LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-117

SIX LECTURES ON DYNAMIC LOGIC

Vaughan R. Pratt

December 1979

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-117

Six Lectures on Dynamic Logic

Vaughan R. Pratt

December, 1978

This paper contains the material for six lectures given at the 3rd Advanced Course on Foundations of Computer Science, at the Mathematisch Centrum, Amsterdam, August 21-25.

This research was supported by the National Science Foundation under NSF grant nos. MCS76-18461 and MCS78-04338.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
CAMBRIDGE MASSACHUSETTS 02139

Six Lectures on Dynamic Logic

Vaughan R. Pratt

Abstract

The distinction made here between static and dynamic logic has a very simple character, yet can play a central and unifying role in logic as a vantage point from which one can compare propositional calculus, predicate calculus, intensional logics such as modal logic and temporal logic, various algorithmic logics (logics of programs), and Quine's notions of transparency and opacity.

Key words

Dynamic logic, logic of programs, semantics of programs, predicate calculus, modal logic, referential transparency.

Six Lectures on Dynamic Logic

Background

Logic is metamathematics, that is, its objects of study are the unremarked-on and unnamed expressions that are used to make remarks and name objects in ordinary mathematics. Typical expressions are "0", "3x+2", "x=y+2", "$(x+y)(x-y) = x^2-y^2$", "$\forall x \exists y[p(x,y) \vee q(y,x)]$". The logician collects a set L of such expressions, calls this set a *language*, and proceeds to study its meaning (*semantics*) and its manipulation (*axiomatics*).

Meaning is specified with the help of a *semantic domain* D; D might contain natural numbers, truth values, functions on the reals, predicates on polynomials, and so on. The connection between L and D is made with a meaning function or *interpretation* μ:L→D, assigning to each expression an object in the semantic domain.

If every expression in L had an agreed-on meaning in the above sense, a single (fixed) interpretation would suffice. Such a language would consist only of constants (that is to say, constant-value expressions), and would contain nothing worthy of the attention of a logician per se. If on the other hand every expression in L could have an arbitrary meaning, any element of L→D could serve as an interpretation, whence L would in effect consist only of variables and again would not be worth studying. What makes logic interesting is that the range of possible interpretations lies between these two extremes.

For a function from L to D to pass muster as an interpretation it must satisfy a collection of constraints. For example we might have the following constraints on μ.

$\mu(0)$ = the zero (unique additive identity) of D
$\mu(x+y)$ = $\mu(x)$ plus $\mu(y)$
$\mu(p \wedge q)$ = *true* if $\mu(p)$ = *true* and $\mu(q)$ = *true*
false otherwise

These constraints have a special form. We shall assume throughout this paper that every expression is of the form $\langle s,t \rangle$, consisting of a symbol s (the *operator*) together with an n-tuple $t = (t_1,\ldots,t_n)$ of expressions (the *arguments*). (Ordinary constants have a zero-tuple of arguments.) The most general form of the above constraints is:

$$\mu(\langle s,t \rangle) = F_s(\mu(t_1),\ldots,\mu(t_n))$$

That is, the meaning of the expression $\langle s,t \rangle$ is defined recursively as some function F_s of the meanings of the arguments, the function depending on s but not on μ. (If we want to have f(x) in L, meaning that f denotes some function (which one depending on μ) to be applied to x, we will write the application of f to x explicitly as $\gamma(f,x)$. The more usual convention of always applying μ to s as well as to the t_i's is less appropriate for our account of dynamic logic.) Constraints of this form we shall call *semantic constraints*.

The domain (in the functional sense) of F_S in the above is significant insofar as it acts as an additional constraint, namely on the possible values of the t_i's. For example, if there is a constraint for the expression $p \wedge q$ with $F_\wedge$ being conjunction, a function with domain {*true*,*false*}, then even if the expression p (or q) has no explicit constraint of its own, its possible values *in the expression* $p \wedge q$ are confined to the domain of $F_\wedge$, *true* and *false*.

The expression x is said to be *interpreted* when some semantic constraint $\mu(x) = \ldots$ applies to it, and otherwise is *uninterpreted*. We write L_0 for the uninterpreted subset of L. It should be apparent that if all expressions are finite (actually, well-founded is sufficient) then for *every* element of $L_0 \rightarrow D$ there is a *unique* extension of that element to an interpretation (i.e. to an element of $L \rightarrow D$ satisfying the semantic constraints). Since every interpretation uniquely determines an element of $L_0 \rightarrow D$, it follows that there is a one-to-one correspondence between the elements of $L_0 \rightarrow D$ and the interpretations. For this reason it is common in logic texts to call instead the elements of $L_0 \rightarrow D$ the interpretations. Our reluctance to follow this convention is due to the fickle nature of L_0, discussed below.

Clearly, what conventionally pass for variables will have to be uninterpreted expressions in this scheme of things. A little loosely perhaps, we will henceforth refer to *any* uninterpreted expression as a variable. (A little later we will discuss some consequences of this somewhat non-standard view of variables.)

Notice that, as defined above, a semantic constraint mentions only one interpretation, the one it is constraining. This is the defining characteristic of a *static* logic. If one uses the constraints to evaluate an expression recursively, the interpretation remains unchanged (static) as the evaluation proceeds; there are no side effects, so to speak. Quine refers to operators defined in this way as being *referentially transparent* [45]; what the operator's arguments refer to can be seen from "outside" the expression, i.e. the operator does not "block the view," so to speak.

The Limits of Static Logic

Consider the following expressions.

(1) $\forall x \exists y(x=y)$
(2) Necessarily $x+y=y+x$
(3) After setting x to 1, $x=1$

Each of these, we argue, involves concepts beyond the scope of static logic. The reason is that there is no function F such that the meaning of $\forall xp$ can be specified with a constraint of the form $\mu(\forall xp) = F(\mu(p))$, and similarly for the other constructs.

States

To give an account of these expressions we introduce the notion of *state*. One quite workable arrangement is to define a state to be an interpretation. However it will be slightly more convenient for us (and consistent with ordinary practice in modal logic) to postulate *a priori*, as part of D, a set of states $W = \{u,v,w,\ldots\}$, along with a function

$\pi:W\to(L\to D)$ which assigns to each state an interpretation. We shall frequently abbreviate $\pi(u)(e)$ to $u\models e$ (think of $\models$ as π on its side). When e is a formula $u\models e$ will be a truth value; this special case coincides with the conventional usage of $\models$. We define $\hat{\pi}:L\to(W\to D)$ as $\hat{\pi}(e)(u) = \pi(u)(e)$. Notice that π need not be an injection (1-1), that is, it is possible to have $u\models e = v\models e$ for all expressions $e\in L$ and still not have $u=v$; such pairs of states are *equivalent* but not *equal*.

Now the meaning of expression (1) above is that no matter what x is changed to y can then be changed so that x=y. Putting it more precisely, we take $u\models\forall x\exists y(x=y)$ to be *true* just when $v\models\exists y(x=y)$ is true for *every* v such that $u\models z = v\models z$ for all variables z other than x. In turn $v\models\exists y(x=y)$ is true just when $w\models x=y$ is true for *some* w such that $v\models z = w\models z$ for all variables z other than y.

