Logics of Program

Dexter Kozen*
Jerzy Tiuryn**

89-962
January 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported by NSF grant CCS-8806979.
**Supported by a grant from the Polish Ministry of Higher Education R.P.I.09.

Logics of Programs*

Dexter Kozen[†]
Cornell University

Jerzy Tiuryn[‡]
University of Warsaw

January 16, 1989

Contents

1 Introduction **3**

1.1 States, Input/Output Relations, Traces 4
1.2 Exogenous and Endogenous Logics 4
1.3 Historical Note 5
1.4 Acknowledgments 5

2 Propositional Dynamic Logic **6**

2.1 Basic Definitions 6
2.1.1 Syntax 6
2.1.2 Semantics 7
2.1.3 Computation Sequences 9
2.2 A Deductive System for *PDL* 10
2.3 Basic Properties 11
2.3.1 The * Operator, Induction, and Reflexive-Transitive Closure 13
2.3.2 Encoding of Hoare Logic 15
2.4 The Small Model Property 16
2.4.1 The Relation $\prec$ and the Fischer-Ladner Closure 16
2.4.2 Filtration 17
2.4.3 Filtration over Nonstandard Models 18
2.5 Deductive Completeness 19
2.5.1 Logical Consequences 19
2.6 Complexity of the Satisfiability Problem for *PDL* 20
2.6.1 A Deterministic Exponential-Time Algorithm 20

---

*To appear in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, North Holland, Amsterdam, 1989.

[†]Supported by NSF grant CCS-8806979.

[‡]Supported by a grant from the Polish Ministry of Higher Education R.P.I.09.

2.6.2 A Lower Bound 22
2.7 Variants of *PDL* 23
2.7.1 Deterministic *PDL* and While Programs 23
2.7.2 Rich and Poor Tests 24
2.7.3 Context-Free Programs 24
2.7.4 Automata Theory and Program Logics 24
2.7.5 Converse 25
2.7.6 Well-Foundedness and Total Correctness 25
2.7.7 Other Work 27

3 First-Order Dynamic Logic **28**
3.1 Syntax 28
3.1.1 Arrays and Stacks 29
3.1.2 Nondeterministic Assignment 30
3.1.3 Rich and Poor Tests 30
3.2 Semantics 30
3.3 Complexity 35
3.3.1 The Validity Problem 35
3.3.2 Spectral Complexity 36
3.4 Deductive Systems 39
3.5 Expressive Power 42
3.6 Operational vs. Axiomatic Semantics 48
3.7 Other Programming Languages 49
3.7.1 Algol-Like Languages 49
3.7.2 Nondeterministic Assignment 50
3.7.3 Auxiliary Data Types 50
3.7.4 Tests 50

4 Other Approaches **51**
4.1 Nonstandard Dynamic Logic 51
4.2 Algorithmic Logic 53
4.3 Logic of Effective Definitions 53
4.4 Temporal Logic 54

References **55**

1 Introduction

Logics of Programs are formal systems for reasoning about computer programs. Traditionally, this has meant formalizing correctness specifications and proving rigorously that those specifications are met by a particular program. Other activities fall into this category as well: determining the equivalence of programs, comparing the expressive power of various operators, synthesing programs from specifications, etc. These activities range from the highly theoretical to the highly practical. Formal systems too numerous to mention have been proposed for these purposes, each with its own peculiarities.

This chapter gives a brief introduction to some of the basic issues in the study of program logics. We have chosen one system, Dynamic Logic, to illustrate these issues. There are perhaps many other reasonable choices: Temporal Logic, Algorithmic Logic, etc. We discuss the relationships among these systems where appropriate. By our choice of Dynamic Logic we do not advocate its use over any other system; we just feel that it is an appropriate vehicle for illustrating the concepts we wish to discuss, and it is the system with which we are the most familiar.

Program logics differ from classical logics in that truth is *dynamic* rather then *static*. In classical predicate logic, the truth value of a formula ϕ is determined by a valuation of its free variables over some structure. The valuation and the truth value of ϕ it induces are regarded as immutable; there is no formalism relating them to any other valuations or truth values. In program logics, there are explicit syntactic constructs called *programs* to change the values of variables, thereby changing the truth values of formulas. For example, the program $x := x + 1$ over the natural numbers changes the truth value of the formula "x is even". Such changes occur on a metalogical level in classical predicate logic, for example in the Tarski definition of truth of a formula: if $u : \{x, y, \ldots\} \to \mathcal{N}$ is a valuation of variables over the natural numbers $\mathcal{N}$, then the formula $\exists x\ x^2 = y$ is defined to be true under the valuation u iff there exists an $a \in \mathcal{N}$ such that the formula $x^2 = y$ is true under the valuation $u[x/a]$, where $u[x/a]$ agrees with u everywhere except x, on which it takes the value a. This definition involves a metalogical operation which produces $u[x/a]$ from u for all possible values $a \in \mathcal{N}$. This operation becomes explicit in Dynamic Logic in the form of the program $x := ?$, called a *nondeterministic assignment*. This is a rather unconventional program, since it is not effective; however, it is quite useful as a descriptive tool. A more conventional way to obtain a square root of y, if it exists, would be the program

$$x := 0;\ \textbf{while}\ x^2 < y\ \textbf{do}\ x := x + 1\ \textbf{od}\ . \tag{1}$$

In Dynamic Logic, programs are first-class objects on a par with formulas, complete with a collection of operators for forming compound programs inductively from a basis of primitive programs. In the simplest version of *DL*, these program operators are $\cup$ (nondeterministic choice), ; (sequential composition), * (iteration), and ? (test). These operators are already sufficient to generate all **while** programs (which over $\mathcal{N}$ are sufficient to compute all partial recursive functions). To discuss the effect of the execution of a program p on the truth of a formula ϕ, *DL* uses a modal construct $\langle p \rangle \phi$, which intuitively states, "It is possible to execute p and halt in a state satisfying ϕ." For example, the first-order formula $\exists x\ x^2 = y$ is equivalent to the *DL* formula

$$\langle x := ?\rangle x^2 = y\ . \tag{2}$$

In order to instantiate the quantifier effectively, we might replace the nondeterministic assignment inside the $\langle\ \rangle$ with the **while** program (1); the resulting formula would be equivalent to (2) in $\mathcal{N}$.

1.1 States, Input/Output Relations, Traces

A *state* is an instantaneous description of reality. In first-order Dynamic Logic, a state is often taken to be a valuation $u : V \to |\mathcal{A}|$ of variables V ranging over a first-order structure $\mathcal{A} = (|\mathcal{A}|, \ldots, f^{\mathcal{A}}, \ldots, R^{\mathcal{A}}, \ldots)$. In practice, variables may be of different types and range over the different sorts of a many-sorted structure, but for simplicity we usually assume that $\mathcal{A}$ is single-sorted. Here $|\mathcal{A}|$ is the carrier of $\mathcal{A}$, $f^{\mathcal{A}}$ represents a typical function of $\mathcal{A}$, and $R^{\mathcal{A}}$ is a typical relation on $\mathcal{A}$.

A program can be viewed as a transformation on states. Given an initial (input) state, the program will go through a series of intermediate states, perhaps eventually halting in a final (output) state. A sequence of states that can obtain from the execution of a program p starting from a particular input state is called a *trace*.

The traces of p need not be uniquely determined by their start states; i.e., p may be *nondeterministic*. A trace can be infinite, in which case p is said to be *looping* or *diverging*. A finite trace corresponds to a halting computation of p, and in this case we say that p *halts*, *terminates*, or *converges*. The set of all pairs of first and last states of finite traces of p is called the *input/output relation* of p.

In Dynamic Logic, programs are interpreted as input/output relations. *DL* cannot be used to reason about program behavior not manifested in the input/output relation. For this reason it is inadequate for dealing with programs that are not normally supposed to halt, such as operating systems. Other program logics, such as Temporal Logic or Process Logic, use an execution sequence semantics which overcomes this limitation. However, for programs that are supposed to halt, correctness criteria are traditionally given in the form of an *input/output specification*, consisting of a formal relation between the input and output states that the program is supposed to maintain. The input/output relation of a program carries all the information necessary to determine whether the program is correct relative to such a specification.

1.2 Exogenous and Endogenous Logics

There are two main approaches to modal logics of programs: the *exogenous* approach, exemplified by Dynamic Logic and its precursor, the Partial Correctness Assertions Method (Hoare Logic) [70], and the *endogenous* approach, exemplified by Temporal Logic and its precursor, the Inductive Assertions Method [42]. A logic is *exogenous* if its programs are explicit in the language. Syntactically, a Dynamic Logic program is a well-formed expression built inductively from primitive programs using a small set of program operators. Semantically, a program is interpreted as its input/output relation. The relation denoted by a compound program is determined by the relations denoted by its parts. This aspect of *compositionality* allows proofs by structural induction. In Temporal Logic, the program is fixed and considered part of the structure over which the logic is interpreted. The current location in the program during execution is stored in a special variable for that purpose, called the *program counter*, and is part of the state along with the values of the program variables. Instead of program operators, there are temporal operators that describe how the program variables, including the program counter, change with time. Thus Temporal Logic sacrifices compositionality for a less restricted and hence more generally applicable formalism.

1.3 Historical Note

The idea of introducing and investigating a formal system dealing with properties of programs in an abstract model was advanced by Thiele [147] and independently by Engeler [37] in the mid 1960s. Research in program verification flourished in the late 1960s and thereafter with the work of many researchers, notably Floyd [42], Hoare [70], and Salwicki [136]. Dynamic Logic, which emphasizes the modal nature of the program/assertion interaction, was introduced by Pratt in [126] (see also [57]) and is a relatively late development.

There are by now a number of books and survey papers treating logics of programs and Dynamic Logic. We refer the reader to [54,56,117,47,48,75].

1.4 Acknowledgments

We would like to thank the following colleagues for their valuable criticism: J. Bergstra, P. van Emde Boas, E. A. Emerson, J. Halpern, D. Harel, M. Karpiński, N. Klarlund, D. McAllester, A. Meyer, R. Parikh, V. Pratt, and M. Vardi. We are especially indebted to David Harel, on whose excellent survey [56] the present work is modeled.

2 Propositional Dynamic Logic

Propositional Dynamic Logic (*PDL*) was first defined by Fischer and Ladner [40,41]. It plays the same role in Dynamic Logic that classical propositional logic plays in classical predicate logic. It describes the properties of the interaction between programs and propositions that are independent of the domain of computation.

2.1 Basic Definitions

2.1.1 Syntax

Syntactically, *PDL* is a blend of propositional logic, modal logic, and the algebra of regular events. It has expressions of two sorts: *programs* $p, q, r, \ldots$ and *propositions* or *formulas* $\phi, \psi, \ldots$. There are countably many atomic symbols of each sort, as well as a variety of operators for forming compound expressions from simpler ones:

- propositional operators $\vee$, $\neg$
- program operators $\cup$, ;, *
- mixed operators ?, $\langle\ \rangle$

Compound programs and propositions are defined by mutual induction, as follows. If ϕ, ψ are propositions and p, q are programs, then

$$\begin{array}{ll} \phi \vee \psi & \textit{(propositional disjunction)} \\ \neg\phi & \textit{(propositional negation)} \\ \langle p \rangle \phi & \textit{(modal possibility)} \end{array}$$

are propositions and

$$\begin{array}{ll} p; q & \textit{(sequential composition)} \\ p \cup q & \textit{(nondeterministic choice)} \\ p^* & \textit{(iteration)} \\ \phi? & \textit{(test)} \end{array}$$

are programs. The intuitive meanings of the less familiar of these constructs are as follows:

$$\begin{array}{rcl} \langle p \rangle \phi & = & \text{“It is possible to execute } p \text{ and terminate in a state satisfying } \phi\text{.”} \\ p; q & = & \text{“Execute } p\text{, then execute } q\text{.”} \\ p \cup q & = & \text{“Choose either } p \text{ or } q \text{ nondeterministically and execute it.”} \\ p^* & = & \text{“Execute } p \text{ repeatedly a nondeterministically chosen finite number of times.”} \\ \phi? & = & \text{“Test } \phi\text{; proceed if true, fail if false.”} \end{array}$$

The set of propositions is denoted Φ. We avoid parentheses by assigning precedence to the operators: unary operators, including $\langle p \rangle$, bind tighter than binary ones, and ; binds tighter than $\cup$. Also, under the semantics to be given in the next section, the operators ; and $\cup$ will turn out to be associative, so we may write $p; q; r$ and $p \cup q \cup r$ without ambiguity.

The primitive operators are chosen for their mathematical simplicity. A number of more conventional programming constructs can be defined from them. The propositional operators $\wedge$, $\rightarrow$, $\leftrightarrow$, *false*, and *true* are defined from $\vee$ and $\neg$ in the usual way. In addition:

$$
\begin{array}{rcl}
\textbf{skip} & = & \mathit{true}? \\
\textbf{fail} & = & \mathit{false}? \\
[p]\phi & = & \neg\langle p\rangle\neg\phi \\
\textbf{if } \phi_1 \rightarrow p_1 \;[\!]\; \cdots \;[\!]\; \phi_n \rightarrow p_n \textbf{ fi} & = & \phi_1?; p_1 \cup \cdots \cup \phi_n?; p_n \\
\textbf{do } \phi_1 \rightarrow p_1 \;[\!]\; \cdots \;[\!]\; \phi_n \rightarrow p_n \textbf{ od} & = & (\phi_1?; p_1 \cup \cdots \cup \phi_n?; p_n)^*; (\neg\phi_1 \wedge \cdots \wedge \neg\phi_n)? \\
\textbf{if } \phi \textbf{ then } p \textbf{ else } q \textbf{ fi} & = & \textbf{if } \phi \rightarrow p \;[\!]\; \neg\phi \rightarrow q \textbf{ fi} \quad = \quad \phi?; p \cup \neg\phi?; q \\
\textbf{while } \phi \textbf{ do } p \textbf{ od} & = & \textbf{do } \phi \rightarrow p \textbf{ od} \quad = \quad (\phi?; p)^*; \neg\phi? \\
\textbf{repeat } p \textbf{ until } \phi & = & p; \textbf{while } \neg\phi \textbf{ do } p \textbf{ od} \quad = \quad p; (\neg\phi?; p)^*; \phi? \\
\{\phi\}p\{\psi\} & = & \phi \rightarrow [p]\psi
\end{array}
$$

The propositions $\langle p\rangle\phi$ and $[p]\phi$ are read "diamond p ϕ" and "box p ϕ", respectively. The latter has the intuitive meaning, "Whenever p terminates, it must do so in a state satisfying ϕ." Unlike $\langle p\rangle\phi$, $[p]\phi$ does not imply that p terminates. Indeed, the formula $[p]$*false* asserts that no computation of p terminates. For fixed program p, the operator $\langle p\rangle$ behaves like a possibility operator of modal logic (see [72,20]). The operator $[p]$ is the modal dual of $\langle p\rangle$ and behaves like a modal necessity operator.

The ternary **if-then-else** operator and the binary **while-do** operator are the usual *conditional test* and *while loop* constructs found in conventional programming languages. The constructs **if-[]-fi** and **do-[]-od** are the *alternative guarded command* and *iterative guarded command* constructs, respectively (see [49]). The construct $\{\phi\}p\{\psi\}$ is the classical Hoare partial correctness assertion [70].

2.1.2 Semantics

The formal semantics of *PDL* comes from modal logic. A *Kripke model* is a pair $\mathcal{M} = (S^{\mathcal{M}}, I^{\mathcal{M}})$ where $S^{\mathcal{M}} = \{u, v, \ldots\}$ is an abstract set of *states* and $I^{\mathcal{M}}$ is an *interpretation function.* Each proposition ϕ is interpreted as a subset $\phi^{\mathcal{M}} \subseteq S^{\mathcal{M}}$, and each program p is interpreted as binary relation $p^{\mathcal{M}}$ on $S^{\mathcal{M}}$. We may think of $\phi^{\mathcal{M}}$ as the set of states satisfying the proposition ϕ, and $p^{\mathcal{M}}$ as the input/output relation of the program p.

The function $I^{\mathcal{M}}$ assigns an arbitrary subset of $S^{\mathcal{M}}$ to each atomic proposition symbol and an arbitrary binary relation on $S^{\mathcal{M}}$ to each atomic program symbol. Compound programs and propositions receive their meanings inductively:

$$
\begin{array}{rcl}
(\phi \vee \psi)^{\mathcal{M}} & = & \phi^{\mathcal{M}} \cup \psi^{\mathcal{M}} \\
(\neg\phi)^{\mathcal{M}} & = & S^{\mathcal{M}} - \phi^{\mathcal{M}} \\
(\langle p\rangle\phi)^{\mathcal{M}} & = & p^{\mathcal{M}} \circ \phi^{\mathcal{M}} \quad = \quad \{u \mid \exists v\ (u,v) \in p^{\mathcal{M}} \text{ and } v \in \phi^{\mathcal{M}}\}
\end{array}
$$

$$
\begin{aligned}
(p;q)^{\mathcal{M}} &= p^{\mathcal{M}} \circ q^{\mathcal{M}} = \{(u,v) \mid \exists w\ (u,w) \in p^{\mathcal{M}} \text{ and } (w,v) \in q^{\mathcal{M}}\} \\
(p \cup q)^{\mathcal{M}} &= p^{\mathcal{M}} \cup q^{\mathcal{M}} \\
(p^*)^{\mathcal{M}} &= \text{the reflexive transitive closure of } p^{\mathcal{M}} \\
(\phi?)^{\mathcal{M}} &= \{(u,u) \mid u \in \phi^{\mathcal{M}}\}
\end{aligned}
$$

where the reflexive-transitive closure of a binary relation ρ is

$$
\bigcup_{n<\omega} \rho^n ,
$$

where

$$
\begin{aligned}
\rho^0 &= \{(u,u) \mid u \in S^{\mathcal{M}}\} \\
\rho^{n+1} &= \rho \circ \rho^n .
\end{aligned}
$$

The symbol $\circ$ denotes relational composition.

The defined operators inherit their meanings from these definitions:

$$
\begin{aligned}
(\phi \wedge \psi)^{\mathcal{M}} &= \phi^{\mathcal{M}} \cap \psi^{\mathcal{M}} \\
([p]\phi)^{\mathcal{M}} &= \{u \mid \forall v\ (u,v) \in p^{\mathcal{M}} \rightarrow v \in \phi^{\mathcal{M}}\} \\
\mathit{true}^{\mathcal{M}} &= S^{\mathcal{M}} \\
\mathit{false}^{\mathcal{M}} &= \emptyset \\
\mathbf{skip}^{\mathcal{M}} &= \{(u,u) \mid u \in S^{\mathcal{M}}\} \\
\mathbf{fail}^{\mathcal{M}} &= \emptyset
\end{aligned}
$$

The **if-then-else**, **while-do**, and guarded commands also receive meanings from the above definitions.

It can be argued that the input/output relations given by these definitions capture the intuitive operational meanings of these constructs. For example, the relation associated with the program **while** ϕ **do** p **od** is the set of pairs (u,v) for which there exist states $u_0, u_1, \ldots, u_n$, $n \geq 0$, such that $u = u_0$, $v = u_n$, $u_i \in \phi^{\mathcal{M}}$ and $(u_i, u_{i+1}) \in p^{\mathcal{M}}$ for $0 \leq i < n$, and $u_n \in \neg\phi^{\mathcal{M}}$.

We often write $\mathcal{M}, u \models \phi$ for $u \in \phi^{\mathcal{M}}$ and say that u *satisfies* ϕ *in* $\mathcal{M}$. We may write $u \models \phi$ when $\mathcal{M}$ is understood. We write $\mathcal{M} \models \phi$ if $\mathcal{M}, u \models \phi$ for all $u \in \mathcal{M}$, and we write $\models \phi$ and say that ϕ is *valid* if $\mathcal{M} \models \phi$ for all $\mathcal{M}$. We say that ϕ is *satisfiable* if $\mathcal{M}, u \models \phi$ for some $\mathcal{M}, u$. If Σ is a set of propositions, we write $\mathcal{M} \models \Sigma$ if $\mathcal{M} \models \phi$ for all $\phi \in \Sigma$. A proposition ψ is said to be a *logical consequence* of Σ if for all $\mathcal{M}$, $\mathcal{M} \models \psi$ whenever $\mathcal{M} \models \Sigma$, in which case we write $\Sigma \models \psi$. (Note that this is *not* the same as saying that $\mathcal{M}, u \models \psi$ whenever $\mathcal{M}, u \models \Sigma$.) We say that an inference rule

$$
\frac{\Sigma}{\psi}
$$

is *sound* if ψ is a logical consequence of Σ.

