MIT/LCS/TM-150

TEN THOUSAND AND ONE LOGICS OF PROGRAMMING

Albert R. Meyer

February 1980

TEN THOUSAND AND ONE LOGICS OF PROGRAMMING

Albert R. Meyer

This research was sponsored in part by NSF Grant #MCS 7719754 and in part by a Grant from ETH, Zurich, Switzerland.

TEN THOUSAND AND ONE LOGICS OF PROGRAMMING

Albert R. Meyer

Before coming here I had the pleasure of visiting with Professor Engeler in Zurich for two weeks along with representatives of two different schools of algorithmic logic in Poland - Salwicki, Mirokowska and Tiuryn - and also with my own colleague Parikh. I thought I would take this opportunity to tell you briefly what our concerns were and to try to summarize approximately two years of my own work and the work of colleagues at M.I.T. on logics of programs

So let me begin by reviewing what one of these logics of programs looks like, naturally the one that I am most familiar with, which was invented by my colleague Pratt, and which is called Dynamic Logic (see slide 1).

The basic concept with which we'll begin is a classical one, namely the notion of an interpretation or a structure. And we use the synonym state of a machine

Slide 1

DYNAMIC LOGIC (Pratt):

"State" of a machine = Structure
= Structure + Assignment
= Interpretation
= Algebra

⟨Domain, functions, relations⟩

Set of symbols = similarity type

x,y,... zero-ary function symbols
f,g,... k-ary function symbols
P,Q,... relation symbols

to refer to that. It means an algebra, a set with operations on it. So we have a domain, functions and relations. In addition, part of the concept of the state is a set of symbols or a <u>similarity type</u>. On Slide 1, x and y are variables or zero-ary function symbols, f and g are function symbols of some-arity, and P and Q are relational symbols. The idea of the <u>state</u> is that a state defines the meaning of those symbols. It tells you the domain and for each function symbol it associates a function on the domain for each predicate, etc. An entirely usual notion from logic. Our understanding of a program is that if we look at some simple program scheme like the one on Slide 2: "while x is not equal to y, set y to f(y)" --- if you begin by telling me everything there is to know about the program, if you tell me the state, namely, what is

Slide 2

β_o: while $x \neq y$ do $y := f(y)$

$s \to t$ iff $(\exists n)[x_s = f_s^{(n)}(y_s)]$

and $y_t = x_s$

and $u_s = u_t$ all $u \neq y$

the domain and what function on the domain is f, and what elements of the domain are x and y, then it's clear how to execute this little while program. We could formally define it but it's obvious what the semantics are, and when we're finished we wind up in general in a new state in which x and y have new values. More generally, even f might have new values if we allowed array assignment to f. So the semantics of a program like this is basically a mapping between states, or, in fact, a relation between states if the program is nondeterministic. Slide 2 displays a simple metalanguage formula defining what the semanitcs of this program is.

It says that state s can reach state t under this program β_o if and only if there is some n such that the value of f in state s applied n times to the value of y in state s is equal to the value of x in state s and the value of y in state t is equal to the value of x in state s, and more stuff like that. The details don't matter.

With that concept I can define for you Dynamic Logic. Namely, its first-order logic (see Slide 3). So we have terms of the atomic formulas, and I can write equality between terms, and if P is a predicate symbol I can write $P(t_1, t_2, t_3)$ as an atomic formula also, and then I can build new formulas by conjunction, negation and universal quantification. That's exactly first-order logic. And we have one more thing to get to Dynamic Logic. Namely, if α is a program, I'll let you write $[\alpha]F$, ("box alpha F"), when F is any formula in the language.