We can put this a little more succinctly if we let R_x denote the binary relation on states such that uR_xv whenever $u\models z = v\models z$ for all variables z other than x. (Thus R_x is an equivalence relation on W.) Then

$$u\models\forall xp = \bigwedge_{uR_xv} v\models p \quad \text{(the conjunction of } v\models p \text{ for all v s.t. } uR_xv)$$

Similarly we may take $\exists xp$ to be defined thus.

$$u\models\exists xp = \bigvee_{uR_xv} v\models p.$$

The only difference is the use of $\vee$ in place of $\wedge$. Notice that $u\models\exists xp$ and $u\models\neg\forall x\neg p$ must be the same for all u in W, that is, these are *equivalent* expressions. Just as $\vee$ is the dual of $\wedge$, so is $\exists x$ the dual of $\forall x$.

The advantage of this way of looking at $\forall x$ and $\exists x$ is that it will also be how we shall look at concepts like "necessarily" and "after setting x to 1." This treatment of "necessarily" was first defined carefully by Kripke [25,26]. While Kripke (deliberately) did not propose any particular binary relation, let us consider the relation R such that uRv for all states u and v, the so-called complete binary relation on states. Then define "necessarily" as follows, writing "[]" for "necessarily".

$$u\models[]p = \bigwedge_{uRv} v\models p.$$

This says that p is necessarily true just when it is true in *all* states v, since R is complete. It follows that $u\models[]p$ is the same for all u's. This particular interpretation of "necessarily" is not implausible.

As with $\forall x$, "necessarily" has a dual, "possibly", written "<>". We have as before

$$u\models\langle\rangle p = \bigvee_{uRv} v\models p$$

We also have the equivalence of $\langle\rangle p$ with $\neg[]\neg p$.

Now consider our third example, "after setting x to 1, x=1." Even this formula can be fitted into the above framework. Let uRv hold just when uR_xv (as defined for example (1)) holds and $v \Vdash x = u \Vdash 1$. That is, x is set to 1, and there are no other effects (on L_0). Then

$$u \Vdash (\text{after setting x to 1, x=1}) = \bigwedge_{uRv} v \Vdash x=1$$

Of course the value of this is *true*, as with the preceding two examples. In this case it doesn't make any difference whether we write $\wedge$ or $\vee$, so that "after setting x to 1" is its own dual.

For notational convenience we abbreviate "set x to 1" as "x:=1", and, in imitation of the notation for "necessarily", we abbreviate "after x:=1" as "[x:=1]", which as we remarked is the same as its dual "$\langle$x:=1$\rangle$."

Form of constraints in DL

We would like to think of the above equations for quantifiers etc. as semantic constraints. To do so, however, we must abandon the requirement that a semantic constraint mention only a single state. In so doing we make the transition from static to dynamic logic.

The general form of a semantic constraint in dynamic logic is

$$\hat{\pi}(\langle s,t\rangle) = F_s(\hat{\pi}(t1),\ldots,\hat{\pi}(t_n)). \qquad \text{(Recall } \hat{\pi}(e)(u) = \pi(u)(e)\text{)}.$$

This is actually the same as the general form for static logic, with $\hat{\pi}$ in place of μ. The difference is that $\hat{\pi}$ is a dynamic meaning function; it yields the meaning of an expression *as a function of state.* In this framework, our original concept of a semantic constraint in static logic takes the form

$$u \Vdash \langle s,t\rangle = F_s(u \Vdash t_1,\ldots,u \Vdash t_n) \quad \text{for all u in W.}$$

We shall henceforth refer to this special form as a *static constraint*, and the more general one above as a *dynamic constraint.* From now on, as a notational matter, we use $u \Vdash$ in place of the (shortlived!) μ.

The general form permits $u \Vdash \langle s,t\rangle$ to depend on values of the t_i's in other states than u. It does not however permit it to depend arbitrarily on the t_i's themselves, which *are* evaluated, even if not in u. In this way, although we cannot substitute equals as we could in static logic (e.g. $u \Vdash (x=y \supset \varphi(x) \equiv \varphi(y))$ is no longer always true where φ is an arbitrary formula - consider $x=y \supset [](x=y) \equiv [](y=y)$), we *can* still substitute equivalents. That is, if $\hat{\pi}(a) = \hat{\pi}(b)$ then we do know that $u \Vdash (\varphi(a) = \varphi(b))$ is *true* for all u in W.

(For LISP afficionados: note that the extent to which dynamic logic is a step up from static logic is less than the extent to which FEXPR's in LISP are a step up from EXPR's. The additional power of a FEXPR over an EXPR is that the FEXPR can inspect the form of the

arguments, for example being able to distinguish $p \wedge q$ from $q \wedge p$, which is beyond the power of dynamic logic.)

Loose ends

We are now in a position to point out a peculiarity of our view of variables. When extra constraints are taken notice of (as might happen in the course of following an argument, when it becomes apparent that say propositional reasoning alone does not support the argument), certain expressions that hitherto were treated as variables now become constrained. A simple example would be the expression 0. As long as 0 remains uninterpreted it acts as a variable, and peculiar expressions such as $\exists 0(x=0)$ then have the same meaning as $\exists y(x=y)$. As soon as 0 is assigned a fixed interpretation, $\exists 0(x=0)$ means something else. According to the definition of $\exists y$ above, $\exists 0(x=0)$ would be equivalent to $x=0$ when 0 is interpreted.

It is a simple enough matter to include a syntactic condition on $\exists$, :=, and other variable-manipulating operators, so that only never-to-be-interpreted variables can be so manipulated. However such a condition would play no significant role in dynamic logic, and would require us to draw a distinction between the unconstrained expressions and variables. Thus we omit it from the theory in the interests of minimizing baggage.

For the remainder of this paper we adopt the convention of writing all expressions we want to be considered uninterpreted using single letters. Thus any occurrence of "x+y" is understood to be interpreted.

An operator definable in dynamic logic but not in static logic is said to be *referentially opaque*, again following Quine. Actually this is a somewhat more mathematically precise definition than Quine had to offer.

The examples of dynamic logic we have seen thus far, without getting into any depth, have already given some idea of the range of domains that can be served by dynamic logic: quantificational calculus, modal logic, and algorithmic logic (i.e. logic of programs, x:=1 being a program).

In this connection it may be worth remarking that the semantics of Lucid [1] are presented with emphasis on $\hat{\pi}$; in fact there is no global set of states in Lucid semantics, and instead each variable takes on values in a series of states defined essentially by the lifetime, or *extent*, of that variable.

The Kripke Operator

All the examples we have seen so far of dynamic logic constraints fit a much narrower description than the above, namely Kripke's semantics for modal logic. The reason for our rather general characterization of a dynamic constraint is that later we will want to deal with certain adverbial constructs that transcend the Kripke formula. For the time being however, we will stick with Kripke semantics.

It is natural to introduce the names of binary relations into the language L. We reserve a,b,c,... as variables for this purpose. We call such expressions *actions*.

It is also natural to take $u \Vdash a$ to be $\{v \mid uR_a v\}$, the set of states accessible from u via R_a. Then $u \Vdash [a]p$ can be defined as $(u \Vdash a) \Vdash p$, if we adopt the usual convention in logic that for a set $U \subseteq W$, $U \Vdash p$ means that $u \Vdash p$ for every u in U.

The notation [a]p, though almost universal in the dynamic logic literature, starting with [39,40], nevertheless obscures what we would like to call the *Kripke operator*. And later we will want to combine a's and p's in other ways, giving rise to other operators similar to the Kripke operator, each needing their own syntax. For these reasons we introduce the Kripke operator $\lrcorner$, and replace the notation [a]p with $a\lrcorner p$. The semantics remains unchanged:

$$u \Vdash (a\lrcorner p) \equiv (u \Vdash a) \Vdash p$$

For the syntax of the language $\{\neg \supset \lrcorner\}$ we adopt the convention that parentheses may be omitted from any of the following without changing the parsing. (We make use of this convention later, as new operators are introduced, to give a succinct account of the syntax of each new operator.)