This version of *PDL* is usually called *regular PDL* because of the primitive operators $\cup, ;, *$, which are familiar from the algebra of regular events (see [71]). Programs can be viewed as regular expressions over the atomic programs and tests. In fact, it can be shown that if ϕ is an atomic proposition symbol, then any two test-free programs p, q are equivalent as regular expressions if and only if the formula $\langle p\rangle\phi \leftrightarrow \langle q\rangle\phi$ is valid.

Example 1 Let ϕ be an atomic proposition, let p be an atomic program, and let $\mathcal{M} = (S^{\mathcal{M}}, I^{\mathcal{M}})$ be a Kripke model with

$$\begin{aligned} S^{\mathcal{M}} &= \{u, v, w\} \\ \phi^{\mathcal{M}} &= \{u, v\} \\ p^{\mathcal{M}} &= \{(u,v),(u,w),(v,w),(w,v)\} \end{aligned}$$

Then $u \models \langle p\rangle\neg\phi \wedge \langle p\rangle\phi$, but $v \models [p]\neg\phi$ and $w \models [p]\phi$. Moreover,

$$\mathcal{M} \models \langle p^*\rangle[(p;p)^*]\phi \wedge \langle p^*\rangle[(p;p)^*]\neg\phi \ .$$

□

Other semantics besides Kripke semantics have been studied [12,110,76,77,129].

2.1.3 Computation Sequences

Let p be a program. A *finite computation sequence* of p is a finite-length string of atomic programs and tests representing a possible sequence of atomic steps that may occur in a halting execution of p. The set of all such sequences is denoted $CS(p)$. We use the word "possible" loosely—$CS(p)$ is determined by the syntax of p alone, and may contain strings that are never executed in any interpretation.

Formally, the set $CS(p)$ is defined by induction on syntax:

$$\begin{aligned} CS(p) &= \{p\},\ p \text{ an atomic program or test} \\ CS(p;q) &= \{\alpha;\beta \mid \alpha \in CS(p),\ \beta \in CS(q)\} \\ CS(p \cup q) &= CS(p) \cup CS(q) \\ CS(p^*) &= \bigcup_{n\geq 0} CS(p^n) \end{aligned}$$

where $p^0 = \textbf{skip}$ and $p^{n+1} = p;p^n$. For example, if p is an atomic program and ϕ is an atomic formula, then the program

$$\textbf{while } \phi \textbf{ do } p \textbf{ od} \ = \ (\phi?;p)^*;\neg\phi?$$

has as computation sequences all strings of the form

$$\phi?;p;\phi?;p;\cdots;\phi?;p;\textbf{skip};\neg\phi? \ .$$

Note that each finite computation sequence q of a program p is itself a program, and

$$CS(q) = \{q\} \ .$$

Moreover, the following proposition is not difficult to prove by induction on syntax:

Proposition 2 $p^{\mathcal{M}} = \bigcup_{q\in CS(p)} q^{\mathcal{M}}$.

2.2 A Deductive System for *PDL*

The following Hilbert-style axiom system for *PDL* was formulated by Segerberg [139].

Axioms of *PDL*

1. *axioms for propositional logic*
2. $\langle p\rangle\phi \wedge [p]\psi \rightarrow \langle p\rangle(\phi \wedge \psi)$
3. $\langle p\rangle(\phi \vee \psi) \leftrightarrow \langle p\rangle\phi \vee \langle p\rangle\psi$
4. $\langle p \cup q\rangle\phi \leftrightarrow \langle p\rangle\phi \vee \langle q\rangle\phi$
5. $\langle p; q\rangle\phi \leftrightarrow \langle p\rangle\langle q\rangle\phi$
6. $\langle \psi?\rangle\phi \leftrightarrow \psi \wedge \phi$
7. $(\phi \vee \langle p\rangle\langle p^*\rangle\phi) \rightarrow \langle p^*\rangle\phi$
8. $\langle p^*\rangle\phi \rightarrow (\phi \vee \langle p^*\rangle(\neg\phi \wedge \langle p\rangle\phi))$

Axioms 2 and 3 are not particular to *PDL*, but hold in all normal modal systems (see [72,20]). Axiom 8 is called the *PDL induction axiom*, and is better known in its dual form (Theorem 3(8) below).

Rules of Inference

1. *modus ponens:*
$$\frac{\phi,\ \phi \rightarrow \psi}{\psi}$$
2. *modal generalization:*
$$\frac{\phi}{[p]\phi}$$

We write $\vdash \phi$ if the proposition ϕ is a theorem of this system, and say that ϕ is *consistent* if not $\vdash \neg\phi$. A set Σ of propositions is *consistent* if all finite conjunctions of elements of Σ are consistent.

The soundness of these axioms and rules over the Kripke semantics can be established by elementary arguments in relational algebra.

2.3 Basic Properties

We list some basic theorems and derived rules of *PDL* that are provable in the deductive system of the previous section.

Theorem 3 *The following are theorems of PDL:*

1. *all propositional tautologies*
2. $[p](\phi \rightarrow \psi) \rightarrow ([p]\phi \rightarrow [p]\psi)$
3. $[p](\phi \wedge \psi) \leftrightarrow [p]\phi \wedge [p]\psi$
4. $[p \cup q]\phi \leftrightarrow [p]\phi \wedge [q]\phi$
5. $[p; q]\phi \leftrightarrow [p][q]\phi$
6. $[\phi?]\psi \leftrightarrow (\phi \rightarrow \psi)$
7. $[p^*]\phi \rightarrow \phi \wedge [p][p^*]\phi$
8. $(\phi \wedge [p^*](\phi \rightarrow [p]\phi)) \rightarrow [p^*]\phi$
9. $\langle p\rangle(\phi \wedge \psi) \rightarrow \langle p\rangle\phi \wedge \langle p\rangle\psi$
10. $[p](\phi \vee \psi) \leftarrow [p]\phi \vee [p]\psi$
11. $\langle p^*; p^*\rangle\phi \leftrightarrow \langle p^*\rangle\phi$
12. $\langle p^{**}\rangle\phi \leftrightarrow \langle p^*\rangle\phi$
13. $(\phi \vee \langle p\rangle\langle p^*\rangle\phi) \leftrightarrow \langle p^*\rangle\phi$
14. $\langle p^*\rangle\phi \leftrightarrow (\phi \vee \langle p^*\rangle(\neg\phi \wedge \langle p\rangle\phi))$
15. $(\phi \wedge [p^*](\phi \rightarrow [p]\phi)) \leftrightarrow [p^*]\phi$

Theorem 4 *The following are sound rules of inference of PDL:*

1. *monotonicity of* $\langle p\rangle$*:*
$$\frac{\phi \rightarrow \psi}{\langle p\rangle\phi \rightarrow \langle p\rangle\psi}$$
2. *monotonicity of* $[p]$*:*
$$\frac{\phi \rightarrow \psi}{[p]\phi \rightarrow [p]\psi}$$

3. *reflexive-transitive closure:*

$$\frac{(\phi \vee \langle p \rangle \psi) \rightarrow \psi}{\langle p^* \rangle \phi \rightarrow \psi}$$

4. *loop invariance rule:*

$$\frac{\psi \rightarrow [p]\psi}{\psi \rightarrow [p^*]\psi}$$

5. *Hoare composition rule:*

$$\frac{\{\phi\}p\{\sigma\}, \ \{\sigma\}q\{\psi\}}{\{\phi\}p; q\{\psi\}}$$

6. *Hoare conditional rule:*

$$\frac{\{\phi \wedge \sigma\}p\{\psi\}, \ \{\neg\phi \wedge \sigma\}q\{\psi\}}{\{\sigma\}\mathbf{if}\ \phi\ \mathbf{then}\ p\ \mathbf{else}\ q\ \mathbf{fi}\{\psi\}}$$

7. *Hoare* **while** *rule:*

$$\frac{\{\phi \wedge \psi\}p\{\psi\}}{\{\psi\}\mathbf{while}\ \phi\ \mathbf{do}\ p\ \mathbf{od}\{\neg\phi \wedge \psi\}}$$

The properties of Theorem 3(2-8) are the modal duals of Axioms 2-8. The converses of Theorem 3(9,10) are *not* valid. They are violated in state u of the model of Example 1. The rules of Theorem 4(1,2) say that the constructs $\langle p \rangle \phi$ and $[p]\phi$ are *monotone* in ϕ with respect to the ordering of logical implication. These constructs are also monotone and antitone in p, respectively, as asserted by the following metatheorem:

Proposition 5 *If $p^{\mathcal{M}} \subseteq q^{\mathcal{M}}$, then for all ϕ,*

1. $\mathcal{M} \models \langle p \rangle \phi \rightarrow \langle q \rangle \phi$
2. $\mathcal{M} \models [q]\phi \rightarrow [p]\phi$.

These follow from Axiom 4 and Theorem 3(4).

2.3.1 The * Operator, Induction, and Reflexive-Transitive Closure

The iteration operator * is interpreted as the reflexive-transitive closure operator on binary relations. It is the means by which looping is coded in *PDL*. Looping introduces a level of complexity to *PDL* beyond the other operators. Because of it, *PDL* is not compact: the set

$$\{\langle p^*\rangle\phi\} \cup \{\neg\phi, \neg\langle p\rangle\phi, \neg\langle p^2\rangle\phi, \ldots\} \tag{3}$$

is finitely satisfiable but not satisfiable. This seems to say that looping is inherently infinitary; it is thus rather surprising that there should be a finitary complete axiomatization.

The dual propositions Axiom 8 and Theorem 3(8) are jointly called the *PDL induction axiom*. Together with Axiom 7, they completely axiomatize the behavior of *. Intuitively, the induction axiom in the form of Theorem 3(8) says: if ϕ is true initially, and if the truth of ϕ is preserved by the program p, then ϕ will be true after any number of iterations of p. It is very similar to the induction axiom of Peano arithmetic:

$$\phi(0) \wedge \forall n\ (\phi(n) \rightarrow \phi(n+1)) \quad \rightarrow \quad \forall n\ \phi(n)\ .$$

Here $\phi(0)$ is the basis of the induction and $\forall n\ (\phi(n) \rightarrow \phi(n+1))$ is the induction step, from which the conclusion $\forall n\ \phi(n)$ may be drawn. In the *PDL* induction axiom, the basis is ϕ and the induction step is $[p^*](\phi \rightarrow [p]\phi)$, from which the conclusion $[p^*]\phi$ may be drawn.

The induction axiom is closely related to the reflexive-transitive closure rule (Theorem 4(3)). The significance of this rule is best described in terms of its relationship to Axiom 7. This axiom is obtained by substituting $\langle p^*\rangle\phi$ for ψ in the premise of the reflexive-transitive closure rule. Axiom 7 thus says that $\langle p^*\rangle\phi$ is a solution of

$$(\phi \vee \langle p\rangle X) \rightarrow X\ ; \tag{4}$$

the reflexive-transitive closure rule says that it is the *least* solution to (4) (with respect to logical implication) among all *PDL* propositions.

The relationship between the induction axiom (Axiom 8), the reflexive-transitive closure rule (Theorem 4(3)), and the rule of loop invariance (Theorem 4(4)) is summed up in the following proposition. We emphasize that this result is purely proof-theoretic and is independent of the semantics of §2.1.2.

Proposition 6 *In PDL without the induction axiom, the following axioms and rules are interderivable:*

(i) the induction axiom (Axiom 8);

(ii) the loop invariance rule (Theorem 4(4));

(iii) the reflexive-transitive closure rule (Theorem 4(3)).

Proof. First we show that the monotonicity rule (Theorem 4(2)) is derivable in *PDL* without induction. Assuming the premise $\phi \to \psi$ and applying modal generalization, we obtain $[p](\phi \to \psi)$; the conclusion $[p]\phi \to [p]\psi$ then follows from Theorem 3(2) and modus ponens. The dual monotonicity rule (Theorem 4(1)) can be derived from this rule by pure propositional reasoning.

(i) $\to$ (ii): Assume the premise of (ii):

$$\phi \quad \to \quad [p]\phi \, .$$

By modal generalization,

$$[p^*](\phi \to [p]\phi) \, ,$$

thus

$$\begin{aligned} \phi \quad &\to \quad \phi \wedge [p^*](\phi \to [p]\phi) \\ &\to \quad [p^*]\phi \, . \end{aligned}$$

The first implication is by propositional reasoning, and the second is the box form of the induction axiom (Theorem 3(8)). By transitivity of implication, we obtain

$$\phi \quad \to \quad [p^*]\phi \, ,$$

which is the conclusion of (ii).

(ii) $\to$ (iii): Dualizing the rule (iii) by purely propositional reasoning, we obtain a rule

$$\frac{\psi \quad \to \quad \phi \wedge [p]\psi}{\psi \quad \to \quad [p^*]\phi} \tag{5}$$

equipollent with (iii). It thus suffices to derive (5) from (ii). From the premise of (5), we obtain by propositional reasoning the two formulas

$$\psi \quad \to \quad \phi \tag{6}$$

$$\psi \quad \to \quad [p]\psi \, . \tag{7}$$

Applying (ii) to (7), we obtain

$$\psi \quad \to \quad [p^*]\psi \, ,$$

which by (6) and monotonicity gives

$$\psi \quad \to \quad [p^*]\phi \, .$$

This is the conclusion of (5).

(iii) $\to$ (i): By Axiom 3, propositional reasoning, and Axiom 7, we have

$$\begin{aligned} &\phi \vee \langle p\rangle(\phi \vee \langle p^*\rangle(\neg\phi \wedge \langle p\rangle\phi)) \\ &\quad \to \quad \phi \vee \langle p\rangle\phi \vee \langle p\rangle\langle p^*\rangle(\neg\phi \wedge \langle p\rangle\phi) \\ &\quad \to \quad \phi \vee (\neg\phi \wedge \langle p\rangle\phi) \vee \langle p\rangle\langle p^*\rangle(\neg\phi \wedge \langle p\rangle\phi) \\ &\quad \to \quad \phi \vee \langle p^*\rangle(\neg\phi \wedge \langle p\rangle\phi) \, . \end{aligned}$$

By transitivity of implication,

$$\phi \vee \langle p \rangle(\phi \vee \langle p^* \rangle(\neg\phi \wedge \langle p \rangle\phi)) \quad \rightarrow \quad \phi \vee \langle p^* \rangle(\neg\phi \wedge \langle p \rangle\phi) \ .$$

Applying (iii), we obtain the induction axiom:

$$\langle p^* \rangle\phi \quad \rightarrow \quad \phi \vee \langle p^* \rangle(\neg\phi \wedge \langle p \rangle\phi) \ .$$

□

2.3.2 Encoding of Hoare Logic

Dynamic Logic subsumes Hoare Logic. As an illustration, we show how to derive the Hoare **while** rule (Theorem 4(7)) in *PDL*. The other Hoare rules are also derivable.

Assume that the premise

$$\{\phi \wedge \psi\}p\{\psi\} \quad = \quad (\phi \wedge \psi) \rightarrow [p]\psi \tag{8}$$

holds. We wish to derive the conclusion

$$\{\psi\}\textbf{while } \phi \textbf{ do } p \textbf{ od}\{\neg\phi \wedge \psi\} \quad = \quad \psi \rightarrow [(\phi?;p)^*;\neg\phi?](\neg\phi \wedge \psi) \ . \tag{9}$$

Using Theorem 3(5,6) and propositional reasoning, the right-hand-side of (8) is equivalent to

$$\psi \quad \rightarrow \quad [\phi?;p]\psi \ .$$

Applying the loop invariance rule (Theorem 4(4)), we obtain

$$\psi \quad \rightarrow \quad [(\phi?;p)^*]\psi \ .$$

By the monotonicity of $[(\phi?;p)^*]$ (Theorem 4(2)) and propositional reasoning,

$$\psi \quad \rightarrow \quad [(\phi?;p)^*](\neg\phi \rightarrow \neg\phi \wedge \psi) \ .$$

Again by Theorem 3(5,6) and propositional reasoning, we obtain the right-hand-side of (9).

2.4 The Small Model Property

The *small model property* comes from modal logic. Although the proof for *PDL* is substantially more complicated than for simpler modal systems, the same proof technique, called *filtration*, applies. The small model property for *PDL* was first proved by Fischer and Ladner [40,41].

The small model property says that if ϕ is satisfiable, then it is satisfied at a state in a model with no more than $2^{|\phi|}$ states, where $|\phi|$ is the length (number of symbols) of ϕ. This immediately gives a naive nondeterministic exponential-time algorithm for the satisfiability problem. A deterministic exponential-time algorithm will be obtained in §2.6.1.

2.4.1 The Relation $\prec$ and the Fischer-Ladner Closure

Many proofs in simpler modal systems use induction on subformulas. In *PDL*, the situation is complicated by the simultaneous inductive definitions of programs and propositions; it is further complicated by the behavior of the * operator. We must therefore do our induction on another well-founded relation that resembles the subformula relation but is somewhat more intricate.

Let $\prec$ be the smallest transitive relation on $\{0,1\} \times \Phi$ containing the following inequalities:

1. $(0,\phi),\ (0,\psi) \prec (0,\phi \vee \psi)$
2. $(0,\phi) \prec (0,\neg\phi)$
3. $(0,\phi),\ (1,\langle p\rangle\phi) \prec (0,\langle p\rangle\phi)$
4. $(1,\langle p\rangle\phi),\ (1,\langle q\rangle\phi) \prec (1,\langle p \cup q\rangle\phi)$
5. $(1,\langle p\rangle\langle q\rangle\phi),\ (1,\langle q\rangle\phi) \prec (1,\langle p;q\rangle\phi)$
6. $(1,\langle p\rangle\langle p^*\rangle\phi) \prec (1,\langle p^*\rangle\phi)$
7. $(0,\phi) \prec (1,\langle \phi?\rangle\psi)$

It is not hard to see that the relation $\prec$ is well-founded.

Define $(i,\phi) \preceq (j,\psi)$ if either $(i,\phi) \prec (j,\psi)$ or $(i,\phi) = (j,\psi)$. The *Fischer-Ladner closure* of ϕ, denoted $FL(\phi)$, is the set

$$FL(\phi) \quad = \quad \{\psi \mid (j,\psi) \preceq (0,\phi),\ j \in \{0,1\}\}\ .$$

Let $|A|$ denote the cardinality of a set A and let $|\phi|$ and $|p|$ denote the length (number of symbols) of ϕ and p, respectively. The following lemma is easily established by induction on the well-founded relation $\prec$.

Lemma 7

1. $|\{(i,\psi) \mid (i,\psi) \preceq (0,\phi)\}| \leq |\phi|$

2. $|\{(i,\psi) \mid (i,\psi) \preceq (1,\langle p\rangle\phi)\}| \leq |p|$

It follows immediately from Lemma 7 that $|FL(\phi)| \leq |\phi|$. We also have

Lemma 8 *If $\langle p\rangle\psi \in FL(\phi)$ then $\psi \in FL(\phi)$.*

2.4.2 Filtration

Given a *PDL* proposition ϕ and a Kripke model $\mathcal{M} = (S^{\mathcal{M}}, I^{\mathcal{M}})$, we define a new model $\mathcal{M}/FL(\phi) = (S^{\mathcal{M}/FL(\phi)}, I^{\mathcal{M}/FL(\phi)})$, called the *filtration of $\mathcal{M}$ by $FL(\phi)$*, as follows. Define a binary relation $\equiv$ on states of $\mathcal{M}$ by:

$$u \equiv v \quad \text{iff} \quad \forall\psi \in FL(\phi)\ (u \in \psi^{\mathcal{M}} \leftrightarrow v \in \psi^{\mathcal{M}})\ .$$

In other words, we collapse u and v if they are not distinguishable by any formula of $FL(\phi)$. Let

$$\begin{aligned}
[u] &= \{v \mid v \equiv u\} \\
S^{\mathcal{M}/FL(\phi)} &= \{[u] \mid u \in S^{\mathcal{M}}\} \\
\phi^{\mathcal{M}/FL(\phi)} &= \{[u] \mid u \in \phi^{\mathcal{M}}\},\ \phi \text{ atomic} \\
p^{\mathcal{M}/FL(\phi)} &= \{([u],[v]) \mid (u,v) \in p^{\mathcal{M}}\},\ p \text{ atomic.}
\end{aligned}$$

$I^{\mathcal{M}/FL(\phi)}$ is extended inductively to compound propositions and programs as in §2.1.2.