Now, the basic property of a formula in a logic is what interpretations make it true or false. The standard notation for that is that a state s satisfies a formula F. And it means F is true under the interpretation given by state s. Well, you know all the rules for predicate calculus, that is, you know that $F \wedge G$ is true at s if and only if F is true at s and G is true at s. The only thing I have to tell you that you don't know is when is [x]F true at S? Well, the meaning of $[\alpha]F$ is to be read as, after doing α, F will surely halt if α holds at all. So that is to say, a state s satisfies $[\alpha]F$ if for all of the t's such that t can be reached from s by a halting computation of α, t satisfies F; and that's a complete semantical definition of Dynamic Logic. That's all there is to it.

Slide 3

Formulas:

First Order:
- $t_1 = t_2$ terms t_1, t_2
- $P(t_1, t_2, t_3)$ P a predicate symbol
- $F \wedge G$
- $\neg F$
- $\forall x F$

DL:
- $[\alpha]$ F

Slide 4

$s \models F$ means "F is true in state S"

$s \models [\alpha]F$ iff

$t \models F$ for all t such that $s \xrightarrow{\alpha} t$

$\models F$ means "$s \models F$ for all s"

"F is valid"

Now let us turn to Slide 4 (above). If I write just $\models$ ("double turnstile symbol"), it means that the formula is valid, i.e., true in all states. So that's the object, the mathematical object that we've been studying and that I want to talk about.

Let's just for practice look at one or two quick examples. Here is a valid formula of Dynamic Logic (Slide 5). It says: after doing an array assignment in which we set the value of f(x) to be y, (this has the effect of course of changing only the function f; x and y don't change)... after setting f(x) to y, P(f(z)) holds, well, that's equivalent to saying that, well, if z were not equal to x, (so that f(z) didn't change) then that's the same as saying what you said before, P(f(z)). On the other hand, if z was equal to x, in which case f(z) has changed, then you say P(y). Again, it doesn't matter whether you follow that in detail, I'm trying to illustrate the language and the sorts of things that can be said quite conveniently with it.

Slide 5

$\models ([f(x):=y]\ P(f(z))) \equiv (z \neq x\ P(f((z)))$

$\vee\ (z=x\ P\wedge(y))$

$\langle x:=g(x);\ \ y:=g(y)\ \ (x=y)\rangle$

neither valid nor invalid

$\langle$while true do NOP$\rangle$ true

is always false

Slide 6

$x[g(f(x))=x] \rightarrow$

$[y:=z;\ \text{REPEAT } z:=f(z)]\ \langle \text{REPEAT } z:=g(z)\rangle\ (y=z)$

$x:=?$	means	"set x to an arbitrary value"
$[x:=?]\ Q$	$\leftrightarrow$	$(\forall_x)\ Q$
$\langle x:=?\rangle\ Q$	$\leftrightarrow$	$(\exists_x)\ Q$

$\models \forall_z [\langle\beta\rangle\ (x=z) \leftrightarrow \langle\gamma\rangle\ (x=z)]$ means "β and γ are equivalent (nondeterministic) programs w.r.t. output x.

Here's one more final example (Slide 6) of using the language. Here's a formula that's also valid. Let's look at the hypothesis. The hypothesis says that f is a one-to-one function and g is its inverse. The conclusion says the following: if I remember the initial value of z someplace, call it y, and then I nondeterministically apply f to z, any finite number of times (that's what REPEAT means, its a nondeterministic operation, sometimes written "star"), then after doing that, it is necessarily the case that it is possible to repeat some finite number of times doing the inverse of f, which is g to get back to where you started. (Diamond, which I read as "it is possible," is an abbreviation for "not box not", so it's not a new construct in the language, it's simply something that I'm so used to using that I didn't realize that I left out the slide defining it when I prepared this talk.) One more illustration of an operation that we sometimes allow in programs - and that may help understanding about boxes and diamonds - is called random assignment statement, x:=?, ("x gets question mark"). The semantics of that is that after you've done it, x can have any value at all in the domain. Nothing else changes. So that, e.g. if I said it's necessarily the case that after setting x to anything at all, Q holds, that's exactly

the same as saying that for all x, Q holds. And likewise, if I said it's possible to set x to something at all and have Q hold, that's exactly the same as saying that there exists an x such that Q holds. Technically speaking, once I've got this random assignment allowed in programs I don't even need the quantifiers. Although in general, real programs wouldn't have this random assignment, we use them for theoretical reasons.

