sim}{S}\text{ , } F_{\mathcal{J}}(\mathcal{S}\models\underset{\sim}{S}) = \mathcal{S}\models T)\\
&= \lambda A.\text{if } A\neq F \text{ then } A_{\mathcal{J}} \text{ else } \lambda\underset{\sim}{x}.F_{\mathcal{J}}(\underset{\sim}{x})\\
&= \lambda A.\text{if } A\neq F \text{ then } A_{\mathcal{J}} \text{ else } F_{\mathcal{J}} \quad (\eta\text{-reduction})\\
&= \lambda A.A_{\mathcal{J}}\\
&= \mathcal{J} \quad (\eta\text{-reduction}).
\end{aligned}$$

∎

We can now complete the proof of Theorem 4(a). It suffices to show that

$$\mathcal{J}\models\langle[\![F(\underset{\sim}{S})\leftarrow T]\!]^{-}\rangle P \equiv \mathcal{J}\models\exists x\underset{\sim}{s}[P' \wedge s=\underset{\sim}{S}' \wedge F(\underset{\sim}{s}) = T'].$$

Now L.H.S. $\equiv \exists\mathcal{S}[\mathcal{S}\models P \wedge \mathcal{S}[\![F(\underset{\sim}{S})\leftarrow T]\!]\mathcal{J}]$

$\equiv \exists\mathcal{S}[\mathcal{S}\models P \wedge \mathcal{S}[\![F(\underset{\sim}{S})\leftarrow T]\!]\mathcal{J} \wedge F_{\mathcal{J}}(\mathcal{S}\models S) = \mathcal{S}\models T]$

(third conjunct implied by second by def. of $[\![F(\underset{\sim}{S})\leftarrow T]\!]$)

$\equiv \exists x_{\mathcal{J}}\underset{\sim}{s}_{\mathcal{J}}[\mathcal{J}\models P' \wedge \underset{\sim}{s}_{\mathcal{J}}=\mathcal{J}\models\underset{\sim}{S}' \wedge F_{\mathcal{J}}(\underset{\sim}{s}_{\mathcal{J}}) = \mathcal{J}\models T']$

($\supset$: take $x_{\mathcal{J}}=\mathcal{S}\models F(\underset{\sim}{S})$, $s_{\mathcal{J}}=\mathcal{S}\models\underset{\sim}{S}$. $\subset$: take $\mathcal{S}$ as in a.c.2)

$\equiv \mathcal{J}\models\exists x\underset{\sim}{s}[P' \wedge \underset{\sim}{s}=\underset{\sim}{S}' \wedge F(\underset{\sim}{s}) = T']$.

$\equiv$ R.H.S.

The preceding lemmas make it straightforward to verify each step.

(b) It suffices to show that for all $\mathcal{S}$,
$\mathcal{J}\models P \equiv \mathcal{S}\models P'$

where $\mathfrak{f} = \lambda\mathfrak{g}.\lambda A.$ if $A \neq F$ then $A_{\mathfrak{g}}$
else $\lambda\underset{\sim}{s}.$ if $\underset{\sim}{s} \neq \mathfrak{g}\models\underset{\sim}{S}$ then $A_{\mathfrak{g}}(\underset{\sim}{s})$
else $\mathfrak{g}\models T$.
and $E' = [\text{IF } \underset{\sim}{Z}=\underset{\sim}{S} \text{ THEN T ELSE } F(\underset{\sim}{Z})/F(\underset{\sim}{Z})]E$.

We proceed by induction on the height of $P = A(\underset{\sim}{B})$.
We take as our induction hypothesis:
$\forall y \in \mathfrak{I}_0 - \{F\}(\mathfrak{g}'' =_y \mathfrak{g} \wedge \mathfrak{g}''[\![F(\underset{\sim}{S}) \leftarrow T]\!]\mathfrak{f}'' \supset \mathfrak{f}''\models P = \mathfrak{g}''\models P')$
where $\mathfrak{g} =_y \mathfrak{f}$ means that $\mathfrak{g}$ and $\mathfrak{f}$ differ only in their assignment to y .