The following lemma relates $\mathcal{M}$ and $\mathcal{M}/FL(\phi)$. Most of the difficulty in the following lemma is in the correct formulation of the induction hypotheses in the statement of the lemma. Once this is done, the proof is a straightforward induction on the well-founded relation $\prec$.

Lemma 9 (Filtration Lemma)

1. *For all $(0,\psi) \preceq (0,\phi)$, $u \in \psi^{\mathcal{M}}$ iff $[u] \in \psi^{\mathcal{M}/FL(\phi)}$.*

2. *For all $(1,\langle p\rangle\psi) \prec (0,\phi)$,*

   (a) *if $(u,v) \in p^{\mathcal{M}}$ then $([u],[v]) \in p^{\mathcal{M}/FL(\phi)}$;*

   (b) *if $([u],[v]) \in p^{\mathcal{M}/FL(\phi)}$ and $v \in \psi^{\mathcal{M}}$, then $u \in \langle p\rangle\psi^{\mathcal{M}}$.*

Using the filtration lemma, we can prove the small model theorem easily:

Theorem 10 (Small Model Theorem) *Let ϕ be a satisfiable formula of PDL. Then ϕ is satisfied in a model with no more than $2^{|\phi|}$ states.*

Proof. If ϕ is satisfiable, then there is a model $\mathcal{M}$ and state $u \in \mathcal{M}$ with $u \in \phi^{\mathcal{M}}$. Let $FL(\phi)$ be the Fischer-Ladner closure of ϕ. By the filtration lemma, $[u] \in \phi^{\mathcal{M}/FL(\phi)}$. Moreover, $\mathcal{M}/FL(\phi)$ has no more states than the number of truth assignments to formulas in $FL(\phi)$, which by Lemma 7(1) is at most $2^{|\phi|}$. □

2.4.3 Filtration over Nonstandard Models

We note for future use that the filtration lemma (Lemma 9) actually holds in more generality. In particular, it holds for *nonstandard models* as well as the standard models defined in §2.1.2.

Definition 11 A *nonstandard Kripke model* is any structure $\mathcal{N} = (S^{\mathcal{N}}, I^{\mathcal{N}})$ that is a Kripke model in the sense of §2.1.2 in every respect, except that $(p^*)^{\mathcal{N}}$ need not be the reflexive-transitive closure of $p^{\mathcal{N}}$, but only a reflexive, transitive relation containing $p^{\mathcal{N}}$ and satisfying the *PDL* induction axiom (Axiom 8). □

It is easily checked that all the axioms and rules of §2.2 are still valid over nonstandard models.

Lemma 12 (Filtration for Nonstandard Models) *The Filtration Lemma (Lemma 9) holds for nonstandard models.*

Proof. All cases are identical to the corresponding cases of Lemma 9, except case 2(a) for p^*. For standard models $\mathcal{M}$, the proof is straightforward, using the fact that $(p^*)^{\mathcal{M}}$ is the reflexive-transitive closure of $p^{\mathcal{M}}$. This does not hold in nonstandard models in general, so we must depend on the weaker induction axiom. We argue this case explicitly.

Let $\mathcal{N}$ be a nonstandard model, and suppose $(u, v) \in (p^*)^{\mathcal{N}}$. We wish to show that $([u], [v]) \in (p^*)^{\mathcal{N}/FL(\phi)}$, or equivalently that $u \in E$, where

$$E \quad = \quad \{w \in \mathcal{N} \mid ([w], [v]) \in (p^*)^{\mathcal{N}/FL(\phi)}\} \; .$$

Since E is a union of equivalence classes defined by truth assignments to the elements of $FL(\phi)$, there is a *PDL* formula ψ_E defining E in $\mathcal{N}$; i.e., $E = \psi_E^{\mathcal{N}}$. There is likewise a *PDL* formula $\psi_{[v]}$ defining just the equivalence class $[v]$. Since $[v] \subseteq E$,

$$\mathcal{N} \models \psi_{[v]} \rightarrow \psi_E \; . \tag{10}$$

Also,

$$\mathcal{N} \models \langle p \rangle \psi_E \rightarrow \psi_E \; , \tag{11}$$

since if $w \in (\langle p \rangle \psi_E)^{\mathcal{N}}$, then there exists a $t \in \psi_E^{\mathcal{N}} = E$ such that $(w, t) \in p^{\mathcal{N}}$; then $([w], [t]) \in p^{\mathcal{N}/FL(\phi)}$ by the induction hypothesis, and $([t], [v]) \in (p^*)^{\mathcal{N}/FL(\phi)}$, thus $([w], [v]) \in (p^*)^{\mathcal{N}/FL(\phi)}$ and therefore $w \in E$.

Combining (10) and (11), we get

$$\mathcal{N} \models (\psi_{[v]} \vee \langle p \rangle \psi_E) \rightarrow \psi_E \; .$$

Using the reflexive-transitive closure rule (Theorem 4(3)), which by Proposition 6 is equivalent to the induction axiom, we have that

$$\mathcal{N} \models \langle p^* \rangle \psi_{[v]} \rightarrow \psi_E \; .$$

But $u \in (\langle p^* \rangle \psi_{[v]})^{\mathcal{N}}$ by assumption; therefore $u \in E$. □

2.5 Deductive Completeness

Completeness of the deductive system of §2.2 was first shown independently by Gabbay [44] and Parikh [115]. A particularly easy-to-follow proof is given in [84]. Completeness is also treated in [127,12,110]. The completeness proof sketched here is from [80] and is based on the approach of [12].

Lemma 13 *Let Σ, Γ be maximal consistent sets of PDL propositions. Then the following two statements are equivalent:*

1. *for all $\psi \in \Gamma$, $\langle p\rangle\psi \in \Sigma$;*
2. *for all $[p]\psi \in \Sigma$, $\psi \in \Gamma$.*

Define a nonstandard model $\mathcal{F}$ by:

$$\begin{aligned} S^{\mathcal{F}} &= \{\text{maximal consistent sets of propositions of } PDL\} \\ \phi^{\mathcal{F}} &= \{u \in S^{\mathcal{F}} \mid \phi \in u\} \\ p^{\mathcal{F}} &= \{(u,v) \mid \forall\phi \in v\ \langle p\rangle\phi \in u\} \\ &= \{(u,v) \mid \forall[p]\phi \in u\ \phi \in v\} \end{aligned}$$

The two definitions of $p^{\mathcal{F}}$ are equivalent, by Lemma 13.

Proposition 14 *$\mathcal{F}$ is a nonstandard Kripke model in the sense of Definition 11.*

Theorem 15 (Completeness of *PDL*) *If $\models \phi$ then $\vdash \phi$.*

Proof. Equivalently, we need to show that if ϕ is consistent, then it is satisfied in a standard Kripke model. If ϕ is consistent, then by Zorn's Lemma it is contained in a maximal consistent set u, which is a state of the nonstandard model $\mathcal{F}$. By the filtration lemma for nonstandard models (Lemma 12), ϕ is satisfied at state $[u]$ in the finite standard model $\mathcal{F}/FL(\phi)$. □

2.5.1 Logical Consequences

In classical logics, a completeness theorem of the form of Theorem 15 is easily adapted to handle the relation of logical consequence $\phi \models \psi$ between formulas, since usually

$$\phi \models \psi \quad \text{iff} \quad \models \phi \rightarrow \psi \tag{12}$$

Unfortunately, (12) fails in *PDL*, as can be seen by taking $\psi = [p]\phi$. However, the following result allows Theorem 15, as well as Algorithm 17 of §2.6.1 below, to be extended to handle logical consequence:

Proposition 16 *Let ϕ, ψ be any PDL formulas. Then*

$$\phi \models \psi \quad \textit{iff} \quad \models [(p_1 \cup \ldots \cup p_n)^*]\phi \rightarrow \psi\ ,$$

where $p_1, \ldots, p_n$ are all atomic programs appearing in ϕ or ψ.

It is shown in [99] that the problem of deciding whether $\Sigma \models \psi$, where Σ is a fixed r.e. set of *PDL* formulas, is Π^1_1-complete.

2.6 Complexity of the Satisfiability Problem for *PDL*

2.6.1 A Deterministic Exponential-Time Algorithm

The naive algorithm for the satisfiability problem that comes from the small model theorem is double-exponential time in the worst case. Here we develop an algorithm that runs in deterministic time $2^{O(|\phi|)}$. In fact, the problem is deterministic exponential-time complete (see §2.6.2), so it is unlikely that a significantly more efficient algorithm will be found. Deterministic exponential-time algorithms were first given by Pratt [127,129]. The algorithm given here is from [129].

The algorithm constructs the small model $\mathcal{N} = \mathcal{F}/FL(\phi)$ obtained in the completeness proof of §2.5 explicitly. If ϕ is satisfiable, then it is consistent, by the soundness of the deductive system of §2.2; then ϕ will be satisfied at some state u of $\mathcal{F}$, and hence at the state $[u]$ of $\mathcal{N}$.

We start with the set S of all truth assignments $u : FL(\phi) \rightarrow \{\mathit{true}, \mathit{false}\}$. By the construction of $\mathcal{N}$, the states of $\mathcal{N}$ are in one-to-one correspondence with the *consistent* truth assignments to $FL(\phi)$, thus we may consider $S^{\mathcal{N}}$ to be a subset of S; i.e., for all $\psi \in FL(\phi)$ and $u \in S^{\mathcal{N}}$,

$$u \in \psi^{\mathcal{N}} \quad \leftrightarrow \quad u(\psi) = \mathit{true} \ . \tag{13}$$

We will approximate $\mathcal{N}$ with a sequence of models $\mathcal{N}_i = (S_i, I_i)$, $i \geq 0$, such that

$$S \supseteq S_0 \supseteq S_1 \supseteq \cdots \supseteq S^{\mathcal{N}} \ ,$$

obtained by deleting from S any truth assignments that we can determine to be inconsistent. When we are done, we will be left with the model $\mathcal{N}$.

The interpretations of the primitive formulas and programs in the models $\mathcal{N}_i$ will be defined in the same way for all i:

$$\psi^{\mathcal{N}_i} \quad = \quad \{u \in S_i \mid u(\psi) = \mathit{true}\} \ , \ \psi \text{ atomic} \tag{14}$$

$$p^{\mathcal{N}_i} \quad = \quad \{(u,v) \in S_i^2 \mid u(\langle p \rangle \psi) = \mathit{true} \text{ whenever } v(\psi) = \mathit{true}\} \ , \ p \text{ atomic} \ . \tag{15}$$

Algorithm 17

1. Construct S.

2. For each $u \in S$, check whether u respects Axioms 1, 4, 5, and 6 of §2.2 and Theorem 2.3(13) of §2.3, all of which can be checked locally. For example, to check Axiom 4, which says

$$\langle p \cup q \rangle \psi \quad \leftrightarrow \quad \langle p \rangle \psi \vee \langle q \rangle \psi \ ,$$

check that $u(\langle p \cup q \rangle \psi) = \mathit{true}$ if and only if either $u(\langle p \rangle \psi) = \mathit{true}$ or $u(\langle q \rangle \psi) = \mathit{true}$. Let S_0 be the set of all $u \in S$ passing this test. The model $\mathcal{N}_0$ is defined by (14) and (15) above.

3. Repeat the following for $i = 0, 1, 2, \ldots$ until no more u are deleted: find a $\preceq$-minimal $(1, \langle p \rangle \psi) \preceq (0, \phi)$ and $u \in S_i$ violating the property

$$u(\langle p \rangle \psi) = true \quad \rightarrow \quad \exists v \ (u, v) \in p^{\mathcal{N}_i} \text{ and } v(\psi) = true \ . \tag{16}$$

Delete u from S_i to get S_{i+1}. □

The correctness of this algorithm will follow from the following lemma (cf. Lemma 9):

Lemma 18 *Let $(j, \xi) \preceq (0, \phi)$ be such that every $(1, \langle p \rangle \psi) \prec (j, \xi)$ and $u \in S_i$ satisfy (16).*

1. *For all $(0, \psi) \preceq (j, \xi)$ and $u \in S_i$, $u(\psi) = true$ iff $u \in \psi^{\mathcal{N}_i}$.*
2. *For all $(1, \langle p \rangle \psi) \preceq (j, \xi)$ and $u, v \in S_i$,*
   (a) *if $(u, v) \in p^{\mathcal{N}}$ then $(u, v) \in p^{\mathcal{N}_i}$;*
   (b) *if $(u, v) \in p^{\mathcal{N}_i}$ and $v(\psi) = true$, then $u(\langle p \rangle \psi) = true$.*

Proof. By induction on $\prec$. □

Since every $u \in S^{\mathcal{N}}$ passes the test of Step 2 of the algorithm, $S^{\mathcal{N}} \subseteq S_0$; and (13) and Lemma 18(2(a)) imply that no $u \in S^{\mathcal{N}}$ is ever deleted in Step 3, since for $u \in S^{\mathcal{N}}$,

$$\begin{aligned} u(\langle p \rangle \psi) = true \quad & \rightarrow \quad u \in (\langle p \rangle \psi)^{\mathcal{N}} \\ & \rightarrow \quad \exists v \ (u, v) \in p^{\mathcal{N}} \text{ and } v \in \psi^{\mathcal{N}} \\ & \rightarrow \quad \exists v \ (u, v) \in p^{\mathcal{N}_i} \text{ and } v(\psi) = true \ . \end{aligned}$$

Thus

$$S^{\mathcal{N}} \subseteq S_i \ , \ i \geq 0 \ .$$

Moreover, when the algorithm terminates with the model $\mathcal{N}_n$, then by Lemma 18(1), every $u \in S_n$ (viewed as a truth assignment) is satisfiable, since it is satisfied by the state u in the model $\mathcal{N}_n$; thus $S_n \subseteq S^{\mathcal{N}}$. We can now test the satisfiability of ϕ by checking whether $u(\phi) = true$ for some $u \in \mathcal{N}_n$.

Algorithm 17 can be programmed to run in exponential time without much difficulty. The efficiency can be further improved by observing that the p in the $\preceq$-minimal $(1, \langle p \rangle \psi)$ violating (16) in Step 3 must be either atomic or of the form q^*, because of the preprocessing in Step 2. This follows easily from Lemma 18. We have shown:

Theorem 19 *There is an exponential-time algorithm for deciding whether a given formula of PDL is satisfiable.*

As previously noted, Proposition 16 of §2.5.1 allows this algorithm to be adapted to test whether one formula is a logical consequence of another.

2.6.2 A Lower Bound

In [40,41], it is shown how *PDL* formulas can encode computations of linear-space-bounded alternating Turing machines. It follows from [18] that the satisfiability problem for *PDL* is deterministic exponential-time hard, therefore requires at least deterministic time $2^{\Omega(n)}$ on formulas of size n.

The satisfiability problem for *PDL* is thus exponential-time complete. In contrast, the satisfiability problem for classical propositional logic is *NP*-complete.

2.7 Variants of *PDL*

A number of interesting variants are obtained by extending or restricting *PDL* in various ways. In this section we describe some of these variants and review some of the known results concerning relative expressive power, complexity, and proof theory.

2.7.1 Deterministic *PDL* and While Programs

A program p is said to be *(semantically) deterministic* in $\mathcal{M}$ if its traces are uniquely determined by their first states. If p is atomic, this is equivalent to the requirement that $p^{\mathcal{M}}$ be a partial function. The class of *deterministic* **while** *programs*, denoted DWP, is the class of programs in which

1. the operators $\cup$, ?, and $*$ may appear only in the context of the conditional test, **while** loop, **skip**, or **fail**;

2. tests in the conditional test and **while** loop are purely propositional (i.e., there is no occurrence of the $\langle\ \rangle$ operator).

The class of *nondeterministic* **while** *programs*, denoted WP, is the same, except unconstrained use of the nondeterministic choice construct $\cup$ is allowed. It is easily shown that if p and q are semantically deterministic in $\mathcal{M}$, then so are **if** ϕ **then** p **else** q **fi** and **while** ϕ **do** p **od**.

Definition 20 We define *PDL*(DWP) to be the syntactically and semantically constrained version of *PDL* in which

- only deterministic **while** programs are allowed;
- every atomic program is semantically deterministic.

□

(This version of *PDL* is sometimes called *strict deterministic PDL* in the literature.)

If ϕ is valid in *PDL*, then ϕ is also valid in *PDL*(DWP), but not conversely: the formula

$$\langle p\rangle\phi \rightarrow [p]\phi \tag{17}$$

is valid in *PDL*(DWP) but not in *PDL*. Also, *PDL*(DWP) is strictly less expressive than *PDL*, since the formula

$$\langle (p \cup q)^*\rangle\phi \tag{18}$$

is not expressible in *PDL*(DWP) [53].

Unlike *PDL*, the satisfiability problem for *PDL*(DWP) is *PSPACE*-complete [53].

If the deductive system of §2.2 is modified so as to refer only to deterministic **while** programs, and if axiom (17) is included, the resulting system is sound and complete for *PDL*(DWP) [8].

Related results are obtained in [12,157,8].

2.7.2 Rich and Poor Tests

Tests $\phi?$ in *PDL* are defined for arbitrary propositions ϕ. This is called *rich-test*. This is substantially more power than one would find in a conventional programming language. Indeed, in the first-order version over the natural numbers, rich test allows undecidable problems to be decided in one step. *Poor-test PDL*, on the other hand, can only test atomic propositions.

The exponential-time hardness result described in §2.6.2 still holds for poor-test *PDL*, since the construction does not require tests. However, it can be shown [124,11,14] that rich-test *PDL* is strictly more expressive than poor-test *PDL*, which in turn is strictly more expressive than test-free *PDL*. These results also hold for *PDL(DWP)* (see §2.7.1). In fact, the formula 2.7.1(18) of test-free *PDL* is not expressible in *PDL(DWP)*.

2.7.3 Context-Free Programs

A *PDL* program can be viewed as a regular expression denoting the set of its computation sequences (§2.1.3). The set of computation sequences is thus a regular set in the sense of the theory of formal languages and automata (see [71]).

More complex sets of computation sequences can be allowed as programs to obtain more expressive versions of *PDL*. In particular, *context-free PDL*, denoted PDL_{cf}, is obtained by allowing programs to be context-free sets of computation sequences. This corresponds to a syntax allowing recursive calls without parameters. For example, the set

$$\{p^n; q^n \mid n \geq 0\} \tag{19}$$

where p and q are atomic, is a program of PDL_{cf} but not of *PDL*. Programs may be built compositionally using a context-free grammar-like syntax, or represented by pushdown automata accepting sets of computation sequences.

The satisfiability problem for PDL_{cf} was shown undecidable by Ladner [unpublished]; the problem was shown to be Π_1^1-complete in [62], even for *PDL* extended with the single context-free program (19). Related results can be found in [63,61].

2.7.4 Automata Theory and Program Logics

A *PDL* program represents a regular set of computation sequences. This same regular set could possibly be represented exponentially more succinctly by a finite automaton. The difference between these two representations corresponds roughly to the difference between **while** programs and flowcharts.

Since finite automata are exponentially more succinct in general, the complexity results of §2.6 could conceivably fail if finite automata were allowed as programs. Moreover, we must also rework the deductive system of §2.2.

However, it turns out that the completeness and exponential-time decidability results of *PDL* are not sensitive to the representation, and still go through in the presence of finite automata as programs, provided the deductive system of §2.2 and the techniques of §§2.4-2.6 are suitably modified [129,130,66].

In very recent years, the automata-theoretic approach to logics of programs has yielded significant insight into propositional logics more powerful than *PDL*, as well as substantial reductions in the complexity of their decision procedures. Especially enlightening are the connections with automata on infinite strings and infinite trees: by viewing a formula as an automaton and a tree-like model as an input to that automaton, the satisfiability problem for a given formula becomes the emptiness problem for a given automaton. Logical questions are thereby transformed into purely automata-theoretic questions.

This connection has prompted renewed inquiry into the complexity of automata on infinite objects, with considerable success [29,31,34,36,96,122,135,141,145,158,159,160,161, 163,164]. Especially noteworthy in this area is the recent result of Safra [135] involving the complexity of converting a nondeterministic automaton on infinite strings to an equivalent deterministic one. This result has already had a significant impact on the complexity of decision procedures for several logics of programs [29,34,35,135].