Now, let me charicature some other very similar approaches to logics of programs. They vary but they all start off with similar motivation. We'd like a formal language in which we can make assertions about programs in a convenient way so that the assertions are easily written and not hard to transcribe from natural language into formal language.

Slide 7

LOGICAL CHARICATURES

How programs are treated:

DL_{Pratt}	Modalities (Quantifiers) (Predicate Transformers)
$AL_{Engeler}$	Atomic Formulas
$AL_{Salwicki}$	Terms (Modality)
LED_{Tiuryn}	Terms (Atomic Formulas)
Assertion $Lang_{Constable}$	Atomic Formulas (Modalities)

Nevertheless, all five approaches are essentially equivalent (in expressive power, degree of undecidability, axiomatizability).

At least five different groups of people have proposed such logics. They differ in how programs are treated. The one I have described to you - and as I say I don't have time for formal definitions, I'm

going to resort to caricature, - is Dynamic Logic. In Dynamic Logic, as Pratt formulated it (what I just showed you), programs in effect are being treated as modalities. The box and diamond are operations from modal logic. And in effect they are generalized quantifiers. As you saw talking about setting x to question mark inside of box (i.e. [x:=?]) was nothing but universal quantification. And we regard programs simply as more general kinds of quantifiers. Or predicate transformers, if you like. In algorithmic logic as Engeler formulated it originally, back in about 1967, and he was a seminal figure here in proposing this kind of thing - in his style of proposed logic, what he called algorithmic logic, and which I call algorithmic logic a la Engeler, programs appeared as atomic formulas.

In algorithmic logic a la Salwicki, invented about 1970 or 1971, programs appeared more as terms, though recently he has shifted more towards treating them as modalities. In the language of effective definition proposed by Tiuryn, again programs are treated primarily as terms, although I'm trying to persuade him that atomic formulas would be somewhat better. I don't know if I've succeeded yet. And finally there's an assertion language that Constable has proposed for talking about programs, in which again programs play the role of atomic formulas. All five of these approaches are essentially equivalent in expressive power, degree of undecidability, convenience of axiomatizability and so on. You haven't seen the ten thousand logics yet, all five of these are actually the same. And each can be transcribed into the other. So, where are the ten thousand logics? (See Slide 8).

Slide 8

TEN THOUSAND LOGICS OF PROGRAMMING

DIFFERENT CLASSES OF PROGRAMS LEAD TO DIFFERENT VERSIONS OF DL:

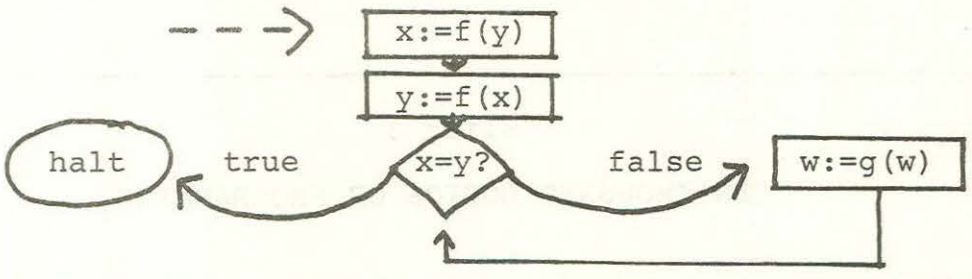
Basically, the one parameter that I have left undefined in explaining Dynamic Logic to you is what class of programs to use. I illustrated it with a simple flow-chart scheme. But, in fact, about four thousand or so of the ten thousand logics arise because of four thousand different plausible notions of what a program scheme is. So basically, different classes of programs lead to different versions of DL. That is to say, we might consider Dynamic Logic in which the only programs that can appear in formulas are programs definable by finite flow charts, not using random assignment, only using simple assignment. Or we might consider some other kind of program. Briefly, what are these kinds of programs? The most basic and familiar programs (See Slide 9) are defined by deterministic finite flow charts in which the tests in the programs are simply atomic formulas, and the only kinds of assignments are the most familiar kind of simple assignment in which one sets a variable to the value of a term. Somewhat more convenient and emphasized by Pratt are regular programs, which are the same thing except nondeterministic. And there's just an illustration, a picture