Case (i). $A=F$.
$\mathfrak{f}\models F(\underset{\sim}{B}) = F_{\mathfrak{f}}(\mathfrak{f}\models\underset{\sim}{B})$
$= F_{\mathfrak{f}}(\mathfrak{g}\models\underset{\sim}{B}')$
$=$ if $\mathfrak{g}\models\underset{\sim}{B}' = \mathfrak{g}\models\underset{\sim}{S}$ then $\mathfrak{g}\models T$ else $F_{\mathfrak{f}}(\mathfrak{g}\models\underset{\sim}{B}')$
$= \mathfrak{g}\models[\text{IF } \underset{\sim}{B}' = \underset{\sim}{S} \text{ THEN T ELSE } F(\underset{\sim}{B}')]$
$= \mathfrak{g}\models F(\underset{\sim}{B})'$.

Case (ii). $A=\exists x$.
$\mathfrak{f}\models\exists xP \equiv \mathfrak{f}\models\exists y[y/x]P$ (α-reduction)
$\equiv \exists\mathfrak{f}''(\mathfrak{f}'' =_y \mathfrak{f} \wedge \mathfrak{f}''\models[y/x]P)$
$\equiv \exists\mathfrak{g}''(\mathfrak{g}'' =_y \mathfrak{g} \wedge \mathfrak{g}''\models([y/x]P)')$
$\equiv \mathfrak{g}\models\exists y([y/x]P)')$
$\equiv \mathfrak{g}\models(\exists xP)'$.

Case (iii). Other A.
$\mathfrak{f}\models A(\underset{\sim}{B}) = A_{\mathfrak{f}}(\mathfrak{f}\models\underset{\sim}{B})$
$= A_{\mathfrak{g}}(\mathfrak{g}\models\underset{\sim}{B}')$
$= \mathfrak{g}\models A(\underset{\sim}{B}')$
$= \mathfrak{g}\models A(\underset{\sim}{B})'$. ∎

Theorem 5. Let $F \leftarrow G$ be a second-order assignment. Then
$[\![F \leftarrow G]\!]P = [G/F]P$
($[G/F]$ is a convenient abbreviation for $[G(\underset{\sim}{Z})/F(\underset{\sim}{Z})]$.)
Hence second-order assignment is backward tidy.

Proof. Essentially the same as for Theorem 4(b).

Theorem 6. $\{a \cup b\} = \{a\} \cap \{b\}$.

Proof. $P\{a \cup b\}Q$
$\equiv \forall\mathfrak{g}\mathfrak{f}(\mathfrak{g}(a \cup b)\mathfrak{f} \supset (\mathfrak{g},\mathfrak{f})\models(P,Q))$
$\equiv \forall\mathfrak{g}\mathfrak{f}((\mathfrak{g}a\mathfrak{f} \vee \mathfrak{g}b\mathfrak{f}) \supset (\mathfrak{g},\mathfrak{f})\models(P,Q))$
$\equiv \forall\mathfrak{g}\mathfrak{f}((\mathfrak{g}a\mathfrak{f} \supset (\mathfrak{g},\mathfrak{f})\models(P,Q)) \wedge (\mathfrak{g}b\mathfrak{f} \supset (\mathfrak{g},\mathfrak{f})\models(P,Q)))$
$\equiv P\{a\}Q \wedge P\{b\}Q$
$\equiv P(\{a\} \cap \{b\})Q$. ∎

Theorem 7A. $\{a \circ b\} \supseteq \{a\} \circ \{b\}$.