2.7.5 Converse

The *converse operator* $^-$ is a program operator which allows a program to be "run backwards":

$$(p^-)^{\mathcal{M}} \quad = \quad \{(s,t) \mid (t,s) \in p^{\mathcal{M}}\} \ .$$

This operator strictly increases the expressive power of *PDL*, since the formula $\langle p^-\rangle\phi$ is not expressible without it. More interestingly, the presence of the converse operator implies that the operator $\langle p\rangle$ is *continuous* in the sense that if Φ is any (possibly infinite) family of formulas possessing a join $\bigvee \Phi$, then $\bigvee \langle p\rangle \Phi$ exists and is logically equivalent to $\langle p\rangle \bigvee \Phi$ [154]. In the absence of the converse operator, there exist nonstandard models for which this fails.

The completeness and single-exponential decidability results of §2.5 and §2.6.1 can be extended to *PDL* with converse [115] provided the following two axioms are added:

$$\begin{aligned} \phi &\rightarrow [p]\langle p^-\rangle\phi \\ \phi &\rightarrow [p^-]\langle p\rangle\phi \ . \end{aligned}$$

The filtration lemma (Lemma 9) still holds in the presence of $^-$, as does the finite model property.

The complexity of *PDL* with converse and various forms of well-foundedness constructs (see §2.7.6 below) is studied in [159].

2.7.6 Well-Foundedness and Total Correctness

If p is a deterministic program, the formula $\phi \rightarrow \langle p\rangle\psi$ asserts the total correctness of p with respect to pre- and postconditions ϕ and ψ, respectively. For *nondeterministic* programs, however, this formula does not express total correctness: it asserts that if ϕ then *there exists* a halting computation sequence of p yielding ψ, whereas we would really like to assert that if ϕ then *all* computation sequences of p terminate and yield ψ. Unfortunately, this property is not expressible in *PDL*.

The problem is essentially concerned with *well-foundedness*. A program p is said to be *well-founded* at a state u_0 if there exists no infinite sequence $u_0, u_1, u_2, \ldots$ with $(u_i, u_{i+1}) \in p^{\mathcal{M}}$ for all $i \geq 0$.

Several very powerful logics have been proposed to deal with this situation. The most powerful is perhaps the propositional μ-calculus [138,69,120,132,81,83,85,113,146,161]. The version of this logic given in [81] is essentially propositional modal logic with a least fixpoint operator μ, which allows syntactic expression of any property that can be formulated as the least fixpoint of a monotone transformation. The well-foundedness of p is expressed

$$\mu x.[p]x \tag{20}$$

in this logic.

Two somewhat weaker ways of capturing well-foundedness without resorting to the full μ-calculus have been studied. One is to add an explicit predicate **wf** for well-foundedness. Another is to add an explicit predicate **halt**, which asserts that all computations of its argument p terminate. These constructs have been investigated in [65,67,113,144,145,146] under the various names **loop**, **repeat**, and Δ.

The predicate **halt** can be defined inductively from **wf**, as follows:

- $\mathbf{halt}(p)$, if p is an atomic program or test,
- $\mathbf{halt}(p;q) \leftrightarrow \mathbf{halt}(p) \wedge [p]\mathbf{halt}(q)$,
- $\mathbf{halt}(p \cup q) \leftrightarrow \mathbf{halt}(p) \wedge \mathbf{halt}(q)$,
- $\mathbf{halt}(p^*) \leftrightarrow [p^*]\mathbf{halt}(p) \wedge \mathbf{wf}(p)$.

The propositional μ-calculus is strictly more expressive than *PDL* with **wf** [113,146], which is strictly more expressive than *PDL* with **halt** [67], which is strictly more expressive than *PDL* [144].

The filtration lemma fails for the propositional μ-calculus and for *PDL* with **halt** or **wf**, as can be seen by considering the model $\mathcal{M} = (S^{\mathcal{M}}, I^{\mathcal{M}})$ with

$$S^{\mathcal{M}} \;=\; \{(i,j) \in \mathcal{N}^2 \mid 0 \leq j \leq i\} \cup \{u\}$$

and atomic program p with

$$I^{\mathcal{M}}(p) \;=\; \{((i,j),(i,j-1)) \mid 1 \leq j \leq i\} \cup \{(u,(i,i)) \mid i \in \mathcal{N}\}$$

The state u satisfies $\mathbf{halt}(p^*)$ and $\mathbf{wf}(p)$, but $[u]$ does not satisfy this formula in any finite filtrate. Despite this failure, these logics do satisfy the finite model property [144,145,83]. In the presence of the converse operator, the finite model property fails, since the formula

$$\neg\mathbf{halt}(p^*) \wedge [p^*]\mathbf{halt}(p^{-*}),\; p \text{ atomic}$$

is satisfiable but has no finite model. All these logics are decidable [144,145,85,161,159]; very recently, they have been shown to be deterministic single-exponential time complete [34,135].

There is no known complete axiomatization for *PDL* with **halt** or **wf**, or for the propositional μ-calculus. A completeness result for a syntactically restricted version of the μ-calculus which includes the formula (20) for p a deterministic **while** program is given in [81], and a complete infinitary deductive system is given in [83].

2.7.7 Other Work

Additional topics related to *PDL*, which space does not permit us to discuss in depth, include work on complementation and intersection of programs [64], nonstandard models [12,13,76,77,129,115], Dynamic Algebra [76,77,78,129,131], Process Logic [116,128,111,58, 162], *PDL* with Boolean assignments [1], restricted forms of the consequence problem [118], concurrency [123], and probabilistic programs [79,134,82,119,29,68,158]. We also refer the reader to Harel's survey [56], which covers many of these topics in more detail.

3 First-Order Dynamic Logic

In this section we define various forms of first-order Dynamic Logic (DL) and discuss their syntax, semantics, proof theory, and expressiveness. The main difference between first-order DL and the propositional version discussed in §2 is the presence of a first-order structure $\mathcal{A}$, called the *domain of computation*, over which first-order quantification is allowed. States are no longer abstract points, but *valuations* of a set of variables over $\mathcal{A}$; primitive programs are no longer abstract binary relations, but *assignments* of the form $x := t$, for example, where x is a variable and t is a term; and primitive assertions are first-order formulas.

3.1 Syntax

Let $L = (\ldots, f, \ldots, R, \ldots)$ be a finite first-order language with equality. Here f and R denote generic function and relation symbols of L, respectively. Each function and relation symbol of L comes with a fixed *arity* (number of inputs). Let $V = \{x_0, x_1, \ldots\}$ be a countable set of *individual variables*. The metasymbols $s, t, \ldots$ range over terms of L.

There are several versions of DL that we will discuss, depending on the choice of primitive constructs. In general, these logics are similar to the propositional version introduced in §2, with the following key exceptions:

- Primitive programs are assignment statements of the form

$$x := t \ , \tag{21}$$

  for example. Here $x \in V$ and t is a term of L; this form of assignment is called a *simple assignment*.

- Primitive assertions are atomic formulas of L, i.e. formulas of the form

$$R(t_1, \ldots, t_n)$$

  where R is an n-ary relation symbol of L and $t_1, \ldots, t_n$ are terms of L.

- In the inductive definition of formulas, we include the clause:

  if ϕ is a formula, then so is $\exists x\ \phi$, where $x \in V$.

Otherwise, compound programs and formulas are formed exactly as in §2.1.1, using the connectives ; (sequential composition), $\cup$ (nondeterministic choice), * (iteration), ? (test), $\langle\ \rangle$ (modal possibility), $\vee$ (propositional disjunction), and $\neg$ (propositional negation).

The class R of *regular programs* contains all programs formed from simple assignments (21), $\cup$, ;, *, and ?, in which any formula ϕ appearing in a test ϕ? must be a quantifier-free first-order formula. For much of the sequel, we will be concerned with **while** *programs* only. The class of *deterministic* **while** *programs*, denoted DWP, is the subclass of R in which the program operators $\cup, ?$, and * are constrained to appear only in the forms

$$\begin{aligned} \textbf{skip} &= \mathit{true}? \\ \textbf{fail} &= \mathit{false}? \\ \textbf{if } \phi \textbf{ then } p \textbf{ else } q \textbf{ fi} &= \phi?; p \cup \neg\phi?; q \\ \textbf{while } \phi \textbf{ do } p \textbf{ od} &= (\phi?; p)^*; \neg\phi? \ . \end{aligned}$$

The class of *(nondeterministic)* **while** *programs*, denoted WP, is the same, except that we allow unrestricted use of the nondeterministic choice construct $\cup$.

The definitions of R, WP, and DWP depend on the language L, but to save notation, we do not make this dependence explicit.

3.1.1 Arrays and Stacks

We will eventually want to discuss the power of auxiliary data structures such as *arrays* and *stacks*, as well as a powerful assignment statement called the *nondeterministic assignment*. We introduce the syntactic machinery now so that we can give the semantics of these constructs all at once in the next section.

To handle arrays, we include a countable set of *array variables*

$$V_{array} = \{F_0, F_1, \ldots\} \ .$$

Each array variable has an associated *arity*, or number of inputs, which we do not represent explicitly. We assume that there are countably many variables of each arity $n \geq 0$. In the presence of array variables, we equate V with the nullary array variables; thus $V \subseteq V_{array}$. The variables in V_{array} of arity n will range over n-ary functions with arguments and values in the domain of computation. In our exposition, elements of the domain of computation play two roles: they are used as *indices* into an array, and as *values* which can be stored in an array. One might equally well introduce a separate sort for array indices; although conceptually simple, this would complicate notation and would give no new insight.

The classes DWP_{array} and WP_{array} of *deterministic* and *nondeterministic* **while** *programs with arrays* are defined similarly to DWP and WP, respectively, except that in addition to simple assignments, we allow *array assignments*. These are similar to simple assignments, except that on the left-hand-side we allow a term in which the outermost symbol is an array variable:

$$F(t_1, \ldots, t_m) := t \ .$$

Here F is an m-ary array variable and $t_1, \ldots, t_m, t$ are terms, possibly involving other array variables. Note that when $m = 0$, this reduces to the ordinary simple assignment.

To handle *stacks* or *pushdown stores*, we introduce *stack variables* σ and two new atomic programs:

$$push(\sigma, t) \qquad\qquad pop(\sigma, y)$$

where t is a term and $y \in V$. Intuitively, σ represents a stack, the *push* operations pushes the current value of t onto the top of the stack σ, and the *pop* operation pops the top value off the top of the stack σ and assigns that value to the variable y. Formally, σ will range over finite strings of elements of the domain of computation. The classes DWP_{pds} and WP_{pds} are obtained by augmenting the classes of deterministic and nondeterministic **while** programs, respectively, with *one* stack variable σ, and allowing the *push* and *pop* operations above as atomic programs in addition to simple assignments. We emphasize that the programs in DWP_{pds} and WP_{pds} may use *only one* stack—the results change dramatically when two or more stacks are allowed.

If we allow both a stack and arrays in deterministic and nondeterministic **while** programs, we obtain the programming languages $DWP_{array+pds}$ and $WP_{array+pds}$, respectively.

3.1.2 Nondeterministic Assignment

The *nondeterministic assignment*

$$x := ?$$

is a device that arises in the study of fairness [5]. It has often been called *random assignment* in the literature, although we prefer the name *nondeterministic assignment*, since it has nothing to do with randomness or probability. Intuitively, it operates by assigning a nondeterministically chosen element of the domain of computation to the variable x. This construct may be considered an extension of the first-order existential quantifier, in the sense that the two formulas

$$\langle x := ?\rangle\phi \qquad\qquad \exists x\ \phi$$

are equivalent. However, the nondeterministic assignment is (at least superficially) more powerful, since it may be iterated.

3.1.3 Rich and Poor Tests

The variants of *DL* we have discussed may all be described as "open-test" or "poor-test". This means that only quantifier-free tests are allowed in the **if-then-else** and **while-do** (cf. §2.7.2). One may allow arbitrary *DL* formulas in these tests to get the "rich-test" versions. These versions are discussed briefly in §3.7.4.

3.2 Semantics

In this section, we assign meanings to all the syntactic constructs described above.

Let $\mathcal{A} = (|\mathcal{A}|, \ldots, f^{\mathcal{A}}, \ldots, R^{\mathcal{A}}, \ldots)$ be a first-order structure for the language L. We call $\mathcal{A}$ the *domain of computation.* Here $|\mathcal{A}|$ is a set, called the *carrier* of $\mathcal{A}$, $f^{\mathcal{A}}$ is an n-ary function $f^{\mathcal{A}} : |\mathcal{A}|^n \rightarrow |\mathcal{A}|$ interpreting the n-ary function symbol f of L, and $R^{\mathcal{A}}$ is an n-ary relation $R^{\mathcal{A}} \subseteq |\mathcal{A}|^n$ interpreting the n-ary relation symbol R of L. (The equality symbol $=$ is always interpreted as the identity relation.) We henceforth dispense with the vertical bars and use the notation $\mathcal{A}$ for both the structure and its carrier.

For $n \geq 0$, let $(\mathcal{A}^n \to \mathcal{A})$ denote the set of all functions $\mathcal{A}^n \to \mathcal{A}$. By convention, we take $(\mathcal{A}^0 \to \mathcal{A}) = \mathcal{A}$. Let $\mathcal{A}^*$ denote the set of all finite-length strings over $\mathcal{A}$.

The structure $\mathcal{A}$ determines a Kripke model, which we will also denote by $\mathcal{A}$, as follows. A *valuation* over $\mathcal{A}$ is a function u assigning an n-ary function over $\mathcal{A}$ to each n-ary array variable, and a finite-length string of elements of $\mathcal{A}$ to each stack variable. That is:

$$\begin{array}{rcll} u(F) & \in & (\mathcal{A}^n \to \mathcal{A}) \ , & \text{if } F \text{ is an } n\text{-ary array variable,} \\ u(\sigma) & \in & \mathcal{A}^* \ , & \text{if } \sigma \text{ is a stack variable.} \end{array}$$

Under the convention $(\mathcal{A}^0 \to \mathcal{A}) = \mathcal{A}$, and assuming that $V \subseteq V_{array}$, the individual variables (i.e., the nullary array variables) are assigned elements of $\mathcal{A}$ under this definition:

$$u(x) \in \mathcal{A} \ , \ \text{if } x \in V \ .$$

The valuation u extends uniquely to terms t by induction:

$$\begin{array}{rcll} u(f(t_1,\ldots,t_m)) & = & f^{\mathcal{A}}(u(t_1),\ldots,u(t_m)) \ , & \text{if } f \text{ is an } m\text{-ary function symbol,} \\ u(F(t_1,\ldots,t_m)) & = & u(F)(u(t_1),\ldots,u(t_m)) \ , & \text{if } F \text{ is an } m\text{-ary array variable.} \end{array}$$

If x is a variable of any type and a is an object of the same type, we denote by $u[x/a]$ the new valuation obtained from u by changing the value of x to a, and leaving the values of all other variables intact. For example, if F is an m-ary array variable and $f : \mathcal{A}^m \to \mathcal{A}$, then $u[F/f]$ is the new valuation which assigns the same value as u to all stack variables and array variables other than F, and

$$u[F/f](F) \ = \ f \ .$$

If $f : \mathcal{A}^m \to \mathcal{A}$ is an m-ary function and $a_1,\ldots,a_m,a \in \mathcal{A}$, we denote by $f[a_1,\ldots,a_m/a]$ the m-ary function that agrees with f everywhere except input $a_1,\ldots,a_m$, on which it takes the value a. I.e.,

$$f[a_1,\ldots,a_m/a](b_1,\ldots,b_m) \ = \ \begin{cases} a \ , & \text{if } b_i = a_i \ , \ 1 \leq i \leq m \\ f(b_1,\ldots,b_m) \ , & \text{otherwise.} \end{cases}$$

Definition 21 We call valuations u and v *finite variants* of each other if

1. $u(F) = v(F)$ for all but finitely many array variables F;
2. for all array variables F of positive arity n,

   $$u(F)(a_1,\ldots,a_n) \ = \ v(F)(a_1,\ldots,a_n)$$

   for all but finitely many n-tuples $a_1,\ldots,a_n \in \mathcal{A}^n$; in other words, u and v may differ on only finitely many array variables, and for those F on which they do differ, the functions $u(F)$ and $v(F)$ may differ on only finitely many values;

3. for all but finitely many stack variables σ, $u(\sigma) = v(\sigma)$.

□

The relation "is a finite variant of" is an equivalence relation on valuations. Since a halting computation can run for only a finite time and therefore execute only finitely many assignments, it will not be able to cross equivalence class boundaries; i.e., in the binary relation semantics given below, if the pair (u, v) is an input/output pair of the program p, then v is a finite variant of u.

We are now ready to define the *states* of our Kripke model.

Definition 22 Let $a \in \mathcal{A}$. Let w_a be the valuation in which all array and individual variables are interpreted as constant functions taking the value a everywhere, and all stacks are empty. A *state* is any valuation that is a finite variant of w_a for some a. The set of states of $\mathcal{A}$ is denoted $S^{\mathcal{A}}$. □

It is meaningful, and indeed useful in some contexts, to take as states the set of all valuations. Our purpose in restricting our attention to states as defined in Definition 22 is to avoid highly complex oracles that might compromise the value of the relative expressiveness results below.

As in §2.1.2, with every program p, we associate a binary relation $p^{\mathcal{A}} \subseteq S^{\mathcal{A}} \times S^{\mathcal{A}}$ (the *input/output relation* of p), and with every formula ϕ, we associate a set $\phi^{\mathcal{A}} \subseteq S^{\mathcal{A}}$. The sets $p^{\mathcal{A}}$ and $\phi^{\mathcal{A}}$ are defined by mutual induction on the structure of p and ϕ.

For the basis of this inductive definition, we first give the semantics of all the assignment statements discussed in §3.1.

1. The array assignment $F(t_1, \ldots, t_m) := t$ is interpreted as the binary relation

$$(F(t_1, \ldots, t_m) := t)^{\mathcal{A}} \quad = \quad \{(u, u[F/u(F)[u(t_1), \ldots, u(t_m)/u(t)]]) \mid u \in S^{\mathcal{A}}\} \ .$$

In other words, the array assignment has the effect of changing the value of F on input $u(t_1), \ldots, u(t_m)$ to $u(t)$, and leaving the value of F on all other inputs and the values of all other variables intact. For $m = 0$, this definition reduces to the following definition of simple assignment:

$$(x := t)^{\mathcal{A}} \quad = \quad \{(u, u[x/u(t)]) \mid u \in S^{\mathcal{A}}\} \ .$$

2. The stack operation $push(\sigma, t)$ is interpreted as the binary relation

$$push(\sigma, t)^{\mathcal{A}} \quad = \quad \{(u, u[\sigma/(u(t) \cdot u(\sigma))]) \mid u \in S^{\mathcal{A}}\} \ .$$

In other words, this operation changes the value of σ from $u(\sigma)$ to the string $u(t) \cdot u(\sigma)$, the concatenation of the value $u(t)$ with the string $u(\sigma)$.

3. The stack operation $pop(\sigma, x)$ is interpreted as the binary relation

$$pop(\sigma,t)^{\mathcal{A}} \;=\; \{(u, u[\sigma/tail(u(\sigma))][x/head(u(\sigma), u(x))]) \mid u \in S^{\mathcal{A}}\} \;,$$

where

$$\begin{aligned} tail(a\cdot\alpha) &= \alpha \\ tail(\epsilon) &= \epsilon \\ head(a\cdot\alpha, b) &= a \\ head(\epsilon, b) &= b \end{aligned}$$

where ϵ is the null string. In other words, if $u(\sigma) \neq \epsilon$, this operation changes the value of σ from $u(\sigma)$ to the string obtained by deleting the first element of $u(\sigma)$, and assigns that element to the variable x; if $u(\sigma) = \epsilon$, then nothing is changed.

4. The nondeterministic assignment $x := ?$ for $x \in V$ is interpreted as the relation

$$(x := ?)^{\mathcal{A}} \;=\; \{(u, u[x/a]) \mid u \in S^{\mathcal{A}},\; a \in \mathcal{A}\} \;.$$

The meanings of the primitive constructs $\cup$ (nondeterministic choice), ; (sequential composition), * (iteration), ? (test), $\langle\;\rangle$ (modal possibility), $\vee$ (propositional disjunction), and $\neg$ (propositional negation) are exactly as in §2.1.2, as are those of the defined constructs **skip**, **fail**, **if-then-else**, **while-do**, [ ], etc.