Slide 9

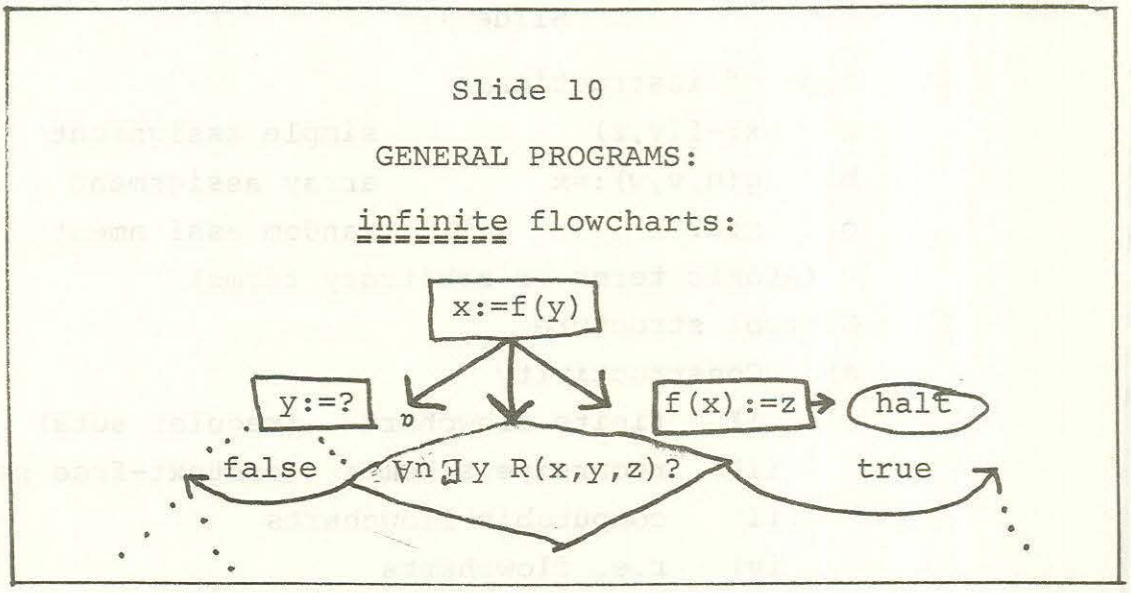
BASIC PROGRAMS

finite flowcharts (deterministic)
atomic tests
simple assignment

REGULAR:: = Nondeterministic BASIC



of one, I don't even remember what it does. But, again, this should be fairly familiar stuff. More generally, though, it's technically convenient and natural to make the leap to considering general programs which are defined by infinite flow charts, instead of finite ones (see Slide 10). And in which also for technical reasons it's quite natural for a logician to consider, as one of the many variants, allowing besides merely atomic tests, first-order formulas to appear as tests in the programs. And that is about the most general notion of program that any of us have considered and that I would consider it even unreasonable to adopt. I'm not advocating this, I'm trying to offer this as an upper bound of what we might in the most inclusive way call a program or an uninterpreted program.



Within this framework I will quickly run through four thousand possibilities. They multiply together, so there are only about five or six categories, with five or six subcategories and I'm merely going to try to give you the flavor of the options. So one has various options on types of instructions (Slide 11). The basic kinds of instructions we're considering are assignment instructions, we could have array assignments, random assignment, and there's a technical distinction about whether we will allow arbitrary nested

terms or only simple terms. And you can choose almost any of the nonempty subsets of these four things and you already get about sixteen possibilities or fifteen.