$p \supset (q \supset r)$	(so $\supset$ is right associative)
$(\neg p) \supset q$	(so $\neg$ binds tighter than $\supset$)
$a\lrcorner(b\lrcorner p)$	(so $\lrcorner$ is right associative)
$(a\lrcorner p) \supset q$	(so $a\lrcorner$ behaves like $\neg$)
$(p \supset q) \supset r$	(so spaces count)

That part of dynamic logic confined to Kripke semantics can draw on a wealth of knowledge about modal logic. Of particular interest is the *theory* of this logic, the set of *valid* formulae, that is, formulae p such that $u \Vdash p$ is *true* for all states u, where π satisfies

$$u \Vdash \neg p \equiv u \not\Vdash p$$
$$u \Vdash (p \supset q) \equiv u \Vdash p \text{ implies } u \Vdash q$$
$$u \Vdash (a\lrcorner p) \equiv (u \Vdash a) \Vdash p$$

Two questions we shall ask about the set of valid formulae are: how hard is it to decide validity, and what useful axiomatizations do they have?

In the case when L contains only one action, that action being in L_0, the theory coincides with the theory of system K of modal logic (that is, just the modal operator [], with no restrictions on the binary relation corresponding to []). The complexity of the theory of K (along with several related systems) was shown by R. Ladner [28] to be log space complete in polynomial space. That is, first there is a computer program that will determine of a given formula of length n whether it is valid using $O(n^d)$ units of storage for some d. (This is to say that the theory of K is in polynomial space. Most reasonable notions of "units of storage" will suffice here.) This by itself is not very exciting, and so second (which is what makes the result much more interesting), one cannot do any better. That is, for *any* set in polynomial space there is a computer program which could determine whether an element of length n was in that set in only O(log n) units of space if only it had access to an "oracle" for the theory of K, a program that gives us at no charge answers to questions of membership in that theory. This is a very strong sense in which validity in K is as hard as any problem solvable in polynomial space.

Ladner's results extend without change to the case when L contains any number of actions, so long as they all belong to L_0. Shortly we shall see what happens when one introduces semantic constraints on actions.

The following will serve as a complete axiomatization of $\{\neg \supset \lfloor\}$.

K	$p \supset q \supset p$
S	$p \supset q \supset r \supset p \supset q \supset p \supset r$
N	$\neg p \supset \neg q \supset \neg p \supset q \supset p$
$\lfloor$	$a\lfloor(p \supset q) \supset a\lfloor p \supset a\lfloor q$

MP	From p, $p \supset q$ derive q	(Modus Ponens)
$\lfloor$Nec	From p derive $a\lfloor p$	(Necessitation)

(The names K and S come from combinatory logic. N is for Negation. Note the similarity between S and M; when a is p? (a test, see below), S and M coincide.)

Regular Dynamic Logic

It is not very interesting to consider just a set of unrelated actions with no visible internal structure. Hence we are inspired to introduce operations on actions. In so doing we shall be combining Tarski's calculus of binary relations [49] with Kripke's semantics for modal logic. This combination is without doubt the most interesting facet of that part of dynamic logic constructed around Kripke semantics. Even more interestingly, when we encounter analogues of the Kripke operator, the combination will become even more fruitful.

Propositional Dynamic Logic, PDL

We begin with four operators, ? ; U *, that together with $\neg \supset \lfloor$ give rise to the language Propositional Dynamic Logic (PDL).

Tests. Conditionals in a programming language are usually introduced with "if-then-else." However the rules of reasoning can be simplified by using a "smaller" notion of conditional, the test, which can be used in conjunction with the next two constructs to synthesize if-then-else. $x>0?$ is an instance of a test, as is $j=0 \vee p(j)=t(k)?$.

A test p? is constructed from a formula p of the logical language. The idea of a test is that a computation may proceed past a test just when that test evaluates to *true* in the current environment, otherwise the computation must block (which for our purposes is equivalent to going into an infinite loop). Formally:

$$u \Vdash p? \quad = \quad \begin{cases} \{u\} & \text{if } u \Vdash p \\ \{\} & \text{otherwise.} \end{cases}$$

Most of what we say holds even for tests containing $\lfloor$, permitting for example the side-effect-free programming construct "if p *would be* the result of running a then..."

The axiom for tests is

T $\quad p?\rfloor q \equiv p \supset q$

Composition. A familiar concept to programmers is that of executing one program after another; we may execute first a and then b. In terms of binary relations this means applying the first relation to a state to nondeterministically yield another state, and then applying the second relation to the resulting state. The *composition* of a and b, written a;b, is the relation describing the net effect of executing first a and then b. Formally:

$$u \models (a;b) = (u \models a) \models b$$

where $U \models b$, for $U \subseteq W$, is the union of the $u \models b$'s for each u in U.

The following axiom completely captures composition in dynamic logic.

C $\quad a;b\rfloor p \equiv a\rfloor b\rfloor p.$ $\quad$ (Syntax: $(a;b)\rfloor p$)

Union. Another concept slightly less familiar to programmers is that of having a choice of which action to carry out. The action aUb offers the choice of actions a or b. Formally:

$$u \models (a \cup b) = u \models a \cup u \models b.$$

Though U may be less familiar than ; it has a static definition, unlike ; . It is a nondeterministic concept; the closest deterministic programming concept is that of the conditional "if p then a else b" where a choice is given between a and b but in the same breath the criterion for making the choice, the formula p, is also given. In dynamic logic these two concepts of *choice* and *testing* are factored out, to simplify the domain of discourse and its attendant rules of reasoning. We can define "if p then a else b" in regular dynamic logic as $(p?;a) \cup (\neg p?;b)$.

The following axiom completely captures union in dynamic logic.

U $\quad a \cup b\rfloor p \equiv a\rfloor p \wedge b\rfloor p.$ $\quad$ (Syntax: $(a \cup b)\rfloor p$)

From these axioms we may infer that the validity problem for the language $\{\neg \supset \rfloor ; \cup\}$ is decidable - in fact in exponential space - simply by using the axioms for ; and U to eliminate all occurrences of ; and U from the input to yield a formula at most exponentially larger.

Iteration. In order to get a program to run for a substantial time some way of executing programs repeatedly is called for. The most elementary form of repetition is *iteration*, which in dynamic logic means execution of an action an arbitrary number of times. We write a^* (a-star) for the iteration of a. Formally

I $\quad u \models I = \{u\}$ $\quad$ (I is the identity action, needed for the next line)

R $\quad u \models a^* = u \models (I \cup a \cup a;a \cup a;a;a \cup \ldots)$

The closest deterministic programming construct to this is "while p do a," which executes the program a a number of times determined by the test p. Again we have reduced things to more fundamental concepts just as we did with if-then-else, this time separating while-do into iteration and testing. We can define "while p do a" as $(p?;a)^*;\neg p?$.

Axioms for iteration are not as easy to come by as for union and composition. In fact when we introduce assignment later we will not be able to get a complete axiomatization of iteration. Without assignment however, we can achieve an axiomatization of iteration as follows.

Refl: $[a^*]p \supset p$
Step: $[a^*]p \supset [a][a^*]p$
Ind: $p \wedge [a^*](p \supset [a]p) \supset [a^*]p$.