Proof. $P\{a\}\circ\{b\}R$

$\equiv \exists Q\forall\vartheta\forall\xi K((\vartheta a\xi \supset (\vartheta,\xi)\models(P,Q)) \wedge (\xi bK \supset (\xi,K)\models(Q,R)))$

$\supset \exists Q\forall\vartheta\forall\xi K((\vartheta a\xi \wedge \xi bK) \supset ((\vartheta,\xi)\models(P,Q) \wedge (\xi,K)\models(Q,R)))$

$\supset \exists Q\forall\vartheta\forall\xi K(\vartheta a\xi bK \supset (\vartheta,K)\models(P,R))$

$\equiv \forall\vartheta\forall\xi K(\vartheta(a\circ b)K \supset (\vartheta,K)\models(P,R))$

$\equiv P\{a\circ b\}R$. ∎

Theorem 7. $\{a\circ b\} = \{a\}\circ\{b\}$ when a is forward tidy or b is backward tidy.

Proof. It suffices to show $\{a\circ b\} \subseteq \{a\}\circ\{b\}$.

(a) $P\{a\circ b\}R$

$\equiv \forall\vartheta K(\vartheta(a\circ b)K \supset (\vartheta,K)\models(P,R))$

$\equiv \forall\xi K(\exists\vartheta(\vartheta a\xi bK \wedge \vartheta\models P) \supset K\models R)$

$\equiv \forall\xi K(\xi bK \supset (\exists\vartheta(\vartheta a\xi \wedge \vartheta\models P) \supset K\models R))$

$\equiv \forall\xi K(\xi bK \supset (\xi\models\langle a^-\rangle P \supset K\models R))$

$\equiv \langle a^-\rangle P\{b\}R$

$\equiv P\{a\}\langle a^-\rangle P\{b\}R$ since $P\{a\}\langle a^-\rangle P$

$\supset P(\{a\}\circ\{b\})R$ ∎

Theorem 8.

(a) If a,b are forward tidy, so are $a\cup b$ and $a\circ b$;

(b) If a,b are backward tidy, so are $a\cup b$ and $a\circ b$.

Proof.

(a) $\langle(a\cup b)^-\rangle P \equiv \langle a^-\cup b^-\rangle P$

$\equiv \langle a^-\rangle P \vee \langle b^-\rangle P$

$\equiv Q\vee R$ where $Q\equiv\langle a^-\rangle P$ and $R\equiv\langle b^-\rangle P$.

$\langle(a\circ b)^-\rangle P \equiv \langle b^-\circ a^-\rangle P$

$\equiv \langle b^-\rangle\langle a^-\rangle P$

$\equiv \langle b^-\rangle Q$ where $Q\equiv\langle a^-\rangle P$

$\equiv R$ where $R\equiv\langle b^-\rangle Q$.

(b) $[a\cup b]P \equiv [a]P\wedge[b]P$

$\equiv Q\wedge R$ where $Q\equiv[a]P$ and $R\equiv[b]P$.

$[a\circ b]P \equiv [a][b]P$

$\equiv [a]Q$ where $Q\equiv[b]P$

$\equiv R$ where $R\equiv[a]Q$. ∎

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. ∎

In the following few theorems, a useful result is:

Lemma 10. $\{a\} \leq_m \{a^-\}$.

Proof. $\{a\} = \neg\{a^-\}^-$ (D)
$\leq_m \{a^-\}$ by the obvious calculation. ∎

We note also that TDL can be strengthened to include the word "recursively" before every occurrence of "tidy." If f is the recursive tidiness function of a then the dual tidiness function g of a^- is defined by $g(P) = \neg f(\neg P)$.

Theorem 10. $\{a \cup b\} \leq_m \{a\} \times \{b\}$ (Cartesian product).

Proof. $\{a \cup b\} = \{a\} \cap \{b\}$ Th 6
$\leq_m \{a\} \times \{b\}$ ∎

Theorem 11.
(a) If a is forward recursively tidy, $\{a \cdot b\} \leq_m \{b\}$.
(b) If b is backward recursively tidy, $\{a \cdot b\} \leq_m \{a\}$.

Proof.
(a) $\{a \cdot b\} = \{a\} \cdot \{b\}$ Th 7
$= (a\Rightarrow) \cdot \{I\} \cdot \{b\}$ TDL
$= (a\Rightarrow) \cdot \{I \cdot b\}$ Th 7
$= (a\Rightarrow) \cdot \{b\}$
Hence to test $P\{a \cdot b\}R$ it suffices to calculate the Q satisfying $P(a\Rightarrow)Q$ and test $Q\{b\}R$.