We consider the first-order quantifier $\exists$ a defined construct:

$$\exists x\, \phi \;\leftrightarrow\; \langle x := ?\rangle\phi \;.$$

Thus,

$$\begin{aligned} (\exists x\, \phi)^{\mathcal{A}} &= (\langle x := ?\rangle\phi)^{\mathcal{A}} \\ &= \{(u, u[x/a]) \mid u \in S^{\mathcal{A}},\; a \in \mathcal{A}\} \circ \phi^{\mathcal{A}} \\ &= \{u \mid \exists a \in A\; u[x/a] \in \phi^{\mathcal{A}}\} \;. \end{aligned}$$

The universal quantifier is then given by

$$\begin{aligned} \forall x\, \phi &\leftrightarrow \neg\exists x\, \neg\phi \\ &\leftrightarrow \neg\langle x := ?\rangle\neg\phi \\ &\leftrightarrow [x := ?]\phi \;. \end{aligned}$$

Note that for *deterministic* programs p, $p^{\mathcal{A}}$ is single-valued, thus a partial function from states to states. The partiality of $p^{\mathcal{A}}$ arises from the possibility that p may diverge when starting its computation in certain states. For example, (**while** *true* **do skip od**)$^{\mathcal{A}}$ is the empty relation. For nondeterministic programs p, the relation $p^{\mathcal{A}}$ need not be single-valued.

If $\mathcal{A}$ is a structure and u is a state of $\mathcal{A}$, the pair $(\mathcal{A}, u)$ is called an *interpretation*. As in §2.1.2, we write $\mathcal{A}, u \models \phi$ for $u \in \phi^{\mathcal{A}}$ and say that u *satisfies* ϕ *in* $\mathcal{A}$. We may write $u \models \phi$ when $\mathcal{A}$ is understood. We write $\mathcal{A} \models \phi$ if $\mathcal{A}, u \models \phi$ for all u in $\mathcal{A}$, and we write $\models \phi$ and say that ϕ is *valid* if $\mathcal{A} \models \phi$ for all $\mathcal{A}$. We say that ϕ is *satisfiable* if $\mathcal{A}, u \models \phi$ for some $(\mathcal{A}, u)$. If Σ is a set of propositions, we write $\mathcal{A} \models \Sigma$ if $\mathcal{A} \models \phi$ for all $\phi \in \Sigma$. A proposition ψ is said to be a *logical consequence* of Σ if for all structures $\mathcal{A}$, $\mathcal{A} \models \psi$ whenever $\mathcal{A} \models \Sigma$. We say that an inference rule

$$\frac{\Sigma}{\psi}$$

is *sound* if ψ is a logical consequence of Σ.

In particular, $\mathcal{A}, u \models \langle p \rangle \phi$ iff there exists a computation of p starting in state u and terminating in a state satisfying ϕ, and $\mathcal{A}, u \models [p]\phi$ iff every terminating computation of p starting in state u terminates in a state satisfying ϕ. For a pure first-order formula ϕ, the metastatement $\mathcal{A}, u \models \phi$ has the same meaning as in first-order logic (see, e.g., [19]).

As noted in §2.3.2, *DL* subsumes Hoare logic [70]. If p is a program and ϕ and ψ are pure first-order formulas, the classical Hoare partial correctness formula $\{\phi\}p\{\psi\}$ of program p with respect to *precondition* ϕ and *postcondition* ψ is expressed $\phi \rightarrow [p]\psi$. The corresponding *total correctness formula* (for deterministic programs only—see §2.7.6) is expressed $\phi \rightarrow \langle p \rangle \psi$.

If K is a given subset of the syntactic constructs introduced in §3.1, we refer to the version of Dynamic Logic built from these constructs as *Dynamic Logic over K*, and denote this logic by $DL(K)$. In particular, we henceforth adopt the following abbreviations:

$$\begin{aligned} DL &= DL(WP) \\ DDL &= DL(DWP) \\ DL_{array} &= DL(WP_{array}) \\ DL_{array+pds} &= DL(WP_{array+pds}) . \end{aligned}$$

3.3 Complexity

3.3.1 The Validity Problem

First, we discuss the complexity of the validity problem for three fragments of Dynamic Logic: full DL, partial correctness formulas, and total correctness formulas. These results give us the first estimate of what can be expected from formal proof systems designed for the above three fragments. The results are stated for nondeterministic **while** programs, but remain true for more powerful programming languages.

The first result of this section is due to Harel, Meyer, and Pratt [57]. The proof can also be found in [56, pp. 551-554].

Theorem 23 *(i) If L contains at least two unary function symbols and a binary function symbol, then the set of valid DL-formulas is a Π^1_1-complete set.*

(ii) If L contains at least one unary function symbol and one binary function symbol, then the set of valid partial correctness formulas

$$\{\phi \rightarrow [p]\psi \mid \phi, \psi \text{ are first-order formulas and } \models \phi \rightarrow [p]\psi\}$$

is a Π^0_2-complete set.

(iii) For every L, the set of valid total correctness formulas

$$\{\phi \rightarrow \langle p\rangle\psi \mid \phi, \psi \text{ are first-order formulas and } \models \phi \rightarrow \langle p\rangle\psi\}$$

is in Σ^0_1.

The assumptions on the language L in the above theorem can presumably be further weakened, but the reader should notice that if L contains no function symbols, then the validity problem for DL is in Σ^0_1.

It follows from Theorem 23 that there is no sound and complete finitary proof system capable of dealing with either of the two fragments described in (i) and (ii). For total correctness, however, the situation is different. Such a system will be presented in the next section. Although the reader may feel comfortable with Theorem 23(iii), it should be stressed that only very simple computations are captured by valid total correctness formulas. This is explained by the next result.

Proposition 24 *Let $\phi \rightarrow \langle p\rangle\psi$ be a valid total correctness formula of DL. There exists a constant $k \geq 0$ such that for every structure $\mathcal{A}$ and state u, if $\mathcal{A}, u \models \phi$, then there exists a computation sequence q of p of length at most k such that $\mathcal{A}, u \models \langle q\rangle\psi$.*

Proof. This is a standard *compactness argument.* Consider the set $CS(p)$ of finite computation sequences of p defined in §2.1.3. For any finite computation sequence q, there is a first-order formula ϕ_q giving necessary and sufficient conditions under which q can execute and terminate successfully; i.e., ϕ_q is logically equivalent to $\langle q\rangle$*true*. The formula ϕ_q is defined by induction on the length of q, as follows:

- if q is the empty string, then $\phi_q = \mathit{true}$.
- if $q = (x := t); q'$, then $\phi_q = \phi_{q'}[x/t]$, where $\phi_{q'}[x/t]$ is the result of substituting t for all free occurrences of x in $\phi_{q'}$, renaming bound variables as necessary to avoid capture.
- if $q = \theta?; q'$, then $\phi_q = \theta \wedge \phi_{q'}$.

By Proposition 2, the total correctness formula in the statement of the Proposition is equivalent to the infinitary formula

$$\phi \quad \rightarrow \quad \bigvee_{q \in CS(p;\psi?)} \phi_q \ ,$$

which by assumption is valid. Hence, by the compactness of first-order logic, there exists a finite subset $F \subseteq CS(p; \psi?)$ such that the finitary formula

$$\phi \quad \rightarrow \quad \bigvee_{q \in F} \phi_q$$

is valid. We may therefore take k to be the maximum of the lengths of the elements of F. $\square$

3.3.2 Spectral Complexity

In this section, we introduce the notion of *spectral complexity* of a programming language. This notion provides a measure of complexity of the halting problem for programs over finite interpretations. The notion of spectral complexity is relatively new in the literature. This name appears for the first time in [153], though the ideas were already present in [150] and [60]. Spectral complexity plays an important role in establishing the expressive power of logics of programs.

In order to introduce the notion of spectral complexity, we shall need some auxiliary definitions.

Definition 25 An interpretation $(\mathcal{A}, u)$ is said to be *Herbrand-like* if the set

$$\{u(x) \mid x \in V\}$$

generates $\mathcal{A}$; i.e., if every $a \in \mathcal{A}$ is $u(t)$ for some term t over V (thus t contains no array variables of positive arity). $\square$

Definition 26 A state u in $\mathcal{A}$ is called *initial* if:

1. there exists an $a \in \mathcal{A}$ such that for all array variables F of positive arity n and $a_1, \ldots, a_n \in \mathcal{A}$,

$$u(F)(a_1, \ldots, a_n) \quad = \quad a \ ;$$

2. for all stack variables σ, $u(\sigma) = \epsilon$.

Let $m \geq 0$. The pair $(\mathcal{A}, u)$ is called an *m-interpretation* if u is an initial state in $\mathcal{A}$ and there exists an $a \in \mathcal{A}$ such that for all $i \geq m$,

$$u(x_i) = a .$$

□

Two interpretations $(\mathcal{A}, u)$ and $(\mathcal{B}, v)$ are said to be *isomorphic* if there exists an isomorphism $h : \mathcal{A} \to \mathcal{B}$ that commutes with the valuations u and v; i.e., for all m-ary array variables F,

$$v(F)(h(a_1), \ldots, h(a_m)) = h(u(F)(a_1, \ldots, a_m))$$

(in particular,

$$v(x) = h(u(x))$$

for individual variables x); and for any stack variable σ,

$$\text{if } u(\sigma) = a_1 a_2 \cdots a_m, \text{ then } v(\sigma) = h(a_1) h(a_2) \cdots h(a_m) .$$

Let L be a finite first-order language. We henceforth assume that L contains at least one function symbol of positive arity (otherwise, only trivial relations can be computed). The language L is said to be *rich* if it contains a function or relation symbol (other than equality) of arity at least two, or at least two function or relation symbols of arity one; otherwise, L is said to be *poor*. Thus, L is poor if it contains exactly one function symbol of arity one and no relation symbols other than equality.

The essential difference between rich and poor languages is that for fixed m, over structures with n elements, a rich language has exponentially (in n) many pairwise nonisomorphic Herbrand-like m-interpretations, whereas poor languages have only polynomially many. For a rich language L and fixed $m \in \omega$, we can encode every finite Herbrand-like m-interpretation $(\mathcal{A}, u)$ by a binary string '$\mathcal{A}, u$' $\in \{0,1\}^*$ in such a way that the following properties hold:

1. The length of '$\mathcal{A}, u$' is polynomial in the cardinality of $\mathcal{A}$.

2. '$\mathcal{A}, u$' = '$\mathcal{B}, v$' iff $(\mathcal{A}, u)$ and $(\mathcal{B}, v)$ are isomorphic.

3. The set of codes

   $$\{\text{'}\mathcal{A}, u\text{'} \mid (\mathcal{A}, u) \text{ is a finite Herbrand-like } m\text{-interpretation}\}$$

   is in $DSPACE(\log n)$.

For a poor language, a similar encoding exists, except the length of '$\mathcal{A}, u$' is logarithmic in the cardinality of A, and the logarithmic function of 3 should be replaced with a linear one. For our purposes, the form of the encoding is not important, only the existence of one; see [150,153] for details.

Now we are ready to define the notion of a *spectrum* of a programming language. Let K be a programming language over L. For $p \in K$ and $m \in \omega$, the m^{th} *spectrum of* p is the set

$$SP_m(p) = \{`\mathcal{A}, u' \mid (\mathcal{A}, u) \text{ is a finite Herbrand-like } m\text{-interpretation, and } \mathcal{A}, u \models \langle p \rangle \mathit{true}\} .$$

The *spectrum* of K is the set

$$SP(K) \quad = \quad \{SP_m(p) \mid p \in K,\ m \in \omega\} .$$

The next three definitions connect spectra with complexity classes. Let $C \subseteq 2^{\{0,1\}^*}$ be a complexity class, and let K be a programming language. We say that the *spectral complexity of K is in C* if $SP_m(p) \in C$ for all $p \in K$ and $m \in \omega$; i.e., $SP(K) \subseteq C$. We say that the *spectral complexity of K is at least C* and write $SP(K) \geq C$ if for every $X \in C$ and $m \in \omega$, if X is a set of codes of finite Herbrand-like m-interpretations, then $X = SP_m(p)$ for some $p \in K$. We say that the *spectral complexity of K is equal to C* and write $SP(K) \approx C$ if both $SP(K) \subseteq C$ and $SP(K) \geq C$.

We conclude this section by establishing the spectral complexity of the programming languages introduced in §3.1. The following result is due to Tiuryn and Urzyczyn [150], except (1e), which follows from a more general result of [153].

Theorem 27 *1. Let L be a rich language.*

   (a) $SP(DWP) \subseteq DSPACE(\log n)$.

   (b) $SP(WP) \subseteq NSPACE(\log n)$.

   (c) $SP(WP_{pds}) = SP(DWP_{pds}) \approx DTIME(2^{O(\log n)}) = PTIME$.

   (d) $SP(WP_{array}) = SP(DWP_{array}) \approx DSPACE(2^{O(\log n)}) = PSPACE$.

   (e) $SP(WP_{array+pds}) = SP(DWP_{array+pds}) \approx DTIME(2^{2^{O(\log n)}}) = EXPTIME$.

2. If L contains a relation symbol of positive arity, exactly one unary function symbol, and no other function symbols, then $\subseteq$ in (1a) and (1b) can be replaced by $\approx$.

3. If L is poor, then (1a)-(1e) carry over, provided $\log n$ *is replaced by* n.

The proofs of Theorem 27(1a,1b,1d) are by mutual simulations of programs and off-line Turing machines. The proofs of Theorem 27(1c,1e) are by mutual simulation of programs and Cook's auxiliary pushdown automata (see [71]).

3.4 Deductive Systems

We will start our discussion of formal deductive systems for first-order DL with the most promising case (cf. Theorem 23(iii)), namely valid total correctness formulas. Here we will consider the programming language R of regular programs. Extensions to handle arrays can be found in [56, pp. 571-575]. Then, by extending the basic formal proof system S (to be introduced below) in various ways, we will obtain:

1. a relatively complete system $S(\ )$ for partial correctness formulas;
2. an arithmetically complete system $S_a(\ )$ for full DL; and
3. an infinitary complete system $S_\infty(\ )$ for full DL.

Some contrasting negative results are contained in [21,90,165].

It should be stressed that unlike *syntax-oriented* formal proof systems of Hoare Logic (cf, the chapter on Hoare logic in this volume), the systems presented in this section are built in a style that resembles first-order reasoning.

Definition 28 (The Deductive System S) The deductive system S consists of the following axioms and rules of inference:

1. all valid first-order formulas;
2. all axioms and rules of PDL (see §2.2);
3. $\langle x := t\rangle\phi \leftrightarrow \phi[x/t]$, where t is a term, ϕ is a first-order formula, and $\phi[x/t]$ denotes the result of substituting t for all free occurrences of x in ϕ, renaming bound variables as necessary to avoid capture.

□

We write $\vdash_S \phi$ if ϕ is a theorem of this system, and say that ϕ is *consistent* if not $\vdash_S \neg\phi$. A set Σ of formulas is *consistent* if all finite conjunctions of elements of Σ are consistent.

The first result, due to Meyer and Halpern [102], establishes the soundness and completeness of S for total correctness formulas.

Theorem 29 *For any first-order formulas ϕ, ψ in L and program p,*

$$\models \phi \rightarrow \langle p\rangle\psi \quad \textit{iff} \quad \vdash_S \phi \rightarrow \langle p\rangle\psi \ .$$

The proof is by induction on p.

It follows from Theorem 23(ii) that we cannot have a result similar to Theorem 29 for partial correctness formulas. A way around this difficulty, suggested by Cook [26], is to consider only *expressive* structures. A structure $\mathcal{A}$ for L is said to be *expressive* for a programming language K with respect to first-order assertions if for every $p \in K$ and for every first-order formula ϕ in L, there exists a first-order formula ψ in L such that

$$\mathcal{A} \models \psi \leftrightarrow [p]\phi \ .$$

Examples of expressive structures for most programming languages are finite structures and *arithmetical structures.* The latter class of structures was introduced by Moschovakis [108] under the name *acceptable structures* (see also [54]). Briefly, a structure $\mathcal{A}$ is *arithmetical* if it contains a first-order-definable copy of the standard model of arithmetic $\mathcal{N} = (\omega, 0, 1, +, \cdot, \leq)$, and has first-order definable functions allowing coding and decoding of finite sequences of elements of $\mathcal{A}$.

Definition 30 (The Deductive System $S(\ )$) For a structure $\mathcal{A}$, let $S(\mathcal{A})$ be the system S extended by adding as axioms all first-order formulas valid in $\mathcal{A}$. □

The next result, essentially due to Cook [26], establishes the soundness and *relative completeness* of $S(\mathcal{A})$ for partial correctness formulas over structures $\mathcal{A}$ expressive for R.

Theorem 31 *For every expressive structure $\mathcal{A}$ for R and for every partial correctness formula $\phi \rightarrow [p]\psi$, where ϕ, ψ are first-order,*

$$\mathcal{A} \models \phi \rightarrow [p]\psi \quad \textit{iff} \quad \vdash_{S(\mathcal{A})} \phi \rightarrow [p]\psi \ .$$

Again, the proof is by induction on p.

If the system $S(\ )$ is further strengthened and restricted to arithmetical structures, then *arithmetical completeness* can be established for full DL.

Definition 32 (The Deductive Systems S_a and $S_a(\ )$) Let S_a be the system S extended with the following two proof rules:

- quantifier generalization:
$$\frac{\phi}{\forall x\ \phi}$$
- rule of convergence:
$$\frac{\phi(n+1) \rightarrow \langle p \rangle \phi(n)}{\phi(n) \rightarrow \langle p^* \rangle \phi(0)}$$
where $p \in R$, $\phi(n)$ is a first-order formula with a free variable n ranging over (a copy of) $\mathcal{N}$ such that n does not occur in p, and $+1$ and 0 replaced by a suitable first-order definition. These requirements can be satisfied in arithmetical structures.

For an arithmetical structure $\mathcal{A}$, let $S_a(\mathcal{A})$ be S_a augmented by adding as axioms all first-order formulas valid in $\mathcal{A}$. □

The next result, due to Harel [54], asserts the *arithmetical completeness* of DL.

Theorem 33 *For every arithmetical structure $\mathcal{A}$ and for every $\phi \in DL$,*

$$\mathcal{A} \models \phi \quad \textit{iff} \quad \vdash_{S_a(\mathcal{A})} \phi \ .$$

By Theorem 23(i), there cannot exist a finitary complete proof system for full DL. A complete infinitary proof system was proposed by Mirkowska for Algorithmic Logic [105]. We follow the exposition [56] where this system is presented for DL.

Definition 34 (The Deductive System S_∞) Let S_∞ be the system S augmented with the following axiom schemes and proof rules:

- $\langle x := t\rangle\phi \leftrightarrow \phi[x/t]$, where $\phi \in DL$. (The substitution of a term for a variable in a DL formula ϕ has to be defined carefully, due to the possible presence of programs in ϕ. The notions of *free* and *bound variable* are not as clear as they are in pure first-order logic. See [56, p. 559] for details.)
- $\phi \leftrightarrow \psi$, where ψ is ϕ with some program p replaced by $z := x; p'; x := z$, for some $z \in V$ not appearing in ϕ or p, and p' is p with all occurrences of x replaced by z.

In addition, we also take quantifier generalization as in S_a, and

(∞) infinitary convergence rule:

$$\frac{\phi \to [p^n]\psi, \ n \in \omega}{\phi \to [p^*]\psi}$$

where $\phi, \psi \in DL$ and $p \in R$. □

Observe that the rule (∞) has infinitely many premises. A *proof* in S_∞ is a possibly infinite sequence of DL formulas, each one either an instance of an axiom scheme, or following from previous formulas by application of a proof rule.

The next result is due to Mirkowska [105].

Theorem 35 *For every $\phi \in DL$, ϕ is valid iff $\vdash_{S_\infty} \phi$.*

There are as many proofs of this theorem as there are proofs of infinitary completeness for $L_{\omega_1\omega}$. In fact, every proof for the latter logic transforms into a proof of Theorem 35. Algebraic methods are used in [105], whereas [56] uses Henkin's method for $L_{\omega_1\omega}$.