It is not at all apparent that these axioms generate all the valid formulae of PDL. The fact that they do was first announced (minus tests) in the Notices of the AMS by K. Segerberg [48]. Later (Jan. 1978) Segerberg found a lacuna in his proof, which he repaired some months after. Meanwhile R. Parikh [36] and the present author [42], working semi-independently, found completeness proofs. Also D. Gabbay [11] gave a sketch of a completeness proof, though much detail would appear to be necessary to convert this sketch into a convincing proof.

It is also not at all apparent that the theory of PDL is decidable; this was shown by M. Fischer and R. Ladner [8]. The proof uses a modal logic technique called filtration to show that a satisfiable formula of PDL has a finite model, whence satisfiability and validity can be determined by a finite search for a model. Normally filtration proofs are straightforward, but in the case of PDL a minor difficulty arises with ;. Fischer and Ladner were able to show that the theory of PDL is in NTIME(2^n) (nondeterministic Turing machine exponential time), but not in DTIME(c^n) for some $c>1$. Given our present ignorance about the extent to which nondeterminism helps in improving running time, this is about as tight a bound as we can hope for.

First Order Regular Dynamic Logic

The transition to any first order logic is made when terms are introduced into the language L. A term denotes an arbitrary domain element, not merely a truth value as in the case of a formula, or a set of states as in the case of an action.

Random Assignment. A random assignment is an action x:=? where x is any expression and ":=?" is the random assignment operator. It is defined by

$$u \models x:=? = \{v \mid u R_x v\} = \{u \models z = v \models z \text{ for all variables } z \text{ other than } x\}$$

Note that since random assignments involve the notion of variable, changing the semantic constraints may affect the meaning of x:=?.

The main role for x:=? is for defining quantifiers. $\forall x$ is just $x:=?\rfloor$. The following two axioms for random assignment are, in the absence of other actions, just enough to completely axiomatize first order predicate calculus.

A1 $\quad p \supset \forall x p$ when x does not occur free in p
A2 $\quad \forall x p(x) \supset p(e)$ for any expression e

The definition of "occurs free in" is as follows. For e in L_0, x occurs free in e just when e is x. x occurs free in a;b ($a\rfloor p$) just when x occurs free in a or $\langle x$ is not bound in a and x occurs free in b $(p)\rangle$. x occurs free in any other expression when x occurs free in one or more of its arguments. (Intuitively, x occurs free in $u\models e$ when there is a chance that the value of e might "depend on" the value of x in u.) x is *bound* in a when a is x:=? or x:=e; when a is b;c and x is bound in b or c; or when a is bUc and x is bound in b and c. (Intuitively, x is bound in a when it is guaranteed that a assigns some value to x.)

In this paper we shall forbid random assignment to action expressions.

Assignment. An assignment is a pair of expressions x:=e. The idea is that an assignment is the action of changing the state so as to make the value of x in the new state that of e in the old. Thus the corresponding binary relation consists of those pairs u,v such that uR_xv and $v\models x = u\models e$. (Recall that R_x was the binary relation corresponding to $\forall x$ and consisting of all pairs u,v such that $u\models z = v\models z$ for all variables z other than x.) So we have

$$u\models x:=e = \{v \mid uR_xv \text{ and } v\models x = u\models e\}$$

There is no axiom for assignments as satisfactory as the axioms we have been encountering for other constructs. If p(x) is a formula involving some "x-e-visible" occurrences of x (an x-e-visible occurrence has only operators "above" it in the expression that are referentially transparent to x and e, i.e. clearly don't change x or e), then the following axiom is adequate.

Ass: $\quad x:=e\rfloor p(x) \equiv p(e)$ (Hoare [20])

As we have thus far only constrained formulae and programs, the only assignments whose effects can be felt thus far are assignments to formulae and programs. We shall forbid the latter kind entirely. Grabowski [12] has shown that algorithmic logic with this construct has a decidable validity problem. Extending this result to dynamic logic poses no insurmountable obstacles.

If we include = in L, with its standard interpretation on whatever domain takes our fancy, matters become more complex. It is now possible for information about the values of non-formulae to propagate up to the formula level; for example, we may now deduce the validity of $x:=y\rfloor x=y$ for variables x and y.

Problem: Determine whether validity is decidable for $\{\neg \supset \rfloor \cup ; * =\}$; for $\{\neg \supset \rfloor \cup ; * = :=\}$.

Including application, γ, (with the condition that γ's first argument be a free variable, i.e. not one occurring inside an expression which contains assignments to that

variable or variable actions) gives us first order predicate calculus with uninterpreted function symbols. We define γ_n (application for n-ary functions) thus.

$$u\models\gamma_n(f,x_1,\ldots,x_n) = (u\models f)(u\models x_1,\ldots,u\models x_n)$$

It was shown in [40] that the theory of what we may take to be $\{\neg \supset \rfloor * \forall := = \gamma\}$ was not recursively enumerable (r.e.), even if attention was restricted to formulae of the form $p\supset(x:=\gamma(f,x))^*\rfloor q$ where p and q were $\rfloor$-free. In [13] this result was strengthened to show that that fragment of the theory was Π^0_2-complete (cf [46]). A. Meyer has shown that the whole theory is Π^1_1-complete (again cf [46]). All these results indicate very definitely that a complete axiomatization of dynamic logic at this level of richness is out of the question.

Applications to Program Verification

Program verification is the art of showing that a program meets its specifications using formal logic. There is no doubt that Hoare's p{a}q construct [20] is of considerable interest to program verification. Since we can embed p{a}q in dynamic logic, as $p\supset a\rfloor q$, it follows that at least that fragment of dynamic logic is relevant to program verification. However, if *total* correctness is to be established, program verification also needs to deal with the problem of termination, which is not expressible using the p{a}q construct. Because of the ability to negate *all* formulae in dynamic logic, termination can be represented with no language extensions or informal arguments.

To test the extent to which dynamic logic could help in program verification, the author, with S. Litvintchouk, implemented a proof checker for dynamic logic proofs [31]. Thus far the largest program we have demonstrated the total correctness of is the Knuth-Morris-Pratt pattern-matching algorithm [22].

One interesting aspect of our perspective on DL is the decomposition of quantifiers such as $\forall x$ into random assignment and the Kripke operator. A result of this is that less code is needed to cope explicitly with quantification, since half of what is know about quantification is actually general knowledge about arbitrary programs. This general knowledge is subsumed under axiom M and rule Nec. The axioms specific to quantification itself are then A1 and A2, which are so like the axioms for reasoning about assignment that only a small amount of additional code is needed to deal explicitly with quantifiers.

This situation should be contrasted with the usual approach to program verification, which is a two-stage affair in which verification conditions are generated and then sent to a theorem prover. Knowledge about programs in general and assignments in particular is kept in the verification condition generator, completely separate from knowledge about quantifiers, which is kept in the theorem prover.

Another point is that one does not always want to generate all verification conditions before starting to work on the sort of logical manipulation done by the theorem prover. Consider for example the two programs

a: while p do b
aa: while p do (b;b)

where p might be "x⩽eps+y" and b might be (x:=y-del×x; y:=x+del×y). Now it turns out that regardless of what p and b are, the termination of aa implies the termination of a, a fact expressible in DL as "¬aa⌋*false* ⊃ ¬a⌋*false*." Yet if this statement, with p and b spelled out in full, is given to a two-stage system (even assuming it could handle things like having two ⌋'s in the problem), it will think hard about the assignments in b before getting to the logic. In a system that works top down (i.e. starting at the "top" of the formula to be proved, a characteristic of natural deduction systems for one), the validity of the above claim can become apparent even before any assignments are contemplated.