(b) $\{a \cdot b\} \leq_m \{b^- \cdot a^-\}$ Lemma D
$\leq \{b^-\}$ Th 11(a)
$\leq \{b\}$ Lemma D. ∎

Theorem 12. If a is recursively tidy, $\{a\} \leq_m \{I\}$.

Proof
(a) If a is forward recursively tidy,
$\{a\} = \{a \cdot I\}$
$\leq_m \{I\}$ Th 11(a).

(b) When a is backward recursively tidy,
$\{a\} \leq_m \{a^-\}$ Lemma D
$\leq_m \{I\}$ TDL, 12(a). ∎

Theorem 13. Instructions are recursively very tidy.

Proof. The strongest consequents and weakest antecedents given by Theorems 3 and 4 are easily calculated. ∎

Theorem 14. If a,b are forward (backward) recursively tidy, so are $a \cup b$ and $a \cdot b$.

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, $\{a\} \leq_m \{1\}$.

(Note that $\{1\}^n = \{1\}x\{1\}x\ldots x\{1\} \leq_m \{1\}$, for any n, since the n questions about membership in $\{1\}$ can be rephrased as a single conjunction.)

Proof. This follows by induction on the height of a program, using Theorems 10-14. ∎

Theorem 16. Let $|\mathfrak{I}_0| \geq 4$, $|\mathfrak{I}_1| \geq 3$, $|\mathfrak{I}_3| \geq 1$, with $V \in \mathfrak{I}_0$, $F \in \mathfrak{I}_1$. Let the symbols of $\mathfrak{I}$ and $\mathfrak{P}$ (excepting $=$) take on all possible interpretations in the universe U . Then $\{[V \leftarrow F(V)]^*\}$ is not r.e., despite $\{1\}$ and $\{[V \leftarrow F(V)]\}$ both being r.e.

Proof. The idea is to make V 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 V) . The basic instructions labelling the edges of the graph will be $X \leftarrow X+1$, $X \leftarrow X-1$, $X=0$, $X \neq 0$, $Y \leftarrow Y+1$, $Y \leftarrow Y-1$, $Y=0$, $Y \neq 0$. (See Minsky [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 ($X=0$, $X \neq 0$ or $Y=0$, $Y \neq 0$) . (If necessary, add edges labelled $X=0$ and $X \neq 0$ from f to f.) 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 that where control goes next be completely specified for every vertex, otherwise F may take V 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 X and Y to 0 first.

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

$$\forall \underset{\sim}{x}\underset{\sim}{y}[C(\underset{\sim}{x}) = C(\underset{\sim}{y}) \supset \underset{\sim}{x}=\underset{\sim}{y}] \quad .$$

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

$$\forall x[U(x)\neq 0 \wedge D(U(x)) = x] \quad .$$

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.

Instruction on edge (i,j) .	Corresponding sentence
$X\leftarrow X+1$	$\forall xy[F(C(x\ ,y,\ U^i(0))) = C(U(x),y,U^j(0))]$
$X\leftarrow X-1$	$\forall xy[F(C(U(x),y,\ U^i(0))) = C(x\ ,y,U^j(0))]$
$X=0$	$\forall y[F(C(0\ ,y,\ U^i(0))) = C(0\ ,y,U^j(0))]$
$X\neq 0$	$\forall xy[F(C(U(x),y,\ U^i(0))) = C(U(x),y,U^j(0))]$

and similarly for Y .