Next choice is the control structure. There are a number of possibilities here. You might insist that this infinite flow chart be at least reasonably constructive. The strongest version of constructive is that it actually be a finite flow chart. But then you might settle for it merely being recursive in the sense of definable by recursive equations, which actually means the flow chart itself is in a certain sense context free.

Slide 11

1) Type of instructions:
   - a) x:-f(y,z) simple assignment
   - b) g(u,v,w):=x array assignment
   - c) x:=? random assignment

   (atomic terms or arbitrary terms)

2) Control structure
   - a) Constructivity
     - i) finite flowchart (regular sets)
     - ii) recursive schemes (context-free sets)
     - iii) computable flowcharts
     - iv) r.e. flowcharts
     - v) arbitrary
   - b) Deterministic or not
   - c) Bounded fan-out or not

Or you might allow the flow chart to be merely recursive in the effective sense of recursion theory, or you might allow it to be recursively enumerable, you might allow it to be absolutely arbitrary. You can also worry about whether it's deterministic and whether a given box in the nondeterministic case has a bounded or

Slide 12

3) Type of tests
   a) atomic
   b) existential
   c) 1st order
   d) "rich" test: any DL formula
4) Number of tests
   a) finite
   b) infinite
      i) bounded quantifier alternations or not
5) Memory structure
   a) finite number of "registers" or not

an unbounded number of arrows are allowed to leave them, or even an infinite number of arrows. You can also consider types of tests: Whether they're atomic or first order and so on, whether the number of kinds of tests is finite or infinite (that can only happen in infinite flow charts) and put various quantifier restrictions upon them. You can also worry about whether or not in the case of infinite flow charts they should be allowed to work with infinitely many temporary registers or only finitely many. That's just about four thousand logics.

There are some other options which I'll mention very briefly, that have come up in the literature and been considered. Salwicki proposed two special constructs that he wanted to add that they thought were appropriate for talking about programs. One of which says it's possible to do α some finite number of times and have P hold (see Slide 13). (This doesn't add anything if your programs are closed under nondeterministic repetition, but in the case that you're dealing with deterministic programs, then adding this to the logical language might increase its power.) They have another operation which I won't define, called intersection.

There's one particular predicate of programs that has gotten a lot of attention because of Dikstra's weakest precondition, in which the notion of looping plays a critical role. It motivated Pratt and Harel to define what they called Dynamic Logic Plus, the "plus" was "plus looping". I won't define that for you except to say, put it in or put it out, it increases the number of possibilities. And then, one can consider, and these variations have been considered, other kinds of path predicates of a kind like looping, fairness and weird things like that. The case of the logic without quantifiers turns out to have quite a rich structure, and also the case where one adds arithmetic as part of the structure with the standard interpretation, which was studied extensively by Harel, is closely related to the weak second-order version. That ten thousand logics. (There's one coming, I haven't forgotten.)

Slide 13

ADDITIONAL LOGICAL CONSTRUCTS

1. $U(\alpha,P) ::= \langle \underline{\text{REPEAT}}\ \alpha \rangle P$

   (i.e., $\langle \alpha^* \rangle P$)

   (redundant if programs are closed under REPEAT)

2. $\mathbf{U}(\alpha,P) ::= \bigvee_n \langle \alpha^n \rangle P$

   (redundant for deterministic programs: $P \wedge [\text{REPEAT } \alpha]\langle\alpha\rangle P$)