3.5 Expressive Power

Let L be a finite first-order language with equality. The subject of study in this section is the relation of *relative expressiveness* between logics $\mathcal{L}, \mathcal{L}'$ over L.

Definition 36 We say that $\mathcal{L}'$ is *more expressive than* $\mathcal{L}$ and write $\mathcal{L} \leq \mathcal{L}'$ iff for every $\phi \in \mathcal{L}$, there exists a $\psi \in \mathcal{L}'$ such that $\mathcal{A}, u \models \phi \leftrightarrow \psi$ for all structures $\mathcal{A}$ and *initial* states u (Definition 26). We write $\mathcal{L} \equiv \mathcal{L}'$ if both $\mathcal{L} \leq \mathcal{L}'$ and $\mathcal{L}' \leq \mathcal{L}$, and $\mathcal{L} < \mathcal{L}'$ if $\mathcal{L} \leq \mathcal{L}'$ but not $\mathcal{L} \equiv \mathcal{L}'$. □

The reason for the restriction to *initial* states in Definition 36 is that if $\mathcal{L}$ and $\mathcal{L}'$ have access to different sets of data types, then they may be trivially incomparable for uninteresting reasons, unless we are careful to limit the states on which they are compared. For example, when comparing DL_{pds} and DL_{array}, we had better disallow input states in which the stack variable σ is nonempty; otherwise, since WP_{pds} programs can access σ and WP_{array} cannot, the DDL_{pds} formula

$$\langle pop(\sigma, x);\ \textbf{if } x = y \textbf{ then skip else fail fi}\rangle true$$

would be trivially equivalent to no formula of DL_{array}. Definition 36 effectively restricts the *input* states of programs to initial states, but still allows non-initial states as intermediate states in a computation. Recall also that stacks and arrays often play an auxiliary role in programs and are not usually used for input/output.

In the definition of $DL(K)$ given in §3.1, the programming language K is an explicit parameter; but the first-order language L over which $DL(K)$ and K are taken should be treated as a parameter as well. It turns out that the relation $\leq$ of relative expressiveness is sensitive not only to K, but also to L. This second parameter is often ignored in the literature, creating a source of potential misinterpretation of the results.

Let $L_{\omega\omega}$ denote the first-order predicate calculus with equality over the language L, and let $L_{\omega_1\omega}$ denote the infinitary language that allows countable disjunctions in addition to the usual formation rules of $L_{\omega\omega}$.

We start with a result which orders the versions of DL of §3.1 linearly.

Proposition 37 *For every first-order language L,*

(i) $L_{\omega\omega} \leq DL \leq DL_{pds} \leq DL_{array} \leq DL_{array+pds} \leq L_{\omega_1\omega}$.

(ii) $DDL \leq DL$ *and* $DDL_* \leq DL_*$, *for* $* \in \{pds, array, array + pds\}$.

All inequalities of the above proposition are easy to establish, except the third one in (i). This inequality will follow from a more general result (Theorem 45) relating $\leq$ with spectra and complexity classes.

It is easy to show that the last inequality in (i) is strict for every language L. Henceforth, we shall assume that L contains at least one function symbol of positive arity. Under this assumption, the first inequality in (i) becomes strict, since it is possible to construct an infinite model for L which is uniquely definable in DL up to isomorphism. By the upward Löwenheim-Skolem Theorem, this is impossible in $L_{\omega\omega}$.

Strictness of the other inequalities in Proposition 37 will be discussed in the remainder of this section. In order to discuss the relationship of these problems to complexity theory, we must introduce some definitions.

For program p and structure $\mathcal{A}$, let $p^{\mathcal{A}}$ be the binary relation associated with p, as defined in §3.2.

Definition 38 A program p is said to be *semantically deterministic* if for every $\mathcal{A}$, $p^{\mathcal{A}}$ is a partial function. □

The remarks immediately following Definition 36 at the beginning of this section motivate the following definition.

Definition 39 Let $\rho \subseteq S^{\mathcal{A}} \times S^{\mathcal{A}}$ be a binary relation on states of a structure $\mathcal{A}$. The *ground relation of* ρ is the binary relation

$$ground(\rho) \quad = \quad \{(u\lceil V, v\lceil V) \mid (u,v) \in \rho \text{ and } u \text{ is an initial state of } \mathcal{A}\} \ .$$

(Here $u\lceil V$ denotes the function u restricted to domain V.) If K and K' are programming languages over L, we write $K \leq K'$ iff for every $p \in K$, there exists a $q \in K'$ such that in every structure $\mathcal{A}$ for L, $ground(p^{\mathcal{A}}) = ground(q^{\mathcal{A}})$. □

Definition 40 The programming language K is said to be *semantically closed* under the n-ary programming construct c if for all programs $p_1, \ldots, p_n \in K$, there exists a $q \in K$ such that in every structure $\mathcal{A}$,

$$c(p_1, \ldots, p_n)^{\mathcal{A}} \quad = \quad q^{\mathcal{A}} \ .$$

□

For example, "K is semantically closed under sequential composition" means that for every $p, p' \in K$, there exists a $q \in K$ such that $q^{\mathcal{A}} = (p;p')^{\mathcal{A}}$ in all structures $\mathcal{A}$.

Note that K need not contain the construct c; however, if it does, then it is trivially semantically closed under c.

We now define a useful programming construct **run-until**, which works as follows. For $p, q \in K$ and first-order formula ϕ, the program

$$\textbf{run } p \textbf{ until } \langle q \rangle \phi$$

runs p, but after each atomic step of p, it runs q from the current state and tests whether q would halt in a state satisfying ϕ. Depending on the outcome, it take the following actions:

- if q diverges, then the **run-until** statement itself diverges;

- if q terminates in a state v satisfying ϕ, the entire **run-until** statement terminates in state v;
- if q terminates in a state v satisfying $\neg\phi$, then control is returned to p in state v to perform the next atomic step.

For regular programs, we can define this construct formally by induction on p:

$$\begin{aligned}
\textbf{run } p \textbf{ until } \langle q\rangle\phi &= p;q \text{ , for } p \text{ an atomic program or test} \\
\textbf{run } p;p' \textbf{ until } \langle q\rangle\phi &= (\textbf{run } p \textbf{ until } \langle q\rangle\phi); \textbf{if } \phi \textbf{ then skip else run } p' \textbf{ until } \langle q\rangle\phi \textbf{ fi} \\
\textbf{run } p\cup p' \textbf{ until } \langle q\rangle\phi &= (\textbf{run } p \textbf{ until } \langle q\rangle\phi) \cup (\textbf{run } p' \textbf{ until } \langle q\rangle\phi) \\
\textbf{run } p^* \textbf{ until } \langle q\rangle\phi &= q \cup (\textbf{run } p \textbf{ until } \langle q\rangle\phi); (\neg\phi?; \textbf{run } p \textbf{ until } \langle q\rangle\phi)^* \ .
\end{aligned}$$

The construct also makes sense in more general programming languages, and can be defined formally for any program p equivalent to its set $CS(p)$ of computation sequences, in the sense of Theorem 2, §2.1.3. The definition gives **run** p **until** $\langle q\rangle\phi$ in terms of its computation sequences. The definition of **run** r **until** $\langle q\rangle\phi$ for r a computation sequence is given above, and for more general programs p,

$$CS(\textbf{run } p \textbf{ until } \langle q\rangle\phi) = \bigcup_{r\in CS(p)} CS(\textbf{run } r \textbf{ until } \langle q\rangle\phi) \ .$$

Let WP_{2pds} denote the class of all **while** programs with two stacks.

Definition 41 A programming language K is said to be *acceptable* if it satisfies the following conditions:

1. $K \leq WP_{2pds}$;
2. K contains all simple assignments $x := t$ and is semantically closed under sequential composition, conditional, and **while** loop, with tests in the last two restricted to quantifier-free formulas of L;
3. K is semantically closed under variable renaming; that is, if $p \in K$ and π is any permutation of V, then there exists a $q \in K$ such that for all $\mathcal{A}$,

   $$q^{\mathcal{A}} = \{(u\circ\pi, v\circ\pi) \mid (u,v) \in p^{\mathcal{A}}\} \ ;$$

4. K is semantically closed under the **run-until** construct

   $$\textbf{run } p \textbf{ until } \langle q\rangle\phi$$

   for p and q semantically deterministic and ϕ a quantifier-free formula of L.

□

Definition 41(1) insures that programs of K perform only effective operations (relative to the interpretation of the symbols in L). WP_{2pds} in (1) can be replaced by any programming language of *universal power*: effective definitional schemes [43], recursive procedures with integer counters, **while** programs with integer-indexed arrays, etc. Conditions (2-4) say that K has some semantical flexibility, so that certain operations on programs can be performed without leaving the class K. It should be clear that all the programming languages of §3.1 are acceptable. The notion of an acceptable programming language was introduced by Lipton [90] and used by many authors, e.g. [22,151].

Definition 42 An acceptable programming language K is *semi-universal* (cf. [151]) if for every $m \in \omega$ there exists a semantically deterministic program $p \in K$ such that for every Herbrand-like m-interpretation $(\mathcal{A}, u)$,

$$\mathcal{A}, u \models \langle p \rangle \mathit{true} \quad \text{iff} \quad (\mathcal{A}, u) \text{ is finite.}$$

□ .

The essence of this definition is that semi-universal programming languages have enough power to search every submodel generated by the input. The notion of semi-universality captures the concept of programs with *unbounded memory* in [56].

We now proceed to the last definition.

Definition 43 A programming language K is *divergence-closed* if for every $p \in K$ there exists a $q \in K$ and two variables $x, y \in V$ such that for every finite Herbrand interpretation $(\mathcal{A}, u)$ with A having at least two elements,

$$\begin{aligned} \mathcal{A}, u \models \langle p \rangle \mathit{true} \quad &\text{iff} \quad \mathcal{A}, u \models \langle q \rangle (x = y) \ , \\ \mathcal{A}, u \models [p] \mathit{false} \quad &\text{iff} \quad \mathcal{A}, u \models \langle q \rangle (x \neq y) \ . \end{aligned}$$

□

Informally, q decides without diverging whether p possibly terminates.

Proposition 44 *The following programming languages are semi-universal and divergence-closed:*

1. *for every L containing at least one function symbol of arity at least two, or at least two unary function symbols: $(D)WP_*$, for $* \in \{pds, array, array + pds\}$ (cf. [151]);*

2. *for every L containing exactly one unary function symbol and no other function symbols of positive arity: DWP.*

It is not known whether WP is divergence-closed for any L.

The next result, due to Tiuryn and Urzyczyn [151,153], plays a key role in applying the hierarchy results of complexity theory to problems concerning $\leq$.

Theorem 45 *Let K_1 and K_2 be programming languages over L such that K_1 is acceptable and K_2 is semi-universal and divergence-closed. Let $C_1, C_2 \subseteq 2^{\{0,1\}^*}$ denote families of sets that are closed downward under logarithmic space or linear space reductions, depending on whether L is rich or poor, respectively. Let $SP(K_i) \approx C_i$ for $i = 1, 2$. The following statements are equivalent:*

1. $DL(K_1) \leq DL(K_2)$
2. $SP(K_1) \subseteq SP(K_2)$
3. $C_1 \subseteq C_2$.

The equivalence of (1) and (2) is proved in [151, Theorem 5]. The equivalence of (2) and (3) is proved in [153, Theorem 3.9].

Combining Theorem 27, Proposition 44, and Theorem 45 with the known hierarchy theorems of complexity theory, we obtain:

Corollary 46 1. *For every L and for every $* \in \{pds, array, array + pds\}$,*

$$DDL_* \equiv DL_* \ .$$

2. *If L has exactly one unary function symbol and no other function symbols of positive arity and at least one relation symbol of positive arity other than $=$, then*

$$DDL \equiv DL \quad \textit{iff} \quad DSPACE(\log n) = NSPACE(\log n) \ .$$

3. *For every L,*

$$DL_{pds} \leq DL_{array} \ .$$

4. *If L is rich, then*

$$DL_{pds} \equiv DL_{array} \quad \textit{iff} \quad DTIME(2^{O(\log n)}) = DSPACE(2^{O(\log n)})$$
$$\textit{(i.e., iff } PTIME = PSPACE\textit{)} \ .$$

5. *For every L,*

$$DL_{pds} < DL_{array+pds} \ .$$

6. *If L is rich, then*

$$DL_{array} \equiv DL_{array+pds} \quad \textit{iff} \quad DSPACE(2^{O(\log n)}) = DTIME(2^{2^{O(\log n)}})$$
$$\textit{(i.e., iff } PSPACE = EXPTIME\textit{)} \ .$$

7. The results of (2), (4), (6) hold for poor languages when $\log n$ *is replaced by* n.

A consequence of Corollary 46 is that many questions on relative expressive power of Dynamic Logics are equivalent to well-known difficult open problems of complexity theory. Most of the results of Corollary 46 were proved in [150].

The following two questions are not settled by the method of spectral complexity.

1. Does $DDL < DL$ hold for every L containing a function symbol of arity at least two or at least two function symbols of arity one?
2. Does $DL < DL_{pds}$ hold for every L?

We conclude this section with a full answer to the first question and a partial answer to the second.

Theorem 47 *If L contains a function symbol of arity at least two or at least two function symbols of arity one, then $DDL < DL$.*

This result is due to Stolboushkin and Taitslin [143] and independently to Berman, Halpern and Tiuryn [15]. The proof of [143] uses Adian's Theorem [2]. It constructs an infinite model $\mathcal{A}$ with two unary functions f, g and a constant c satisfying the property:

> for every program $p \in DWP$ there exists a constant $k \in \omega$ such that every terminating computation of p in $\mathcal{A}$ takes at most k steps.

This property of $\mathcal{A}$, together with the compactness of first-order logic, imply that the formula

$$\langle z := c; \textbf{while } z \neq x \textbf{ do } z := f(z) \cup z := g(z) \textbf{ od}\rangle \mathit{true} \ ,$$

which expresses a nondeterministic search, is equivalent to no DDL formula.

The proof of [15] uses a purely combinatorial argument to construct a model $\mathcal{A}$ with the above property. In both cases the construction of $\mathcal{A}$ together with a proof that $\mathcal{A}$ has the desired property is the major difficulty of the proof of Theorem 47. Other proofs of Theorem 47 can be found in [156,74,152].

A partial answer to (2) is given by the next result.

Theorem 48 *If L contains a function symbol of arity at least two, then $DL < DL_{pds}$.*

This result is due to Erimbetov [38] and independently to Tiuryn [148]. The method of proof in both cases is essentially the same, though [148] contains a slightly more general statement. It says that two certain infinite models, constructed from constants, can be distinguished by no formula of a certain fragment $L^{BM}_{\omega_1\omega}$ of the infinitary language $L_{\omega_1\omega}$. The proof uses two techniques: a *pebbling argument* invented by M. Paterson and C. Hewitt [121] and independently by H. Friedman [43], and the technique of Ehrenfeucht-Fraisse games [30]. The pebbling argument works in this framework only for languages that satisfy the assumption of Theorem 48. The proof is completed by checking that $DL \leq L^{BM}_{\omega_1\omega}$, and that the two models can be distinguished by a formula of DL_{pds}.

A stronger result holds for deterministic programs. It follows from Corollary 46(1) and Theorem 47 that if L contains a function symbol of arity at least two or at least two function symbols of arity one, then $DDL < DDL_{pds}$. For languages with exactly one unary function symbol and no function symbols of higher arity, the problems of comparing DL to DL_{pds} and DDL to DDL_{pds} reduce to open problems in complexity theory, such as whether the classes $DSPACE(\log n)$ and $PTIME$ are equal. A challenging open problem not known to be equivalent to an open problem in complexity theory is the question of whether $DL < DL_{pds}$ for all languages containing a function symbol of arity at least two or at least two function symbols of arity one.

3.6 Operational vs. Axiomatic Semantics

In this section, we survey some results connected with the following question:

> Can the semantics of a programming language be specified by partial correctness formulas with first-order assertions?

This question is of fundamental importance for so-called *axiomatic semantics.*

Definition 49 Let K be a programming language. For $p, q \in K$, we say that p is *semantically contained in* q and write $p \subseteq q$ iff for every model $\mathcal{A}$, $p^{\mathcal{A}} \subseteq q^{\mathcal{A}}$. □

For definiteness, we choose $K = WP_{2pds}$, **while** programs with two stacks.

Consider the two propositions

1. $p \subseteq q$
2. for all first-order formulas ϕ, ψ in L, if $\phi \rightarrow [q]\psi$ is valid, then so is $\phi \rightarrow [p]\psi$.

The implication (1) $\rightarrow$ (2) follows immediately from Proposition 5. The essence of the question raised above is whether the converse holds. A. Meyer conjectured that this was indeed the case, and this conjecture was confirmed by Meyer and Halpern [102] and independently by Bergstra, Tiuryn and Tucker [9].

Theorem 50 *Let L be any first-order language except one containing exclusively unary relations and at most one unary function symbol. Then for every $p, q \in WP_{2pds}$, if for all first-order formulas ϕ, ψ in L, $\models \phi \rightarrow [q]\psi$ implies $\models \phi \rightarrow [p]\psi$, then $p \subseteq q$.*

It is noteworthy that Theorem 50 extends with the same proof of [102,9] to programming languages whose power goes beyond any acceptable programming language; e.g., flowcharts with arbitrary first-order tests, nondeterministic assignments, stacks, arrays, etc. We also remark that the assertions ϕ, ψ and programs p, q in Theorem 50 are all over the same language L. It is not known whether Theorem 50 extends to a language L consisting of only one unary function symbol and some number of unary relation symbols. The reader is referred to [88] for a simpler proof of Theorem 50 for the special case of **while** programs. However, the result of [88] is weaker than Theorem 50 for **while** programs, since the former allows the assertions ϕ, ψ to be in an extension of L.

The definitions of this section can easily be relativized to a first-order theory T, i.e., when interpretations are restricted to models of T. Currently, it is not well understood what conditions on T might be sufficient to imply a relativized version of Theorem 50. It is shown in [9] that the relativized version of Theorem 50 fails for as simple a theory as the equational theory

$$\{f(g(x)) = g(f(x)) = x\}$$

[9, Theorem 5.8], and for as complex a theory as *Complete Number Theory* [9, Theorem 5.10]. A positive result in this direction by Csirmaz [27] shows that Theorem 50 admits relativization to *Peano Arithmetic*, confirming a conjecture posed in [9].

We conclude this section with a result showing that the validity of the partial correctness formula $\phi \rightarrow [q]\psi$ of Theorem 50 can even be proved in Hoare Logic, provided one allows ϕ and ψ to be in an extension of L. This result is due independently to Leivant [89] and Meyer [104]. The method of proof given in [104] adapts to any programming language other than *WP*for which there is a sound and relatively complete proof system.

Theorem 51 *Let L be any first-order language. There exists an extension $L' \supseteq L$ such that for any $p \in WP_{2pds}$ and $q \in WP$ over L, if $p \not\subseteq q$, then there exist first-order formulas ϕ, ψ in L' such that $\phi \rightarrow [q]\psi$ is a theorem of Hoare logic, but $\phi \rightarrow [p]\psi$ is not valid.*

It is not known, even for L satisfying the assumption of Theorem 50, whether the extension of the language L in Theorem 51 is essential.

The reader is also referred to [10] where various schemes for establishing program inclusion are studied.

3.7 Other Programming Languages

For uniformity of exposition, we have concentrated on the language of **while** programs, occasionally augmented with arrays and stacks. Other definitions appearing in the literature (e.g., [54,56]) may differ slightly from ours, but the reader should have no difficulty establishing that these versions are equivalent in expressive power.

Below we discuss briefly some possible extensions of *WP*.

3.7.1 Algol-Like Languages

One can add to *WP* recursive procedures without parameters, recursive procedures with individual parameters passed by name, reference, value/result, or other mechanism, or recursive procedures with higher-order procedure parameters. Blocks with local declarations can be added as well. Depending on scope rules (dynamic vs. static) and the features allowed, one easily arrives at a family consisting of thousands of programming languages.

Proof theory for Algol-like languages is well developed by now. The reader is referred to the chapter in this volume on Hoare Logic and [4,46] for typical results and further references.

The expressive power of logics based on Algol-like programmming languages has been studied to a lesser extent. A typical result in this area, which follows from [16,24], is that Dynamic Logic with recursive procedures in which individual parameters are passed by value/result is equivalent to DL_{pds}.

