III FIRST-ORDER DYNAMIC LOGIC

11 First-Order Dynamic Logic 283
11.1 Basic Syntax 283
11.2 Richer Programs 287
11.3 Semantics 291
11.4 Satisfiability and Validity 297
11.5 Bibliographical Notes 298
Exercises 298
12 Relationships with Static Logics 301
12.1 The Uninterpreted Level 301
12.2 The Interpreted Level 307
12.3 Bibliographical Notes 311
Exercises 311
13 Complexity 313
13.1 The Validity Problem 313
13.2 Spectral Complexity 317
13.3 Bibliographical Notes 324
Exercises 325
14 Axiomatization 327
14.1 The Uninterpreted Level 327
14.2 The Interpreted Level 333
14.3 Bibliographical Notes 341
Exercises 341
15 Expressive Power 343
15.1 The Unwind Property 344
15.2 Spectra and Expressive Power 347
15.3 Bounded Nondeterminism 355

15.4 Unbounded Memory 369
15.5 The Power of a Boolean Stack 376
15.6 Unbounded Nondeterminism 377
15.7 Bibliographical Notes 378
Exercises 380
16 Variants of DL 383
16.1 Algorithmic Logic 383
16.2 Nonstandard Dynamic Logic 384
16.3 Well-Foundedness 386
16.4 Dynamic Algebra 389
16.5 Probabilistic Programs 391
16.6 Concurrency and Communication 393
16.7 Bibliographical Notes 394
17 Other Approaches 397
17.1 Logic of Effective Definitions 397
17.2 Temporal Logic 398
17.3 Process Logic 408
17.4 The μ-Calculus 415
17.5 Kleene Algebra 418

References 425
Notation and Abbreviations 439
Index 449

11 First-Order Dynamic Logic

In this chapter we begin the study of first-order Dynamic Logic. The main difference between first-order DL and the propositional version PDL discussed in Part II of the book is the presence of a first-order structure $\mathfrak{A}$, 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 A, the *carrier* of $\mathfrak{A}$. Atomic programs in DL are no longer abstract binary relations, but *assignment statements* of various forms, all based on assigning values to variables during the computation. The most basic example of such an assignment is the *simple assignment* $x := t$, where x is a variable and t is a term. The atomic formulas of DL are generally taken to be atomic first-order formulas.

In addition to the constructs introduced in Part II, the basic DL syntax contains individual variables ranging over A, function and predicate symbols for distinguished functions and predicates of $\mathfrak{A}$, and quantifiers ranging over A, exactly as in classical first-order logic. More powerful versions of the logic contain array and stack variables and other constructs, as well as primitive operations for manipulating them, and assignments for changing their values. Sometimes the introduction of a new construct increases expressive power and sometimes not; sometimes it has an effect on the complexity of deciding satisfiability and sometimes not. Indeed, one of the central goals of Part III of the book is to classify these constructs in terms of their relative expressive power and complexity.

In this chapter we lay the groundwork for this by defining the various logical and programming constructs we shall be needing.

11.1 Basic Syntax

The language of first-order Dynamic Logic is built upon classical first-order logic as described in Section 3.4. There is always an underlying first-order vocabulary Σ, which involves a vocabulary of function symbols and predicate (or relation) symbols. On top of this vocabulary, we define a set of *programs* and a set of *formulas*. These two sets interact by means of the modal construct [ ] exactly as in the propositional case. Programs and formulas are usually defined by mutual induction.

Let $\Sigma = \{f, g, \ldots, p, r, \ldots\}$ be a finite first-order vocabulary. Here f and g denote typical function symbols of Σ, and p and r denote typical relation symbols. Associated with each function and relation symbol of Σ is a fixed *arity* (number of arguments), although we do not represent the arity explicitly. We assume

that Σ always contains the equality symbol $=$, whose arity is 2. Functions and relations of arity $0, 1, 2, 3$ and n are called *nullary*, *unary*, *binary*, *ternary*, and *n-ary*, respectively. Nullary functions are also called *constants*. We shall be using a countable set of *individual variables* $V = \{x_0, x_1, \ldots\}$.

The definitions of DL programs and formulas below depend on the vocabulary Σ, but in general we shall not make this dependence explicit unless we have some specific reason for doing so.

Atomic Formulas and Programs

In all versions of DL that we will consider, atomic formulas are atomic formulas of the first-order vocabulary Σ; that is, formulas of the form

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

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

As in PDL, programs are defined inductively from atomic programs using various programming constructs. The meaning of a compound program is given inductively in terms of the meanings of its constituent parts. Different classes of programs are obtained by choosing different classes of atomic programs and programming constructs.

In the basic version of DL, an atomic program is a *simple assignment*

$$x := t, \tag{11.1.1}$$

where $x \in V$ and t is a term of Σ. Intuitively, this program assigns the value of t to the variable x. This is the same form of assignment found in most conventional programming languages.

More powerful forms of assignment such as stack and array assignments and nondeterministic “wildcard” assignments will be discussed later. The precise choice of atomic programs will be made explicit when needed, but for now, we use the term *atomic program* to cover all of these possibilities.

Tests

As in PDL, DL contains a test operator ?, which turns a formula into a program. In most versions of DL that we shall discuss, we allow only quantifier-free first-order formulas as tests. We sometimes call these versions *poor test*. Alternatively, we might allow any first-order formula as a test. Most generally, we might place no restrictions on the form of tests, allowing any DL formula whatsoever, including those that contain other programs, perhaps containing other tests, etc. These

versions of DL are labeled *rich test* as in Section 10.2. Whereas programs can be defined independently from formulas in poor test versions, rich test versions require a mutually inductive definition of programs and formulas.

As with atomic programs, the precise logic we consider at any given time depends on the choice of tests we allow. We will make this explicit when needed, but for now, we use the term *test* to cover all possibilities.

Regular Programs

For a given set of atomic programs and tests, the set of *regular programs* is defined as in PDL (see Section 5.1):

- any atomic program or test is a program;
- if α and β are programs, then $\alpha\,;\,\beta$ is a program;
- if α and β are programs, then $\alpha \cup \beta$ is a program;
- if α is a program then α^* is a program.

While Programs

Some of the literature on DL is concerned with the class of **while** *programs*. This class was defined formally in Section 10.1 for PDL (see also Section 5.1); the definition is the same here.

Formally, *deterministic* **while** *programs* form the subclass of the regular programs in which the program operators $\cup$, ?, and * are constrained to appear only in the forms

$$\begin{aligned}
\textbf{skip} &\stackrel{\text{def}}{=} \mathbf{1}? \\
\textbf{fail} &\stackrel{\text{def}}{=} \mathbf{0}? \\
\textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta &\stackrel{\text{def}}{=} (\varphi?;\alpha) \cup (\neg\varphi?;\beta) && (11.1.2) \\
\textbf{while } \varphi \textbf{ do } \alpha &\stackrel{\text{def}}{=} (\varphi?;\alpha)^*;\neg\varphi? && (11.1.3)
\end{aligned}$$

The class of *nondeterministic* **while** *programs* is the same, except that we allow unrestricted use of the nondeterministic choice construct $\cup$. Of course, unrestricted use of the sequential composition operator is allowed in both languages.

Restrictions on the form of atomic programs and tests apply as with regular programs. For example, if we are allowing only poor tests, then the φ occurring in the programs (11.1.2) and (11.1.3) must be a quantifier-free first-order formula.

The class of deterministic **while** programs is important because it captures the basic programming constructs common to many real-life imperative programming

languages. Over the standard structure of the natural numbers $\mathbb{N}$, deterministic **while** programs are powerful enough to define all partial recursive functions, and thus over $\mathbb{N}$ they are as as expressive as regular programs. A similar result holds for a wide class of models similar to $\mathbb{N}$, for a suitable definition of "partial recursive functions" in these models. However, it is not true in general that **while** programs, even nondeterministic ones, are universally expressive. We discuss these results in Chapter 15.

Formulas

A *formula* of DL is defined in way similar to that of PDL, with the addition of a rule for quantification. Equivalently, we might say that a formula of DL is defined in a way similar to that of first-order logic, with the addition of a rule for modality. The basic version of DL is defined with regular programs:

- the false formula $\mathbf{0}$ is a formula;
- any atomic formula is a formula;
- if φ and ψ are formulas, then $\varphi \to \psi$ is a formula;
- if φ is a formula and $x \in V$, then $\forall x\ \varphi$ is a formula;
- if φ is a formula and α is a program, then $[\alpha]\varphi$ is a formula.

The only missing rule in the definition of the syntax of DL are the tests. In our basic version we would have:

- if φ is a quatifier-free first-order formula, then $\varphi?$ is a test.

For the rich test version, the definitions of programs and formulas are mutually dependent, and the rule defining tests is simply:

- if φ is a formula, then $\varphi?$ is a test.

We will use the same notation as in propositional logic that $\neg\varphi$ stands for $\varphi \to \mathbf{0}$. As in first-order logic, the first-order existential quantifier $\exists$ is considered a defined construct: $\exists x\ \varphi$ abbreviates $\neg\forall x\ \neg\varphi$. Similarly, the modal construct $< >$ is considered a defined construct as in Section 5.1, since it is the modal dual of $[\ ]$. The other propositional constructs $\wedge$, $\vee$, $\leftrightarrow$ are defined as in Section 3.2. Of course, we use parentheses where necessary to ensure unique readability.

Note that the individual variables in V serve a dual purpose: they are both program variables and logical variables.

11.2 Richer Programs

Seqs and R.E. Programs

Some classes of programs are most conveniently defined as certain sets of seqs. Recall from Section 5.3 that a *seq* is a program of the form $\sigma_1; \cdots ; \sigma_k$, where each σ_i is an assignment statement or a quantifier-free first-order test. Each regular program α is associated with a unique set of seqs $CS(\alpha)$ (Section 5.3). These definitions were made in the propositional context, but they apply equally well to the first-order case; the only difference is in the form of atomic programs and tests.

Construing the word in the broadest possible sense, we can consider a *program* to be an arbitrary set of seqs. Although this makes sense semantically—we can assign an input/output relation to such a set in a meaningful way—such programs can hardly be called executable. At the very least we should require that the set of seqs be recursively enumerable, so that there will be some effective procedure that can list all possible executions of a given program. However, there is a subtle issue that arises with this notion. Consider the set of seqs

$$\{x_i := f^i(c) \mid i \in \mathbb{N}\}.$$

This set satisfies the above restriction, yet it can hardly be called a program. It uses infinitely many variables, and as a consequence it might change a valuation at infinitely many places. Another pathological example is the set of seqs

$$\{x_{i+1} := f(x_i) \mid i \in \mathbb{N}\},$$

which not only could change a valuation at infinitely many locations, but also depends on infinitely many locations of the input valuation.

In order to avoid such pathologies, we will require that each program use only finitely many variables. This gives rise to the following definition of *r.e. programs*, which is the most general family of programs we will consider. Specifically, an r.e. program α is a Turing machine that enumerates a set of seqs over a finite set of variables. The set of seqs enumerated will be called $CS(\alpha)$. By $FV(\alpha)$ we will denote the finite set of variables that occur in seqs of $CS(\alpha)$.

An important issue connected with r.e. programs is that of *bounded memory*. The assignment statements or tests in an r.e. program may have infinitely many terms with increasingly deep nesting of function symbols (although, as discussed, these terms only use finitely many variables), and these could require an unbounded amount of memory to compute. We define a set of seqs to be *bounded memory* if the depth of terms appearing in it is bounded. In fact, without sacrificing computational

power, we could require that all terms be of the form $f(x_1, \ldots, x_n)$ in a bounded-memory set of seqs (Exercise 15.4).

Arrays and Stacks

Interesting variants of the programming language we use in DL arise from allowing auxiliary data structures. We shall define versions with *arrays* and *stacks*, as well as a version with a nondeterministic assignment statement called *wildcard assignment*.

Besides these, one can imagine augmenting **while** programs with many other kinds of constructs such as blocks with declarations, recursive procedures with various parameter passing mechanisms, higher-order procedures, concurrent processes, etc. It is easy to arrive at a family consisting of thousands of programming languages, giving rise to thousands of logics. Obviously, we have had to restrict ourselves. It is worth mentioning, however, that certain kinds of recursive procedures are captured by our stack operations, as explained below.

Arrays

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

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

Each array variable has an associated *arity*, or number of arguments, 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 the set V of individual variables with the set of nullary array variables; thus $V \subseteq V_{\text{array}}$.

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 both as *indices* into an array and as *values* that can be stored in an array. One might equally well introduce a separate sort for array indices; although conceptually simple, this would complicate the notation and would give no new insight.

We extend the set of first-order terms to allow the unrestricted occurrence of array variables, provided arities are respected.

The classes of *regular programs with arrays* and *deterministic* and *nondeterministic* **while** *programs with arrays* are defined similarly to the classes without, except that we allow *array assignments* in addition to simple assignments. Array assignments are similar to simple assignments, but on the left-hand side we allow a term in which the outermost symbol is an array variable:

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

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

Recursion via an Algebraic Stack

We now consider DL in which the programs can manipulate a stack. The literature in automata theory and formal languages often distinguishes a stack from a pushdown store. In the former, the automaton is allowed to inspect the contents of the stack but to make changes only at the top. We shall use the term stack to denote the more common pushdown store, where the only inspection allowed is at the top of the stack.

The motivation for this extension is to be able to capture recursion. It is well known that recursive procedures can be modeled using a stack, and for various technical reasons we prefer to extend the data-manipulation capabilities of our programs than to introduce new control constructs. When it encounters a recursive call, the stack simulation of recursion will push the return location and values of local variables and parameters on the stack. It will pop them upon completion of the call. The LIFO (last-in-first-out) nature of stack storage fits the order in which control executes recursive calls.

To handle the stack in our stack version of DL, we add two new atomic programs

$$\mathbf{push}(t) \quad \text{and} \quad \mathbf{pop}(y),$$

where t is a term and $y \in V$. Intuitively, $\mathbf{push}(t)$ pushes the current value of t onto the top of the stack, and $\mathbf{pop}(y)$ pops the top value off the top of the stack and assigns that value to the variable y. If the stack is empty, the pop operation does not change anything. We could have added a test for stack emptiness, but it can be shown to be redundant (Exercise 11.3). Formally, the stack is simply a finite string of elements of the domain of computation.

The classes of *regular programs with stack* and *deterministic* and *nondeterministic* **while** *programs with stack* are obtained by augmenting the respective classes of programs with the **push** and **pop** operations as atomic programs in addition to simple assignments.

In contrast to the case of arrays, here there is only a single stack. In fact, expressiveness changes dramatically when two or more stacks are allowed (Exercise 15.7). Also, in order to be able to simulate recursion, the domain must have at least two distinct elements so that return addresses can be properly encoded in the stack. One way of doing this is to store the return address itself in unary using one ele-

ment of the domain, then store one occurrence of the second element as a delimiter symbol, followed by domain elements constituting the current values of parameters and local variables.

The kind of stack described here is often termed *algebraic*, since it contains elements from the domain of computation. It should be contrasted with the Boolean stack described next.

Parameterless Recursion via a Boolean Stack

An interesting special case is when the stack can contain only two distinct elements. This version of our programming language can be shown to capture recursive procedures without parameters or local variables. This is because we only need to store return addresses, but no actual data items from the domain of computation. This can be achieved using two values, as described above. We thus arrive at the idea of a Boolean stack.

To handle such a stack in this version of DL, we add three new kinds of atomic programs and one new test. The atomic programs are

push-1 **push-0** **pop**,

and the test is simply

top?.

Intuitively, **push-1** and **push-0** push the corresponding distinct Boolean values on the stack, **pop** removes the top element, and the test **top**? evaluates to true iff the top element of the stack is **1**, but with no side effect.

With the test **top**? only, there is no explicit operator that distinguishes a stack with top element **0** from the empty stack. We might have defined such an operator, and in a more realistic language we would certainly do so. However, it is mathematically redundant, since it can be simulated with the operators we already have (Exercise 11.1).

Wildcard Assignment

The nondeterministic assignment

$x := ?$

is a device that arises in the study of fairness; see Apt and Plotkin (1986). It has often been called *random assignment* in the literature, although it has nothing to do with randomness or probability. We shall call it *wildcard assignment*. Intuitively,

it operates by assigning a nondeterministically chosen element of the domain of computation to the variable x. This construct together with the [ ] modality is similar to the first-order universal quantifier, since it will follow from the semantics that the two formulas

$$[x := ?]\varphi \qquad \text{and} \qquad \forall x\ \varphi$$

are equivalent. However, wildcard assignment may appear in programs and can therefore be iterated.

11.3 Semantics

In this section we assign meanings to the syntactic constructs described in the previous sections. We interpret programs and formulas over a first-order structure $\mathfrak{A}$. Variables range over the carrier of this structure. We take an *operational* view of program semantics: programs change the values of variables by sequences of simple assignments $x := t$ or other assignments, and flow of control is determined by the truth values of tests performed at various times during the computation.

States as Valuations

An instantaneous snapshot of all relevant information at any moment during the computation is determined by the values of the program variables. Thus our *states* will be *valuations* $u, v, \ldots$ of the variables V over the carrier of the structure $\mathfrak{A}$. Our formal definition will associate the pair (u, v) of such valuations with the program α if it is possible to start in valuation u, execute the program α, and halt in valuation v. In this case, we will call (u, v) an *input/output pair* of α and write $(u, v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha)$. This will result in a Kripke frame exactly as in Chapter 5.

Let

$$\mathfrak{A} \quad = \quad (A,\ \mathfrak{m}_{\mathfrak{A}})$$

be a first-order structure for the vocabulary Σ as defined in Section 3.4. We call $\mathfrak{A}$ the *domain of computation*. Here A is a set, called the *carrier* of $\mathfrak{A}$, and $\mathfrak{m}_{\mathfrak{A}}$ is a *meaning function* such that $\mathfrak{m}_{\mathfrak{A}}(f)$ is an n-ary function $\mathfrak{m}_{\mathfrak{A}}(f) : A^n \to A$ interpreting the n-ary function symbol f of Σ, and $\mathfrak{m}_{\mathfrak{A}}(r)$ is an n-ary relation $\mathfrak{m}_{\mathfrak{A}}(r) \subseteq A^n$ interpreting the n-ary relation symbol r of Σ. The equality symbol $=$ is always interpreted as the identity relation.

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

The structure $\mathfrak{A}$ determines a Kripke frame, which we will also denote by $\mathfrak{A}$, as follows. A *valuation* over $\mathfrak{A}$ is a function u assigning an n-ary function over A to each n-ary array variable. It also assigns meanings to the stacks as follows. We shall use the two unique variable names STK and $BSTK$ to denote the algebraic stack and the Boolean stack, respectively. The valuation u assigns a finite-length string of elements of A to STK and a finite-length string of Boolean values $\mathbf{1}$ and $\mathbf{0}$ to $BSTK$. Formally:

$$\begin{array}{rcll} u(F) & \in & A^n \to A, & \text{if } F \text{ is an } n\text{-ary array variable,} \\ u(STK) & \in & A^*, & \\ u(BSTK) & \in & \{\mathbf{1},\mathbf{0}\}^*. & \end{array}$$

By our convention $A^0 \to A = A$, and assuming that $V \subseteq V_{\text{array}}$, the individual variables (that is, the nullary array variables) are assigned elements of A under this definition:

$u(x) \in A$ if $x \in V$.