3. Loop_α ::= "Infinitely many boxes of α are accessible"

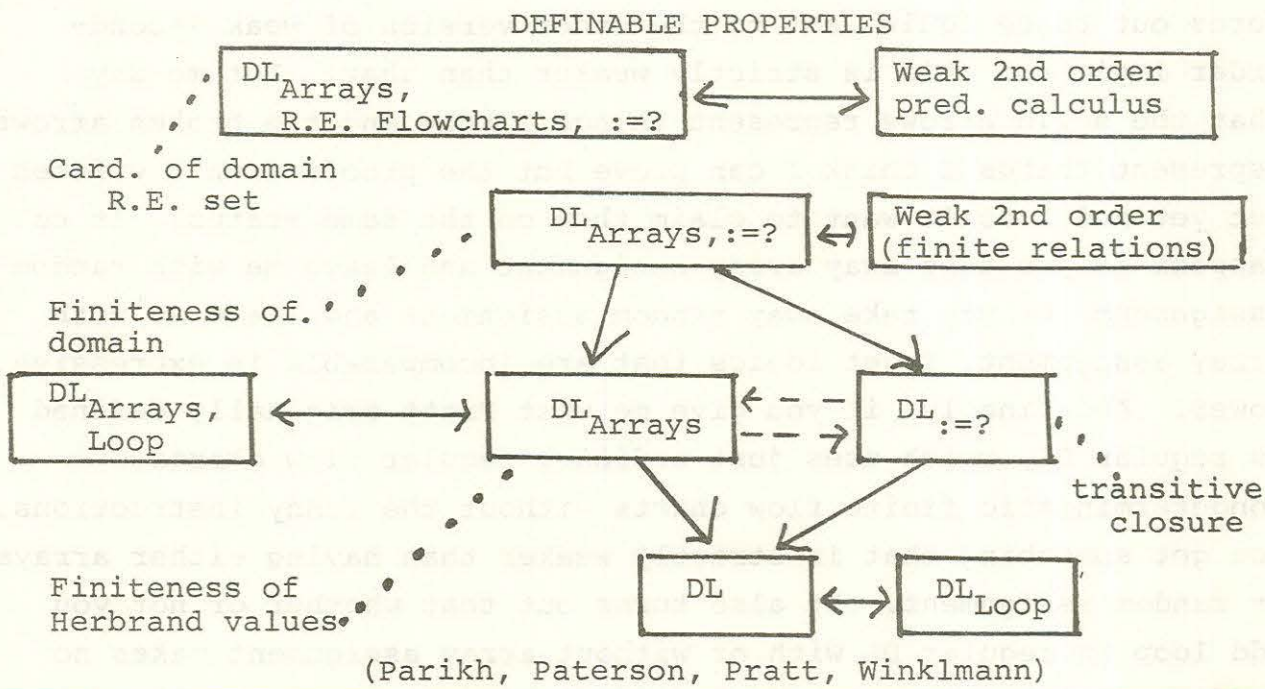
4. Other "path" predicates: until (α,P,Q), fair (α,P)

5. Quantifier-free DL

6. Arithmetic DL (Harel), Weak Second-Order DL

Now what sort of results are there about this thing to give you the flavor of our concerns. Well, the main concern that I had and still have when I'm faced with a situation like this is, with ten thousand possibilities, it's very likely that if you choose one at random it will be the wrong one, and since answering questions about these things tends to be quite challenging technically, it would be a disaster to spend years of your life working on the wrong version of logic. On the other hand, it's not at all clear what the criteria should be to choose the right one, and we have been floundering. I at least have been floundering, in trying to bring some order to this set of possibilities, get some understanding for what various options give you, how these logics relate to classically understood ones and also get some sense of the additional expressive power that allowing certain constructs will provide that others will not. The objective roughly is, that we would like to have the absolutely simplest logic that is adequate to talk about programs and prove things about them.

Slide 14



Because the world being as it is, you can be sure that if you have more technical machinery and power than you need to do the job at hand, you will pay for it in things like degrees of undecidability, or in the ease of theorem proving, or in some other unanticipated way. This led me to the question of asking, can we prove equivalences? Surely now, these ten thousand are not all different. Well, they're not all different, although there are more than one, when you're all finished. Above are some partial results of some landmarks of what some of these systems turn out to be (Slide 14).