3.7.2 Nondeterministic Assignment

The nondeterministic assignment $x := ?$, introduced in §3.1, chooses an element of the domain of computation nondeterministically and assigns it to x. Thus, it is a device representing *unbounded nondeterminism*, as opposed to the *binary nondeterminism* of the nondeterministic choice construct $\cup$. The programming language *WP* augmented with the nondeterministic assignment is not an acceptable language. Adding nondeterministic assignment to a programming language usually increases expressive power. For more information, the reader is referred to [56].

3.7.3 Auxiliary Data Types

WP can be augmented with other data types, such as counters, binary stacks, or higher-order arrays and stacks. Data types can be combined.

Proof theory for programming languages with these data types is not sufficiently developed. Their relative expressive power has been studied more intensively. One interesting result, due to Urzyczyn [155], is that adding a binary stack to *DWP* results in a logic strictly more expressive than *DDL*. (The corresponding question for nondeterministic **while** programs is open.) It follows from [148] that this logic is strictly weaker than DL_{pds}. An infinite hierarchy of logics over *WP* with higher-order arrays and stacks is studied in [153].

3.7.4 Tests

In previous sections, we allowed tests to be quantifier-free first-order formulas. One can increase the power of programs by allowing arbitrary first-order formulas as tests. One can even go further and define programs and formulas by simultaneous induction, allowing programs to test arbitrary formulas. This is called a *rich-test* logic of programs. The reader is referred to [56] for more information on this topic.

4 Other Approaches

4.1 Nonstandard Dynamic Logic

Nonstandard Dynamic Logic (*NDL*) was introduced by Andréka, Németi, and Sain in 1979. The reader is referred to [109,3] for a full exposition and further references. The main idea behind *NDL* is to allow nonstandard models of time by referring only to first-order properties of time when measuring the length of a computation. The approach described in [3] and further research in Nonstandard Dynamic Logic is concentrated on proving properties of flowcharts, i.e., programs built up of assignments, conditionals and **go to**'s.

Nonstandard Dynamic Logic is well suited to comparing the reasoning power of various program verification methods. This is usually done by providing a model-theoretic characterization of a given method for program verification. To illustrate this approach, we briefly discuss a characterization of Hoare Logic for partial correctness formulas. For the present exposition, we choose a somewhat simpler formalism which still conveys the basic idea of nonstandard time.

Let L be a first-order language. We fix for the remainder of this section a deterministic **while** program p over L in which the **while-do** construct does not occur. (Such a program is called *loop-free.*) Let $\overline{z} = (z_1, \ldots, z_n)$ contain all variables occurring in p, and let $\overline{y} = (y_1, \ldots, y_n)$ be a vector of n distinct individual variables disjoint from $\overline{z}$.

Since p is loop-free, it has only finitely many computation sequences. One can easily define a quantifier-free first-order formula θ_p with all free variable among $\overline{y}, \overline{z}$ which defines the input/output relation of p in all structures $\mathcal{A}$ for L, in the sense that the pair of states (u, v) is in $p^{\mathcal{A}}$ if and only if

$$\mathcal{A}, v[y_1/u(z_1), \ldots, y_n/u(z_n)] \models \theta_p$$

and $u(x) = v(x)$ for all $x \in V - \{z_1, \ldots, z_n\}$.

Let p^+ be the following deterministic **while** program:

$$\begin{array}{l} \overline{y} := \overline{z}; \\ p; \\ \textbf{while } \overline{z} \neq \overline{y} \textbf{ do } \overline{y} := \overline{z};\ p \textbf{ od} \end{array}$$

where $\overline{z} \neq \overline{y}$ stands for $z_1 \neq y_1 \vee \cdots \vee z_n \neq y_n$ and $\overline{y} := \overline{z}$ stands for $y_1 := z_1; \cdots; y_n = z_n$. Thus program p^+ performs p iteratively until p does not change the state.

The remainder of this section is devoted to giving a model-theoretic characterization, using *NDL*, of Hoare's system for proving partial correctness properties of p^+ relative to a given first-order theory T in L. We denote provability in Hoare Logic by $\vdash_{HL}$.

Due to the very specific form of p^+, the Hoare system reduces to the following rule:

$$\frac{\phi \to \chi,\ \chi[\overline{z}/\overline{y}] \wedge \theta_p \to \chi,\ \chi[\overline{z}/\overline{y}] \wedge \theta_p \wedge \overline{z} = \overline{y} \to \psi}{\phi \to [p^+]\psi}$$

where ϕ, χ, ψ are first-order formulas, and no variable of $\overline{y}$ occurs free in χ.

The next series of definitions introduces a variant of *NDL*. A structure $\mathcal{I}$ for the language consisting of a unary function symbol $+1$ (*successor*), a constant symbol 0, and equality is called a *time model* if the following axioms are valid in $\mathcal{I}$:

1. $x + 1 = y + 1 \rightarrow x = y$
2. $x + 1 \neq 0$
3. $x \neq 0 \rightarrow \exists y\; y + 1 = x$
4. $x \neq x \underbrace{+1 + 1 + \cdots + 1}_{n}$, for any $n = 1, 2, \ldots$

Let $\mathcal{A}$ be a structure for L, and let $\mathcal{I}$ be a time model. A function $\rho : \mathcal{I} \rightarrow \mathcal{A}^n$ is called a *run* of p in $\mathcal{A}$ if the following two infinitary formulas are valid in $\mathcal{A}$:

1. $\bigwedge_{i \in \mathcal{I}} \theta_p[\overline{y}/\rho(i), \overline{z}/\rho(i+1)]$;
2. for every first-order formula $\phi(\overline{z})$ in L,

$$\phi(\rho(0)) \wedge \bigwedge_{i \in \mathcal{I}} (\phi(\rho(i)) \rightarrow \phi(\rho(i+1))) \quad \rightarrow \quad \bigwedge_{i \in \mathcal{I}} \phi(\rho(i)) \;.$$

The first formula says that for $i \in \mathcal{I}$, $\rho(i)$ is the valuation obtained from $\rho(0)$ after i iterations of the program p. The second formula is the induction scheme along the run ρ.

Finally, we say that a partial correctness formula $\phi \rightarrow [p^+]\psi$ *follows from T in nonstandard time semantics* and write $T \models_{NT} \phi \rightarrow [p^+]\psi$ if for every model $\mathcal{A}$ of T, time model $\mathcal{I}$, and run ρ of p in $\mathcal{A}$,

$$\mathcal{A} \models \phi[\overline{z}/\rho(0)] \rightarrow \bigwedge_{i \in \mathcal{I}} (\rho(i) = \rho(i+1) \rightarrow \psi[\overline{z}/\rho(i)]) \;.$$

The following characterization theorem is due to Csirmaz [28].

Theorem 52 *For every first-order theory T in L and first-order formulas ϕ, ψ in L, the following conditions are equivalent:*

1. $T \vdash_{HL} \phi \rightarrow [p^+]\psi$;
2. $T \models_{NT} \phi \rightarrow [p^+]\psi$.

Other proof methods have been characterized in the same spirit. The reader is referred to [93] for more information on this issue and further references.

4.2 Algorithmic Logic

Algorithmic Logic (AL), first defined by Salwicki in 1970 [136], foreshadowed Dynamic Logic in many respects. Research in AL has centered on program verification, infinitary completeness, normal forms for programs, recursive procedures with parameters, and data type specification; see [7,137] for surveys.

The original version of AL allowed deterministic **while** programs and formulas built from the constructs

$$p\phi \quad \cup p\phi \quad \cap p\phi$$

corresponding in our terminology to

$$\langle p\rangle\phi \quad \langle p^*\rangle\phi \quad \bigwedge_{n\in\omega}\langle p^n\rangle\phi \ ,$$

respectively, where p is a **while** program and ϕ is a quantifier-free first-order formula. Mirkowska [106,107] extended AL to allow nondeterministic **while** programs and the constructs

$$\nabla p\phi \quad \Delta p\phi$$

corresponding in our terminology to

$$\langle p\rangle\phi \quad \mathbf{halt}(p) \wedge [p]\phi \wedge \langle p\rangle\phi \ ,$$

respectively. The latter asserts that all traces of p are finite and terminate in a state satisfying ϕ. Complete infinitary deductive systems are given for first-order and propositional versions [106,107].

Constable [23,25] and Goldblatt [47] present logics similar to AL and DL for reasoning about deterministic **while** programs.

4.3 Logic of Effective Definitions

The Logic of Effective Definitions (LED), introduced by Tiuryn in 1978 (see [149]), was intended to study notions of computability over abtract models and to provide a universal framework for the study of logics of programs over such models. It consists of first-order logic augmented with new atomic formulas of the form $p = q$, where p and q are *effective definitional schemes* [43]:

$$\begin{array}{l}\mathbf{if}\ \phi_1\ \mathbf{then}\ t_1\\ \qquad \mathbf{else\ if}\ \phi_2\ \mathbf{then}\ t_2\\ \qquad\qquad \mathbf{else\ if}\ \phi_3\ \mathbf{then}\ t_3\\ \qquad\qquad\qquad \mathbf{else\ if}\ \ldots\end{array}$$

where the ϕ_i are quantifier-free formulas and t_i are terms over a bounded set of variables, and the function $i \mapsto (\phi_i, t_i)$ is recursive. The formula $p = q$ is defined to be true in state u if both p and q terminate and yield the same value, or neither terminates.

Model theory and infinitary completeness of LED are treated in [149].

4.4 Temporal Logic

Temporal Logic (*TL*) is an alternative application of modal logic to program specification and verification. It was first proposed as a useful tool in program verification by Pnueli [125], and has since been developed by many authors in various forms. This topic is surveyed in depth elsewhere in this volume [32].

TL differs from *DL* chiefly in that it is *endogenous*, i.e., programs are not explicit in the language. Every application has a single program associated with it, and the language may contain program-specific statements such as $at(\ell)$, meaning "execution is currently at location ℓ in the program." Models can be a linear sequence of program states (so-called *linear-time TL*), representing the execution sequence of a deterministic program or a possible execution sequence of a nondeterministic or concurrent program; or a tree of program states (so-called *branching-time TL*), representing the space of all possible computation sequences of a nondeterministic or concurrent program.

Modal constructs used in *TL* include

$\Box\phi$ "ϕ holds in all future states"
$\Diamond\phi$ "ϕ holds in some future state"
$\circ\phi$ "ϕ holds in the next state"

for linear-time logic, as well as constructs for expressing

"for all paths starting from the present state..."
"for some path starting from the present state..."

for branching-time logic.

Temporal Logic is useful in situations where programs are not normally supposed to halt, such as operating systems, and is particularly well suited to the study of concurrency. Many of the classical program verification methods such as the *intermittent assertions method* are treated quite elegantly in this framework.

References

[1] Abrahamson, K., "Decidability and Expressiveness of Logics of Processes," Ph.D. Thesis, Tech. Rep. 80-08-01, Dept. of Comput. Sci., Univ. of Washington, Seattle, 1980.

[2] Adian, S.I., *The Burnside Problem and Identities in Groups.* Springer, Heidelberg, 1979.

[3] Andréka, H., I. Németi, and I. Sain, "A Complete Logic for Reasoning about Programs via Nonstandard Model Theory. Parts I,II," *Theor. Comput. Sci.* 17 (1982), 193-212, 259-278.

[4] Apt, K. R., "Ten Years of Hoare's Logic: a Survey—Part 1," *ACM Trans. Prog. Lang. Syst.* 3 (1981), 431-483.

[5] Apt, K. R. and G. Plotkin, "Countable Nondeterminism and Random Assignment," *J. ACM* 33 (1986), 724-767.

[6] de Bakker, J., *Mathematical Theory of Program Correctness.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[7] Banachowski, L., A. Kreczmar, G. Mirkowska, H. Rasiowa, and A. Salwicki, "An Introduction to Algorithmic Logic: Metamathematical Investigations in the Theory of Programs," in: Mazurkiewitz and Pawlak (eds.), *Math. Found. Comput. Sci.*, Banach Center Publications, Warsaw, 1977, 7-99.

[8] Ben-Ari, M., J. Y. Halpern, and A. Pnueli, "Deterministic Propositional Dynamic Logic: Finite Models, Complexity and Completeness," *J. Comput. Syst. Sci.* 25 (1982), 402-417.

[9] Bergstra, J. A., J. Tiuryn, and J.V. Tucker, "Floyd's Principle, Correctness Theories and Program Equivalence," *Theor. Comput. Sci.* 17 (1982), 113-149.

[10] Bergstra, J. A. and J. W. Klop, "Proving Program Inclusion Using Hoare's Logic," *Theor. Comput. Sci.* 30 (1984), 1-48.

[11] Berman, F., "Expressiveness Hierarchy for *PDL* with Rich Tests," Tech. Rep. 78-11-01, Dept. of Comput. Sci., Univ. of Washington, Seattle, 1978.

[12] Berman, F., "A Completeness Technique for *D*-Axiomatizable Semantics," *Proc. 11th ACM Symp. Theory of Comput.*, 1979, 160-166.

[13] Berman, F., "Semantics of Looping Programs in Propositional Dynamic Logic," *Math. Syst. Theory* 15 (1982), 285-294.

[14] Berman, F. and M. Paterson, "Propositional Dynamic Logic is Weaker Without Tests," *Theor. Comput. Sci.* 16 (1981), 321-328.

[15] Berman, P., J. Y. Halpern, and J. Tiuryn, "On the Power of Nondeterminism in Dynamic Logic," in: Nielsen and Schmidt (eds.), *Proc. 9th Colloq. Automata Lang. Prog.*, Lect. Notes in Comput. Sci. 140, Springer, 1982, 48-60.

[16] Brown, S., D. Gries, and T. Szymanski, "Program Schemes with Pushdown Stores," *SIAM J. Comput.* 1 (1972), 242-268.

[17] Burstall, R. M., "Program Proving as Hand Simulation with a Little Induction," in: *Information Processing 1974*, North-Holland, Amsterdam, 308-312.

[18] Chandra, A. K., D. Kozen, and L. Stockmeyer, "Alternation," *J. Assoc. Comput. Mach.* 28:1 (1981), 114-133.

[19] Chang, C. C. and H.J. Keisler, *Model Theory*. North-Holland, Amsterdam, 1973.

[20] Chellas, B. F., *Modal Logic: an Introduction*. Cambridge University Press, 1980.

[21] Clarke, E. M., "Programming Language Constructs for which it is Impossible to Obtain Good Hoare Axiom Systems," *J. ACM* 26 (1979), 129-147.

[22] Clarke, E. M., S. M. German, and J. Y. Halpern, "Effective Axiomatizations of Hoare Logics," *J. ACM* 30 (1983), 612-636.

[23] Constable, R. L., "On the Theory of Programming Logics," *Proc. 9th ACM Symp. Theory of Comput.*, 1977, 269-285.

[24] Constable, R. L. and D. Gries, "On Classes of Program Schemata," *SIAM J. Comput.* 1 (1972), 66-118.

[25] Constable, R. L. and M. O'Donnell, *A Programming Logic*. Winthrop, Cambridge, Mass., 1978.

[26] Cook, S. A., "Soundness and Completeness of an Axiom System for Program Verification," *SIAM J. Comput.* 7 (1978), 70-80.

[27] Csirmaz, L., "Determinateness of Program Equivalence over Peano Axioms," *Theor. Comput. Sci.* 21 (1982), 231-235.

[28] Csirmaz, L., "A Completeness Theorem for Dynamic Logic," *Notre Dame J. Formal Logic* 26 (1985), 51-60.

[29] Courcoubetis, C. and M. Yannakakis, "Verifying Temporal Properties of Finite-State Probabilistic Programs," *Proc. 29th IEEE Symp. Foundations of Comput. Sci.*, October 1988, 338-345.

[30] Ehrenfeucht, A., "An Application of Games in the Completeness Problem for Formalized Theories," *Fund. Math.* 49 (1961), 129-141.

[31] Emerson, E. A., "Automata, Tableax, and Temporal Logics," in: R. Parikh (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 193, Springer, 1985, 79-88.

[32] Emerson, E. A., "Temporal and Modal Logic," *this volume.*

[33] Emerson, E. A. and J. Y. Halpern, "Decision Procedures and Expressiveness in the Temporal Logic of Branching Time," *Proc. 14th ACM Symp. Theory of Comput.*, 1982, 169-180.

[34] Emerson, E. A. and C. Jutla, "The Complexity of Tree Automata and Logics of Programs," *Proc. 29th IEEE Symp. Foundations of Comput. Sci.*, October 1988, 328-337.

[35] Emerson, E. A. and C. Jutla, "On Simultaneously Determinizing and Complementing ω-Automata," *Proc. IEEE Symp. Logic in Computer Science*, Asilomar, California, June 1989.

[36] Emerson, E. A. and P. A. Sistla, "Deciding Full Branching-Time Logic," *Infor. and Control* 61 (1984), 175-201.

[37] Engeler, E., "Algorithmic Properties of Structures," *Math. Syst. Theory* 1 (1967), 183-195.

[38] Erimbetov, M. M., "On the Expressive Power of Programming Logics," *Proc. Conf. Research in Theoretical Programming*, Alma-Ata, 1981, 49-68 (in Russian).

[39] Feldman, Y. A. and D. Harel, "A Probabilistic Dynamic Logic," *Proc. 14th ACM Symp. Theory of Comput.*, 1982, 181-195.

[40] Fischer, M. J. and R. E. Ladner, "Propositional Modal Logic of Programs," *Proc. 9th ACM Symp. Theory of Comput.*, 1977, 286-294.

[41] Fischer, M. J. and R. E. Ladner, "Propositional Dynamic Logic of Regular Programs," *J. Comput. Syst. Sci.* 18:2 (1979), 194-211.

[42] Floyd, R. W., "Assigning Meanings to Programs," *Proc. AMS Symp. Appl. Math.* 19, Amer. Math. Soc., Providence, 1967, 19-31.

[43] Friedman, H., "Algorithmic Procedures, Generalized Turing Algorithms, and Elementary Recursion Theory," in: Gandy and Yates (eds.), *Logic Colloq. 1969*, North-Holland, Amsterdam, 1971, 361-390.

[44] Gabbay, D., "Axiomatizations of Logics of Programs," unpublished manuscript, Bar-Ilan Univ., Ramat-Gan, Israel, 1977.

[45] Gabbay, D., A. Pnueli, S. Shelah, and J. Stavi, "On the Temporal Analysis of Fairness," *Proc. 7th ACM Symp. Princip. Prog. Lang.*, 1980, 163-173.

[46] German, S. M., E. M. Clarke, and J. T. Halpern, "True Relative Completeness of an Axiom System for the Language *L4*," *Proc. 1st IEEE Symp. Logic in Comput. Sci.*, 1986, 11-25.

[47] Goldblatt, R., *Axiomatising the Logic of Computer Programming.* Lect. Notes in Comput. Sci. 130, Springer, 1982.

[48] Goldblatt, R., *Logics of Time and Computation.* Center for the Study of Language and Information Lect. Notes 7, Stanford Univ., 1987.

[49] Gries, D., *The Science of Programming.* Springer, New York, 1981.

[50] Hajek, P. and P. Kurka, "A Second-Order Dynamic Logic with Array Assignments," *Fund. Informaticae* IV (1981), 919-933.

[51] Halpern, J. Y., "On the Expressive Power of Dynamic Logic II," Tech. Rep. TM-204, MIT Lab. for Comput. Sci., Cambridge, Mass., 1981.

[52] Halpern, J. Y., "Deterministic Process Logic is Elementary," *Proc. 23rd IEEE Symp. Found. Comput. Sci.*, 1982, 204-216.

[53] Halpern, J. Y. and J. H. Reif, "The Propositional Dynamic Logic of Deterministic, Well-Structured Programs," *Proc. 22nd IEEE Symp. Found. Comput. Sci.*, 1981, 322-334.

[54] Harel, D., *First-Order Dynamic Logic.* Lect. Notes in Comput. Sci. 68, Springer, 1979.

[55] Harel, D., "Proving the Correctness of Regular Deterministic Programs: A Unifying Survey Using Dynamic Logic," *Theor. Comput. Sci.* 12 (1980), 61-81.

[56] Harel, D., "Dynamic Logic," in: Gabbay and Guenthner (eds.), *Handbook of Philosophical Logic. II: Extensions of Classical Logic*, D. Reidel, Boston, 1984, 497-604.