The valuation u extends uniquely to terms t by induction. For an n-ary function symbol f and an n-ary array variable F,

$$\begin{array}{rcl} u(f(t_1,\ldots,t_n)) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{A}}(f)(u(t_1),\ldots,u(t_n)) \\ u(F(t_1,\ldots,t_n)) & \stackrel{\text{def}}{=} & u(F)(u(t_1),\ldots,u(t_n)). \end{array}$$

Recall the *function-patching* operator defined in Section 1.3: if X and D are sets, $f : X \to D$ is any function, $x \in X$, and $d \in D$, then $f[x/d] : X \to D$ is the function defined by

$$f[x/d](y) \stackrel{\text{def}}{=} \begin{cases} d, & \text{if } x = y \\ f(y), & \text{otherwise.} \end{cases}$$

We will be using this notation in several ways, both at the logical and metalogical levels. For example:

- If u is a valuation, x is an individual variable, and $a \in A$, then $u[x/a]$ is the new valuation obtained from u by changing the value of x to a and leaving the values of all other variables intact.
- If F is an n-ary array variable and $f : A^n \to A$, then $u[F/f]$ is the new valuation that assigns the same value as u to the stack variables and to all array variables other than F, and

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

• If $f : A^n \to A$ is an n-ary function and $a_1, \ldots, a_n, a \in A$, then the expression $f[a_1, \ldots, a_n/a]$ denotes the n-ary function that agrees with f everywhere except for input $a_1, \ldots, a_n$, on which it takes the value a. More precisely,

$$f[a_1, \ldots, a_n/a](b_1, \ldots, b_n) \quad = \quad \begin{cases} a, & \text{if } b_i = a_i,\ 1 \le i \le n \\ f(b_1, \ldots, b_n), & \text{otherwise.} \end{cases}$$

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

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

for all but finitely many array variables F and n-tuples $a_1, \ldots, a_n \in A^n$. In other words, u and v differ on at most finitely many array variables, and for those F on which they do differ, the functions $u(F)$ and $v(F)$ differ on at most finitely many values.

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

We are now ready to define the *states* of our Kripke frame. For $a \in A$, let w_a be the valuation in which the stacks are empty and all array and individual variables are interpreted as constant functions taking the value a everywhere. A *state* of $\mathfrak{A}$ is any finite variant of a valuation w_a. The set of states of $\mathfrak{A}$ is denoted $S^{\mathfrak{A}}$.

Call a state *initial* if it differs from some w_a only at the values of individual variables.

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 above is to prevent arrays from being initialized with highly complex oracles that would compromise the value of the relative expressiveness results of Chapter 15.

Assignment Statements

As in Section 5.2, with every program α we associate a binary relation

$$\mathfrak{m}_{\mathfrak{A}}(\alpha) \quad \subseteq \quad S^{\mathfrak{A}} \times S^{\mathfrak{A}}$$

(called the *input/output relation* of p), and with every formula φ we associate a set

$$\mathfrak{m}_{\mathfrak{A}}(\varphi) \quad \subseteq \quad S^{\mathfrak{A}}.$$

The sets $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ and $\mathfrak{m}_{\mathfrak{A}}(\varphi)$ are defined by mutual induction on the structure of α and φ.

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

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

$$\mathfrak{m}_{\mathfrak{A}}(F(t_1, \ldots, t_n) := t) \quad \stackrel{\text{def}}{=} \quad \{(u, u[F/u(F)[u(t_1), \ldots, u(t_n)/u(t)]]) \mid u \in S^{\mathfrak{A}}\}.$$

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

$$\mathfrak{m}_{\mathfrak{A}}(x := t) \quad \stackrel{\text{def}}{=} \quad \{(u, u[x/u(t)]) \mid u \in S^{\mathfrak{A}}\}.$$

- The push operations, $\mathbf{push}(t)$ for the algebraic stack and **push-1** and **push-0** for the Boolean stack, are interpreted as the binary relations

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{A}}(\mathbf{push}(t)) \quad &\stackrel{\text{def}}{=} \quad \{(u, u[STK/(u(t) \cdot u(STK))]) \mid u \in S^{\mathfrak{A}}\} \\
\mathfrak{m}_{\mathfrak{A}}(\mathbf{push\text{-}1}) \quad &\stackrel{\text{def}}{=} \quad \{(u, u[BSTK/(\mathbf{1} \cdot u(BSTK))]) \mid u \in S^{\mathfrak{A}}\} \\
\mathfrak{m}_{\mathfrak{A}}(\mathbf{push\text{-}0}) \quad &\stackrel{\text{def}}{=} \quad \{(u, u[BSTK/(\mathbf{0} \cdot u(BSTK))]) \mid u \in S^{\mathfrak{A}}\},
\end{aligned}$$

respectively. In other words, $\mathbf{push}(t)$ changes the value of the algebraic stack variable STK from $u(STK)$ to the string $u(t) \cdot u(STK)$, the concatenation of the value $u(t)$ with the string $u(STK)$, and everything else is left intact. The effects of **push-1** and **push-0** are similar, except that the special constants **1** and **0** are concatenated with $u(BSTK)$ instead of $u(t)$.

- The pop operations, $\mathbf{pop}(y)$ for the algebraic stack and **pop** for the Boolean stack, are interpreted as the binary relations

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{A}}(\mathbf{pop}(y)) \quad &\stackrel{\text{def}}{=} \quad \{(u, u[STK/\mathbf{tail}(u(STK))][y/\mathbf{head}(u(STK), u(y))]) \mid u \in S^{\mathfrak{A}}\} \\
\mathfrak{m}_{\mathfrak{A}}(\mathbf{pop}) \quad &\stackrel{\text{def}}{=} \quad \{(u, u[BSTK/\mathbf{tail}(u(BSTK))]) \mid u \in S^{\mathfrak{A}}\},
\end{aligned}$$

respectively, where

$$\begin{aligned}
\mathbf{tail}(a \cdot \sigma) &\stackrel{\text{def}}{=} \sigma\\
\mathbf{tail}(\varepsilon) &\stackrel{\text{def}}{=} \varepsilon\\
\mathbf{head}(a \cdot \sigma, b) &\stackrel{\text{def}}{=} a\\
\mathbf{head}(\varepsilon, b) &\stackrel{\text{def}}{=} b
\end{aligned}$$

and ε is the empty string. In other words, if $u(STK) \neq \varepsilon$, this operation changes the value of STK from $u(STK)$ to the string obtained by deleting the first element of $u(STK)$ and assigns that element to the variable y. If $u(STK) = \varepsilon$, then nothing is changed. Everything else is left intact. The Boolean stack operation **pop** changes the value of $BSTK$ only, with no additional changes. We do not include explicit constructs to test whether the stacks are empty, since these can be simulated (Exercise 11.3). However, we do need to be able to refer to the value of the top element of the Boolean stack, hence we include the **top**? test.

- The Boolean test program **top**? is interpreted as the binary relation

$$\mathfrak{m}_{\mathfrak{A}}(\mathbf{top}?) \quad \stackrel{\text{def}}{=} \quad \{(u, u) \mid u \in S^{\mathfrak{A}},\ \mathbf{head}(u(BSTK)) = \mathbf{1}\}.$$

In other words, this test changes nothing at all, but allows control to proceed iff the top of the Boolean stack contains **1**.

- The wildcard assignment $x := ?$ for $x \in V$ is interpreted as the relation

$$\mathfrak{m}_{\mathfrak{A}}(x := ?) \quad \stackrel{\text{def}}{=} \quad \{(u, u[x/a]) \mid u \in S^{\mathfrak{A}},\ a \in A\}.$$

As a result of executing this statement, x will be assigned some arbitrary value of the carrier set A, and the values of all other variables will remain unchanged.

Programs and Formulas

The meanings of compound programs and formulas are defined by mutual induction on the structure of α and φ essentially as in the propositional case (see Section 5.2). We include these definitions below for completeness.

Regular Programs and While Programs

Here are the semantic definitions for the four constructs of regular programs.

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{A}}(\alpha\,;\,\beta) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha)\circ\mathfrak{m}_{\mathfrak{A}}(\beta)\\
&= \{(u,v)\mid \exists w\ (u,w)\in\mathfrak{m}_{\mathfrak{A}}(\alpha)\text{ and }(w,v)\in\mathfrak{m}_{\mathfrak{A}}(\beta)\} &(11.3.1)\\
\mathfrak{m}_{\mathfrak{A}}(\alpha\cup\beta) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha)\cup\mathfrak{m}_{\mathfrak{A}}(\beta) &(11.3.2)\\
\mathfrak{m}_{\mathfrak{A}}(\alpha^*) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{A}}(\alpha)^* \;=\; \bigcup_{n\geq 0}\mathfrak{m}_{\mathfrak{A}}(\alpha)^n\\
\mathfrak{m}_{\mathfrak{A}}(\varphi?) &\stackrel{\text{def}}{=} \{(u,u)\mid u\in\mathfrak{m}_{\mathfrak{A}}(\varphi)\}. &(11.3.3)
\end{aligned}$$

The semantics of defined constructs such as **if-then-else** and **while-do** are obtained using their definitions exactly as in PDL.

Seqs and R.E. Programs

Recall that an r.e. program is a Turing machine enumerating a set $CS(\alpha)$ of seqs. If α is an r.e. program, we define

$$\mathfrak{m}_{\mathfrak{A}}(\alpha) \stackrel{\text{def}}{=} \bigcup_{\sigma\in CS(\alpha)}\mathfrak{m}_{\mathfrak{A}}(\sigma).$$

Thus, the meaning of α is defined to be the union of the meanings of the seqs in $CS(\alpha)$. The meaning $\mathfrak{m}_{\mathfrak{A}}(\sigma)$ of a seq σ is determined by the meanings of atomic programs and tests and the sequential composition operator.

There is an interesting point here regarding the translation of programs using other programming constructs into r.e. programs. This can be done for arrays and stacks (for Booleans stacks, even into r.e. programs with bounded memory), but not for wildcard assignment. Since later in the book we shall be referring to the r.e. set of seqs associated with such programs, it is important to be able to carry out this translation. To see how this is done for the case of arrays, for example, consider an algorithm for simulating the execution of a program by generating only ordinary assignments and tests. It does not generate an array assignment of the form $F(t_1,\ldots,t_n) := t$, but rather "remembers" it and when it reaches an assignment of the form $x := F(t_1,\ldots,t_n)$ it will aim at generating $x := t$ instead. This requires care, since we must keep track of changes in the variables inside t and $t_1,\ldots,t_n$ and incorporate them into the generated assignments. We leave the details to the reader (Exercises 11.5–11.7).

Formulas

Here are the semantic definitions for the constructs of formulas of DL. The reader is referred to Section 3.4 for the semantic definitions of atomic first-order formulas.

$$\mathfrak{m}_{\mathfrak{A}}(\mathbf{0}) \stackrel{\text{def}}{=} \varnothing \tag{11.3.4}$$

$$\mathfrak{m}_{\mathfrak{A}}(\varphi \to \psi) \stackrel{\text{def}}{=} \{u \mid \text{if } u \in \mathfrak{m}_{\mathfrak{A}}(\varphi) \text{ then } u \in \mathfrak{m}_{\mathfrak{A}}(\psi)\} \tag{11.3.5}$$

$$\mathfrak{m}_{\mathfrak{A}}(\forall x\ \varphi) \stackrel{\text{def}}{=} \{u \mid \forall a \in A\ u[x/a] \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\} \tag{11.3.6}$$

$$\mathfrak{m}_{\mathfrak{A}}([\alpha]\varphi) \stackrel{\text{def}}{=} \{u \mid \forall v \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha) \text{ then } v \in \mathfrak{m}_{\mathfrak{A}}(\varphi)\}. \tag{11.3.7}$$

Equivalently, we could define the first-order quantifiers $\forall$ and $\exists$ in terms of the wildcard assignment:

$$\forall x\ \varphi \quad \leftrightarrow \quad [x := ?]\varphi \tag{11.3.8}$$

$$\exists x\ \varphi \quad \leftrightarrow \quad \langle x := ?\rangle\varphi. \tag{11.3.9}$$

Note that for *deterministic* programs α (for example, those obtained by using the **while** programming language instead of regular programs and disallowing wildcard assignments), $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ is a partial function from states to states; that is, for every state u, there is at most one v such that $(u,v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha)$. The partiality of the function arises from the possibility that α may not halt when started in certain states. For example, $\mathfrak{m}_{\mathfrak{A}}$(**while 1 do skip**) is the empty relation. In general, the relation $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ need not be single-valued.

If K is a given set of syntactic constructs, we refer to the version of Dynamic Logic with programs built from these constructs as *Dynamic Logic with* K or simply as DL(K). Thus, we have DL(r.e.), DL(array), DL(stk), DL(bstk), DL(wild), and so on. As a default, these logics are the poor-test versions, in which only quantifier-free first-order formulas may appear as tests. The unadorned DL is used to abbreviate DL(reg), and we use DL(dreg) to denote DL with **while** programs, which are really deterministic regular programs. Again, **while** programs use only poor tests. Combinations such as DL(dreg+wild) are also allowed.

11.4 Satisfiability and Validity

The concepts of satisfiability, validity, etc. are defined as for PDL in Chapter 5 or as for first-order logic in Section 3.4.

Let $\mathfrak{A} = (A, \mathfrak{m}_{\mathfrak{A}})$ be a structure, and let u be a state in $S^{\mathfrak{A}}$. For a formula φ, we write $\mathfrak{A}, u \vDash \varphi$ if $u \in \mathfrak{m}_{\mathfrak{A}}(\varphi)$ and say that u *satisfies* φ *in* $\mathfrak{A}$. We sometimes write $u \vDash \varphi$ when $\mathfrak{A}$ is understood. We say that φ is $\mathfrak{A}$-*valid* and write $\mathfrak{A} \vDash \varphi$ if $\mathfrak{A}, u \vDash \varphi$

for all u in $\mathfrak{A}$. We say that φ is *valid* and write $\vDash \varphi$ if $\mathfrak{A} \vDash \varphi$ for all $\mathfrak{A}$. We say that φ is *satisfiable* if $\mathfrak{A}, u \vDash \varphi$ for some $\mathfrak{A}, u$.

For a set of formulas Δ, we write $\mathfrak{A} \vDash \Delta$ if $\mathfrak{A} \vDash \varphi$ for all $\varphi \in \Delta$.

Informally, $\mathfrak{A}, u \vDash [\alpha]\varphi$ iff every terminating computation of α starting in state u terminates in a state satisfying φ, and $\mathfrak{A}, u \vDash \texttt{<}\alpha\texttt{>}\varphi$ iff there exists a computation of α starting in state u and terminating in a state satisfying φ. For a pure first-order formula φ, the metastatement $\mathfrak{A}, u \vDash \varphi$ has the same meaning as in first-order logic (Section 3.4).

11.5 Bibliographical Notes

First-order DL was defined in Harel et al. (1977), where it was also first named Dynamic Logic. That paper was carried out as a direct continuation of the original work of Pratt (1976).

Many variants of DL were defined in Harel (1979). In particular, DL(stk) is very close to the context-free Dynamic Logic investigated there.

Exercises

11.1. Show that in the presence of the Boolean stack operations **push-1**, **push-0**, **pop**, and **top**?, there is no need for a Boolean stack operation that tests whether the top element is **0**.

11.2. Show how to write the recursive procedure appearing in Section 9.1 using a Boolean stack.

11.3. Show that a test for stack emptiness is redundant in DL with an algebraic stack.

11.4. Prove that the meaning of a regular program is the same as the meaning of the corresponding (regular) r.e. program.

11.5. Show how to translate any regular program with array assignments into an r.e. set of seqs with simple assignments only.

11.6. Show how to translate any regular program with an algebraic stack into an r.e. set of seqs with simple assignments only.

11.7. Show how to translate any regular program with a Boolean stack into a bounded-memory r.e. set of seqs with simple assignments only.

11.8. Define DL with integer counters. Show how to translate this logic into bounded-memory DL(r.e.).

11.9. Prove the equivalences (11.3.8) and (11.3.9) for relating wildcard assignment to quantification.

12 Relationships with Static Logics

Reasoning in first-order Dynamic Logic can take two forms: *uninterpreted* and *interpreted*. The former involves properties expressible in the logic that are independent of the domain of interpretation. The latter involves the use of the logic to reason about computation over a particular domain or a limited class of domains. In this chapter we discuss these two levels of reasoning and the relationships they engender between DL and classical static logics.

12.1 The Uninterpreted Level

Uninterpreted Reasoning: Schematology

In contrast to the propositional version PDL discussed in Part II, DL formulas involve variables, functions, predicates, and quantifiers, a state is a mapping from variables to values in some domain, and atomic programs are assignment statements. To give semantic meaning to these constructs requires a first-order structure $\mathfrak{A}$ over which to interpret the function and predicate symbols. Nevertheless, we are not obliged to assume anything special about $\mathfrak{A}$ or the nature of the interpretations of the function and predicate symbols, except as dictated by first-order semantics. Any conclusions we draw from this level of reasoning will be valid under all possible interpretations. *Uninterpreted reasoning* refers to this style of reasoning.

For example, the formula

$$p(f(x), g(y, f(x))) \quad \rightarrow \quad \texttt{<}z := f(x)\texttt{>}p(z, g(y, z))$$

is true over any domain, irrespective of the interpretations of p, f, and g.

Another example of a valid formula is

$$\begin{aligned}&z = y \ \wedge\ \forall x\ f(g(x)) = x\\ &\quad\rightarrow\quad \texttt{[}\mathbf{while}\ p(y)\ \mathbf{do}\ y := g(y)\texttt{]}\texttt{<}\mathbf{while}\ y \neq z\ \mathbf{do}\ y := f(y)\texttt{>}\mathbf{1}.\end{aligned}$$

Note the use of `[ ]` applied to `< >`. This formula asserts that under the assumption that f "undoes" g, any computation consisting of applying g some number of times to z can be backtracked to the original z by applying f some number of times to the result.

This level of reasoning is the most appropriate for comparing features of programming languages, since we wish such comparisons not to be influenced by the coding capabilities of a particular domain of interpretation. For example, if we aban-

don the uninterpreted level and assume the fixed domain $\mathbb{N}$ of the natural numbers with zero, addition and multiplication, all reasonable programming languages are equivalent in computation power—they all compute exactly the partial recursive functions. In contrast, on the uninterpreted level, it can be shown that recursion is a strictly more powerful programming construct than iteration. Research comparing the expressive power of programming languages on the uninterpreted level is sometimes called *schematology*, and uninterpreted programs are often called *program schemes*.

As an example, let us consider regular programs and nondeterministic **while** programs. The former are as powerful as the latter, since every **while** program is obviously regular, as can be seen by recalling the definitions from Section 11.1:

$$\begin{aligned} \textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta &\stackrel{\text{def}}{=} (\varphi?;\alpha) \cup (\neg\varphi?;\beta) \\ \textbf{while } \varphi \textbf{ do } \alpha &\stackrel{\text{def}}{=} (\varphi?;\alpha)^*;\neg\varphi?. \end{aligned}$$

Conversely, over any structure, nondeterministic **while** programs are as powerful as regular programs (Exercise 12.2). We define our logics using the regular operators since they are simpler to manipulate in mathematical arguments, but the **while** program operators are more natural for expressing algorithms.

If we do not allow nondeterminism in **while** programs, the situation is different. We show in Chapter 15 that DL with deterministic **while** programs is strictly less expressive than DL with regular programs when considered over all structures. However, over $\mathbb{N}$ they are equivalent (Theorem 12.6).

Failure of Classical Theorems

We now show that three basic properties of classical (uninterpreted) first-order logic, the *Löwenheim–Skolem theorem*, *completeness*, and *compactness*, fail for even fairly weak versions of DL.

The Löwenheim–Skolem theorem (Theorem 3.59) states that if a formula φ has an infinite model then it has models of all infinite cardinalities. Because of this theorem, classical first-order logic cannot define the structure of elementary arithmetic

$$\mathbb{N} \quad = \quad (\omega, +, \cdot, 0, 1, =)$$

up to isomorphism. That is, there is no first-order sentence that is true in a structure $\mathfrak{A}$ if and only if $\mathfrak{A}$ is isomorphic to $\mathbb{N}$. However, this can be done in DL.

Proposition 12.1: There exists a formula $\Theta_{\mathbb{N}}$ of DL(dreg) that defines $\mathbb{N}$ up to isomorphism.