Applications to Natural Language

Consider the following sentences.

(1) Whether you strike a match or operate a cigarette lighter you get a flame.

This may be formalized as MUC⌋F, where M stands for the proposition that you have struck a match, C that you have operated a cigarette lighter, and F that you have a flame.

(2) If you strike a match you get a flame, and if you operate a cigarette lighter you get a flame.

Similarly this amounts to M⌋F∧C⌋F. The intuitive equivalence between (1) and (2) is formalized (and therefore subject to automatic verification) by the assertion MUC⌋F ≡ M⌋F∧C⌋F.

(3) When you open the door and walk through it you enter the room.

(4) When you open the door then when you walk through it you enter the room.

The equivalence of (3) and (4) is summarized in O;W⌋E ≡ O⌋W⌋E. Notice that we do not get as equivalent

(5) When you walk through the door and open it you enter the room.

or

(6) When you walk through the door, then when you open it you enter the room.

even though (5) and (6) are equivalent to each other, W;O⌋E ≡ W⌋O⌋E. If we were to try to capture the meaning of 3 or 4 using the propositional calculus alone, we might end up with O∧W⊃E ≡ O⊃E⊃W, which is certainly valid. Unfortunately O∧W⊃E is equivalent to W∧O⊃E, which reveals the limitation of propositional calculus for reasoning about action in this way.

(7) If your TV won't work and you kick it it still won't work.

(8) If your TV won't work then no matter how many times you kick it it still won't work.

If 7 is true in all circumstances then 8 ought also to be true in all circumstances. This amounts to the soundness of the rule, from $W \supset K\rfloor W$ derive $W \supset K^*W$. This rule can be derived by starting with $W \supset K\rfloor W$, applying Necessitation to get $K^*(W \supset K\rfloor W)$, then applying Modus Ponens to it and the induction axiom (reformulated slightly using propositional reasoning to read $K^*\rfloor(W \supset K\rfloor W) \supset W \supset K^*\rfloor W$) to get $W \supset K^*\rfloor W$ as desired.

Reasoning About Processes

So far the Kripke operator $\rfloor$ has been our only operator relating actions to formulae. We now introduce some other operators that, like $\rfloor$, find application to both algorithmic logic and to natural language reasoning. A price we must pay for these operators is the redefinition of the meaning of actions, which as defined so far do not contain enough information.

So far we have taken $u \Vdash a$ to be the set of states that a might halt in when started in state u. We now take it to be the set of *sequences* of states that a goes through, starting from u. We let s,t,... range over sequences. Sequences can be viewed as functions from an initial segment of ordinals to states. In the event that a runs forever, the sequence will be infinite. If a is blocked by a test that evaluates to *false*, the limbo state $\Lambda \in W$ is entered. Sequences always have a final state, called s_f, whence infinite sequences need a limit element, indexed by the ordinal ω, which will always be Λ. Λ may not appear as a non-final state of a sequence.

The distinguished state Λ has a special behavior as regards formulae; $\Lambda \Vdash p$ is *true* for all formulae. For actions, $\Lambda \Vdash a$ is $\{(\Lambda)\}$ for all actions. Semantic constraints of the form $u \Vdash e = \ldots$ do not include the case $u = \Lambda$.

We also insist that $u \Vdash a$ never be the empty set, for any action a, interpreted or not, even if this means taking $u \Vdash a$ to be $\{(u,\Lambda)\}$.

With this notion of an action it becomes possible to define the new operators. But first we should adjust the definition of $\rfloor$ so that it retains its meaning.

$$u \Vdash a\rfloor p = \bigwedge_{s \in u \Vdash a} s_f \Vdash p$$

The next operator is $\Box\!\!\Box$, as in $a\Box\!\!\Box p$, pronounced "a maintains p." The idea is that $u \Vdash a\Box\!\!\Box p$ is *true* just when $v \Vdash p$ is *true* in every state v of every sequence of $u \Vdash a$. Formally:

$$u \Vdash a\Box\!\!\Box p = \bigwedge_{s \in u \Vdash a} \bigwedge_{v \in s} v \Vdash p$$

(We loosely write $v \in s$ to mean $v = s_i$ for some element s_i of s.)

The following completely axiomatizes $\{\neg \supset \mathbb{W}\}$, if we take S, K, N and MP.

$\mathbb{W}_1$ $\quad a\mathbb{W}(p \supset q) \supset a\mathbb{W}p \supset a\mathbb{W}q$

$\mathbb{W}_2$ $\quad a\mathbb{W}p \supset p$

$\mathbb{W}_3$ $\quad$ From p derive $a\mathbb{W}p$

The proof of completeness may be found in either of [42] or [44].

The third operator is $\perp$, as in $a\perp p$, pronounced "a promises p." Here $u \models a\mathbb{W}p$ is *true* just when $v \models p$ is *true* in some state v of every sequence of $u \models a$. Formally:

$$u \models a\perp p \;=\; \bigwedge_{s \in u \models a} \; \bigvee_{v \in s} v \models p$$

Notice that in any sequence ending in Λ, everything is "promised." The idea here is that if a sequence ends in limbo you aren't supposed to care, just as for $\perp$. This is important for programs where iteration is implemented using * and tests. Careful inspection of the possible sequences reveals many ending in Λ that we would not want to compromise the intuitive notion of "promises" in conjunction with while loops.

The following completely axiomatizes $\{\neg \supset \perp\}$.

$\perp_1$ $\quad p \supset a\perp p$

$\perp_2$ $\quad$ From $p \supset q$ infer $a\perp p \supset a\perp q$

The proof of completeness may be found in [44].

Finally we have $\int$, as in $a\int p$, pronounced "a preserves p." Here $u \models a\int p$ is *true* just when if $v \models p$ is *true* for any state v in $s \in u \models a$ then $w \models p$ is *true* for all states w in s after v. Formally:

$$u \models a\int p \;=\; \bigwedge_{s \in u \models p} \; \bigwedge_{v \leqslant w \in s} v \models p \supset w \models p$$

where "$v \leqslant w \in s$" means that $v = s_i$, $w = s_j$, with $i \leqslant j$. The axiomatization is somewhat more complex; again see [44]:

$\int_1$ $\quad p \supset a\int p \supset a\int(p \supset q) \supset a\int q$

$\int_2$ $\quad p \equiv q \supset a\int(p \equiv q) \supset a\int p \supset a\int q$

$\int_3$ $\quad p \supset a\int p \supset a\int \neg p$

$\int_4$ $\quad a\int p \supset a\int q \supset a\int(p \wedge q)$

$\int_5$ $\quad a\int p \supset a\int q \supset a\int(p \vee q)$

$\int_6$ $\quad$ From p infer $a\int p$

So far we have considered just the languages $\{\neg \supset \times\}$ for various operators $\times$. When these are combined to form $\{\neg \supset \rfloor \; \mathbb{W} \; \perp \; \int\}$ we need some additional axioms.

$a[\![]p \equiv p \wedge a\lfloor p$ (suggests taking $[\![]$ as abbreviation only)
$a\lfloor p \supset a\bot p \supset a\_p$
$\neg(a[\![]p \wedge a\bot\neg p)$ (depends on fact that $u \models a$ is never empty)

None of this deals with operations on actions. The definitions of ∪ and * need not change. We do however require definitions for ? ; and :=. First let us define the operation ; on individual sequences, as

$(s;t)_i = s_i$ where s_i is defined
$(s;t)_{S+i} = t_i$ where S is the length of s and t_i is defined.