[57] Harel, D., A. R. Meyer, and V. R. Pratt, "Computability and Completeness in Logics of Programs," *Proc. 9th ACM Symp. Theory of Comput.*, 1977, 261-268.

[58] Harel, D., D. Kozen, and R. Parikh, "Process Logic: Expressiveness, Decidability, Completeness," *Proc. 21st IEEE Symp. Found. Comput. Sci.*, 1980, 129-142; *J. Comput. Syst. Sci.* 25:2 (1982), 144-170.

[59] Harel, D. and D. Kozen, "A Programming Language for the Inductive Sets, and Applications," *Infor. and Control* 63:1-2 (1984), 118-139.

[60] Harel, D. and D. Peleg, "On Static Logics, Dynamic Logics, and Complexity Classes," *Infor. and Control* 60 (1984), 86-102.

[61] Harel, D. and M. S. Paterson, "Undecidability of *PDL* with $L = \{a^{2^i} \mid i \geq 0\}$," *J. Comput. Syst. Sci.* 29 (1984), 359-365.

[62] Harel, D., A. Pnueli, and J. Stavi, "Propositional Dynamic Logic of Nonregular Programs," *J. Comput. Syst. Sci.* 26 (1983), 222-243.

[63] Harel, D., A. Pnueli, and J. Stavi, "Further Results on Propositional Dynamic Logic of Nonregular Programs," in: D. Kozen (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 131, Springer, 1981, 124-136.

[64] Harel, D., A. Pnueli, and M. Vardi, "Two Dimensional Temporal Logic and *PDL* with Intersection," unpublished manuscript, The Weizmann Inst., Rehovot, Israel, 1982.

[65] Harel, D. and V. R. Pratt, "Nondeterminism in Logics of Programs," *Proc. 5th ACM Symp. Princip. Prog. Lang.*, 1978, 203-213.

[66] Harel, D. and R. Sherman, "Propositional Dynamic Logic with Programs as Automata," unpublished manuscript, The Weizmann Inst., Rehovot, Israel, 1982.

[67] Harel, D. and R. Sherman, "Looping vs. Repeating in Dynamic Logic," *Infor. and Control* 55 (1982), 175-192.

[68] Hart, S., M. Sharir, and A. Pnueli, "Termination of Probabilistic Concurrent Programs," *Proc. 9th ACM Symp. Princip. Prog. Lang.*, 1982, 1-6.

[69] Hitchcock, P. and D. Park, "Induction Rules and Termination Proofs," in: M. Nivat (ed.), *Int. Colloq. Automata Lang. Prog.*, North-Holland, Amsterdam, 1973.

[70] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Comm. ACM* 12 (1967), 516-580.

[71] Hopcroft, J. E. and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[72] Hughes, G. E. and M. J. Cresswell, *An Introduction to Modal Logic.* Methuen, 1968.

[73] Ianov, Y. I., "The Logical Schemes of Algorithms," in: *Problems of Cybernetics 1*, Pergamon Press, New York, 1960, 82-140.

[74] Kfoury, A. J., "Definability by Deterministic and Nondeterministic Programs with Applications to First-Order Dynamic Logic," *Infor. and Control* 65:2-3 (1985), 98-121.

[75] Knijnenburg, P. M. W., "On Axiomatizations for Propositional Logics of Programs," Tech. Report RUU-CS-88-34, Rijksuniversiteit Utrecht, November 1988.

[76] Kozen, D., "On the Duality of Dynamic Algebras and Kripke Models," in: E. Engeler (ed.), *Proc. Workshop on Logic of Programs*, Lect. Notes in Comput. Sci. 125, Springer, 1981, 1-11.

[77] Kozen, D., "A Representation Theorem for Models of *-free *PDL*," in: J. W. de Bakker and J. van Leeuwen (eds.), *Proc. 7th Int. Colloq. Automata Languages and Programming,* Lect. Notes in Comput. Sci. 85, Springer, 1980, 351-362.

[78] Kozen, D., "On Induction vs. *-Continuity," in: D. Kozen (ed.), *Proc. Workshop on Logics of Programs 1981*, Lect. Notes in Comput. Sci. 131, Springer, 1981, 167-176.

[79] Kozen, D., "Semantics of Probabilistic Programs," *J. Comput. Syst. Sci.* 22 (1981), 328-350.

[80] Kozen, D., "Logics of Programs," unpublished lecture notes, University of Aarhus, Denmark, 1981.

[81] Kozen, D., "Results on the Propositional μ-Calculus," in: M. Neilsen and E. M. Schmidt (eds.), *Proc. 9th Int. Colloq. Automata Languages and Programming,* Lect. Notes in Comput. Sci. 140, Springer, 1982, 348-359.

[82] Kozen, D., "A Probabilistic *PDL*," *J. Comput. Syst. Sci.* 30:2 (1985), 162-178.

[83] Kozen, D., "A Finite Model Theorem for the Propositional μ-Calculus," *Studia Logica* 47:3 (1988), 53-61.

[84] Kozen, D. and R. Parikh, "An Elementary Proof of the Completeness of *PDL*," *Theor. Comput. Sci.* 14 (1981), 113-118.

[85] Kozen, D. and R. Parikh, "A Decision Procedure for the Propositional μ-Calculus," in: E. Clarke and D. Kozen (eds.), *Proc. Workshop on Logics of Programs 1983*, Lect. Notes in Comput. Sci. 164, Springer, 1983, 313-325.

[86] Lamport, L., "'Sometime' is Sometimes 'Not Never'," *Proc. 7th ACM Symp. Princip. Prog. Lang.*, 1980, 174-185.

[87] Lehmann, D. and S. Shelah, "Reasoning with Time and Chance," *Infor. and Control* 53:3 (1982), 165-198.

[88] Leivant, D., "Logical and Mathematical Reasoning about Imperative Programs," *Proc. 12th ACM Symp. Princip. Prog. Lang.*, 1985, 132-140.

[89] Leivant, D., "Hoare's Logic Captures Program Semantics (extended summary)," Tech. Rep., Comput. Sci. Dept., Carnegie Mellon Univ., 1985.

[90] Lipton, R. J., "A Necessary and Sufficient Condition for the Existence of Hoare Logics," *Proc. 18th IEEE Symp. Found. Comput. Sci.*, 1977, 1-6.

[91] Luckham, D. C., D. Park, and M. Paterson, "On Formalized Computer Programs," *J. Comput. Syst. Sci.* 4 (1970), 220-249.

[92] Makowski, J. A., "Measuring the Expressive Power of Dynamic Logics: An Application of Abstract Model Theory," *Proc. 7th Int. Colloq. Automata Lang. Prog.*, Lect. Notes in Comput. Sci. 80, Springer, 1980, 409-421.

[93] Makowsky, J. A. and I. Sain, "On the Equivalence of Weak Second-Order and Nonstandard Time Semantics for Various Program Verification Systems," *Proc. 1st IEEE Symp. Logic in Comput. Sci.*, Boston, 1986, 293-300.

[94] Manders, K. L. and R. F. Daley, "The Complexity of the Validity Problem for Dynamic Logic," unpublished manuscript, Univ. of Pittsburgh, 1982.

[95] Manna, Z. and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles," in: D. Kozen (ed.), *Proc. Workshop on Logics of Programs,* Lect. Notes in Comput. Sci. 131, Springer, 1981, 200-252.

[96] Manna, Z. and A. Pnueli, "Specification and Verification of Concurrent Programs by $\forall$-Automata," *Proc. 14th ACM Symp. Principles of Programming Languages*, Munich, January 1987, 1-12.

[97] Meyer, A. R., "Ten Thousand and One Logics of Programming," *Bull. Europ. Assoc. Theor. Comput. Sci.* 10 (1980), 11-29.

[98] Meyer, A. R. and R. Parikh, "Definability in Dynamic Logic," *J. Comput. Syst. Sci.* 23 (1981), 279-298.

[99] Meyer, A. R., R. S. Streett, and G. Mirkowska, "The Deducibility Problem in Propositional Dynamic Logic," in: E. Engeler (ed.), *Proc. Workshop Logic of Programs*, Lect. Notes in Comput. Sci. 125, Springer, 1981, 12-22.

[100] Meyer, A. R. and J. Tiuryn, "A Note on Equivalences Among Logics of Programs," in: D. Kozen (ed.), *Proc. Workshop on Logics of Programs,* Lect. Notes in Comput. Sci. 131, Springer, 1981, 282-299.

[101] Meyer, A. R., and K. Winklmann, "Expressing Program Looping in Regular Dynamic Logic," *Theor. Comput. Sci.* 18 (1982), 301-323.

[102] Meyer, A. R. and J. Y. Halpern, "Axiomatic Definitions of Programming Languages: a Theoretical Assessment," *J. ACM* 29 (1982), 555-576.

[103] Meyer, A. R. and J. C. Mitchell, "Termination Assertions for Recursive Programs: Completeness and Axiomatic Definability," Tech. Rep. TM-214, MIT Lab. for Comput. Sci., Cambridge, Mass., 1982.

[104] Meyer, A. R., "Floyd-Hoare Logic Defines Semantics (preliminary version)," *Proc. IEEE Symp. Logic in Comput. Sci.*, 1986, 44-48.

[105] Mirkowska, G., "On Formalized Systems of Algorithmic Logic," *Bull. Acad. Polon. Sci., Ser. Sci. Math. Astron. Phys.* 19 (1971), 421-428.

[106] Mirkowska, G., "Algorithmic Logic with Nondeterministic Programs," *Fund. Informaticae* III (1980), 45-64.

[107] Mirkowska, G., "*PAL*—Propositional Algorithmic Logic," in: E. Engeler (ed.), *Proc. Workshop Logic of Programs*, Lect. Notes in Comput. Sci. 125, Springer, 1981, 23-101; *Fund. Informaticae* IV (1981), 675-760.

[108] Moschovakis, Y. N., *Elementary Induction on Abstract Structures.* North-Holland, Amsterdam, 1974.

[109] Németi, I., "Nonstandard Dynamic Logic," in: D. Kozen (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 131, Springer, 1981, 311-348.

[110] Nishimura, H., "Sequential Method in Propositional Dynamic Logic," *Acta Informatica* 12 (1979), 377-400.

[111] Nishimura, H., "Descriptively Complete Process Logic," *Acta Informatica* 14 (1980), 359-369.

[112] Nishimura, H., "Arithmetical Completeness in First-Order Dynamic Logic for Concurrent Programs," *Publ. Research Inst. Math. Sci.* 17, Kyoto Univ., 1981, 297-309.

[113] Niwinski, D., "The Propositional μ-Calculus is More Expressive than the Propositional Dynamic Logic of Looping," unpublished manuscript, 1984.

[114] Olshanski, T. and A. Pnueli. "There Exist Decidable Context Free Propositional Dynamic Logics," unpublished manuscript, The Weizmann Inst., Rehovot, Israel, 1981.

[115] Parikh, R., "The Completeness of Propositional Dynamic Logic," *Proc. 7th Symp. on Math. Found. of Comput. Sci.*, Lect. Notes in Comput. Sci. 64, Springer, 1978, 403-415.

[116] Parikh, R., "A Decidability Result for Second Order Process Logic," *Proc. 19th IEEE Symp. Found. Comput. Sci.*, 1978, 177-183.

[117] Parikh, R., "Propositional Dynamic Logics of Programs: A Survey," in: E. Engeler (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 125, Springer, 1981, 102-144.

[118] Parikh, R., "Propositional Logics of Programs," *Proc. 7th ACM Symp. Princip. Prog. Lang.*, 1980, 186-192.

[119] Parikh, R. and A. Mahoney, "A Theory of Probabilistic Programs," in: E. Clarke and D. Kozen (eds.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 164, Springer, 1983, 396-402.

[120] Park, D., "Finiteness is μ-Ineffable," *Theor. Comput. Sci.* 3 (1976), 173-181.

[121] Paterson, M. S. and C. E. Hewitt, "Comparative Schematology," in: *Record Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM, New York, 1970, 119-128.

[122] Pecuchet, J. P., "On the Complementation of Büchi Automata," *Theor. Comput. Sci.* 47 (1986), 95-98.

[123] Peleg, D., "Concurrent Dynamic Logic," *J. ACM* 34:2 (1987), 450-479.

[124] Peterson, G. L., "The Power of Tests in Propositional Dynamic Logic," Tech. Rep. 47, Dept. of Comput. Sci., Univ. of Rochester, 1978.

[125] Pnueli, A., "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. Found. Comput. Sci.*, 1977, 46-57.

[126] Pratt, V. R., "Semantical Considerations on Floyd-Hoare Logic," *Proc. 17th IEEE Symp. Found. Comput. Sci.* 1976, 109-121.

[127] Pratt, V. R., "A Practical Decision Method for Propositional Dynamic Logic," *Proc. 10th ACM Symp. Theory of Comput.,* 1978, 326-337.

[128] Pratt, V. R., "Process Logic," *Proc. 6th ACM Symp. Princip. Prog. Lang.,* 1979, 93-100.

[129] Pratt, V. R., "Models of Program Logics," *Proc. 20th IEEE Symp. Found. Comput. Sci.*, 1979, 115-122.

[130] Pratt, V. R., "Using Graphs to Understand *PDL*," in: D. Kozen (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 131, Springer, 1981, 387-396.

[131] Pratt, V. R., "Dynamic Algebras and the Nature of Induction," *Proc. 12th ACM Symp. Theory of Comput.*, 1980, 22-28.

[132] Pratt, V. R., "A Decidable μ-Calculus: Preliminary Report," *Proc. 22nd IEEE Symp. Found. Comput. Sci.*, 1981, 421-427.

[133] Rabin, M. O., "Decidability of Second Order Theories and Automata on Infinite Trees," *Trans. Amer. Math. Soc.* 141 (1969), 1-35.

[134] Ramshaw, L. H., "Formalizing the Analysis of Algorithms," Ph.D. thesis, Stanford Univ., 1981.

[135] Safra, S., "On the Complexity of ω-Automata," *Proc. 29th IEEE Symp. Foundations of Comput. Sci.*, October 1988, 319-327.

[136] Salwicki, A., "Formalized Algorithmic Languages," *Bull. Acad. Polon. Sci., Ser. Sci. Math. Astron. Phys.* 18 (1970), 227-232.

[137] Salwicki, A., "Algorithmic Logic: A Tool for Investigations of Programs," in: Butts and Hintikka (eds.), *Logic, Foundations of Mathematics, and Computability Theory*, Reidel, 1977, 281-295.

[138] Scott, D. S. and J. W. de Bakker, "A Theory of Programs," unpublished notes, IBM Vienna, 1969.

[139] Segerberg, K., "A Completeness Theorem in the Modal Logic of Programs (Preliminary Report)," *Not. Amer. Math. Soc.* 24:6, A-552 (1977).

[140] Sistla, A. P. and E.M. Clarke, "The Complexity of Propositional Linear Temporal Logics," *Proc. 14th ACM Symp. Theory of Comput.*, 1982, 159-168.

[141] Sistla, A. P., M. Y. Vardi, and P. Wolper, "The Complementation Problem for Büchi Automata with Application to Temporal Logic," *Theor. Comput. Sci.* 49 (1987), 217-237.

[142] Sokolowski, S., "Programs as Term Transformers," *Fund. Informaticae* III (1980), 419-432.

[143] Stolboushkin, A. P. and M. A. Taitslin, "Deterministic Dynamic Logic is Strictly Weaker than Dynamic Logic," *Infor. and Control* 57 (1983), 48-55.

[144] Streett, R. S., "Propositional Dynamic Logic of Looping and Converse," *Proc. 13th ACM Symp. Theory of Comput.,* 1981, 375-381.

[145] Streett, R. S., "Propositional Dynamic Logic of Looping and Converse is Elementarily Decidable," *Infor. and Control* 54 (1982), 121-141.

[146] Streett, R. S., "Fixpoints and Program Looping: Reductions from the Propositional μ-Calculus into Propositional Dynamic Logics of Looping," in: R. Parikh (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 193, Springer, 1985, 359-372.

[147] Thiele, H., *Wissenschaftstheoretische Untersuchungen in Algorithmischen Sprachen. Theorie der Graphschemata-Kalkale.* Veb. Deutscher Verlag der Wissenschaften, Berlin, 1966.

[148] Tiuryn, J., "Unbounded Program Memory Adds to the Expressive Power of First-Order Programming Logics," *Proc. 22nd IEEE Symp. Found. Comput. Sci.*, 1981, 335-339; *Infor. and Control* 60 (1984), 12-35.

[149] Tiuryn, J., "A Survey of the Logic of Effective Definitions," in: E. Engeler (ed.), *Proc. Workshop on Logics of Programs,* Lect. Notes in Comput. Sci. 125, Springer, 1981, 198-245.

[150] Tiuryn, J., and P. Urzyczyn, "Some Relationships between Logics of Programs and Complexity Theory," *Proc. 24th IEEE Symp. Found. Comput. Sci.* 1983, 180-184; *Theor. Comput. Sci.* 60 (1988), 83-108.

[151] Tiuryn, J. and P. Urzyczyn, "Remarks on Comparing Expressive Power of Logics of Programs," in: Chytil and Koubek (eds.), *Proc. Math. Found. Comput. Sci.,* Lect. Notes in Comput. Sci. 176, Springer, 1984, 535-543.

[152] Tiuryn, J., "A Simplified Proof of $DDL < DL$," Tech. Rep., Dept. Comput. Sci., Washington State Univ., Pullman, 1985; *Infor. and Control* (1989), to appear.

[153] Tiuryn, J., "Higher-Order Arrays and Stacks in Programming: An Application of Complexity Theory to Logics of Programs," in: Gruska and Rovan (eds.), *Proc. Math. Found. Comput. Sci.*, Lect. Notes in Comput. Sci. 233, Springer, 1986, 177-198.

[154] Trnkova, V., and J. Reiterman, "Dynamic Algebras which are not Kripke Structures," *Proc. 9th Symp. on Math. Found. Comput. Sci.*, 1980, 528-538.

[155] Urzyczyn, P., "Deterministic Context-Free Dynamic Logic is More Expressive than Deterministic Dynamic Logic of Regular Programs," in: M. Karpinski (ed.), *Proc. 1983 Conf. Fund. Comput. Theory,* Lect. Notes in Comput. Sci. 158, Springer, 1983, 496-504.

[156] Urzyczyn, P., "Nontrivial Definability by Flowchart Programs," *Infor. and Control* 58 (1983), 59-87.

[157] Valiev, M.K., "Decision Complexity of Variants of Propositional Dynamic Logic," *Proc. 9th Symp. Math. Found. Comput. Sci.,* Lect. Notes in Comput. Sci. 88, Springer, 1980, 656-664.

[158] Vardi, M. Y., "Automatic Verification of Probabilistic Concurrent Finite-State Programs," *Proc. 26th IEEE Symp. Foundations of Comput. Sci.*, October 1985, 327-338.

[159] Vardi, M. Y., "The Taming of the Converse: Reasoning about Two-Way Computations," in: R. Parikh (ed.), *Proc. Workshop on Logics of Programs*, Lect. Notes in Comput. Sci. 193, Springer, 1985, 413-424.

[160] Vardi, M. Y., "Verification of Concurrent Programs: the Automata-Theoretic Framework," *Proc. IEEE Symp. Logic in Computer Science*, Ithaca, New York, June 1987, 167-176.

[161] Vardi, M. and L. Stockmeyer, "Improved Upper and Lower Bounds for Modal Logics of Programs: Preliminary Report," *Proc. 17th ACM Symp. Theory of Comput.*, May 1985, 240-251.

[162] Vardi, M. Y. and P. Wolper, "Yet Another Process Logic," in: E. Clarke and D. Kozen (eds.), *Proc. Workshop on Logics of Programs 1983*, Lect. Notes in Comput. Sci. 164, Springer, 1983, 501-512.

[163] Vardi, M. Y. and P. Wolper, "Automata-Theoretic Techniques for Modal Logics of Programs," *Proc. 14th ACM Symp. Theory of Computing*, May 1984, 446-455.

[164] Vardi, M. Y. and P. Wolper, "An Automata-Theoretic Approach to Automatic Program Verification," *Proc. IEEE Symp. on Logic in Computer Science*, Cambridge, June 1986, 332-344.

[165] Wand, M., "A New Incompleteness Result for Hoare's System," *J. ACM* 25 (1978), 168-175.