Proof Take as $\Theta_{\mathbb{N}}$ the conjunction of the following six first-order formulas:

- $\forall x\ x + 1 \neq 0$
- $\forall x\ \forall y\ x + 1 = y + 1 \to x = y$
- $\forall x\ x + 0 = x$
- $\forall x\ \forall y\ x + (y + 1) = (x + y) + 1$
- $\forall x\ x \cdot 0 = 0$
- $\forall x\ \forall y\ x \cdot (y + 1) = (x \cdot y) + x$,

plus the DL(dreg) formula

$$\forall x\ \langle y := 0\,;\ \mathbf{while}\ y \neq x\ \mathbf{do}\ y := y + 1\rangle \mathbf{1}. \tag{12.1.1}$$

The sentence (12.1.1) says that the program inside the diamond halts for all x; in other words, every element of the structure is obtained from 0 by adding 1 a finite number of times. This is inexpressible in first-order logic. A side effect of (12.1.1) is that we may use the induction principle in all models of $\Theta_{\mathbb{N}}$.

The first two of the above first-order formulas imply that every model of $\Theta_{\mathbb{N}}$ is infinite. The remaining first-order formulas are the inductive definitions of addition and multiplication. It follows that every model of $\Theta_{\mathbb{N}}$ is isomorphic to $\mathbb{N}$. ■

The Löwenheim–Skolem theorem does not hold for DL, because $\Theta_{\mathbb{N}}$ has an infinite model (namely $\mathbb{N}$), but all models are isomorphic to $\mathbb{N}$ and are therefore countable.

Besides the Löwenheim–Skolem Theorem, compactness fails in DL as well. Consider the following countable set Γ of formulas:

$$\{\langle \mathbf{while}\ p(x)\ \mathbf{do}\ x := f(x)\rangle \mathbf{1}\} \quad \cup \quad \{p(f^n(x)) \mid n \geq 0\}.$$

It is easy to see that Γ is not satisfiable, but it is finitely satisfiable, i.e. each finite subset of it is satisfiable.

Worst of all, completeness cannot hold for any deductive system as we normally think of it (a finite effective system of axioms schemes and finitary inference rules). The set of theorems of such a system would be r.e., since they could be enumerated by writing down the axioms and systematically applying the rules of inference in all possible ways. However, the set of valid statements of DL is not r.e. (Exercise 12.1). In fact, we will show in Chapter 13 exactly how bad the situation is.

This is not to say that we cannot say anything meaningful about proofs and deduction in DL. On the contrary, there is a wealth of interesting and practical results on axiom systems for DL that we will cover in Chapter 14.

Expressive Power

In this section we investigate the power of DL relative to classical static logics on the uninterpreted level. In particular, we will introduce *rich test* DL *of r.e. programs* and show that it is equivalent to the infinitary language $L_{\omega_1^{\mathrm{ck}}\omega}$. Some consequences of this fact are drawn in later sections.

First we introduce a definition that allows to compare different variants of DL. Let us recall from Section 11.3 that a state is *initial* if it differs from a constant state w_a only at the values of individual variables. If DL_1 and DL_2 are two variants of DL over the same vocabulary, we say that DL_2 is *as expressive as* DL_1 and write $\mathsf{DL}_1 \leq \mathsf{DL}_2$ if for each formula φ in DL_1 there is a formula ψ in DL_2 such that $\mathfrak{A}, u \vDash \varphi \leftrightarrow \psi$ for all structures $\mathfrak{A}$ and initial states u. If DL_2 is as expressive as DL_1 but DL_1 is not as expressive as DL_2, we say that DL_2 is *strictly more expressive than* DL_1, and write $\mathsf{DL}_1 < \mathsf{DL}_2$. If DL_2 is as expressive as DL_1 and DL_1 is as expressive as DL_2, we say that DL_1 and DL_2 are of *equal expressive power*, or are simply *equivalent*, and write $\mathsf{DL}_1 \equiv \mathsf{DL}_2$. We will also use these notions for comparing versions of DL with static logics such as $L_{\omega\omega}$.

There is a technical reason for the restriction to initial states in the above definition. If DL_1 and DL_2 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. We shall see examples of this in Chapter 15.

Also, in the definition of $\mathsf{DL}(K)$ given in Section 11.4, the programming language K is an explicit parameter. Actually, the particular first-order vocabulary Σ over which $\mathsf{DL}(K)$ and K are considered should be treated as a parameter too. It turns out that the relative expressiveness of versions of DL is sensitive not only to K, but also to Σ. This second parameter is often ignored in the literature, creating a source of potential misinterpretation of the results. For now, we assume a fixed first-order vocabulary Σ.

Rich Test Dynamic Logic of R.E. Programs

We are about to introduce the most general version of DL we will ever consider. This logic is called *rich test Dynamic Logic of r.e. programs*, and it will be denoted DL(rich-test r.e.). Programs of DL(rich-test r.e.) are r.e. sets of seqs as defined in Section 11.2, except that the seqs may contain tests $\varphi?$ for any previously constructed formula φ.

The formal definition is inductive. All atomic programs are programs and all atomic formulas are formulas. If φ, ψ are formulas, α, β are programs, $\{\alpha_n \mid n \in \omega\}$

is an r.e. set of programs over a finite set of variables (free or bound), and x is a variable, then

- $\mathbf{0}$
- $\varphi \to \psi$
- $[\alpha]\varphi$
- $\forall x\ \varphi$

are formulas and

- $\alpha\,;\,\beta$
- $\{\alpha_n \mid n \in \omega\}$
- $\varphi?$

are programs. The set $CS(\alpha)$ of computation sequences of a rich test r.e. program α is defined as usual.

Recall from Section 3.6 that $L_{\omega_1\omega}$ is the language with the formation rules of the first-order language $L_{\omega\omega}$, but in which countably infinite conjunctions and disjunctions $\bigwedge_{i\in I}\varphi_i$ and $\bigvee_{i\in I}\varphi_i$ are also allowed. In addition, if $\{\varphi_i \mid i \in I\}$ is recursively enumerable, then the resulting language is denoted $L_{\omega_1^{\mathrm{ck}}\omega}$ and is sometimes called *constructive* $L_{\omega_1\omega}$.

PROPOSITION 12.2: DL(rich-test r.e.) $\equiv L_{\omega_1^{\mathrm{ck}}\omega}$.

Proof In the translations below, φ ranges over $L_{\omega_1^{\mathrm{ck}}\omega}$ formulas, ψ ranges over DL(rich-test r.e.) formulas, and α ranges over rich test r.e. programs. The translation from $L_{\omega_1^{\mathrm{ck}}\omega}$ to DL(rich-test r.e.) is obtained via the mapping μ. The main clause of its definition is given below. Recall that $\neg\varphi$ stands for $\varphi \to \mathbf{0}$ and $<\alpha>\varphi$ stands for $\neg[\alpha]\neg\varphi$.

$$\mu(\bigvee_{i\in I}\varphi_i) \quad \stackrel{\mathrm{def}}{=} \quad <\{\mu(\varphi_i)? \mid i \in I\}>\mathbf{1}.$$

The reverse translation is obtained via a mapping ν with the help of a mapping $(\ )_\alpha$ that tranforms $L_{\omega_1^{\mathrm{ck}}\omega}$ formulas into $L_{\omega_1^{\mathrm{ck}}\omega}$ formulas. Here α is an arbitrary rich test r.e. program. The main clause of the definition of ν is

$$\nu(<\alpha>\psi) \quad \stackrel{\mathrm{def}}{=} \quad \nu(\psi)_\alpha,$$

and the main defining clauses for $(\ )_\alpha$ are as follows:

$$\begin{aligned}
\varphi_{x:=t} &\stackrel{\text{def}}{=} \varphi[x/t] \\
\varphi_{\alpha\,;\,\beta} &\stackrel{\text{def}}{=} (\varphi_\alpha)_\beta \\
\varphi_{\{\alpha_n \mid n\in\omega\}} &\stackrel{\text{def}}{=} \bigvee_{n\in\omega} \varphi_{\alpha_n} \\
\varphi_{\psi?} &\stackrel{\text{def}}{=} \varphi \wedge \nu(\psi).
\end{aligned}$$

■

Since r.e. programs as defined in Section 11.2 are clearly a special case of general rich-test r.e. programs, it follows that DL(rich-test r.e.) is as expressive as DL(r.e.). In fact they are not of the same expressive power.

THEOREM 12.3: DL(r.e.) $<$ DL(rich-test r.e.).

Proof sketch. One can use an Ehrenfeucht–Fraïssé argument to show that DL(r.e.) cannot distinguish between the recursive ordinals ω^ω and $\omega^\omega \cdot 2$, whereas any recursive ordinal can be defined by a formula of DL(rich-test r.e.) up to isomorphism. Details can be found in Meyer and Parikh (1981). ■

Henceforth, we shall assume that the first-order vocabulary Σ contains at least one function symbol of positive arity. Under this assumption, DL can easily be shown to be strictly more expressive than $L_{\omega\omega}$:

THEOREM 12.4: $L_{\omega\omega} <$ DL.

Proof In Section 12.1 we saw how to construct an infinite model for Σ that is uniquely definable in DL up to isomorphism. By the upward Löwenheim–Skolem theorem, this is impossible in $L_{\omega\omega}$. ■

COROLLARY 12.5:

$$L_{\omega\omega} \ < \ \mathsf{DL} \ \leq \ \mathsf{DL}(\text{r.e.}) \ < \ \mathsf{DL}(\text{rich-test r.e.}) \ \equiv \ L_{\omega_1^{\text{ck}}\omega}.$$

The situation with the intermediate versions of DL, e.g. DL(stk), DL(bstk), DL(wild), etc., is of interest. We deal with the relative expressive power of these in Chapter 15, where we also show that the second inequality in Corollary 12.5 is strict.

12.2 The Interpreted Level

Interpreted Reasoning: Arithmetical Structures

This is the most detailed level we will consider. It is the closest to the actual process of reasoning about concrete, fully specified programs. Syntactically, the programs and formulas are as on the uninterpreted level, but here we assume a fixed structure or class of structures.

In this framework, we can study programs whose computational behavior depends on (sometimes deep) properties of the particular structures over which they are interpreted. In fact, almost any task of verifying the correctness of an actual program falls under the heading of interpreted reasoning.

One specific structure we will look at carefully is the natural numbers with the usual arithemetic operations:

$$\mathbb{N} \quad = \quad (\omega,\ 0,\ 1,\ +,\ \cdot,\ =).$$

Let $-$ denote the (first-order-definable) operation of subtraction and let $\mathrm{gcd}(x, y)$ denote the first-order-definable operation giving the greatest common divisor of x and y. The following formula of DL is $\mathbb{N}$-valid, i.e., true in all states of $\mathbb{N}$:

$$x = x' \wedge y = y' \wedge xy \geq 1 \quad \rightarrow \quad <\alpha>(x = \mathrm{gcd}(x', y')) \qquad (12.2.1)$$

where α is the **while** program of Example 4.1 or the regular program

$$(x \neq y?; ((x > y?; x := x - y) \cup (x < y?; y := y - x)))^* x = y?.$$

Formula (12.2.1) states the correctness and termination of an actual program over $\mathbb{N}$ computing the greatest common divisor.

As another example, consider the following formula over $\mathbb{N}$:

$$\forall x \geq 1 <(\textbf{if even}(x) \textbf{ then } x := x/2 \textbf{ else } x := 3x + 1)^*>(x = 1).$$

Here / denotes integer division, and **even**( ) is the relation that tests if its argument is even. Both of these are first-order definable. This innocent-looking formula asserts that starting with an arbitrary positive integer and repeating the following two operations, we will eventually reach 1:

- if the number is even, divide it by 2;
- if the number is odd, triple it and add 1.

The truth of this formula is as yet unknown, and it constitutes a problem in number theory (dubbed "the $3x + 1$ problem") that has been open for over 60 years. The

formula $\forall x \geq 1$ **<**α**>1**, where α is

while $x \neq 1$ **do if even**(x) **then** $x := x/2$ **else** $x := 3x + 1$,

says this in a slightly different way.

The specific structure $\mathbb{N}$ can be generalized, resulting in the class of *arithmetical structures*. We shall not give a full definition here. Briefly, a structure $\mathfrak{A}$ is *arithmetical* if it contains a first-order-definable copy of $\mathbb{N}$ and has first-order definable functions for coding finite sequences of elements of $\mathfrak{A}$ into single elements and for the corresponding decoding.

Arithmetical structures are important because (i) most structures arising naturally in computer science (e.g., discrete structures with recursively defined data types) are arithmetical, and (ii) any structure can be extended to an arithmetical one by adding appropriate encoding and decoding capabilities. While most of the results we present for the interpreted level are given in terms of $\mathbb{N}$ alone, many of them hold for any arithmetical structure, so their significance is greater.

Expressive Power over $\mathbb{N}$

The results of Section 12.1 establishing that

$$L_{\omega\omega} \quad < \quad \mathsf{DL} \quad \leq \quad \mathsf{DL}(\text{r.e.}) \quad < \quad \mathsf{DL}(\text{rich-test r.e.})$$

were on the uninterpreted level, where all structures are taken into account.[1] Thus first-order logic, regular DL, and DL(rich-test r.e.) form a sequence of increasingly more powerful logics when interpreted uniformly over all structures.

What happens if one fixes a structure, say $\mathbb{N}$? Do these differences in expressive power still hold? We now address these questions.

First, we introduce notation for comparing expressive power over $\mathbb{N}$. If DL_1 and DL_2 are variants of DL (or static logics, such as $L_{\omega\omega}$) and are defined over the vocabulary of $\mathbb{N}$, we write $\mathsf{DL}_1 \leq_{\mathbb{N}} \mathsf{DL}_2$ if for each $\varphi \in \mathsf{DL}_1$ there is $\psi \in \mathsf{DL}_2$ such that $\mathbb{N} \vDash \varphi \leftrightarrow \psi$. We define $<_{\mathbb{N}}$ and $\equiv_{\mathbb{N}}$ from $\leq_{\mathbb{N}}$ the same way $<$ and $\equiv$ were defined from $\leq$ in Section 12.1.

We now show that over $\mathbb{N}$, DL is no more expressive than first-order logic $L_{\omega\omega}$. This is true even for finite-test DL. The result is stated for $\mathbb{N}$, but is actually true for any arithmetical structure.

THEOREM 12.6: $\quad L_{\omega\omega} \equiv_{\mathbb{N}} \mathsf{DL} \equiv_{\mathbb{N}} \mathsf{DL}(\text{r.e.})$.

1 As mentioned, the second inequality is also strict.

Proof The direction $\leq$ of both equivalences is trivial. For the other direction, we sketch the construction of a first-order formula φ_L for each $\varphi \in \mathsf{DL}$(r.e.) such that $\mathbb{N} \models \varphi \leftrightarrow \varphi_L$.

The construction of φ_L is carried out by induction on the structure of φ. The only nontrivial case is for φ of the form $[\alpha]\psi$. For a formula of this form, suppose ψ_L has been constructed. Let $FV(\alpha) \subseteq \{x_1, \ldots, x_k\}$ for some $k \geq 0$. Consider the set of seqs σ over the vocabulary of arithmetic such that $FV(\sigma) \subseteq \{x_1, \ldots, x_k\}$. Every such σ is a finite expression, therefore can be encoded as a natural number $\ulcorner\sigma\urcorner$. Now consider the set

$$R \stackrel{\text{def}}{=} \{(\ulcorner\sigma\urcorner, n_1, \ldots, n_k, m_1, \ldots, m_k) \in \mathbb{N}^{2k+1} \mid (\overline{n}, \overline{m}) \in \mathfrak{m}_{\mathbb{N}}(\sigma)\},$$

where $\overline{n}$ is the state that assigns n_i to x_i for $1 \leq i \leq k$ and 0 to the remaining variables. The state $\overline{m}$ is defined similarly. Clearly R is a recursive set and there is first-order formula $\gamma(y, x_1, \ldots, x_k, z_1, \ldots, z_k)$ that defines R in $\mathbb{N}$. We can assume that the variables $y, z_1, \ldots, z_k$ do not occur in ψ_L. Let $\varphi_\alpha(y)$ be a formula defining the set $\{\ulcorner\sigma\urcorner \mid \sigma \in CS(\alpha)\}$. The desired formula φ_L is

$$\forall y\ \forall z_1\ \ldots\ \forall z_k\ (\varphi_\alpha(y) \wedge \gamma(y, x_1, \ldots, x_k, z_1, \ldots, z_k) \quad \rightarrow \quad \psi_L[x_1/z_1, \ldots, x_k/z_k]).$$

The remaining cases we leave as an exercise (Exercise 12.5). ■

The significance of this result is that in principle, one can carry out all reasoning about programs interpreted over $\mathbb{N}$ in the first-order logic $L_{\omega\omega}$ by translating each DL formula into a first-order equivalent. The translation is effective, as this proof shows. Moreover, Theorem 12.6 holds for any arithmetical structure containing the requisite coding power. As mentioned earlier, every structure can be extended to an arithmetical one.

However, the translation of Theorem 12.6 produces unwieldly formulas having little resemblance to the original ones. This mechanism is thus somewhat unnatural and does not correspond closely to the type of arguments one would find in practical program verication. In Section 14.2, a remedy is provided that makes the process more orderly.

We now show that over $\mathbb{N}$, DL(rich-test r.e.) has considerably more power than the equivalent logics of Theorem 12.6. This too is true for any arithmetical structure.

THEOREM 12.7: Over $\mathbb{N}$, DL(rich-test r.e.) defines precisely the Δ_1^1 (hyperarithmetic) sets.

Proof We will show in Theorem 13.6 that the set

$$\{\psi \in \mathsf{DL}(\text{rich-test r.e.}) \mid \mathbb{N} \vDash \psi\} \tag{12.2.2}$$

is hyperarithmetic. Any DL(rich-test r.e.)-definable set

$$\{(a_1, \ldots, a_n) \mid \mathbb{N} \vDash \varphi[x_1/a_1, \ldots, x_n/a_n]\} \tag{12.2.3}$$

defined by a DL(rich-test r.e.) formula φ with free variables $x_1, \ldots, x_n$ reduces by simple substitution[2] to (12.2.2). The set (12.2.3) is therefore hyperarithmetic.

For the other direction, we use the characterization of Δ_1^1 as the subsets of $\mathbb{N}$ defined by total IND programs; equivalently, by IND programs that always halt within "time" bounded by a recursive ordinal. This generalized notion of time is defined formally by the ordinal mapping $\mathrm{ord} : T \to \mathbf{Ord}$ on recursive well-founded trees as discussed in Section 2.2. The time of a halting IND computation is the ordinal associated with the root of the computation tree.

Given an IND program π over $\mathbb{N}$ with program variables $x_1, \ldots, x_n$ and a recursive ordinal represented by a well-founded recursive tree $T \subseteq \omega^*$ as described in Section 2.2, we define a family of DL(rich-test r.e.) formulas φ_ℓ^w with free variables $x_1, \ldots, x_n$, where $w \in T$ and ℓ is a statement label of π. The formula $\varphi_\ell^w[x_1/a_1, \ldots, x_n/a_n]$ says that π halts and accepts in at most $\mathrm{ord}(w)$ steps when started at statement ℓ in a state in which x_i has value a_i, $1 \le i \le n$.

The definition of φ_ℓ^w is inductive on the well-founded tree T. In the following definition, $c(\ell)$ refers to the continuation of statement ℓ in π; that is, the first statement of π if ℓ is the last statement, otherwise the statement immediately following ℓ.