Then the definitions of the action operators are:

$u \models I = \{(u)\}$
$u \models p? = \{(u)\}$ if $u \models p$
$\{(u,\Lambda)\}$ otherwise
$u \models a;b = \{s;t \mid s \in u \models a,\ t \in u \models b\}$
$u \models x:=e = \{(u,v) \mid uR_x v$ and $v \models x = u \models e\}$

The following axiomatize $\lfloor$ in conjunction with these. (With the axiom $a[\![]p \equiv p \wedge a\lfloor p$ it becomes unnecessary to give further axioms for $[\![]$.)

$p?\lfloor q$
$a \cup b\lfloor p \equiv a\lfloor p \wedge b \wedge p$
$a;b\lfloor p \equiv a\lfloor p \wedge a\_b\lfloor p$
$a^*\lfloor p \equiv a^*\_a\lfloor p$

This leaves open the problem of axiomatizing $\bot$ with the action operators. A little reflection shows that while ∪ and * can be axiomatized ($a^*\bot p \equiv p$), ; cannot be axiomatized with a single equivalence in any obvious way. To get around this we introduce a new operator, $[\![]$, as in $a[\![]p,q$, which takes two arguments p and q. It is defined thus.

$u \models a[\![]p,q = \forall s \in u \models a(\exists i(s_i \models p) \vee s_f \models q)$.

This says that for every sequence s in $u \models a$, either p holds in some state of s or q holds in the final state of s (including the case when $s_f = \Lambda$, which satisfies both p and q). This rather odd construct has the properties that ; can be axiomatized with it, and both $\_$ and $\bot$ can be treated as abbreviations, thus:

$a\_p \equiv a\_\mathit{false},p$
$a\bot p \equiv a\_p,\mathit{false}$.

The following axiomatize $[\![]$ in conjunction with $\lfloor$, treating $\_$, $[\![]$ as mere abbreviations.

$[\![]1$ $a[\![]p,(q \supset r) \supset (a[\![]p,q \supset a[\![]p,r)$
$[\![]2$ $a[\![]p,\neg p$
$[\![]3$ $p \supset a[\![]p,q$
$[\![]4$ from $p \supset q$ derive $a[\![]p,r \supset a[\![]q,r$

$\Vdash 5$ $\quad a\lceil p \supset (a\perp p \supset a\lfloor p)$

$\Vdash 6$ $\quad \neg(a\Vdash p \wedge a\perp\neg p)$

$\Vdash 7$ $\quad (aUb)\Vdash p,q \equiv a\Vdash p,q \wedge b\Vdash p,q$

$\Vdash 8$ $\quad (a;b)\Vdash p,q \equiv a\Vdash p,(b\Vdash p,q)$

$\Vdash 9$ $\quad a^*\Vdash p,q \supset p \vee q$

$\Vdash 10$ $\quad a^*\Vdash p,q \supset a\Vdash p,(a^*\Vdash p,q)$

$\Vdash 11$ $\quad a^*\Vdash p,(q \supset a\Vdash p,q) \supset (q \supset a^*\Vdash p,q)$ (Harel induction)

$\Vdash 12$ $\quad p?\Vdash q,r \equiv q \vee (p \supset r)$

Applications of process logic to algorithmic logics

The $\Vdash$ operator is perhaps the simplest operator one might wish to apply to a program that was designed to run forever (e.g. an operating system, or an interactive editor), for which the $\lfloor$ operator is worthless.

The $\perp$ operator is relevant to the issues of fairness and starvation, concepts that arise occasionally in the literature on verification of operating systems. If we view the scheduler as a nondeterministic program (and even if we assume that the operating system is a deterministic mechanism we cannot really work that determinacy into our proofs in practice), then we would like to be able to say of the system as a whole, nondeterminism and all, that there will come a time when a certain state of affairs (e.g. such-and-such a process getting service) will hold.

The $\lceil$ operator arises naturally in talking about a system that only manages to keep p true throughout its execution by assuming it is true to begin with and depending throughout on its staying true. This idea is embodied in the axiom $p \wedge a\lceil p \supset a\Vdash p$. The $\lceil$ operator is used implicitly by Owicki in her thesis [35].

Applications of process logic to natural language

We give a further series of examples of natural language formulae embodying arguments formalizable within dynamic logic and using other operators besides the Kripke operator.

(1) While stacking up blocks, if the box becomes empty it will remain empty for the duration of the stacking process.

(2) Sometime during the stacking of blocks the box is guaranteed to be empty.

(3) When you stack up blocks you end up with the box being empty.

It is apparent that if both 1 and 2 are the case then so is 3, as can be seen from the valid formula $S\lceil E \wedge S\perp E \supset S\lfloor E$.

(4) If a defect appears in the wall while laying bricks the defect will stay there for the rest of the bricklaying.

(5) After laying any number of bricks, if a defect is found in the wall while laying the next brick the defect will stay there for the rest of the laying of that brick.

With a little thought it can be seen that 4 and 5 are equivalent, as formalized by $L^*\lceil D \equiv L^*\rfloor L\lceil P$.

Pointers to the dynamic logic literature

The earliest formal dynamic logic was Frege's quantificational calculus [10]. The idea of viewing quantifiers in terms of a relation R_X, thereby making the connection between modal logic and the quantificational calculus and so permitting the latter to viewed as a dynamic logic, seems not to have been made until [40]. Modal logic as discussed here is due to Lewis [29]. The semantics we are using is due principally to Kripke [25], who also contributed to issues of decidability in [26]. An excellent reference work on modal logic is [21].

Following Engeler [7] and the Polish school of algorithmic logic [4,47], we shall call a dynamic logic whose actions are deterministic programs, described either by flowcharts or if's and while's, an *algorithmic logic*. The earliest work on proving programs correct [50,51] amounted to informal algorithmic logics for flowchart programs. In the early sixties J. McCarthy [32] proposed the use of a static approach to program correctness by programming with recursively defined functions, thereby avoiding the problem of reasoning about states.

In 1967 Floyd [9] described in detail for the first time an algorithmic logic, built around flowcharts as with [50,51]. In 1969 Hoare [20] described a more conventional algorithmic logic oriented towards textual programs using if's and while's. Hoare's logic introduced the notion of a partial correctness assertion $p\{a\}q$ as an expression having a status different from that of an ordinary formula, in particular not being subject to Boolean operations. Though Hoare gave only an informal semantics for $p\{a\}q$, it seems beyond debate that he meant it to have the semantics of $\vdash p \supset a\rfloor q$. In 1970 Salwicki developed a similar algorithmic logic (and applied Engeler's term algorithmic logic [7] to it). The most striking difference from Hoare's logic was that all of Salwicki's formulae were subject to Boolean operations; as such, Salwicki's logic is the first true algorithmic logic. It may be characterized as dynamic logic using $\rfloor$, if, while, and having functional rather than relational actions, as behoves a deterministic programming language. (Engeler's algorithmic logic [7] is rather weaker, permitting in effect only *false* as the second argument to $\rfloor$.) Salwicki's work prompted a veritable flood of papers from Warsaw on algorithmic logic, mostly by members of H. Rasiowa's group; a comprehensive survey of work up to 1974, including a bibliography of some 40 papers on algorithmic logic, may be found in [4].