Claim 1. Given any interpretation $\mathcal{I}$ satisfying P_N , in which all symbols save F are assigned interpretations, let N denote $\{\mathcal{I}\models U^n(0) \mid n\geq 0\}$ and let M denote $\{\mathcal{I}\models U^n(0) \mid m$ labels a vertex of $p\}$. Thus N is that subset of D reachable via $U_{\mathcal{I}}$ from $0_{\mathcal{I}}$, and M is that subset of D corresponding to the vertices of the flowchart. Then the above table consistently and completely determines $F_{\mathcal{I}}(C_{\mathcal{I}}(x,y,z))$ for all $x,y \in N$ and $z \in M$, except when $("z",i)$ is labelled with $X\leftarrow X-1$, in which case it is undetermined when $x = 0_{\mathcal{I}}$, and similarly for $Y\leftarrow Y-1$. ("z" is the necessarily unique natural number satisfying $U_{\mathcal{I}}^{"z"}(0_{\mathcal{I}})=z$.)

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 $F_{\mathcal{I}}(C_{\mathcal{I}}(x,y,i))$ is completely specified except for the decrement instructions. In the latter case, $F_{\mathcal{I}}(C_{\mathcal{I}}(0,y,i))$ is specified, as is $F_{\mathcal{I}}(C_{\mathcal{I}}(U(x),y,i))$, accounting for all elements of N . Consistency follows from P_C and P_N which together ensure that each of the above equations specifies $F_{\mathcal{I}}$ at a different element of the domain. ∎

Claim 2. If $x,y\in N$ and $z\in M$ then $F_{\mathcal{I}}(C_{\mathcal{I}}(x,y,z)) = C_{\mathcal{I}}(a,b,c)$ where, if p is started with "control" at vertex "z" and X,Y contain "x","y" respectively, then running p for one step yields "a" in X and "b" in Y , with control at vertex "c" .

Proof. Straightforward. ∎

Now assume that $\{[\![V\leftarrow F(V)]\!]^*\}$ is r.e. Then we can decide whether p started with i in register X and 0 in register Y 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 $\{[\![V\leftarrow F(V)]\!]^*\}$ looking for $(P_C \wedge P_N \wedge P_F \wedge P_I\ ,\ V \neq C(0,0,U^f(0))$ where P_I is $V=C(U^i(0),0,U^s(0))$.
The crucial observation is that we will find this pca if and only if p does not halt. For certainly if we find it we know that by Lemma 2 the machine cannot get into the state (0,0,f). Conversely, if the machine cannot get into this state, then by Lemmas 1 and 2 $V \neq C(0,0,U^f(0))$ will remain true no matter how often F is applied to V .

This completes the proof of Theorem 16. ∎

Corollary 17. When $\{I\}$ is r.e., $[\![V\leftarrow F(V)]\!]^*$ is not recursively tidy.

Proof. Suppose $[\![V\leftarrow F(V)]\!]^*$ to be recursively tidy. Then

$$\{[\![V\leftarrow F(V)]\!]^*\} = \{[\![V\leftarrow F(V)]\!]^* \cdot I\} \leq_m \{I\} \qquad \text{Th. 11}$$

but this would imply that $\{[\![V\leftarrow F(V)]\!]^*\}$ is r.e. , contradicting Theorem 16. ∎

Theorem 18. If $\leq \in \mathcal{P}$ then $\{[\![V\leftarrow V+1]\!]^*\}$ is recursively very tidy.

Proof. $\langle [\![V\leftarrow V+1]\!]^- \rangle P \equiv \exists n (n \leq V \wedge [n/V]P)$.
$[[\![V\leftarrow V+1]\!]^*]P \equiv \forall n (V \leq n \supset [n/V]P)$. ∎

Theorem 19. $(\mathcal{P}) = (I) = I_{\mathcal{L}_f}$.

Proof. Straightforward. ∎

Theorem 20. $(a \cup b) = (a) \cap (b)$.

Proof. Straightforward. ∎

Theorem 21. $(a \circ b) \supseteq (a) \circ (b) = (a) \cap (b)$.

Proof. Straightforward. ∎

Corollary 22. For a given program a, the structure $(\{(a^n) \mid n \geq 0\}, \subseteq)$ is a homomorph of the natural number division lattice $(N, |)$, with (a) as the least element and (I) as the greatest. Further, when $a = [\![X\leftarrow F(X)]\!]$ with F uninterpreted, the homomorphism becomes an isomorphism.