The formulas φ_ℓ^w are defined as follows. If ℓ is the statement $x_i := \exists$, define

$$\varphi_\ell^w \stackrel{\text{def}}{=} \mathtt{<}\{x_i := m \mid m \in \omega\}\mathtt{><}\{\varphi_{c(\ell)}^{wn}? \mid n \in \omega,\ wn \in T\}\mathtt{>}\mathbf{1}.$$

If ℓ is the statement $x_i := \forall$, define

$$\varphi_\ell^w \stackrel{\text{def}}{=} \mathtt{[}\{x_i := m \mid m \in \omega\}\mathtt{]<}\{\varphi_{c(\ell)}^{wn}? \mid n \in \omega,\ wn \in T\}\mathtt{>}\mathbf{1}.$$

If ℓ is either **accept** or **reject**, define φ_ℓ^w to be $\mathbf{1}$ or $\mathbf{0}$, respectively. Finally, if ℓ is the statement **if** r **then go to** ℓ', define

$$\varphi_\ell^w \stackrel{\text{def}}{=} \mathtt{<}\mathbf{if}\ r\ \mathbf{then}\ \{\varphi_{\ell'}^{wn}? \mid n \in \omega,\ wn \in T\}\ \mathbf{else}\ \{\varphi_{c(\ell)}^{wn}? \mid n \in \omega,\ wn \in T\}\mathtt{>}\mathbf{1}.$$

The top-level statement asserting that π halts and accepts in ordinal time bounded by $\mathrm{ord}(T)$ is $\varphi_{\ell_0}^{\varepsilon}$, where ℓ_0 is the first statement of π and ε is the null string. ■

---

2 We assume the coding scheme for DL(rich-test r.e.) formulas has been designed to permit effective identification of and substitution for free variables.

Theorem 12.6 says that over $\mathbb{N}$, the languages DL and DL(r.e.) each define precisely the arithmetic (first-order definable) sets, and Theorem 12.7 says that DL(rich-test r.e.) defines precisely the hyperarithmetic or Δ_1^1 sets. Since the inclusion between these classes is strict—for example, first-order number theory is hyperarithmetic but not arithmetic—we have

Corollary 12.8: DL(r.e.) $<_{\mathbb{N}}$ DL(rich-test r.e.).

12.3 Bibliographical Notes

Uninterpreted reasoning in the form of program schematology has been a common activity ever since the work of Ianov (1960). It was given considerable impetus by the work of Luckham et al. (1970) and Paterson and Hewitt (1970); see also Greibach (1975). The study of the correctness of interpreted programs goes back to the work of Turing and von Neumann, but seems to have become a well-defined area of research following Floyd (1967), Hoare (1969) and Manna (1974).

Embedding logics of programs in $L_{\omega_1\omega}$ is based on observations of Engeler (1967). Theorem 12.3 is from Meyer and Parikh (1981). Theorem 12.6 is from Harel (1979) (see also Harel (1984) and Harel and Kozen (1984)); it is similar to the expressiveness result of Cook (1978). Theorem 12.7 and Corollary 12.8 are from Harel and Kozen (1984).

Arithmetical structures were first defined by Moschovakis (1974) under the name *acceptable structures*. In the context of logics of programs, they were reintroduced and studied in Harel (1979).

Exercises

12.1. Consider DL with deterministic **while** programs over the first-order vocabulary of $\mathbb{N}$. Show that the set of valid DL formulas over this vocabulary is not recursively enumerable. (*Hint*. Using the formula $\Theta_{\mathbb{N}}$ defined in Section 12.1 that defines the natural numbers up to isomorphism, show that if the set of valid DL formulas were r.e., then so would be the set of formulas true in $\mathbb{N}$, thus contradicting Gödel's incompleteness theorem.)

12.2. Show that nondeterministic **while** programs and regular programs are equivalent over any structure.

12.3. Show that in the uninterpreted sense, allowing only atomic formulas instead of all quantifier-free formulas as tests does not diminish the expressive power of DL.

12.4. Argue by induction on the well-founded recursive tree T that the construction of the DL(rich-test r.e.) formulas φ_ℓ^w in the proof of Theorem 12.7 is correct.

12.5. Fill in the missing cases in the proof of Theorem 12.6.

12.6. Give a precise definition of an arithmetical structure. Let $L_1 \leq_A L_2$ denote relative expressibility in arithmetical structures; that is, $L_1 \leq_A L_2$ holds if for any arithmetical structure $\mathfrak{A}$ and any formula φ in L_1, there is a formula ψ in L_2 such that $\mathfrak{A} \vDash \varphi \leftrightarrow \psi$. Define $L_1 \equiv_A L_2$ accordingly. Show that Theorem 12.6 holds for arithmetical structures; that is,

$$L_{\omega\omega} \quad \equiv_A \quad \mathsf{DL} \quad \equiv_A \quad \mathsf{DL}(\text{r.e.}).$$

13 Complexity

This chapter addresses the complexity of first-order Dynamic Logic.

Section 13.1 discusses the difficulty of establishing validity in DL. As in Chapter 12, we divide the question into uninterpreted and interpreted versions. On the uninterpreted level, we deal with the complexity of deciding validity of a given formula of an arbitrary signature over all interpretations for that signature. On the interpreted level, we are interested in the truth in $\mathbb{N}$ of a number-theoretic DL formula or validity over arithmetical structures.

In Section 13.2 we turn our attention to some of the programming languages defined in Chapter 11 and analyze their *spectral complexity*, a notion that measures the difficulty of the halting problem over finite structures. Spectral complexity will become useful in comparing the expressive power of variants of DL in Chapter 15.

13.1 The Validity Problem

Since all versions of DL subsume first-order logic, truth can be no easier to establish than in $L_{\omega\omega}$. Also, since DL(r.e.) is subsumed by $L_{\omega_1^{\mathrm{ck}}\omega}$, truth will be no harder to establish than in $L_{\omega_1^{\mathrm{ck}}\omega}$. These bounds hold for both uninterpreted and interpreted levels of reasoning.

The Uninterpreted Level: Validity

In this section we discuss the complexity of the validity problem for DL. By the remarks above and Theorems 3.60 and 3.67, this problem is between Σ_1^0 and Π_1^1. That is, as a lower bound it is undecidable and can be no better than recursively enumerable, and as an upper bound it is in Π_1^1. This is a rather large gap, so we are still interested in determining more precise complexity bounds for DL and its variants. An interesting related question is whether there is some nontrivial[1] fragment of DL that is in Σ_1^0, since this would allow a complete axiomatization.

In the following, we consider these questions for full DL(reg), but we also consider two important subclasses of formulas for which better upper bounds are derivable:

- partial correctness assertions of the form $\psi \to [\alpha]\varphi$, and
- termination or total correctness assertions of the form $\psi \to \langle\alpha\rangle\varphi$,

1 *Nontrivial* here means containing $L_{\omega\omega}$ and allowing programs with iteration. The reason for this requirement is that loop-free programs add no expressive power over first-order logic.

where φ and ψ are first-order formulas. The results are stated for regular programs, but they remain true for the more powerful programming languages too. They also hold for deterministic **while** programs (Exercises 13.3 and 13.4).

We state the results without mentioning the underlying first-order vocabulary Σ. For the upper bounds this is irrelevant. For the lower bounds, we assume the Σ contains a unary function symbol and ternary predicate symbols to accommodate the proofs.

THEOREM 13.1: The validity problem for DL is Π^1_1-hard, even for formulas of the form $\exists x\ [\alpha]\varphi$, where α is a regular program and φ is first-order.

Proof For convenience, we phrase the proof in terms of satisfiablity instead of validity, carrying out a reduction from the Σ^1_1-complete tiling problem of Proposition 2.22: Given a finite set T of tile types, can the infinite $\omega \times \omega$ grid with blue south and west boundaries be tiled so that the color red occurs infinitely often?

We will adapt the encoding of Theorem 3.67 to our needs. Let us recall that the vocabulary contains one constant symbol a, one unary function symbol f, and four ternary relation symbols SOUTH, NORTH, WEST and EAST.

As in the proof of Theorem 3.67, define the formula

$$\begin{aligned}\mathrm{RED}(x,y) \quad \stackrel{\mathrm{def}}{\iff} \quad & \mathrm{NORTH}(x,y,f^{\mathrm{red}}(a)) \vee \mathrm{SOUTH}(x,y,f^{\mathrm{red}}(a)) \\ & \vee \mathrm{EAST}(x,y,f^{\mathrm{red}}(a)) \vee \mathrm{WEST}(x,y,f^{\mathrm{red}}(a)),\end{aligned}$$

which says intuitively that the tile at position x, y has a red side. Let ψ_T be the conjunction of the five formulas (3.4.3)–(3.4.7) used in the proof of Theorem 3.60 and the formula

$$\forall x <y := x; z := x; (y := f(y))^*; (z := f(z))^*> \mathrm{RED}(y,z). \tag{13.1.1}$$

The claim is that ψ_T is satisfiable iff T can tile the $\omega \times \omega$ grid so that the color red occurs infinitely often. The explanations of the encoding in the proof of Theorem 3.60 apply here. Clause (13.1.1) forces the red to appear infinitely often in the tiling. It asserts that no matter how far we go, we always find at least one point with a tile containing red.

As for the required form of the DL formulas, note that the first five clauses can be "pushed under" the diamond and attached as conjuncts to RED(x, y). Negating the resulting formula in order to accommodate the phrasing of the theorem in terms of validity yields the desired result. ■

The following is an immediate corollary of Theorem 13.1:

THEOREM 13.2: The validity problem for DL and DL(rich-test r.e.), as well as all intermediate versions, is Π^1_1-complete.

To soften the negative flavor of these results, we now show that the special cases of unquantified one-program DL(r.e.) formulas have easier validity problems (though, as mentioned, they are still undecidable). We first need a lemma.

LEMMA 13.3: For every r.e. program α and for every first-order formula φ, there exists an r.e. set $\{\varphi_\sigma \mid \sigma \in CS(\alpha)\}$ of first-order formulas such that

$$\models \; [\alpha]\varphi \;\; \leftrightarrow \;\; \bigwedge_{\sigma\in CS(\alpha)} \varphi_\sigma .$$

Proof For every seq σ, we define a mapping $(\;)_\sigma$ that transforms first-order formulas into first-order formulas as follows:[2]

$$\begin{aligned} \varphi_\varepsilon &\stackrel{\text{def}}{=} \varphi, \quad \text{where } \varepsilon \text{ is the null seq;} \\ \varphi_{x:=t\,;\,\sigma} &\stackrel{\text{def}}{=} \varphi_\sigma[x/t]; \\ \varphi_{\psi?\,;\,\sigma} &\stackrel{\text{def}}{=} \psi \rightarrow \varphi_\sigma . \end{aligned}$$

Verification of the conclusion of the lemma is left to the reader. ■

THEOREM 13.4: The validity problem for the sublanguage of DL(r.e.) consisting of formulas of the form $<\alpha>\varphi$, where φ is first-order and α is an r.e. program, is Σ^0_1-complete.

Proof It suffices to show that the problem is in Σ^0_1, since the sublanguage $L_{\omega\omega}$ is already Σ^0_1-complete. By Lemma 13.3, $<\alpha>\varphi$ is equivalent to $\bigvee_{\sigma\in CS(\alpha)} \varphi_\sigma$, and all the φ_σ are first-order. By the compactness of first-order logic, there is some finite subset $\Gamma \subseteq \{\varphi_\sigma \mid \sigma \in CS(\alpha)\}$ such that $\models <\alpha>\varphi$ iff $\models \bigvee \Gamma$. Each such finite disjunction is a first-order formula, hence the finite subsets Γ can be generated and checked for validity in a recursively enumerable manner. ■

It is easy to see that the result holds for formulas of the form $\psi \rightarrow <\alpha>\varphi$, where ψ is also first-order (Exercise 13.1). Thus, termination assertions for nondeterministic programs with first-order tests (or total correctness assertions for deterministic programs), on the uninterpreted level of reasoning, are recursively enumerable and

2 The reader may wish to compare this mapping with the mapping defined in the proof of Proposition 12.2.

therefore axiomatizable. We shall give an explicit axiomatization in Chapter 14.

We now turn to partial correctness.

Theorem 13.5: The validity problem for the sublanguage of DL(r.e.) consisting of formulas of the form $[\alpha]\varphi$, where φ is first-order and α is an r.e. program, is Π^0_2-complete. The Π^0_2-completeness property holds even if we restrict α to range over deterministic **while** programs.

Proof For the upper bound, we have by Lemma 13.3 that $\models [\alpha]\varphi$ iff $\models \bigwedge_{\sigma\in CS(\alpha)} \varphi_\sigma$. It follows that the validity of the latter is co-r.e. in the r.e. problem of validity of first-order formulas, hence it is in Π^0_2.

For the lower bound, we carry out a reduction (to the dual satisfiability problem) from the Σ^0_2-complete tiling problem of Proposition 2.21. Let us recall that this problem calls for a finite set T of tile types to tile the positive quadrant of the integer grid in such a way that the colors on the south boundary form a finite sequence of colors followed by an infinite sequence of blue.

For our encoding, we again adapt the proof of Theorem 3.60. We use the notation from that proof. We take ψ'_T to be the conjunction of the clauses (3.4.3), (3.4.6), and (3.4.7) used in the proof of Theorem 3.60 together with the clause

$$\forall x\ \textsc{South}(x, a, f^{\text{blue}}(a)) \quad\rightarrow\quad \textsc{South}(f(x), a, f^{\text{blue}}(a)).$$

This clause expresses the property that the color blue, when occurring on the south boundary, remains there from the first occurrence on. Now we can combine ψ'_T with the requirement that blue actually occurs on the south boundary to obtain the formula

$$\psi_T \quad\stackrel{\text{def}}{=}\quad \langle x := a; \textbf{while}\ \neg\textsc{South}(x, a, f^{\text{blue}}(a))\ \textbf{do}\ x := f(x)\rangle\psi'_T.$$

The claim is that ψ_T is satisfiable iff T can tile the grid with the additional constraint on the colors of south boundary. We leave the verification of this claim to the reader. ■

Theorem 13.5 extends easily to partial correctness assertions; that is, to formulas of the form $\psi \rightarrow [\alpha]\varphi$, where ψ is also first-order (Exercise 13.2). Thus, while Π^0_2 is obviously better than Π^1_1, it is noteworthy that on the uninterpreted level of reasoning, the truth of even simple correctness assertions for simple programs is not r.e., so that no finitary complete axiomatization for such validities can be given.

The Interpreted Level: Validity over $\mathbb{N}$

The characterizations of the various versions of DL in terms of classical static logics established in Section 12.2 provide us with the precise complexity of the validity problem over $\mathbb{N}$.

THEOREM 13.6: The $\mathbb{N}$-validity problem for DL(dreg) and DL(rich-test r.e.), as well as all intermediate versions, when defined over the vocabulary of $\mathbb{N}$, is hyperarithmetic (Δ_1^1) but not arithmetic.

Proof Let

$$X \stackrel{\text{def}}{=} \{\varphi \in \mathsf{DL}(\text{rich-test r.e.}) \mid \mathbb{N} \models \varphi\}.$$

Let $\Theta_{\mathbb{N}}$ be the DL(dreg) formula that defines $\mathbb{N}$ up to isomorphism (see Proposition 12.1). Since for every $\varphi \in \mathsf{DL}$(rich-test r.e.) we have

$$\varphi \in X \iff \models \Theta_{\mathbb{N}} \rightarrow \varphi,$$

by Theorem 13.2 we have that X is in Π_1^1. On the other hand, since for every sentence φ we have $\varphi \notin X$ iff $\neg\varphi \in X$, it follows that X is also in Σ_1^1, hence it is in Δ_1^1.

That $\mathbb{N}$-validity for any of the intermediate versions is not arithmetic follows from the fact that the first-order theory of $\mathbb{N}$ is already not arithmetic. ■

13.2 Spectral Complexity

We now introduce the *spectral complexity* of a programming language. As mentioned, this notion provides a measure of the complexity of the halting problem for programs over finite interpretations.

Recall that a *state* is a finite variant of a constant valuation w_a for some $a \in A$ (see Section 11.3), and a state w is *initial* if it differs from w_a for individual variables only. Thus, an initial state can be uniquely defined by specifying its relevant portion of values on individual variables. For $m \in \mathbb{N}$, we call an initial state w an *m-state* if for some $a \in A$ and for all $i \geq m$, $w(x_i) = a$. An m-state can be specified by an $(m+1)$-tuple of values $(a_0, \ldots, a_m)$ that represent values of w for the first $m+1$ individual variables $x_0, \ldots, x_m$. Call an m-state $w = (a_0, \ldots, a_m)$ *Herbrand-like* if the set $\{a_0, \ldots, a_m\}$ generates A; that is, if every element of A can be obtained as a value of a term in the state w.

Coding Finite Structures

Let Σ be a finite first-order vocabulary, and assume that the symbols of Σ are linearly ordered as follows. Function symbols are smaller in the order than predicate symbols. Function symbols are ordered according to arity; that is, symbols of smaller arity are smaller than symbols of larger arity. Function symbols of the same arity are ordered in an arbitrary but fixed way. Predicate symbols are ordered similarly.

Let $\mathfrak{A}$ be a structure for Σ. We define a *natural chain* in $\mathfrak{A}$ as a particular way of linearly ordering all elements in the substructure of $\mathfrak{A}$ generated by the empty set. A natural chain is a partial function $C_{\mathfrak{A}} : \mathbb{N} \to A$ defined for $k \in \mathbb{N}$ as follows:

$$C_{\mathfrak{A}}(k) \stackrel{\text{def}}{=} \begin{cases} f_i^{\mathfrak{A}}(C_{\mathfrak{A}}(i_1), \ldots, C_{\mathfrak{A}}(i_n)), & \text{if } (i, i_1, \ldots, i_n) \text{ is the first vector in lexicographic order such that } f_i \text{ is an } n\text{-ary function symbol in } \Sigma,\ i_1, \ldots, i_n < k, \text{ and } f_i^{\mathfrak{A}}(C_{\mathfrak{A}}(i_1), \ldots, C_{\mathfrak{A}}(i_n)) \notin \{C_{\mathfrak{A}}(j) \mid j < k\}; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Observe that if Σ has no constant symbols, then $C_{\mathfrak{A}} = \varnothing$. From now on, we assume that Σ has at least one constant symbol.

Let Σ be a first-order vocabulary and let $c_0, \ldots, c_m$ be symbols not occurring in Σ. An *expanded vocabulary* $\Sigma \cup \{c_0, \ldots, c_m\}$ is obtained from Σ by adding $c_0, \ldots, c_m$ as constant symbols. If the symbols of Σ were linearly ordered in some way, then assuming a linear order on the new constants, the symbols of $\Sigma \cup \{c_0, \ldots, c_m\}$ are ordered as in Σ, except that the new constants come just after the old constants and before the function symbols of Σ.

Let $\Sigma' = \Sigma \cup \{c_0, \ldots, c_m\}$. For every Σ-structure $\mathfrak{A}$ and for every m-state $w = (a_0, \ldots, a_m)$ in $\mathfrak{A}$, we expand $\mathfrak{A}$ into a Σ'-structure $\mathfrak{A}_w$ by interpreting each c_i by a_i and leaving the interpretation of the old symbols unchanged.

The next result shows that the natural chain in $\mathfrak{A}_w$ can be uniformly computed by a deterministic program with an algebraic stack.

PROPOSITION 13.7: For every $m > 0$, there exists a deterministic program $\textsc{Next}_m$ with an algebraic stack such that for every Σ-structure $\mathfrak{A}$, m-state w in $\mathfrak{A}$, and $b \in A$,

$$\mathfrak{A}, w[x_{m+1}/b] \vDash \texttt{<}\textsc{Next}_m\texttt{>}\mathbf{1} \iff b \in C_{\mathfrak{A}_w}(\mathbb{N}).$$

Moreover, if $b = C_{\mathfrak{A}_w}(k)$ for some k, then $\textsc{Next}_m$ terminates for the input $w[x_{m+1}/b]$ in some state in which x_{m+1} has value $C_{\mathfrak{A}_w}(k+1)$ if $C_{\mathfrak{A}_w}(k+1)$ is defined, b if not.

Proof Following the recursive definition of $C_{\mathfrak{A}}$, it is easy to write a recursive procedure that computes the successor of $b \in A$ with respect to the natural chain in $\mathfrak{A}_w$. This procedure is further translated into the desired deterministic program with an algebraic stack (see Section 11.2). ■

It follows that for every structure $\mathfrak{A}$ and input w that is an m-state, there is a canonical way of computing a successor function on the elements generated by the input.