Starting at the top, if I look at Dynamic Logic for the class of programs that contain array assignment and have infinite effective flow charts, and also random assignments, I get something that turns out to be equivalent to a nice classical object (semiclassical anyway), weak second-order predicate calculus. And I recently discovered that weak second-order predicate calculus is not well defined. There are two different versions of it that are not equivalent. They agree on states with infinite domains differently. So, we can look at Dynamic Logic with finite flow charts that have array assignment and random assignment and that turns out to be equivalent to the other version of weak second-order logic and this is strictly weaker than that. Let me say that the solid arrows represent things I know and the broken arrows represent things I think I can prove but the proofs aren't written out yet and I don't want to claim them on the same status. It so happens if you take away array assignment and leave me with random assignment, or you take away random assignment and leave me with array assignment, I get logics that are incomparable in expressive power. And finally, if you give me what Pratt originally defined as regular DL, which uses just ordinary regular flow charts, nondeterministic finite flow charts without the funny instructions, you get something that is strictly weaker than having either arrays or random assignment. It also turns out that whether or not you add loop in regular DL with or without array assignment makes no difference; the extension to Dynamic Logic Plus was actually

unnecessary, although the proofs in both these cases were nontrivial and these two proofs are utterly different. In general you might expect that adding loop to some other logic might change things, but in these cases of most interest it doesn't. To get a feel for how some of these distinctions are made, here are some things that can be said (indicated by dots) in some logics and not in others.

At the top, the most powerful of these logics can do this: pick your favorite recursively enumerable set, and it's possible to write a formula in this language whose meaning is, the domain is finite and its cardinality is in your recursively enumerable set. No such predicate is definable in any of the other weaker logics. In the next one down, there is a formula which is true precisely of those states whose domain is finite. That's definable with array assignment and with random assignment, but you need both. If you have random assignment but not array assignemnt, you can define transitive closure of a relation; if you have array assignment but not random assignment you can define finiteness of the set of values of terms in the language. Neither of these things are definable in DL if you just have simple assignment and no random assignment. But even in DL, the most primitive of these languages, you're well beyond first order and you can, for example, define any ordinal less than omega to the omega (ω^{ω}) up to isomorphism, although Parikh and I are able to show you can't get beyond ω^{ω}. These results are variously due to Parikh, Paterson, Pratt, Winklmann and myself.

Now, let's take a bigger overview of what's the gist of all of this, what have we learned in our two years or so of study and two weeks of conferring together? And this is the big picture which appears to be emerging (Slide 15). All of these variations are equivalent to various fragments of the language $L_{\omega_1,\omega}$, which is an infinitary extension of first-order language. That observation was in effect first made by Engeler and subsequently independently

made by lots of people, most everybody who worked in the language. All of them have a very high degree of undecidability for deciding validity, it's not even arithmetic, it's at level Π^1_1. We've shown that there are at least five distinct languages -- distinct in terms of their expressive power. There may be more, my own judgment is there are probably not more than a dozen, but there might be several hundred. The open problems at the present time are such that several hundred possibilities remain among the ten thousand. The model theoretic properties of these logics are similar to those of $L_{\omega_1,\omega}$, and so far have been amenable to classical methods, the deepest results obtained so far being due to Parikh and Tiuryn, using omitting-type theorems, Ehrenfeucht games, and all kinds of neat stuff.

Slide 15

GENERAL THEORETICAL PROPERTIES

1. All variations are equivalent to various fragments of $L_{\omega_1,\omega}$ (Engeler et.al.)
2. "All" have Π^1_1 decision problems for validity.
3. There are at least 5 but perhaps not many more inequivalent versions of DL (Parikh, Winklmann).
4. Model theoretic properties similar to $L_{\omega_1,\omega}$ and so far amenable to classical methods (Parikh, Tiuryn).
5. Easily axiomatizable (relative to arithmetic with uninterpreted extra symbols, or by infinitary inference rules). (Harel, Mirkowska, Pratt, Salwicki, Tiuryn).