Proof. If $m|n$ then $(a^m) \subseteq (a^n)$.
Further, when $a = [\![X\leftarrow F(X)]\!]$, if $m \not| n$ then the formula

$P(X) \wedge \forall x(P(x) \supset \neg P(F(X)) \wedge \neg P(F^2(X)) \wedge \ldots \wedge P(F^m(X)))$
(which makes P hold once every m applications of F)
is an invariant of (strictly, is a projection of an invariant of)
a^m but not of a^n. ∎

Theorem 23. $\langle a^* \rangle = \langle a \rangle$

Proof. This follows immediately from $a^* = \cup\{a^n \mid n \geq 0\}$, theorem 20 and corollary 22 (the part of the corollary that says that $\langle a \rangle$ is the least element of $\{\langle a^n \rangle \mid n \geq 0\}$). ∎

Theorem 24 (Star Interpolation Theorem). Let a^* be tidy, with $P\{a^*\}R$. Then there exists Q satisfying $P \supset Q \supset R$ and $Q\{a\}Q$. (An equivalent statement of the theorem is that if a^* is tidy, $\{a^*\} = \{1\} \circ \langle a \rangle \circ \{1\}$.)

Proof. We need only treat the case when a^* is forward tidy; the other case is the exact dual. Choose $Q = P\langle a^* \Rightarrow \rangle$. Then $Q \supset R$ since Q is the strongest consequent of P, and $P \supset Q$ since $I_U \subseteq a^*$. Moreover (using an improved version of our original argument suggested by R. Rivest)

	$P\{a^*\}Q$	
so	$P\{a^* \circ a\}Q$	since $a^* \circ a \subseteq a^*$
so	$P(\{a^*\} \circ \{a\})Q$	Theorem 7; a^* is forward tidy
so	$P\{a^*\}S\{a\}Q$	for some $S \in \mathcal{L}_f$
thus	$Q \supset S$	$P\{a^*\}S$ and Q is strongest
whence	$Q\{a\}Q$	$S\{a\}Q$. ∎

Corollary 25. When all regular programs are tidy, $\{a\} \leq_e \{1\}$.

Proof. We proceed by induction on the height of a regular expression representing a. If a is an instruction, the result follows from Theorems 12 and 13. If a is the union or composition of two programs then Theorems 10 and 11 together with the induction hypothesis apply. If $a = b^*$ then by Theorem 24 $\{a\} = \{1\} \circ \langle b \rangle \circ \{1\}$. By induction, all the components of this composition are r.e. reducible to $\{1\}$, hence so is $\{a\}$. ∎

Corollary 26. Under the conditions of Theorem 16, if $\{1\}$ is r.e. then $[\![V \leftarrow F(V)]\!]^*$ is not tidy.

Proof. If it were tidy, then by Theorem 24 $\{[\![V \leftarrow F(V)]\!]\}$ would be $\{1\} \circ \langle [\![V \leftarrow F(V)]\!] \rangle \circ \{1\}$, which is r.e. because all of its components are r.e. But this would then contradict Theorem 16. ∎

Acknowledgments

The material in this paper evolved during three semantics courses taught by the author. Some of the students were of especial help: the key ideas in Theorem 4 are due to R. Hale,

while the connection between Kripke semantics and programs was suggested by R. Moore. Several discussions with M. Fischer proved valuable. J. Schwarz pointed out the absence of an induction axiom in an earlier formulation of the modal axioms. A. Meyer proved that {[X←F(X)]*} was not finitely axiomatizable, prompting G. Plotkin to ask us whether it was r.e. A. Meyer, R. Rivest and Y. Shestov commented helpfully on early drafts of this paper.

References

[1] Aho, A.V. Indexed Grammars. JACM 15, 4, 647-671, 1968.

[2] --------, J. E. Hopcroft and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass. 1974.

[3] Ashcroft, E. and Z. Manna. The translation of 'go to' programs to 'while' programs. STAN-CS-71-188, Stanford, CA. 1971.