PROPOSITION 13.8: Let $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be Σ-structures on the same carrier generated by the empty set (that is, every element is named by a ground term), and assume that $C_{\mathfrak{A}_1} = C_{\mathfrak{A}_2}$. Then $\mathfrak{A}_1$ and $\mathfrak{A}_2$ are isomorphic iff $\mathfrak{A}_1 = \mathfrak{A}_2$.

Proof Let $f : \mathfrak{A}_1 \to \mathfrak{A}_2$ be an isomorphism. One proves by a straightforward induction on k in the domain of $C_{\mathfrak{A}_1}$ that $f(C_{\mathfrak{A}_1}(k)) = C_{\mathfrak{A}_2}(k)$. Thus f is the identity and $\mathfrak{A}_1 = \mathfrak{A}_2$. ■

Let Σ be a first-order vocabulary. Recall that we assume that Σ contains at least one function symbol of positive arity. In this section we actually assume that Σ is *rich*; that is, either it contains at least one predicate symbol[3] or the sum of arities of the function symbols is at least two. Examples of rich vocabularies are: two unary function symbols, or one binary function symbol, or one unary function symbol and one unary predicate symbol. A vocabulary that is not rich will be called *poor*. Hence a poor vocabulary has just one unary function symbol and possibly some constants, but no relation symbols other than equality. The main difference between rich and poor vocabularies is that the former admit exponentially many pairwise non-isomorphic structures of a given finite cardinality, whereas the latter admit only polynomially many. In this section we will cover rich vocabularies. The case of poor vocabularies will be covered in the exercises (Exercises 13.9, 13.13, and 13.14).

We say that the vocabulary Σ is *mono-unary* if it contains no function symbols other than a single unary one. It may contain constants and predicate symbols.

Let $\mathfrak{A}$ be a Σ-structure generated by the empty set and let $\#A = n$. Without loss of generality, we can assume that $A = \{0, 1, \ldots, n-1\}$ and that $C_{\mathfrak{A}}(k) = k$ for all $k < n$. Every structure can be transformed into one satisfying this property by renaming elements if necessary. Let S_n be the set of all such structures over a fixed vocabulary Σ. Clearly, the set S_n depends on the vocabulary Σ. We shall

3 The equality symbol is not counted here.

write S_n^L when we want to make the dependence on Σ explicit. It follows from Proposition 13.8 that if $\mathfrak{A}, \mathfrak{B} \in S_n$ are different, then they are not isomorphic. Also every n-element Σ-structure with no proper substructures is isomorphic to precisely one element of S_n.

We encode every element $\mathfrak{A}$ of S_n by a binary string $\ulcorner\mathfrak{A}\urcorner \in \{0,1\}^*$ as follows. All elements of $\{0,\ldots,n-1\}$ are encoded in binary using the same length, $\lfloor \log(n-1) \rfloor + 1$. The code of $\mathfrak{A}$ consists of concatenating the values of consecutive symbols of Σ in the order in which they occur in Σ, where the values of any function or predicate[4] in $\mathfrak{A}$ are listed for consecutive arguments in lexicographic order with respect to the natural order in $\{0,\ldots,n-1\}$. It is easy to see that for every $\mathfrak{A} \in S_n$, the length of $\ulcorner\mathfrak{A}\urcorner$ is polynomial in n.[5]

Let us illustrate the coding technique with an example.

Example 13.9: Let $\mathfrak{A} = (\{0,1,2\}, c, f, \leq)$, where c is a constant that denotes 1, f is the binary operation of addition modulo 3, and $\leq$ is the linear order $0 \leq 1 \leq 2$. Clearly, $\mathfrak{A}$ is generated by the empty set. The natural chain in $\mathfrak{A}$ is $1, 2, 0$, thus $\mathfrak{A} \notin S_3$. However, $\mathfrak{A}$ is isomorphic to $\mathfrak{A}' = (\{0,1,2\}, c', f', \leq')$, where c' denotes 0, $f'(x,y) = x + y + 1 \pmod 3$, and $\leq'$ is the linear order $2 \leq' 0 \leq' 1$. The natural chain in $\mathfrak{A}'$ is $0, 1, 2$, therefore $\mathfrak{A}' \in S_3$. In order to help read off the code of $\mathfrak{A}'$, we abbreviate 00 by 0, 01 by 1, and 10 by 2. The code of $\mathfrak{A}'$ is given below.

$$0\ \underbrace{1\,2\,0\,2\,0\,1\,0\,1\,2}_{\text{code of } f'}\ \underbrace{1\,1\,0\,0\,1\,0\,1\,1\,1}_{\text{code of } \leq'}$$

Spectra

We are now ready to define the notion of a *spectrum* of a programming language. Let K be a programming language and let $\alpha \in K$ and $m \geq 0$. The mth *spectrum of* α is the set

$$SP_m(\alpha) \stackrel{\text{def}}{=} \{\ulcorner\mathfrak{A}_w\urcorner \mid \mathfrak{A} \text{ is a finite } \Sigma\text{-structure, } w \text{ is an } m\text{-state in } \mathfrak{A}, \text{ and } \mathfrak{A}, w \vDash \texttt{<}\alpha\texttt{>}\mathbf{1}\}.$$

The *spectrum* of K is the set

$$SP(K) \stackrel{\text{def}}{=} \{SP_m(\alpha) \mid \alpha \in K,\ m \in \mathbb{N}\}.$$

4 Truth values of a predicate are represented using the correspondence 0 for **0** and 1 for **1**.

5 However, this polynomial depends on Σ.

Given $m \geq 0$, observe that structures in $S_n^{\Sigma \cup \{c_0,\ldots,c_m\}}$ can be viewed as structures of the form $\mathfrak{A}_w$ for a certain Σ-structure $\mathfrak{A}$ and an m-state w in $\mathfrak{A}$. This representation is unique.

In this section we establish the complexity of spectra; that is, the complexity of the halting problem in finite interpretations. Let us fix $m \geq 0$, a rich vocabulary Σ, and new constants $c_0, \ldots, c_m$. Since not every binary string is of the form $\ulcorner \mathfrak{A} \urcorner$ for some Σ-structure $\mathfrak{A}$ and m-state w in $\mathfrak{A}$, we will restrict our attention to strings that are of this form. Let

$$H_m^\Sigma \quad \stackrel{\text{def}}{=} \quad \{\ulcorner \mathfrak{A} \urcorner \mid \mathfrak{A} \in S_n^{\Sigma \cup \{c_0,\ldots,c_m\}} \text{ for some } n \geq 1\}.$$

It is easy to show that the language H_m^Σ is in *LOGSPACE* for every vocabulary Σ and $m \geq 0$. Later, we shall need the following result.

LEMMA 13.10: Let $m \geq 0$ and let L be a rich vocabulary. For every language $X \subseteq \{0,1\}^*$, there is a language $Y \subseteq H_m^\Sigma$ such that

$$X \quad \leq_{\log} \quad Y \quad \leq_{\log} \quad X.$$

Proof The proof is structured according to the symbols that belong to Σ. Let us consider the case in which Σ contains a unary relation symbol r and a unary function symbol f. The other cases are dealt with similarly, and we leave them to the reader.

Let $x \in \{0,1\}^*$. We define a Σ-structure $\mathfrak{B}_x$ and an Herbrand-like m-state u. Let $n = |x|$ be the length of x. The carrier of $\mathfrak{B}_x$ is the set $U = \{0, 1, \ldots, n\}$. The interpretation of f in $\mathfrak{B}_x$ is the successor function modulo $n+1$. The interpretation of r is as follows. For $i \in U$, we let $r^{\mathfrak{B}_x}(i)$ hold iff $1 \leq i \leq n$ and the ith bit in x is 1.

All other function symbols, including constants, are interpreted as functions constantly equal to 0. All other relation symbols are interpreted as empty relations. The state u assigns 0 to every variable. We leave it to the reader to show that there is a *LOGSPACE*-computable function $\Theta : \{0,1\}^* \to \{0,1\}^*$ such that $\Theta(x) = \ulcorner \mathfrak{B}_x, u \urcorner$. Since $\mathfrak{B}_x$ and $\mathfrak{B}_y$ are not isomorphic for $x \neq y$, it follows that Θ is one-to-one.

Let us describe a computation of another function $\Psi : \{0,1\}^* \to \{0,1\}^*$. Given an input $y \in \{0,1\}^*$, it checks whether $y \in H_m^\Sigma$. If so, it finds the cardinality (in binary) of a structure $\mathfrak{A}$ whose code is y. It then reads off from the code whether $f^{\mathfrak{A}}$ is the successor, whether all other operations of $\mathfrak{A}$ are constantly equal to 0, and whether all relations besides $r^{\mathfrak{A}}$ are empty. If so, it reads off from the code

of $r^{\mathfrak{A}}$ the bits of a string x such that $\ulcorner \mathfrak{B}_x, u \urcorner = y$. If on any of these tests the machine computing Ψ should fail, the computation is aborted and the value of $\Psi(y)$ is the empty string. The reader can easily check that Ψ is indeed computable by a *LOGSPACE* transducer and that

$$\begin{aligned} \Psi(\Theta(x)) &= x \quad \text{for all } x \in \{0,1\}^*, \\ \Theta(\Psi(y)) &= y \quad \text{for all } y \in \Theta(\{0,1\}^*). \end{aligned}$$

Given $X \subseteq \{0,1\}^*$, let $Y = \Theta(X)$. It follows that Θ establishes the reduction $X \leq_{\log} Y$, while Ψ establishes the reduction $Y \leq_{\log} X$. ■

We are now ready to connect complexity classes with spectra. Let K be any programming language and let $C \subseteq 2^{\{0,1\}^*}$ be a family of sets. We say that $SP(K)$ *captures* C, denoted $SP(K) \approx C$, if

- $SP(K) \subseteq C$, and
- for every $X \in C$ and $m \geq 0$, if $X \subseteq H_m^{\Sigma}$, then there is a program $\alpha \in K$ such that $SP_m(\alpha) = X$.

For example, if C is the class of all sets recognizable in polynomial time, then $SP(K) \approx P$ means that

- the halting problem over finite interpretations for programs from K is decidable in polynomial time, and
- every polynomial-time-recognizable set of codes of finite interpretations is the spectrum of some program from K.

We conclude this section by establishing the spectral complexity of some of the programming languages introduced in Chapter 11.

THEOREM 13.11: Let Σ be a rich vocabulary. Then

(i) $SP(\text{dreg}) \subseteq$ *LOGSPACE*.
(ii) $SP(\text{reg}) \subseteq$ *NLOGSPACE*.

Moreover, if Σ is mono-unary, then $SP(\text{dreg})$ captures *LOGSPACE* and $SP(\text{reg})$ captures *NLOGSPACE*.

Proof We first show (i). Let α be a deterministic regular program and let $m \geq 0$. A deterministic off-line $O(\log n)$-space-bounded Turing machine M_α that accepts $SP_m(\alpha)$ can be constructed as follows. For a given input string $z \in \{0,1\}^*$, it checks

whether z is the code of an expanded structure $\mathfrak{A}_w \in S_n^{\Sigma \cup \{c_0,\ldots,c_m\}}$ for some $n \geq 1$. This can be done in $O(\log n)$ space. If so, it starts a simulation of a computation of α in $\mathfrak{A}$, taking the values given by w as initial values for the registers of α. At any stage of the simulation, the current values of the registers of α are stored on the work tape of M_α using their binary representations of length $O(\log n)$. The necessary tests and updates of values of the registers of α can be read off from the input string z. The machine M_α halts iff α halts for $(\mathfrak{A},\ w)$.

The proof of (ii) is essentially the same, except that M_α will be nondeterministic.

For the second part of the theorem, assume that Σ is mono-unary. We show that *SP*(dreg) captures *LOGSPACE*. The argument for *SP*(reg) is similar and is omitted. Let $X \in \mathit{LOGSPACE}$ and $X \subseteq H_m^\Sigma$ for some $m \geq 0$. We describe a deterministic regular program α such that for every $n \geq 1$ and every $\mathfrak{A}_w \in S_n^{\Sigma \cup \{c_0,\ldots,c_m\}}$,

$$\mathfrak{A}, w \models \texttt{<}\alpha\texttt{>}\mathbf{1} \iff \ulcorner \mathfrak{A}_w \urcorner \in X.$$

First, let us consider the case in which the carrier of $\mathfrak{A}$ has only one element. There are only finitely many pairwise nonisomorphic structures over a one-element carrier. They differ only by different interpretations of the predicate symbols. Let $\mathfrak{A}_1, \ldots, \mathfrak{A}_k$ all be one-element structures such that $\ulcorner \mathfrak{A}_w^i \urcorner \in X$. Since $\mathfrak{A}^i$ has only one element, it follows that w is uniquely determined.

The program α first checks whether the structure generated by the input has exactly one element. If so, it checks whether this structure is one of the $\mathfrak{A}^i$ listed above, in which case it halts. Otherwise it diverges.

From now on, we assume that $\mathfrak{A}$ has more than one element. Let M be a deterministic off-line $O(\log n)$-space-bounded Turing machine that accepts X. Without loss of generality, we can assume that M's tape alphabet is $\{0, 1\}$. Moreover, since the length of the input $\ulcorner \mathfrak{A}_w \urcorner$ for M is polynomial in $\#\mathfrak{A} = n$, we can assume without loss of generality that the work tape of M has length $k\lfloor \log n \rfloor$, where k is constant. Hence, the contents of this tape can be stored by α in k registers, each holding a value $a \in A$ whose binary expansion represents the relevant portion of the work tape.

In order to store head positions of M, the program α uses counters, which are simulated as follows. Since Σ is mono-unary, one can define a deterministic regular program that plays the role of the program $\textsc{Next}_m$ of Proposition 13.7. This is the only place where we crucially use the assumption about Σ. Hence, α can compute the successor function that counts up to $n - 1$ in an n-element structure. Using several registers, α can thus count up to a polynomial number of steps.

The bits of the code $\ulcorner\mathfrak{A}_w\urcorner$ can be read off directly from $\mathfrak{A}$ and the first $m+1$ registers $x_0, \ldots, x_m$, which store the initial values of w. For this it is enough to have polynomial-size arithmetic on counters, as explained above.

Now α can simulate the computation of M step by step, updating the contents of M's work tape and M's head positions. It halts if and only if M eventually reaches an accepting state. ■

Theorem 13.12: Over a rich vocabulary Σ, *SP*(dstk) and *SP*(stk) capture *P*.

Proof The proof is very similar to the proof of Theorem 13.11. Instead of mutual simulation with $O(\log n)$-space-bounded Turing machines, we work with Cook's $O(\log n)$ *auxiliary pushdown automata* (APDAs); see Chapter 14 of Hopcroft and Ullman (1979) for the definition. The pushdown store of the APDA directly simulates the algebraic stack of a regular program. It follows from Cook's theorem (see Theorem 14.1 of Hopcroft and Ullman (1979)) that languages accepted by deterministic/nondeterministic $O(\log n)$ APDAs coincide with *P*. The program $\textsc{Next}_m$ of Proposition 13.7 is used to simulate counters as in the proof of Theorem 13.11. ■

Theorem 13.13: If Σ is a rich vocabulary, then *SP*(darray) and *SP*(array) capture *PSPACE*.

Proof Again, the proof is very similar to that of Theorem 13.11. This time we mutually simulate deterministic/nondeterministic regular programs with arrays and deterministic/nondeterministic polynomial space Turing machines. By Savitch's theorem (Hopcroft and Ullman, 1979, Theorem 12.11) it follows that both models of Turing machines accept the same class of languages, namely *PSPACE*. To simulate counters for the backwards reduction, we need a deterministic regular program with arrays that performs the same function as the program $\textsc{Next}_m$ of Proposition 13.7. The easy details are left to the reader. ■

13.3 Bibliographical Notes

The Π^1_1-completeness of DL was first proved by Meyer, and Theorem 13.1 appears in Harel et al. (1977). The proof given here is from Harel (1985). Theorem 13.4 is from Meyer and Halpern (1982). That the fragment of DL considered in Theorem 13.5 is not r.e., was proved by Pratt (1976). Theorem 13.6 follows from Harel and Kozen (1984).

The name "spectral complexity" was proposed by Tiuryn (1986), although the main ideas and many results concerning this notion were already present in Tiuryn and Urzyczyn (1983); the reader may consult Tiuryn and Urzyczyn (1988) for the full version. This notion is an instance of the so-called *second-order spectrum* of a formula. First-order spectra were investigated by Sholz (1952), from which originates the well known *Spectralproblem*. The reader can find more about this problem and related results in the survey paper by Börger (1984). Proposition 13.7 and the notion of a natural chain is from Urzyczyn (1983a). The results of Section 13.2 are from Tiuryn and Urzyczyn (1983, 1988); see the latter for the complete version. A result similar to Theorem 13.12 in the area of finite model theory was obtained by Sazonov (1980) and independently by Gurevich (1983). Higher-order stacks were introduced in Engelfriet (1983) to study complexity classes. Higher-order arrays and stacks in DL were considered by Tiuryn (1986), where a strict hierarchy within the class of elementary recursive sets was established. The main tool used in the proof of the strictness of this hierarchy is a generalization of Cook's auxiliary pushdown automata theorem for higher-order stacks, which is due to Kowalczyk et al. (1987).

Exercises

13.1. Prove Theorem 13.4 for termination or total correctness formulas of the form $\psi \to \langle\alpha\rangle\varphi$.

13.2. Prove Theorem 13.5 for partial correctness assertions of the form $\psi \to [\alpha]\varphi$.

13.3. Prove Theorem 13.2 for DL(dreg).

13.4. Prove Theorem 13.5 for DL(dreg).

13.5. Show that for every structure $\mathfrak{A}$, the image $C_{\mathfrak{A}}(\mathbb{N})$ is the substructure of $\mathfrak{A}$ generated by the empty set; that is, the least substructure of $\mathfrak{A}$.

13.6. Write a recursive procedure that computes the successor function with respect to the natural chain in $\mathfrak{A}_w$ (see Proposition 13.7).

13.7. Show that if a vocabulary Σ contains no function symbols of positive arity, then DL(r.e.) reduces to first-order logic over all structures.

13.8. Show that for a rich vocabulary Σ and a given $n > 0$, there are exponentially many (in n) pairwise nonisomorphic Σ-structures $\mathfrak{A}$ such that $\#A = n$ and $\mathfrak{A}$ is generated by the empty set.

13.9. Show that for a poor vocabulary Σ and for a given $n > 0$, there are polynomially many (in n) pairwise nonisomorphic Σ-structures $\mathfrak{A}$ such that $\#A = n$ and $\mathfrak{A}$ is generated by the empty set.

13.10. Let Σ be a rich vocabulary. Show that for every $\mathfrak{A} \in S_n$, the length of $\ulcorner\mathfrak{A}\urcorner$ is polynomial in n.

13.11. Show that for every rich vocabulary Σ and $m \geq 0$, the language H_m^Σ is in *LOGSPACE*.

13.12. (Tiuryn (1986)) Show that if the vocabulary Σ is rich, then the spectra of deterministic/nondeterministic regular programs with an algebraic stack and arrays capture *EXPTIME*.

13.13. Let Σ be a poor vocabulary. Give an encoding $\ulcorner\ulcorner\mathfrak{A}\urcorner\urcorner \in \{0,1\}^*$ of finite structures $\mathfrak{A} \in S_n$ such that the length of $\ulcorner\ulcorner\mathfrak{A}\urcorner\urcorner$ is $O(\log n)$.

13.14. Let Σ be a poor vocabulary. Redefine the notion of a spectrum following the encoding of structures for poor vocabularies, and show that the complexity classes thus captured by spectra become exponentially higher. For example:

- spectra of deterministic regular programs capture $DSPACE(n)$;
- spectra of nondeterministic regular programs capture $NSPACE(n)$;
- spectra of regular programs with an algebraic stack capture $DTIME(2^{O(n)})$;
- spectra of regular programs with arrays capture $DSPACE(2^{O(n)})$ (see Tiuryn and Urzyczyn (1988)).