And these logics are all easily axiomatizable, modulo the fact that they are totally undecidable of an unimaginably high degree of undecidability. So when you axiomatize them, something funny has to happen. One way out is to choose as axioms, axioms of arithmetic with uninterpreted function symbols. Take all the valid sentences

of arithmetic as axioms. Well, that's where you get the incredible degree of undecidability. Once you've got that around, it's easy to get a relatively complete axiom system for your programming logic. The other way to go is to have what's called an infinitary axiom system, and for what that's worth, that's been done.

The title of this talk as originally announced was "Ten Thousand Logics of Programming", but after hearing Manna's opening lecture, I had to add one.

The one more logic is the temporal logic that Manna talked about due to Pnueli (Slide 16). The main theorem as far as I am concerned about temporal logic is that in the propositional case, which is the main case that's been studied, temporal logic is precisely equivalent in expressive power to the first-order theory of the natural numbers under the order relation "less than" (with uninterpreted monadic predicates also allowed). That's the theorem of Gabbay, Pnueli, Shelah and Stavi. The corollary due to Meyer - I have to get in my controversial remark - is that that makes it not interesting theoretically. Now that we know their theorem, it seems to me quite apparent that you're not going to learn anything special about programs by talking about the order of the integers. Unfortunately I don't have time to really get into the argument about that, but that's what my interpretation of their recent main theorem is.

Slide 16

The "one" more logic is

TEMPORAL LOGIC (Pnueli)

Thm: (Gabbay, Pnueli, Shelah, Stavi)
Propositional Temproal Logic is equivalent to the first-order monadic theory of $(|N,<)$:

Cor: (Meyer) Therefore it is not theoretically interesting.

One more question issue: why are these Dynamic Logic logics, all the different variations, why have they come out so classically? (See Slide 17.) I'm disappointed in them, too, because we have not yet encountered the real combinatorial aspects of programs which I would hope the right logic of programs would have to embody. We're getting things like weak second-order logic and fragments of infinitary first-order logic. The reason, I think, is because we began with the classical notion of state from logic, that is, an algebraic structure. It seems to me that the direction for the future is when we begin looking at richer notions of state. Like reflexive domains, and structures with pointers and things like that. Some preliminary steps, I regard them as preliminary, have been taken by Milner and Scott in LCF and by Van Emde-Boas with his work with intensional logic. And I'm hoping that some synthesis will come about from merging our own work with this classical notion of state and infinitary logics with the work that's previously been done in this other direction.

Slide 17

Why are the DL-logics and results so "Classical"?

Because of the classical notion of "state" chosen!

For the future: Logics of richer notions of states:

- Reflexive domains - LCF (Milner, Scott)
- Structures with pointers - Intensive Logic (Van Emde-Boas)

REFERENCES

1. Banachowski, et.al. An Introduction to Algorithmic Logic, Mathematical Investigations in the Theory of Programs, Banach Center Publications, Volume 2, ed. C. Olech, Polish Scientific Publishers, Warsaw, 1977, 7-100.

2. Constable, R. and M. O'Donnell. A Programming Logic, Winthrop, Cambridge, Mass. c. 1978, 389 pp.

3. Engeler, E. Algorithmic Logic in Foundations of Computer Science, ed. J. W. DeBakker Mathematisch Centrum, Amsterdam, 1975, 57-88.

4. Harel, D. First-Order Dynamic Logic, Lecture Notes in Computer Science, Springer-Verlag, New York, 1979, 133 pp.

5. Harel, D. Proving the Correctness of Regular Deterministic Programs: A Unifying Survey Using Dynamic Logic, Theoretical Computer Science, 1979, to appear.

6. Manna, Z. and A. Pnueli. The Model Logic of Programs, Automata, Languages and Programming, ed. H. A. Mourer, Lectures Notes in Computer Science, 1979, 385-409.

7. Tiuryn, J. Logic of Effective Definitions, unpublished report, Lehrstuhl fur Informatik II, Achen, BRD, July, 1979, 29 pp.