The idea of modelling programs with binary relations, taking advantage of Tarski's calculus of U ; * [49], goes at least as far back as Eilenberg and Elgot [6]. De Bakker [2], with de Roever [3], developed the idea considerably further, adding a fixed point operator to Tarski's calculus to model recursion. Independently of de Bakker, but motivated by Eilenberg and Elgot, D. Park [19], with P. Hitchcock, also used the fixed point operator in a relational treatment of flowchart programs.

The combination of modal logic and Tarski's operators was first developed by the author [39] in response to a suggestion of R. Moore, a student in the author's program semantics course. It was brought to the attention of a wider audience in [40] some two and a half years later, in a paper that prompted several people, including M. Fischer, D. Harel, R. Ladner and A. Meyer, to work on dynamic logic. This gave rise to a paper by Harel, Meyer and the author [13] on the complexity of the theory of first-order dynamic logic, along with a relative-completeness proof of the axiomatization given in [40], and another paper by Fischer and Ladner [8] on the validity problem for PDL, including not only the result that it was decidable but giving good bounds on the complexity of the problem. A little later, Harel and the author reported on work on Dijkstra's notion of total correctness ("*weakest precondition*"), proposing definitions for the concepts Dijkstra was attempting to define via axioms, and giving a relatively complete axiom system for Dijkstra's language [14].

At about the same time several people began asking questions about definability in dynamic logic. A. Meyer addressed the question of whether DL^+, the language defined in [14] in part to formalize Dijkstra's language, was expressible in regular first-order dynamic logic. This problem turned out to have a very elusive answer. Meyer was able to show that DL^+ was no more expressive than DL provided the programming language permitted array assignments [33]. Later Winklmann [52] was able to obtain the same result without requiring array assignments, but using $\bot$ within tests. Eventually he was able to eliminate $\bot$ from tests [53]. Meanwhile the author showed that PDL^+ is strictly stronger than PDL, complementing [53].

F. Berman and M. Paterson, in a remarkably delicate argument, showed that PDL was strictly strengthened by the inclusion of tests [5]. Meyer and Parikh showed that regular first-order DL with $\bot$-free tests was strictly weaker than constructive $L\omega_1\omega$ [34].

D. Harel developed further the relative completeness ideas of [13], drawing a distinction between relative completeness and arithmetic completeness. Using a result of Lipton [30], Harel showed in essence that arithmetic completeness is all that one wants.

The question of finding a complete axiomatization for PDL was raised in [8]. There is an account earlier in this paper of the origins of [48,36,42,11] as answers to this question.

A very thorough and detailed treatment of Harel's many contributions to DL may be found in his thesis [18]. In addition Harel has authored a close-to-exhaustive survey of logics of deterministic programs, using the $[a]p/\langle a\rangle p$ notation as a lingua franca in order to make it easier to see the similarities and differences between the various logics.

Motivated by the concept of a Boolean algebra underlying propositional calculus, Harel asks the question, what is the appropriate algebra for PDL? A partial answer to this may be found in [17].

The author, with S. Litvintchouk, explored the question of how to implement a proof checker for DL. A proposal for such a system is described in [31]. Some of the techniques used in the first implementation of the system are alluded to briefly in [41]. The system has been operational since August 1977, when it was able to check a 20-theorem proof of the total correctness of the Knuth-Morris-Pratt pattern-matching algorithm, taking 45 seconds to do so.

In the process of making the system more automatic, some theoretical questions about deciding validity in PDL arose, giving rise to an algorithm described in [42] that is more "practical" than the algorithm of [8]. This algorithm has been very recently further improved by Pratt to require time one exponential in the worst case [43]. To within a polynomial, this meets the lower bound of one exponential given in [8].

In [42] the issue of discussing programs intended not to halt is raised. Several constructs are proposed, namely those discussed above in the section on logics of processes. The author has recently been able to show completeness of various axiom systems for some of those constructs [44]. The semantics for processes is intimately related to Pnueli's semantics for temporal logic [38]. Parikh has shown, using Rabin's remarkable decidability result for the weak second order theory of n successors, that a very much stronger language has a decidable theory, although unlike the theory of [42], it is not elementary recursive.

This survey, combined with the pointers in [4] and [18], covers the bulk of what is known about dynamic logic. Missing will be the Polish work from 1974-1978, and most of the work to date on classical modal logic, much of which however is subsumed by the recent DL work (see, e.g., the last page of [8] to see how one DL result can be translated into several modal logic results). Also missing is the bulk of a century's work on various quantificational calculi, for which we can only point the reader at the tremendous volume of logic literature that has been accumulating.

The Interest in Nondeterministic Programs

The following is taken from [14], and may be of interest to those wondering why someone interested in deterministic algorithmic logic would want to get involved in the greater generality of dynamic logic and Tarski's relational calculus.

First, nondeterministic programs have been proposed as a model of parallel processes. Such parallelism arises in timeshared computers, where nondeterminism expresses the apparent capriciousness of the scheduler. It also arises in the management of external physical devices, where the nondeterminism captures the unpredictable behavior of physical devices.

Second, nondeterminism is gaining credence as a component of a programming style that imposes the fewest constraints on the processor executing the program. For example a certain program may run correctly provided that initially x is even. If the programmer requires the processor to set x to an even number of the programmer's choosing, the processor may be unduly constrained. On a byte oriented machine where integers are represented as four-byte quantities, setting x to a particular number requires four operations, but if the programmer has merely requested setting it to an arbitrary even number the processor can satisfy the request with one operation, by setting the low-order byte to, say, zero.

Third, nondeterminism supplies one methodology for interfacing two procedures that, though written independently, are intended to cooperate on solving a single problem. The approach is to make one procedure an "intelligent" interpreter for the other. Wood's Augmented Transition Networks supply an instance of the style. The user of this system writes a grammar for a specific natural language which amounts to a nondeterministic program to be run on Wood's interpreter, which though ignorant of the details of specific languages

nevertheless contributes much domain-independent parsing knowledge to the problem of making choices left unspecified by the user's program. This technique is in wide use in other areas of Artificial Intelligence, and supplies a way of viewing such AI programming languages as QA-3, PLANNER, and a number of more recent languages.

Fourth, from a strictly mathematical viewpoint, there is something dissatisfying about taking such constructs as if-then-else and while-do as primitive constructs. If-the-else involves the two concepts of *testing* and *choosing*, and while-do involves the two concepts *testing* and *iterating*. A more basic approach is to develop these concepts separately. However, in isolating the concept of testing from the concepts of choosing and iterating, we have removed the parts of the if-then-else and while-do constructs responsible for their determinism.

Fifth, from a practical point of view, when reasoning about deterministic programs it can sometimes be convenient to make what amounts to claims about nondeterministic programs. When we argue that "if x>0 then x:=x-1 else x:=x+1" cannot affect y, a part of our argument might be that, whether we execute x:=x-1 or x:=x+1, y will not change. The fact that the whole program is deterministic played no role in this argument, which amounts to the observation that the nondeterministic program x:=x-1Ux:=x+1 cannot change y. (aUb is a program calling for execution of either program a or program b, the choice being made arbitrarily, i.e. nondeterministically.) By the same token the observation that "while x<0 do x:=x+2" leaves the parity of x unchanged depends principally on the fact that executing x:=x+2 arbitrarily often, i.e. executing (x:=x+2)*, leaves the parity of x unchanged. (a* is a program calling for a number of executions of program a, the choice of number being made nondeterministically.) This illustrates the appropriateness of applying nondeterministic reasoning to deterministic programs.