14 Axiomatization

This chapter deals with axiomatizing first-order Dynamic Logic. We divide our treatment along the same lines taken in Chapters 12 and 13, dealing with the uninterpreted and interpreted cases separately. We must remember, though, that in both cases the relevant validity problems are highly undecidable, something we will have to find a way around.

14.1 The Uninterpreted Level

Recall from Section 13.1 that validity in DL is Π^1_1-complete, but only r.e. when restricted to simple termination assertions. This means that termination (or total correctness when the programs are deterministic) can be fully axiomatized in the standard sense. This we do first, and we then turn to the problem of axiomatizing full DL.

Completeness for Termination Assertions

Although the reader may feel happy with Theorem 13.4, it should be stressed that only very simple computations are captured by valid termination assertions:

PROPOSITION 14.1: Let $\varphi \to \langle\alpha\rangle\psi$ be a valid formula of DL, where φ and ψ are first-order and α contains first-order tests only. There exists a constant $k \geq 0$ such that for every structure $\mathfrak{A}$ and state u, if $\mathfrak{A}, u \vDash \varphi$, there is a computation sequence $\sigma \in CS(\alpha)$ of length at most k such that $\mathfrak{A}, u \vDash \langle\sigma\rangle\psi$.

Proof The proof is left as an exercise (Exercise 14.1). ■

Nevertheless, since the validity problem for such termination assertions is r.e., it is of interest to find a nicely-structured complete axiom system. We propose the following.

AXIOM SYSTEM 14.2:

Axiom Schemes

- all instances of valid first-order formulas;
- all instances of valid formulas of PDL;
- $\varphi[x/t] \to \langle x := t\rangle\varphi$, where φ is a first-order formula.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

We denote provability in Axiom System 14.2 by $\vdash_{S1}$.

LEMMA 14.3: For every first-order formula ψ and for every sequence σ of atomic assignments and atomic tests, there is a first-order formula ψ_σ such that

$$\vDash \ \psi_\sigma \quad \leftrightarrow \quad \langle\sigma\rangle\psi.$$

Proof The proof is left as an exercise (Exercise 14.2). ■

THEOREM 14.4: For any DL formula of the form $\varphi \to \langle\alpha\rangle\psi$, for first-order φ and ψ and program α containing first-order tests only,

$$\vDash \varphi \to \langle\alpha\rangle\psi \quad \Longleftrightarrow \quad \vdash_{S1} \varphi \to \langle\alpha\rangle\psi.$$

Proof Soundness ($\Longleftarrow$) is obvious. The proof of completeness ($\Longrightarrow$) proceeds by induction on the structure of α and makes heavy use of the compactness of first-order logic. We present the case for $\varphi \to \langle\beta \cup \gamma\rangle\psi$.

By assumption, $\vDash \varphi \to \langle\beta \cup \gamma\rangle\psi$, therefore $\vDash \varphi \to \bigvee_{\sigma \in CS(\beta\cup\gamma)} \psi_\sigma$, where ψ_σ is the first-order equivalent to $\langle\sigma\rangle\psi$ from Lemma 14.3. By the compactness of first-order logic, $\vDash \varphi \to \bigvee_{\sigma\in C} \psi_\sigma$ for some finite set of seqs $C \subseteq CS(\beta \cup \gamma) = CS(\beta) \cup CS(\gamma)$. This can be written

$$\vDash \ \varphi \quad \to \quad (\bigvee_{\sigma\in C_1} \psi_\sigma \vee \bigvee_{\tau\in C_2} \psi_\tau)$$

for some finite sets $C_1 \subseteq CS(\beta)$ and $C_2 \subseteq CS(\gamma)$. Since the last formula is first-order and valid, by the completeness of first-order logic we have

$$\vdash_{S1} \ \varphi \quad \to \quad (\bigvee_{\sigma\in C_1} \psi_\sigma \vee \bigvee_{\tau\in C_2} \psi_\tau). \tag{14.1.1}$$

However, since $C_1 \subseteq CS(\beta)$ and $C_2 \subseteq CS(\gamma)$, we have $\vDash \bigvee_{\sigma\in C_1} \psi_\sigma \to \langle\beta\rangle\psi$ and $\vDash \bigvee_{\tau\in C_2} \psi_\tau \to \langle\gamma\rangle\psi$. Applying the inductive hypothesis to each yields $\vdash_{S1} \bigvee_{\sigma\in C_1} \psi_\sigma \to \langle\beta\rangle\psi$ and $\vdash_{S1} \bigvee_{\tau\in C_2} \psi_\tau \to \langle\gamma\rangle\psi$. By (14.1.1) and propositional

reasoning, we obtain

$$\vdash_{S1} \varphi \quad \rightarrow \quad (\langle\beta\rangle\psi \vee \langle\gamma\rangle\psi),$$

which together with an instance of the PDL tautology $\langle\beta\rangle\psi \vee \langle\gamma\rangle\psi \rightarrow \langle\beta \cup \gamma\rangle\psi$ yields $\vdash_{S1} \varphi \rightarrow \langle\beta \cup \gamma\rangle\psi$. ■

Remark 14.5: The result also holds if α is allowed to involve tests that are themselves formulas as defined in the theorem.

Infinitary Completeness for the General Case

Given the high undecidability of validity in DL, we cannot hope for a complete axiom system in the usual sense. Nevertheless, we do want to provide an orderly axiomatization of valid DL formulas, even if this means that we have to give up the finitary nature of standard axiom systems.

In this section, we present a complete infinitary axiomatization of DL that includes an inference rule with infinitely many premises. Before doing so, however, we must get a certain technical complication out of the way. We would like to be able to consider valid first-order formulas as axiom schemes, but instantiated by general formulas of DL. In order to make formulas amenable to first-order manipulation, we must be able to make sense of such notions as "a free occurrence of x in φ" and the substitution $\varphi[x/t]$. For example, we would like to be able to use the axiom scheme of the predicate calculus $\forall x\ \varphi \rightarrow \varphi[x/t]$, even if φ contains programs.

The problem arises because the dynamic nature of the semantics of DL may cause a single occurrence of a variable in a DL formula to act as both a free and bound occurrence. For example, in the formula $\langle\mathbf{while}\ x \leq 99\ \mathbf{do}\ x := x+1\rangle\mathbf{1}$, the occurrence of x in the expression $x+1$ acts as both a free occurrence (for the first assignment) and as a bound occurrence (for subsequent assignments).

There are several reasonable ways to deal with this, and we present one for definiteness. Without loss of generality, we assume that whenever required, all programs appear in the special form

$$\langle\overline{z} := \overline{x}\ ;\ \alpha\ ;\ \overline{x} := \overline{z}\rangle\varphi \tag{14.1.2}$$

where $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{z} = (z_1, \ldots, z_n)$ are tuples of variables, $\overline{z} := \overline{x}$ stands for

$$z_1 := x_1\ ;\ \cdots\ ;\ z_n := x_n$$

(and similarly for $\overline{x} := \overline{z}$), the x_i do not appear in α, and the z_i are new variables

appearing nowhere in the relevant context outside of the program α. The idea is to make programs act on the "local" variables z_i by first copying the values of the x_i into the z_i, thus freezing the x_i, executing the program with the z_i, and then restoring the x_i. This form can be easily obtained from any DL formula by consistently changing all variables of any program to new ones and adding the appropriate assignments that copy and then restore the values. Clearly, the new formula is equivalent to the old. Given a DL formula in this form, the following are bound occurrences of variables:

- all occurrences of x in a subformula of the form $\exists x\ \varphi$;
- all occurrences of z_i in a subformula of the form (14.1.2) (note, though, that z_i does not occur in φ at all);
- all occurrences of x_i in a subformula of the form (14.1.2) except for its occurrence in the assignment $z_i := x_i$.

Every occurrence of a variable that is not bound is free. Our axiom system will have an axiom that enables free translation into the special form discussed, and in the sequel we assume that the special form is used whenever required (for example, in the assignment axiom scheme below).

As an example, consider the formula:

$$\begin{aligned}&\forall x\ (\langle y := f(x);\ x := g(y,x)\rangle p(x,y))\\ &\quad\to\quad \langle z_1 := h(z);\ z_2 := y;\ z_2 := f(z_1);\ z_1 := g(z_2,z_1);\ x := z_1;\ y := z_2\rangle p(x,y).\end{aligned}$$

Denoting $\langle y := f(x); x := g(y,x)\rangle p(x,y)$ by φ, the conclusion of the implication is just $\varphi[x/h(z)]$ according to the convention above; that is, the result of replacing all free occurrences of x in φ by $h(z)$ after φ has been transformed into special form. We want the above formula to be considered a legal instance of the assignment axiom scheme below.

Now consider the following axiom system.

Axiom System 14.6:

Axiom Schemes

- all instances of valid first-order formulas;
- all instances of valid formulas of PDL;
- $\langle x := t\rangle\varphi \leftrightarrow \varphi[x/t]$;
- $\varphi \leftrightarrow \widehat{\varphi}$, where $\widehat{\varphi}$ is φ in which some occurrence of a program α has been replaced

by the program $z := x;\ \alpha';\ x := z$ for z not appearing in φ, and where α' is α with all occurrences of x replaced by z.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

- generalization:

$$\frac{\varphi}{[\alpha]\varphi} \quad \text{and} \quad \frac{\varphi}{\forall x\varphi}$$

- infinitary convergence:

$$\frac{\varphi \to [\alpha^n]\psi,\ n \in \omega}{\varphi \to [\alpha^*]\psi}$$

Provability in Axiom System 14.6, denoted by $\vdash_{S2}$, is the usual concept for systems with infinitary rules of inference; that is, deriving a formula using the infinitary rule requires infinitely many premises to have been previously derived.

Axiom System 14.6 consists of an axiom for assignment, facilities for propositional reasoning about programs and first-order reasoning with no programs (but with programs possibly appearing in instantiated first-order formulas), and an infinitary rule for $[\alpha^*]$. The dual construct, $<\alpha^*>$, is taken care of by the "unfolding" validity of PDL:

$$<\alpha^*>\varphi \quad \leftrightarrow \quad (\varphi \vee <\alpha;\alpha^*>\varphi).$$

See Example 14.8 below.

The main result here is:

THEOREM 14.7: For any formula φ of DL,

$$\vDash \varphi \quad \Longleftrightarrow \quad \vdash_{S2} \varphi.$$

Proof sketch. Soundness is straightforward. Completeness can be proved by adapting any one of the many known completeness proofs for the classical infinitary logic, $L_{\omega_1\omega}$. Algebraic methods are used in Mirkowska (1971), whereas Harel (1984) uses Henkin's method. For definiteness, we sketch an adaptation of the proof given in Keisler (1971).

Take the set At of *atoms* to consist of all consistent finite sets of formulas possibly involving elements from among a countable set G of new constant symbols. By an atom A being consistent, we mean that it is not the case that $\vdash_{S2} \neg\widehat{A}$, where $\widehat{A} = \bigwedge_{\varphi\in A}\varphi$. It is now shown how to construct a model for any $A \in At$. The result will then follow from the fact that for any consistent formula φ, $\{\varphi\} \in At$.

Given an atom A, we define its *closure* $CL(A)$ to be the least set of formulas containing all formulas of A and their subformulas, exactly as is done for the Fischer-Ladner closure $FL(\varphi)$ in Section 6.1, but which is also closed under substitution of constants from G for arbitrary terms, and which contains $c = d$ for each $c, d \in G$. An infinite sequence of atoms $A = A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots$ is now constructed. Given $A_i \in At$, A_{i+1} is constructed by considering φ_i, the ith closed formula of $CL(A)$ in some fixed ordering, and checking whether $A_i \cup \{\varphi_i\} \in At$. If so, certain formulas are added to A_i to produce A_{i+1}, depending on the form of φ_i.

A typical rule of this kind is the following. If $\varphi_i = \texttt{<}\alpha^*\texttt{>}\psi$, then we claim that there must be some n such that $\widehat{A}_i \vee \texttt{<}\alpha^n\texttt{>}\psi$ is consistent; then we take A_{i+1} to be $A_i \cup \{\varphi_i,\ \texttt{<}\alpha^n\texttt{>}\psi,\ \ t_i = c\}$, where t_i is the ith item in some fixed enumeration of the basic terms over the current vocabulary, but with constants from G, and where $c \in G$ does not occur in A_i. To see that such an n exists, assume to the contrary that $\vdash_{S2} \neg(\widehat{A}_i \wedge \texttt{<}\alpha^n\texttt{>}\psi)$ for every n. Then $\vdash_{S2} \widehat{A}_i \to \texttt{[}\alpha^n\texttt{]}\neg\psi$ for each n. By the infinitary convergence rule, $\vdash_{S2} \widehat{A}_i \to \texttt{[}\alpha^*\texttt{]}\psi$, which is $\vdash_{S2} \neg(\widehat{A}_i \wedge \texttt{<}\alpha^*\texttt{>}\psi)$. But this contradicts the fact that $A_i \cup \{\varphi_i\} \in At$.

Now let $A_\infty = \bigcup_i A_i$ and let $\widehat{c} = \{d \in G \mid (c = d) \in A_\infty\}$. The structure $\mathfrak{A} = (D, \mathfrak{m}_\mathfrak{A})$ is obtained by taking the carrier to be $D = \{\widehat{c} \mid c \in G\}$ and for example setting $\mathfrak{m}_\mathfrak{A}(p)(\widehat{c}_1, \ldots, \widehat{c}_k)$ to be true iff $p(c_1, \ldots, c_k) \in A_\infty$. A straightforward induction on the complexity of formulas shows that all formulas of A_∞ are true in $\mathfrak{A}$. ■

Example 14.8: We use Axiom System 14.6 to prove the validity of the following formula:

$$x = y \quad\to\quad \texttt{[}(x := f(f(x)))^*\texttt{]}\texttt{<}(y := f(y))^*\texttt{>}x = y.$$

To that end, we show that for every n,

$$\vdash_{S2}\ x = y \quad\to\quad \texttt{[}(x := f(f(x)))^n\texttt{]}\texttt{<}(y := f(y))^*\texttt{>}x = y$$

and then apply the infinitary convergence rule to obtain the result. Let n be fixed.

We first prove

$$\vdash_{S2} \ x = y \quad \rightarrow \quad [x := f(f(x))]\,[x := f(f(x))] \ldots [x := f(f(x))] \\ \langle y := f(y)\rangle\langle y := f(y)\rangle \ldots \langle y := f(y)\rangle \ x = y \qquad (14.1.3)$$

with n occurrences of $[x := f(f(x))]$ and $2n$ occurrences of of $\langle y := f(y)\rangle$. This is done by starting with the first-order validity $\vdash_{S2} x = y \rightarrow f^{2n}(x) = f^{2n}(y)$ (where $f^{2n}(\cdot)$ abbreviates $f(f(\cdots(\cdot)\cdots)$ with $2n$ occurrences of f), and then using the assignment axiom with propositional manipulation n times to obtain

$$\vdash_{S2} \ x = y \quad \rightarrow \quad [x := f(f(x))] \ldots [x := f(f(x))]\,x = f^{2n}(y),$$

and again $2n$ times to obtain (14.1.3). Having proved (14.1.3), we use the PDL validity $\varphi \rightarrow \langle\alpha^*\rangle\varphi$ with φ taken to be $x = y$ and α taken to be $y := f(y)$, then apply the PDL validity $\langle\alpha\rangle\langle\alpha^*\rangle\varphi \rightarrow \langle\alpha^*\rangle\varphi$ $2n$ times with the same instantiation, using the monotonicity rules

$$\frac{\varphi \rightarrow \psi}{\langle\alpha\rangle\varphi \rightarrow \langle\alpha\rangle\psi} \qquad\qquad \frac{\varphi \rightarrow \psi}{[\alpha]\varphi \rightarrow [\alpha]\psi}$$

to obtain

$$\vdash_{S2} \ x = y \quad \rightarrow \quad [x := f(f(x))] \ldots [x := f(f(x))]\langle(y := f(y))^*\rangle x = y.$$

Now $n - 1$ applications of the PDL validity $[\alpha][\beta]\varphi \rightarrow [\alpha;\beta]\varphi$ yield the desired result.

14.2 The Interpreted Level

Proving properties of real programs very often involves reasoning on the interpreted level, where one is interested in $\mathfrak{A}$-validity for a particular structure $\mathfrak{A}$. A typical proof might use induction on the length of the computation to establish an invariant for partial correctness or to exhibit a decreasing value in some well-founded set for termination. In each case, the problem is reduced to the problem of verifying some domain-dependent facts, sometimes called *verification conditions*. Mathematically speaking, this kind of activity is really an effective transformation of assertions about programs into ones about the underlying structure.

In this section, we show how for DL this transformation can be guided by a direct induction on program structure using an axiom system that is complete *relative to* any given arithmetical structure $\mathfrak{A}$. The essential idea is to exploit the existence, for any given DL formula, of a first-order equivalent in $\mathfrak{A}$, as guaranteed by Theorem

12.6. In the axiom systems we construct, instead of dealing with the Π_1^1-hardness of the validity problem by an infinitary rule, we take all $\mathfrak{A}$-valid first-order formulas as additional axioms. Relative to this set of axioms, proofs are finite and effective.

In Section 14.2 we take advantage of the fact that for partial correctness assertions of the form $\varphi \rightarrow [\alpha]\psi$ with φ and ψ first-order and α containing first-order tests, it suffices to show that DL reduces to the first-order logic $L_{\omega\omega}$, and there is no need for the natural numbers to be present. Thus, the system we present in Section 14.2 works for finite structures too. In Section 14.2, we present an *arithmetically complete* system for full DL that does make explicit use of natural numbers.

Relative Completeness for Correctness Assertions

It follows from Theorem 13.5 that for partial correctness formulas we cannot hope to obtain a completeness result similar to the one proved in Theorem 14.4 for termination formulas. A way around this difficulty is to consider only *expressive* structures.

A structure $\mathfrak{A}$ for the first-order vocabulary Σ is said to be *expressive* for a programming language K if for every $\alpha \in K$ and for every first-order formula φ, there exists a first-order formula ψ_L such that $\mathfrak{A} \models \psi_L \leftrightarrow [\alpha]\varphi$. Examples of structures that are expressive for most programming languages are finite structures and arithmetical structures.

Consider the following axiom system:

Axiom System 14.9:

Axiom Schemes

- all instances of valid formulas of PDL;
- $<x := t>\varphi \leftrightarrow \varphi[x/t]$ for first-order φ.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \rightarrow \psi}{\psi}$$

- generalization:

$$\frac{\varphi}{[\alpha]\varphi}.$$

Note that Axiom System 14.9 is really the axiom system for PDL from Chapter 7 with the addition of the assignment axiom. Given a DL formula φ and a structure $\mathfrak{A}$, denote by $\mathfrak{A} \vdash_{S3} \varphi$ provability of φ in the system obtained from Axiom System 14.9 by adding the following set of axioms:

- all $\mathfrak{A}$-valid first-order sentences.

THEOREM 14.10: For every expressive structure $\mathfrak{A}$ and for every formula ξ of DL of the form $\varphi \to [\alpha]\psi$, where φ and ψ are first-order and α involves only first-order tests, we have

$$\mathfrak{A} \vDash \xi \iff \mathfrak{A} \vdash_{S3} \xi.$$

Proof Soundness is trivial. For completeness, one proceeds by induction on the structure of α. We present the case for $\alpha = \beta^*$.