[4] Berztiss, A.T. Data Structures. (2nd Ed.) Academic Press, N.Y. 1975.

[5] Burstall, R.M. Program Proving as Hand Simulation with a Little Induction. IFIP 1974, Stockholm.

[6] Chomsky, N. Aspects of the Theory of Syntax. MIT Press, Cambridge, Mass. 1965.

[7] Cook, S.A. Axiomatic and Interpretive Semantics for an Algol Fragment. TR-79, Toronto, Feb. 1975.

[8] Craig, W. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. JSL 22, 250-268, 1957.

[9] de Bakker, J.W., and D. Scott. An outline of a theory of programs. Unpublished manuscript, 1969.

[10] ----------, and W.P. de Roever. A calculus for recursive program schemes. in Automata, Languages and Programming (ed. Nivat), 167-196. North Holland, 1972.

[11] ----------, and L.G.L.T. Meertens. On the Completeness of the Inductive Assertion Method. JCSS 11, 323-357, 1975.

[12] Dijkstra, E. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N.J. 1976.

[13] Eilenberg, S. and C. Elgot. Recursiveness. Academic Press, N.Y. 1970.

[14] Elgot, C.C. Structured Programming With and Without GO TO Statements. IEEE Transactions on Software Engineering, SE-2, 1, 41-53, March 1976.

[15] Floyd, R.W. Assigning Meanings to Programs. in Mathematical Aspects of Computer Science (ed. J.T. Schwartz), 19-32, 1967.

[16] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM 12, 576-580, 1969.

[17] Karp, R.M. Some applications of logical syntax to digital computing. Ph. D. Thesis, Harvard, 1959.

[18] King, J. A Program Verifier. Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa, 1969.

[19] Knuth, D. E. The Art of Computer Programming, 1. Addison-Wesley, Reading, MA. 1968.

[20] Kripke, S. Semantical considerations on Modal Logic. Acta Philosophica Fennica, 83-94, 1963.

[21] Kroeger, F. Logical Rules of Natural Reasoning about Programs. In Automata, Languages and Programming 3 (ed. Michaelson, S. and R. Milner), 87-98. Edinburgh University Press, 1976.

[22] Luckham, D. and N. Suzuki. Automatic Program Verification IV: Proof of Termination within a Weak Logic of Programs. STAN-CS-75-522, Stanford, October 1975.

[23] Manna, Z. Mathematical Theory of Computation. McGraw-Hill, 1974.

[24] -----, and A. Pnueli. Axiomatic Approach to Total Correctness of Programs. Acta Informatica, 3, 243-263, 1974.

[25] Mendelsohn, E. Introduction to Mathematical Logic. Van Nostrand, N.Y. 1964.

[26] Minsky, M.L. Computation - Finite and Infinite Machines. Prentice-Hall, N.J. 1967.

[27] Pratt, V.R. Semantics of Programming Languages. Lecture notes for 6.892, Fall 1974, M.I.T.

[28] Quine, W.V.O. Word and Object. MIT Press, MA. 1960.

[29] Reingold, E. M. On the Optimality of Some Set Algorithms. JACM 19, 4, 649-659, April 1972.

[30] Rogers, H. Theory of Recursive Functions and Effective Computability. McGraw-Hill, 1967.

[31] Scott, D. Toward a Mathematical Semantics for Computer Languages. Symposium on Computers and Automata, Microwave Research Institute Proceedings, 21, Polytechnical Institute of Brooklyn, 1971.

[32] Schwarz, J.S. Semantics of Partial Correctness Formalisms. Ph.D. Dissertation, Syracuse, Dec 1974.

[33] --------. Event Based Reasoning - A System for Proving Correct Termination of Programs. In Automata, Languages and Programming 3 (ed. Michaelson, S. and R. Milner), 131-146. Edinburgh University Press, 1976.

[34] Tarski, A. The semantic conception of truth and the foundations of semantics. Philos. and Phenom. Res, 4, 341-376, 1944.

[35] Wirth, N. The programming language Pascal. Acta Informatica, 1, 35-63, 1971.