Bibliography

[1] Ashcroft, E.A. and W.W. Wadge. Lucid, a Nonprocedural Language with Iteration. Comm. ACM, *20*, 7, 519-526. July 1977.

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

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

[4] Banachowski, L., A. Kreczmar, G. Mirkowska, H. Rasiowa, A. Salwicki. An Introduction to Algorithmic Logic; Metamathematical Investigations in the Theory of Programs . In Math. Found. of Comp. Sc. (eds. Mazurkiewicz and Pawlak), Banach Center Publications, Warsaw. 1977.

[5] Berman, F. and M. Paterson. Test-Free Propositional Dynamic Logic is Strictly Weaker than PDL. T.R. 77-10-02, Dept. of Computer Science, Univ. of Washington, Seattle. Nov. 1977.

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

[7] Engeler, E. Algorithmic properties of structures. Math. Sys. Thy. *1*, 183-195. 1967.

[8] Fischer, M.J. and R.E. Ladner. Propositional Modal Logic of Programs. Proc. 9th Ann. ACM Symp. on Theory of Computing, 286-294, Boulder, Col., May 1977.

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

[10] Frege, G. *Begriffsschrift.* Halle, 1879.

[11] Gabbay, D. Axiomatizations of Logics of Programs. Manuscript, under cover dated Nov. 1977.

[12] Grabowski, M. The Set of Tautologies of Zero-order Algorithmic Logic is Decidable. Bull. Acad. Pol. Sci., Ser. Math. Astr. Phys., *20*, 575-582. 1972.

[13] Harel, D., A.R. Meyer and V.R. Pratt. Computability and Completeness in Logics of Programs. Proc. 9th Ann. ACM Symp. on Theory of Computing, 261-268, Boulder, Col., May 1977.

[14] Harel, D. and V.R. Pratt. Nondeterminism in Logics of Programs. Proc. 5th Ann. ACM Symp. on Principles of Programming Languages, 203-213, Tucson, Arizona, Jan. 1978.

[15] Harel, D. Arithmetical Completeness in Logics of Programs, Proceedings of the 5th International Colloq. on Automata, Languages and Programming, Udine, Italy, July, 1978.

[16] Harel, D. On the Correctness of Regular Deterministic Programs; A Unified Survey. Submitted for publication.

[17] Harel, D. Relational Logic, Internal report, MIT, 1977.

[18] Harel, D. Logics of Programs: Axiomatics and Descriptive Power. Ph.D. thesis, Dept. of EECS, MIT, May 1978.

[19] Hitchcock, P. and D. Park. Induction Rules and Termination Proofs. In *Automata, Languages and Programming* (ed. Nivat, M.), IRIA. North-Holland, 1973.

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

[21] Hughes, G.E. and M.J. Cresswell. *An Introduction to Modal Logic.* London: Methuen and Co. Ltd. 1972.

[22] Knuth, D.E., J.H. Morris and V.R. Pratt. Fast Pattern Matching in Strings. SIAM J. on Computing, *6*, 2, 323-350. June 1977.

[23] Kreczmar, A. The set of all tautologies of algorithmic logic is hyperarithmetical. Bull. Acad. Pol. Sci., Ser. Sci. Math. Astr. Phys. Vol. 19. 781-783. 1971.

[24] Kreczmar, A. Degree of recursive unsolvability of algorithmic logic. Bull. Acad. Pol. Sci., Ser. Sci. Math. Astr. Phys. Vol. 20. 615-617. 1972.

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

[26] Kripke, S. A. Semantical analysis of modal logic I: normal modal propositional calculi. Zeitschr. f. Math. Logik und Grundlagen d. Math., *9*, 67-96. 1963.

[27] 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.

[28] Ladner, R. The Computational Complexity of Provability in Systems of Modal Propositional Logic. SIAM J. on Computing, *6*, 3, 467-480. Sept. 1977.

[29] Lewis, C.I. *A Survey of Symbolic Logic.* Berkeley, 1918.

[30] Lipton, R.J. A Necessary and Sufficient Condition for the Existence of Hoare Logics. 18th IEEE Symp. on Foundations of Computer Science, Providence, R.I. Oct. 1977.

[31] Litvintchouk, S.D. and V.R. Pratt. A Proof-checker for Dynamic Logic. Proc. 5th Int. Joint Conf. on AI, 552-558, Boston, Aug. 1977.

[32] McCarthy, J. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, 33-70 (eds. P. Braffort and D. Hirschberg), North Holland, Amsterdam. 1963.

[33] Meyer, A.R. Equivalence of DL, DL+ and ADL for Regular Programs with Array Assignments. Unpublished report, MIT. August 1977.

[34] Meyer, A.R. and R. Parikh. Definability in Dynamic Logic. Talk given at NSF-CBMS Research Conference on the Logic of Computer Programming, Troy, N.Y., May 1978.

[35] Owicki, S. A consistent and complete deductive system for the verification of parallel programs. Proc. 8th Ann. ACM Symp. on Theory of Computing, 73-86. Hershey PA. May 1976.

[36] Parikh, R. A Completeness Result for PDL. Symposium on Mathematical Foundations of Computer Science, Zakopane, Warsaw, Sept. 1978.

[37] Parikh, R. Second Order Process Logic. 19th IEEE Symposium on Foundations of Computer Science. Oct. 1978.

[38] Pnueli, A. The Temporal Logic of Programs. 18th IEEE Symposium on Foundations of Computer Science, 46-57. Oct. 1977.

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

[40] Pratt, V.R. Semantical Considerations on Floyd-Hoare Logic. Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci., 109-121. 1976.

[41] Pratt, V.R. Two Easy Theories Whose Combination is Hard. Internal Report, MIT LCS, Sept. 1977.

[42] Pratt, V.R. A Practical Decision Method for Propositional Dynamic Logic. Proc. 10th Ann. ACM Symp. on Theory of Computing, San Diego, May 1978.

[43] Pratt, V. R. A Near-Optimal Decision Method for Reasoning about Action. MIT LCS TM-113, Sept. 1978.

[44] Pratt, V.R. Process Logic. Proc. 6th Ann. ACM Symp. on Principles of Programming Languages, San Antonio, Texas, Jan. 1979.

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

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

[47] Salwicki, A. Formalized Algorithmic Languages. Bull. Acad. Pol. Sci., Ser. Sci. Math. Astr. Phys. Vol. 18. No. 5. 1970.

[48] Segerberg, K. A Completeness Theorem in the Modal Logic of Programs. Preliminary report Notices of the AMS, *24*, 6, A-552. Oct. 1977.

[49] Tarski, A. On the Calculus of Relations. J. Symbolic Logic, *6*, 73-89. 1941.

[50] Turing, A. Checking a Large Routine. In *Rep. Conf. High Speed Automatic Calculating Machines.* Inst. of Comp. Sci. Univ. of Toronto. Ontario, Can. Jan. 1950.

[51] Von Neumann, J. *Collected Works.* 5. pp. 91-99. Macmillan, New York. 1963.

[52] Winklmann, K. Equivalence of DL and DL$^+$ for regular programs without array assignments but with DL-formulas in tests. Manuscript, Lab. for Comp. Sci., M.I.T. 1978.

[53] Winklmann, K. Equivalence of DL and DL$^+$ for regular programs. Manuscript, Lab. for Comp. Sci., M.I.T. 1978.