By the assumption, $\mathfrak{A} \vDash \varphi \to [\beta^*]\psi$. Consider the first-order formula $([\beta^*]\psi)_L$, which exists by the expressiveness of $\mathfrak{A}$, and denote it by χ. Clearly, $\mathfrak{A} \vDash \varphi \to \chi$ and $\mathfrak{A} \vDash \chi \to \psi$. Since both these formulas are first-order and are $\mathfrak{A}$-valid, they are axioms, so we have:

$$\mathfrak{A} \vdash_{S3} \varphi \to \chi \tag{14.2.1}$$

$$\mathfrak{A} \vdash_{S3} \chi \to \psi. \tag{14.2.2}$$

However, by the semantics of β^* we also have $\mathfrak{A} \vDash \chi \to [\beta]\chi$, from which the inductive hypothesis yields $\mathfrak{A} \vdash_{S3} \chi \to [\beta]\chi$. Applying the generalization rule with $[\beta^*]$ and using modus ponens with the PDL induction axiom of Chapter 7 yields $\mathfrak{A} \vdash_{S3} \chi \to [\beta^*]\chi$. This together with (14.2.1), (14.2.2), and PDL manipulation yields $\mathfrak{A} \vdash_{S3} \varphi \to [\beta^*]\psi$. ■

REMARK 14.11: The theorem holds also if α is allowed to involve tests of the form $<\alpha>\chi$, where χ is first-order and α is constructed inductively in the same way.

Arithmetical Completeness for the General Case

In this section we prove the completeness of an axiom system for full DL. It is similar in spirit to the system of the previous section in that it is complete relative to the formulas valid in the structure under consideration. However, this system works for arithmetical structures only. It is not tailored to deal with other expressive structures, notably finite ones, since it requires the use of the natural numbers. The kind of completeness result proved here is thus termed *arithmetical*.

As in Section 12.2, we will prove the results for the special structure $\mathbb{N}$, omitting the technicalities needed to deal with general arithmetical structures, a task we leave to the exercises. The main difference in the proofs is that in $\mathbb{N}$ we can use variables n, m, etc., knowing that their values will be natural numbers. We can thus write $n + 1$, for example, assuming the standard interpretation. When working in an unspecified arithmetical structure, we have to precede such usage with appropriate predicates that guarantee that we are indeed talking about that part of the domain that is isomorphic to the natural numbers. For example, we would often have to use the first-order formula, call it $\mathrm{nat}(n)$, which is true precisely for the elements representing natural numbers, and which exists by the definition of an arithmetical structure.

Consider the following axiom system:

Axiom System 14.12:

Axiom Schemes

- all instances of valid first-order formulas;
- all instances of valid formulas of PDL;
- $<x := t>\varphi \leftrightarrow \varphi[x/t]$ for first-order φ.

Inference Rules

- modus ponens:

$$\frac{\varphi,\ \varphi \to \psi}{\psi}$$

- generalization:

$$\frac{\varphi}{[\alpha]\varphi} \quad \text{and} \quad \frac{\varphi}{\forall x \varphi}$$

- convergence:

$$\frac{\varphi(n+1) \to <\alpha>\varphi(n)}{\varphi(n) \to <\alpha^*>\varphi(0)}$$

for first order φ and variable n not appearing in α.

Remark 14.13: For general arithmetical structures, the $+1$ and 0 in the rule of convergence denote suitable first-order definitions.

As in Axiom System 14.9, denote by $\mathfrak{A} \vdash_{S4} \varphi$ provability of φ in the system obtained from Axiom System 14.12 by adding all $\mathfrak{A}$-valid first-order sentences as axioms.

Interestingly, the infinitary system 14.6 and the arithmetical system 14.12 deal with α^* in dual ways. Here we have the arithmetical convergence rule for $<\alpha^*>$, and $[\alpha^*]$ is dealt with by the PDL induction axiom, whereas in 14.6 we have the infinitary rule for $[\alpha^*]$, and $<\alpha^*>$ is dealt with by the PDL unfolding axiom.

Before we address arithmetical completeness, we prove a slightly more specific version of the expressiveness result of Theorem 12.6. Again, we state it for $\mathbb{N}$, but an appropriately generalized version of it holds for any arithmetical structure.

LEMMA 14.14: For any DL formula φ and program α, there is a first-order formula $\chi(n)$ with a free variable n such that for any state u in the structure $\mathbb{N}$, we have

$$\mathbb{N}, u \vDash \chi(n) \iff \mathbb{N}, u \vDash <\alpha^{u(n)}>\varphi.$$

(Recall that $u(n)$ is the value of variable n in state u.)

Proof The result is obtained as in the proof of Theorem 12.6 in the following way: $\chi(n)$ will be constructed just as $(<\alpha>\varphi)_L$ in that proof. Instead of taking $\varphi_\alpha(y)$ which defines the set $\{\ulcorner\sigma\urcorner \mid \sigma \in CS(\alpha)\}$, we take a formula $\varphi_\alpha(n, y)$ defining the r.e. set

$$\{(n, \ulcorner\sigma_1 \cdots \sigma_n\urcorner) \in \mathbb{N}^2 \mid \sigma_1, \ldots, \sigma_n \in CS(\alpha)\}.$$

The rest of the proof is as in Theorem 12.6. ■

We first show that Axiom System 14.12 is arithmetically complete for first-order termination assertions.

THEOREM 14.15: For every formula ξ of DL of the form $\varphi \to <\alpha>\psi$, for first-order formulas φ and ψ and program α involving only first-order tests,

$$\mathbb{N} \vDash \xi \iff \mathbb{N} \vdash_{S4} \xi.$$

Proof Soundness is trivial. For completeness, we proceed by induction on the structure of α. As in Theorem 14.10, we present the case for $\alpha = \beta^*$.

By assumption, $\mathbb{N} \vDash \varphi \to <\beta^*>\psi$. Consider the first-order formula $\chi(n)$ of Lemma 14.14 for ψ and α. Clearly $\mathbb{N} \vDash \varphi \to \exists n\, \chi(n)$ for n not appearing in φ, ψ

or α, and $\mathbb{N} \vDash \chi(0) \rightarrow \psi$. Hence, these being first-order, we have

$$\mathbb{N} \vdash_{S4} \varphi \rightarrow \exists n\, \chi(n),$$
$$\mathbb{N} \vdash_{S4} \chi(0) \rightarrow \psi.$$

However, from the meaning of $\chi(n)$, we also have $\mathbb{N} \vDash \chi(n+1) \rightarrow \langle\beta\rangle\chi(n)$. By the inductive hypothesis, we obtain $\mathbb{N} \vdash_{S4} \chi(n+1) \rightarrow \langle\beta\rangle\chi(n)$. The convergence rule now yields $\mathbb{N} \vdash_{S4} \chi(n) \rightarrow \langle\beta^*\rangle\chi(0)$. Applying the generalization rule with $\forall n$ and using first-order manipulation, we obtain $\mathbb{N} \vdash_{S4} \exists n\ \chi(n) \rightarrow \langle\beta^*\rangle\chi(0)$, which together with the two formulas above gives the result. ■

The main result here is the following, which holds for any arithmetical structure (see Exercise 14.6):

THEOREM 14.16: For every formula ξ of DL,

$$\mathbb{N} \vDash \xi \iff \mathbb{N} \vdash_{S4} \xi.$$

Proof Soundness is obvious. For completeness, let $\mathbb{N} \vDash \xi$. Define k_ξ to be the sum of the number of programs in ξ and the number of quantifiers prefixing non-first-order formulas in ξ. (Of course, we also count those quantifiers and programs that appear within tests.) We proceed by induction on k_ξ.

If $k_\xi = 0$, ξ must be first-order, so that $\mathbb{N} \vdash_{S4} \xi$. For $k_\xi > 0$, we can assume that ξ is in conjunctive normal form and then deal with each conjunct separately. Without loss of generality (see Exercise 14.7), it suffices to deal with formulas of the form $\varphi \rightarrow op\, \psi$, where $op \in \{\forall x, \exists x, \langle\alpha\rangle, [\alpha]\}$ for some x or α, and where $op\, \psi$ is not first-order. This way, we have k_φ, $k_\psi < k_\xi$.

Now consider the first-order formulas φ_L and ψ_L, which exist by the expressiveness of $\mathbb{N}$. Clearly, since $\mathbb{N} \vDash \varphi \rightarrow op\, \psi$, we also have $\mathbb{N} \vDash \varphi_L \rightarrow op\, \psi_L$. We now claim that this implication is in fact provable:

$$\mathbb{N} \vdash_{S4} \varphi_L \rightarrow op\, \psi_L. \qquad (14.2.3)$$

For $op \in \{\forall x, \exists x\}$, the claim is trivial, since the formula is first-order. For the cases $[\alpha]$ and $\langle\alpha\rangle$, the proof proceeds by induction on α exactly as in the proofs of Theorems 14.10 and 14.15, respectively. The only difference is that the main inductive hypothesis of the present proof is employed in dealing with non-first-order tests.

Now, from $\mathbb{N} \vDash \varphi \rightarrow \varphi_L$ and $\mathbb{N} \vDash \psi_L \rightarrow \psi$, we deduce $\mathbb{N} \vdash_{S4} \varphi \rightarrow \varphi_L$ and $\mathbb{N} \vdash_{S4} \psi_L \rightarrow \psi$ by the inductive hypothesis, since k_φ, $k_\psi < k_\xi$. These combine with

(14.2.3) and some PDL and first-order manipulation to yield $\mathbb{N} \vdash_{S4} \varphi \to op\,\psi$, as desired. ■

The use of the natural numbers as a device for counting down to 0 in the convergence rule of Axiom System 14.12 can be relaxed. In fact, any well-founded set suitably expressible in any given arithmetical structure suffices. Also, it is not necessary to require that an execution of α causes the truth of the parameterized $\varphi(n)$ in that rule to decrease exactly by 1; it suffices that the decrease is positive at each iteration.

Example 14.17: Consider the following program for computing v^w for natural numbers v and w.

$(z, x, y) := (1, v, w);$
while $y > 0$ **do**
  if $even(y)$
    then $(x, y) := (x^2, y/2)$
    else $(z, y) := (zx, y - 1)$

We shall prove using Axiom System 14.12 that this program terminates and correctly computes v^w in z. Specifically, we show

$$\begin{aligned}\mathbb{N} \quad \vdash_{S4} \quad & (z = 1 \land x = v \land y = w) \\ & \to \ <(((y > 0 \land even(y))?;\ x := x^2;\ y := y/2) \\ & \qquad \cup\ (odd(y)?;\ z := z \cdot x;\ y := y - 1))^*>\ (y = 0 \land z = v^w).\end{aligned}$$

Consider the formula above as $\varphi \to <(\alpha \cup \beta)^*>\psi$. We construct a first-order formula $\chi(n)$, for which we show

(i) $\mathbb{N} \vdash_{S4} \varphi \to \exists n\ \chi(n)$
(ii) $\mathbb{N} \vdash_{S4} \chi(0) \to \psi$
(iii) $\mathbb{N} \vdash_{S4} \chi(n+1) \to <\alpha \cup \beta>\chi(n)$.

Application of the convergence rule to (iii) and further manipulation yields the result.

Let

$$\chi(n) \quad \stackrel{\text{def}}{=} \quad zx^y = v^w \ \land \ n = \lfloor \log_2 y \rfloor + 1bin(y).$$

Here $1bin(y)$ is the function yielding the number of 1's in the binary representation of y. Clearly, $1bin(y)$, $even(y)$, $odd(y)$ and $\lfloor \log_2 y \rfloor$ are all computable, hence they

are first-order definable in $\mathbb{N}$. We consider their appearance in $\chi(n)$ as abbreviations. Also, consider $y := y/2$ as an abbreviation for the obvious equivalent program over $\mathbb{N}$, which need be defined only for even y.

To prove (i) and (iii), all we need to show is that the formulas therein are $\mathbb{N}$-valid, since they are first-order and will thus be axioms. For example, $\chi(0) \rightarrow \psi$ is

$$(zx^y = v^w \ \wedge \ 0 = \lfloor \log_2 y \rfloor + 1bin(y)) \quad \rightarrow \quad (y = 0 \ \wedge \ z = v^w),$$

which is clearly $\mathbb{N}$-valid, since $1bin(y) = 0$ implies $y = 0$, which in turn implies $zx^y = z$.

To prove (iii), we show

$$\mathbb{N} \quad \vdash_{S4} \quad (\chi(n+1) \wedge y > 0 \wedge even(y)) \ \rightarrow \ \texttt{<}\alpha\texttt{>}\chi(n) \tag{14.2.4}$$

and

$$\mathbb{N} \quad \vdash_{S4} \quad (\chi(n+1) \wedge odd(y)) \ \rightarrow \ \texttt{<}\beta\texttt{>}\chi(n). \tag{14.2.5}$$

PDL and first-order reasoning will then yield the desired (iii). Indeed, (14.2.4) is obtained by applying the assignment axiom and the PDL axiom for tests to the following formula:

$$\begin{aligned}&(zx^y = v^w \ \wedge \ n+1 = \lfloor \log_2 y \rfloor + 1bin(y) \ \wedge \ y > 0 \ \wedge \ even(y)) \\ &\quad \rightarrow \quad (y > 0 \ \wedge \ even(y) \ \wedge \ z = (x^2)^{y/2} = v^w \ \wedge \ n = \lfloor \log_2 (y/2) \rfloor + 1bin(y/2)).\end{aligned}$$

This formula is $\mathbb{N}$-valid (and hence an axiom), since for any even y, $1bin(y) = 1bin(y/2)$ and $\lfloor \log_2(y) \rfloor = 1 + \lfloor \log_2(y/2) \rfloor$.

Similarly, (14.2.5) is obtained from the formula:

$$\begin{aligned}&(zx^y = v^w \ \wedge \ n+1 = \lfloor \log_2 y \rfloor + 1bin(y) \ \wedge \ odd(y)) \\ &\quad \rightarrow (odd(y) \ \wedge \ zxx^{y-1} = v^w \ \wedge \ n = \lfloor \log_2 (y-1) \rfloor + 1bin(y-1)).\end{aligned}$$

This formula is also $\mathbb{N}$-valid, since for odd y, $1bin(y) = 1 + 1bin(y-1)$ and $\lfloor \log_2 y \rfloor = \lfloor \log_2 (y-1) \rfloor$.

Note that the proof would have been easier if the truth of $\chi(n)$ were allowed to "decrease" by more than 1 each time around the loop. In such a case, and with a more liberal rule of convergence (see Exercise 14.8), we would not have had to be so pedantic about finding the exact quantity that decreases by 1. In fact, we could have taken $\chi(n)$ to be simply $zx^y = v^w \wedge n = y$. The example was chosen in its present form to illustrate the fact (which follows from the completeness result) that in principle the strict convergence rule can always be used.

In closing, we note that appropriately restricted versions of all axiom systems of this chapter are complete for DL(dreg). In particular, as pointed out in Section 5.7, the Hoare **while**-rule

$$\frac{\varphi \wedge \xi \to [\alpha]\varphi}{\varphi \to [\textbf{while } \xi \textbf{ do } \alpha](\varphi \wedge \neg\xi)}$$

results from combining the generalization rule with the induction and test axioms of PDL, when * is restricted to appear only in the context of a **while** statement; that is, only in the form $(\xi?; p)^*; (\neg\xi)?$.

14.3 Bibliographical Notes

Completeness for termination assertions (Theorem 14.4) is from Meyer and Halpern (1982). Infinitary completeness for DL (Theorem 14.7) is based upon a similar result for Algorithmic Logic (see Section 16.1) by Mirkowska (1971). The proof sketch presented here is an adaptation of Henkin's proof for $L_{\omega_1\omega}$ appearing in Keisler (1971).

The notion of relative completeness and Theorem 14.10 are due to Cook (1978). The notion of arithmetical completeness and Theorems 14.15 and 14.16 are from Harel (1979).

The use of invariants to prove partial correctness and of well-founded sets to prove termination are due to Floyd (1967). An excellent survey of such methods and the corresponding completeness results appears in Apt (1981).

Some contrasting negative results are contained in Clarke (1979), Lipton (1977), and Wand (1978).

Exercises

14.1. Prove Proposition 14.1.

14.2. Prove Lemma 14.3.

14.3. Complete the proof of Theorem 14.4.

14.4. Show that every finite structure is expressive for the regular programs of DL.

14.5. Complete the proof of Theorem 14.10.

14.6. Phrase and prove Theorems 14.15 and 14.16 for general arithmetical structures.

14.7. Justify the special form of formulas that is used in the proof of Theorem 14.16.

14.8. Formulate a more liberal rule of convergence as in the discussion following Theorem 14.16. Prove that if the convergence rule of Axiom System 14.12 is replaced with the new one, the resulting system is arithmetically complete.

14.9. Extend Axiom Systems 14.6 and 14.12 to handle array assignments, and prove the infinitary and arithmetical completeness, respectively, of the resulting systems.

15 Expressive Power

The subject of study in this chapter is the relative expressive power of languages. We will be primarily interested in comparing, on the uninterpreted level, the expressive power of various versions of DL. That is, for programming languages P_1 and P_2 we will study whether $\mathsf{DL}(P_1) \leq \mathsf{DL}(P_2)$ holds. Recall from Chapter 12 (Section 12.1) that the latter relation means that for each formula φ in $\mathsf{DL}(P_1)$, there is a formula ψ in $\mathsf{DL}(P_2)$ such that $\mathfrak{A}, u \vDash \varphi \leftrightarrow \psi$ for all structures $\mathfrak{A}$ and initial states u.

Before describing the contents of this chapter, we pause to make two comments. The first is that by studying the expressive power of logics, rather than the computational power of programs, we are able to compare, for example, deterministic and nondeterministic programming languages. More on this will appear in Section 15.2. The second comment is that the answer to the fundamental question "$\mathsf{DL}(P_1) \leq \mathsf{DL}(P_2)$?" may depend crucially on the vocabulary over which we consider logics and programs. Indeed, as we will see later, the answer may change from "yes" to "no" as we move from one vocabulary to another. For this reason we always make clear in the theorems of this chapter our assumptions on the vocabulary.

Section 15.1 introduces the very useful concept of the unwinding of a program. Some basic properties of this notion are proved there. Section 15.2 establishes the fundamental connection between spectra of formulas (i.e. codes of finite interpretations in which a given formula holds) and the relative expressive power of logics of programs. This section also makes some connections with computational complexity theory.

Section 15.3 studies the important question of the role nondeterminism plays in the expressive power of logic. We discuss separately the case of regular programs (Section 15.3) and regular programs with a Boolean stack (Section 15.3). The more powerful programs are discussed in Section 15.3.

In Section 15.4 we study the question of the impact on the expressive power of bounded vs. unbounded memory. We discuss separately the cases of a polyadic vocabulary (Section 15.4) and a monadic vocabulary (Section 15.4).

The power of a Boolean stack vs. an algebraic stack and vs. pure regular programs is discussed in Section 15.5. Finally, in Section 15.6 we discuss some of the aspects of adding wildcard assignment to other programming constructs.

For now, we adopt a very liberal notion of a program. Let Σ be a finite vocabulary. All we assume about the programming language is that for every program α we have a set $CS(\alpha)$ of seqs that describe the semantics of α in all structures of the same signature. Hence, for every Σ-structure $\mathfrak{A}$ we have a binary

input/output relation $\mathfrak{m}_\mathfrak{A}(\alpha) \subseteq S^\mathfrak{A} \times S^\mathfrak{A}$ defined by the equation

$$\mathfrak{m}_\mathfrak{A}(\alpha) = \bigcup\{\mathfrak{m}_\mathfrak{A}(\sigma) \mid \sigma \in CS(\alpha)\}.$$

We assume that with each program $\alpha \in K$ there is associated a finite set of individual variables $FV(\alpha) \subseteq V$ that occur in α. The property that we need is that for all $u, v \in S^\mathfrak{A}$, if $(u, v) \in \mathfrak{m}_\mathfrak{A}(\alpha)$ then $u(x) = v(x)$ for all $x \in V - FV(\alpha)$; that is, α does not change the values of individual variables that are not in $FV(\alpha)$.

15.1 The Unwind Property

We present a powerful technique that can be sometimes used to establish that one logic is strictly less expressive than another. This technique is based on the notion of the *unwind property*. We say that α *unwinds* in a structure $\mathfrak{A}$ if there exists $m \in \mathbb{N}$ and seqs $\sigma_1, \ldots, \sigma_m \in CS(\alpha)$ such that

$$\mathfrak{m}_\mathfrak{A}(\alpha) = \mathfrak{m}_\mathfrak{A}(\sigma_1) \cup \cdots \cup \mathfrak{m}_\mathfrak{A}(\sigma_m).$$

The next result says that the unwind property is invariant under elementary equivalence of structures.

Proposition 15.1: The unwind property is invariant under elementary equivalence of structures. That is, for every program α and for all structures $\mathfrak{A}$ and $\mathfrak{B}$ that are elementarily equivalent,

$$\mathfrak{m}_\mathfrak{A}(\alpha) = \mathfrak{m}_\mathfrak{A}(\sigma_1) \cup \cdots \cup \mathfrak{m}_\mathfrak{A}(\sigma_m) \implies \mathfrak{m}_\mathfrak{B}(\alpha) = \mathfrak{m}_\mathfrak{B}(\sigma_1) \cup \cdots \cup \mathfrak{m}_\mathfrak{B}(\sigma_m),$$

where $\sigma_1, \ldots, \sigma_m \in CS(\alpha)$.

Proof Assume that α unwinds in $\mathfrak{A}$; that is, there are $m \in \mathbb{N}$ and $\sigma_1, \ldots, \sigma_m \in CS(\alpha)$ such that

$$\mathfrak{m}_\mathfrak{A}(\alpha) = \mathfrak{m}_\mathfrak{A}(\sigma_1) \cup \cdots \cup \mathfrak{m}_\mathfrak{A}(\sigma_m). \quad (15.1.1)$$

For each $i \in \mathbb{N}$, let φ_i be a first-order formula describing the input-output relation of σ_i; that is, if $x_1, \ldots, x_n$ are all registers of α and $y_1, \ldots, y_n$ are new variables, then

$$\models \varphi_i \leftrightarrow \langle\sigma_i\rangle(x_1 = y_1 \wedge \cdots \wedge x_n = y_n).$$

By Lemma 14.3, we know that there exists such a formula.

It follows from (15.1.1) that for all $i \in \mathbb{N}$, the formula

$$\forall x_1 \ldots \forall x_n\ \forall y_1 \ldots \forall y_n\ (\varphi_i \rightarrow (\varphi_1 \vee \cdots \vee \varphi_n))$$

holds in $\mathfrak{A}$. Thus, it holds in $\mathfrak{B}$ as well, therefore

$$\mathfrak{m}_{\mathfrak{B}}(\alpha) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{B}}(\sigma_1) \cup \cdots \cup \mathfrak{m}_{\mathfrak{B}}(\sigma_m).$$

Since the opposite inclusion always holds, it follows that α unwinds in $\mathfrak{B}$. ■

Lemma 15.2: If φ is a DL formula over a programming language P and $\mathfrak{A}$ is a structure such that all programs that occur in φ unwind in $\mathfrak{A}$, then there is a first-order formula $\overline{\varphi}$ such that

$$\mathbf{Th}\,\mathfrak{A} \quad \models \quad \varphi \ \leftrightarrow \ \overline{\varphi}.$$

Proof The proof is by induction on φ. The only non-trivial step is when φ is $[\alpha]\varphi'$. If the program α unwinds in $\mathfrak{A}$, then for some $m \in \mathbb{N}$ and for some $\sigma_1, \ldots, \sigma_m \in CS(\alpha)$, the programs α and $\sigma_1 \cup \cdots \cup \sigma_m$ are equivalent in $\mathfrak{A}$, and by Proposition 15.1 they are equivalent in all models of $Th(\mathfrak{A})$. By Lemma 14.3, there is a first-order formula ψ_α that in all models describes the input-output relation of $\sigma_1 \cup \cdots \cup \sigma_m$; that is,

$$\models \ \psi_\alpha \quad \leftrightarrow \quad <\sigma_1 \cup \cdots \cup \sigma_m>(x_1 = y_1 \wedge \cdots \wedge x_n = y_n),$$

where $x_1, \ldots, x_n$ are all the registers of α and $y_1, \ldots, y_n$ are fresh variables. By the inductive hypothesis, there is a first-order formula $\overline{\varphi}'$ such that

$$\mathbf{Th}\,\mathfrak{A} \quad \models \quad \varphi' \ \leftrightarrow \ \overline{\varphi}'.$$

Assuming that $y_1, \ldots, y_n$ do not occur free in φ', we have

$$\mathbf{Th}\,\mathfrak{A} \quad \models \quad [\alpha]\varphi' \ \leftrightarrow \ \forall y_1 \ldots \forall y_n\ (\psi_\alpha \ \rightarrow \ \overline{\varphi'}[x_1/y_1, \ldots, x_n/y_n]),$$

which completes the proof. ■

Lemma 15.2 gives a useful method for showing that some programs do not unwind. We illustrate it with the program Next_0 of Proposition 13.7.

Proposition 15.3: If $\mathfrak{A}$ is an infinite structure without proper substructures, then Next_0 does not unwind in $\mathfrak{A}$.

Proof Observe that the formula

$$\forall x_0 \; \forall x_1 \; \langle \textsc{Next}_0 \rangle \mathbf{1}$$

holds in a structure $\mathfrak{A}$ iff $\mathfrak{A}$ has no proper substructures. Now, take an infinite structure $\mathfrak{A}$ without proper substructures. If $\textsc{Next}_0$ unwinds in $\mathfrak{A}$, then by Lemma 15.2 there is a first-order formula φ such that

$$\mathbf{Th}\,\mathfrak{A} \quad \models \quad \varphi \;\leftrightarrow\; \forall x_0 \forall x_1 \; \langle \textsc{Next}_0 \rangle \mathbf{1}.$$

This contradicts the upward Löwenheim–Skolem theorem (Theorem 3.59) since $\mathbf{Th}\,\mathfrak{A}$ contains uncountable models. ■

The following result shows that the unwind property can be used to separate the expressive power of logics of programs.

THEOREM 15.4: Let P_1 and P_2 be two programming languages over the same first-order vocabulary, and assume that there is a program $\alpha \in P_1$ such that for an arbitrary finite set $\{\beta_1, \ldots, \beta_m\} \subseteq P_2$ there exists a structure $\mathfrak{A}$ with the property that all the $\beta_1, \ldots, \beta_m$ unwind in $\mathfrak{A}$ but α does not. Then $\mathsf{DL}(P_2)$ is not reducible to $\mathsf{DL}(P_1)$.

Proof Let $CS(\alpha) = \{\sigma_i \mid i \geq 0\}$. Let $FV(\alpha) = \{x_1, \ldots, x_n\}$ be all the input registers of α. Let $y_1, \ldots, y_n$ be new variables. We prove that the formula

$$\psi \quad = \quad \langle \alpha \rangle (x_1 = y_1 \wedge \cdots \wedge x_n = y_n)$$

is equivalent to no formula of $\mathsf{DL}(P_2)$. Indeed, assume that ψ is equivalent to a formula φ of $\mathsf{DL}(P_2)$. Let $\beta_1, \ldots, \beta_m$ be all the programs occurring in φ, and take a structure $\mathfrak{A}$ in which each β_i unwinds and α does not. The latter property means that the set

$$\{\psi\} \quad \cup \quad \{\neg \langle \sigma_0 \cup \cdots \cup \sigma_k \rangle (x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \mid k \geq 0\}$$

is finitely satisfiable in $\mathfrak{A}$.

For $k \geq 0$, let ψ_k be a first-order formula that is equivalent to

$$\neg \langle \sigma_0 \cup \cdots \cup \sigma_k \rangle (x_1 = y_1 \wedge \cdots \wedge x_n = y_n)$$

in all models (see Lemma 14.3).

By Lemma 15.2, there is a first-order formula $\overline{\varphi}$ that is equivalent to φ over all

structures elementarily equivalent to $\mathfrak{A}$; that is,

$$\mathbf{Th}\,\mathfrak{A} \models \varphi \leftrightarrow \overline{\varphi}.$$

Since φ is equivalent to ψ, it follows that the set

$$\mathbf{Th}\,\mathfrak{A} \cup \{\overline{\varphi}\} \cup \{\psi_k \mid k \geq 0\}$$

is finitely satisfiable. By the compactness property for predicate logic (Theorem 3.57), it has a model $\mathfrak{B}$. This model is such that φ holds but ψ does not. This contradiction completes the proof. ■

15.2 Spectra and Expressive Power

The goal of the present section is to relate the question of comparing the expressive power of Dynamic Logic over various programming languages to the complexity of spectra of the corresponding programming languages. As we will see later, for sufficiently powerful programming languages, the only way to distinguish between the corresponding logics is by investigating the behavior of the programs over finite interpretations.

An important notion often studied in the area of comparative schematology is that of translatability of one programming language into another. Let K_1 and K_2 be programming languages. We say that a program $\beta \in K_2$ *simulates* a program $\alpha \in K_1$ if for every Σ-structure $\mathfrak{A}$ the following holds:

$$\begin{aligned}&\{(u,\, v{\upharpoonright}FV(\alpha)) \mid (u,v) \in \mathfrak{m}_{\mathfrak{A}}(\alpha),\ u \text{ is initial}\}\\ &\quad = \{(u,\, v{\upharpoonright}FV(\alpha)) \mid (u,v) \in \mathfrak{m}_{\mathfrak{A}}(\beta),\ u \text{ is initial}\}.\end{aligned}$$

The reason for restricting v in the above formula to $FV(\alpha)$ is to allow β to use auxiliary variables and perhaps some other data types. We say that a program $\alpha \in K_1$ is *translatable into* K_2, if there exists $\beta \in K_2$ that simulates α. Finally, K_1 is translatable into K_2, denoted $K_1 \leq K_2$, if every program of K_1 is translatable into K_2.

A programming language K is said to be *admissible* if:

(i) it is translatable into the class of r.e. programs;

(ii) all atomic regular programs and all tests are translatable into K;

(iii) K is semantically closed under composition, **if-then-else** and **while-do**; e.g., closure under composition means that if $\alpha, \beta \in K$, then there is $\gamma \in K$ such that for every $\mathfrak{A}$, $\mathfrak{m}_{\mathfrak{A}}(\gamma) = \mathfrak{m}_{\mathfrak{A}}(\alpha) \circ \mathfrak{m}_{\mathfrak{A}}(\beta)$, and similarly for the other constructs.

Thus, if K is admissible, we will treat it as being syntactically closed under the above constructs. This will allow us to write expressions like **if** φ **then** α **else** β, where φ is a quantifier free formula and $\alpha, \beta \in K$. Such expressions, even though they do not necessarily belong to K, are semantically equivalent to programs in K. This convention will simplify notation and should never lead to confusion.

The relation of translatability can be further weakened if all we care about is the expressive power of the logic. For example, as we will see later, there are programming languages K_1 and K_2 such that $\mathsf{DL}(K_1) \leq \mathsf{DL}(K_2)$ holds, even though K_1 contains nondeterministic programs and K_2 contains only deterministic programs, so that $K_1 \leq K_2$ is impossible. It follows from the next result that all that really matters for the expressive power of the relevant logic are the termination properties of programs in the programming language.

Proposition 15.5: Let K be admissible. For every formula φ of $\mathsf{DL}(K)$ there is a formula φ' of $\mathsf{DL}(K)$ that is equivalent in all interpretations to φ and such that for every program α that occurs in φ', if α occurs in the context $[\alpha]\psi$, then $\psi = \mathbf{0}$.

Proof If α occurs in φ in the context $[\alpha]\psi$ with $\psi \neq \mathbf{0}$, then we replace $[\alpha]\psi$ with

$$\forall y_1 \ldots \forall y_m\ (\neg[\alpha; (x_1 = y_1 \wedge \cdots \wedge x_m = y_m)?]\mathbf{0} \ \rightarrow\ \psi[x_1/y_1, \ldots, x_m/y_m]),$$

where $x_1, \ldots, x_m$ are all variables that occur freely in α and $y_1, \ldots, y_m$ are fresh variables that occur neither in α nor in ψ. Since K is admissible, it follows that $\alpha; (x_1 = y_1 \wedge \cdots \wedge x_m = y_m)?$ belongs to K. After a finite number of steps we transform φ into the desired formula φ'. ■

The above comments motivate the following definition. K_2 is said to *termination-subsume* K_1, denoted $K_1 \preceq_T K_2$, if for every $\alpha \in K_1$ there is $\beta \in K_2$ such that for every Σ-structure $\mathfrak{A}$ and for every state $u \in S^{\mathfrak{A}}$, we have

$$\mathfrak{A}, u \vDash \langle\alpha\rangle\mathbf{1} \quad \Longleftrightarrow \quad \mathfrak{A}, u \vDash \langle\beta\rangle\mathbf{1}.$$

Notice that the above is equivalent to

$$\mathfrak{A}, u \vDash [\alpha]\mathbf{0} \quad \Longleftrightarrow \quad \mathfrak{A}, u \vDash [\beta]\mathbf{0}.$$

Proposition 15.6: Let K_1 and K_2 be admissible programming languages.

(i) If $K_1 \leq K_2$, then $K_1 \preceq_T K_2$.
(ii) If $K_1 \preceq_T K_2$, then $\mathsf{DL}(K_1) \leq \mathsf{DL}(K_2)$.

Proof The first part is immediate. The second follows immediately from Proposition 15.5. ■

An admissible programming language K is said to be *semi-universal* if for every $m > 0$, the program Next_m of Proposition 13.7 is translatable into K.

Examples of semi-universal programming languages include r.e. programs, regular programs with an algebraic stack, and regular programs with arrays. A corollary of the following result is that the expressive power of DL over a semi-universal programming language can be determined by investigating finite interpretations only.

Recall that a state u is Herbrand-like (see the beginning of Section 13.2) if the values assigned by u to the individual variables (there are finitely many of them) generate the structure.

PROPOSITION 15.7: If K is semi-universal, then for every r.e. program α there is $\beta \in K$ such that α and β have the same termination properties over all infinite interpretations; that is, for every infinite Σ-structure $\mathfrak{A}$ and for every Herbrand-like state u in $\mathfrak{A}$,

$$\mathfrak{A}, u \vDash <\alpha>\mathbf{1} \iff \mathfrak{A}, u \vDash <\beta>\mathbf{1}.$$

Proof sketch. We sketch the proof, leaving the details to the reader. Let α be an arbitrary r.e. program and let $FV(\alpha) \subseteq \{x_0, \ldots, x_m\}$. Clearly, the termination of α in any interpretation depends only on the values of variables in $FV(\alpha)$ and on the substructure generated by these values. Thus, we can assume that the state u in the conclusion of the proposition is an Herbrand-like m-state. Let us start by observing that using Next_m and working in an infinite interpretation gives us *counters*, with a successor function that corresponds to the particular order in which all elements of the substructure generated by the input occur in the natural chain. Zero is represented here as the first element of the natural chain; testing for zero and testing the equality of counters can be done easily. The control structure of a deterministic regular program with these counters is strong enough to compute every partial recursive function.

Now, we can use counters to simulate the Turing machine that computes $CS(\alpha) = \{\sigma_n \mid n \in \mathbb{N}\}$. The regular program β that will simulate α searches through all seqs σ_n starting with σ_0, trying to find the first one that terminates. It halts as soon as it finds one such σ_n.

In order to simulate the computation of σ_n, β has to be able to compute the value of any term t with variables in $\{x_0, \ldots, x_m\}$. This can be done as follows. Given t, the program β computes the value of t with respect to the actual values

stored in $x_0, \ldots, x_m$ by first computing the values for subterms of t of depth 1, then of depth 2, etc. Of course, in order to do this, β has to store the intermediate values. For this we use the power of counters and the program $\textsc{Next}_m$. Using counters, β can encode arbitrary finite sequences of natural numbers. Using $\textsc{Next}_m$ gives β a natural encoding of all the elements of the substructure generated by the input.

Now, it should be clear that being able to compute the value of any term with variables in $\{x_0, \ldots, x_m\}$, the program β can perform the computation of every σ_n. Since K is admissible, it follows that the program described above is equivalent to a program in K. ■

An admissible programming language K is *divergence-closed* if for every $\alpha \in K$ there exists $\beta \in K$ and two variables $x, y \in V$ such that for every finite Herbrand-like interpretation $(\mathfrak{A}, u)$ with A having at least two elements,

$$\begin{aligned} \mathfrak{A}, u \models \langle\alpha\rangle\mathbf{1} &\iff \mathfrak{A}, u \models \langle\beta\rangle(x = y), \\ \mathfrak{A}, u \models [\alpha]\mathbf{0} &\iff \mathfrak{A}, u \models \langle\beta\rangle(x \neq y). \end{aligned}$$

Informally, β decides without diverging whether α possibly terminates.

Lemma 15.8: If K is divergence-closed, then for every $\alpha \in K$ there exists $\gamma \in K$ such that for every finite Herbrand-like interpretation $(\mathfrak{A}, u)$ with A having at least two elements, we have both

$$\begin{aligned} \mathfrak{A}, u \models \langle\alpha\rangle\mathbf{1} &\iff \mathfrak{A}, u \models [\gamma]\mathbf{0} \\ \mathfrak{A}, u \models [\alpha]\mathbf{0} &\iff \mathfrak{A}, u \models \langle\gamma\rangle\mathbf{1}. \end{aligned}$$

Proof Take as γ the program $\beta; (x \neq y)?$, where β is a program that corresponds to α by the definition of K being divergence-closed. Since K is admissible, it follows that γ (semantically) belongs to K. ■

We now list some languages that are semi-universal and divergence-closed. In some cases this depends on the vocabulary Σ.

Proposition 15.9: The following programming languages are semi-universal and divergence-closed:

(i) For every Σ containing at least one function symbol of arity at least two, or at least two unary function symbols:

- (deterministic/nondeterministic) regular programs with algebraic stack;

MIT Press Math7X9/2010/08/25:15:15 Page 351

- (deterministic/nondeterministic) regular programs with arrays.

(ii) For every mono-unary Σ:

- deterministic regular programs;
- deterministic regular programs with a Boolean stack.

Proof sketch. First, we sketch the proof of (i). In the proof of Theorem 13.12 there is a sketch of a mutual simulation between Cook's $\log n$-APDA's and deterministic regular programs with an algebraic stack. It follows from the proof of Cook's theorem (see Chapter 14 of Hopcroft and Ullman (1979)) that we can assume without loss of generality that deterministic $\log n$-APDA's halt for every input. Since the simulation of a deterministic $\log n$-APDA by a deterministic regular program α with algebraic stack is step by step, it follows that α can find out in a finite number of steps whether the $\log n$-APDA accepts or rejects the input. Then α halts, assigning the same value to the special variables x and y if the input is accepted, and assigning two different values to x and y otherwise. The same remarks hold for nondeterministic regular programs with an algebraic stack.

The same argument applies to regular programs with arrays. Here the mutual simulation is with polynomial-space bounded Turing machines, and without loss of generality we may assume that these Turing machines halt for every input. This proves (i).

For part (ii), we use an argument similar to the one used above, except that now we work with log-space bounded Turing machines (see Theorem 13.11). This proves the result for deterministic regular programs. The second part of (ii) follows immediately from (i) and from the fact that over a mono-unary vocabulary regular programs with a Boolean stack are computationally equivalent to regular programs with an algebraic stack (Exercise 15.10). ■

It is not known whether the class of all regular programs is divergence-closed for vocabularies richer than mono-unary.

It turns out that for semi-universal and divergence-closed programming languages, the DL theory of finite interpretations reduces to termination properties.

PROPOSITION 15.10: If K is semi-universal and divergence-closed, then for every formula φ of $\mathsf{DL}(K)$ there exists a program $\alpha_\varphi \in K$ such that for every finite Σ-structure $\mathfrak{A}$, for every $m \geq 0$, and for every Herbrand-like m-state w in $\mathfrak{A}$, we have

$$\mathfrak{A}, w \models \varphi \leftrightarrow \langle\alpha_\varphi\rangle\mathbf{1}.$$

Proof Let us fix $m \geq 0$. We first prove the conclusion of the proposition by induction on φ, assuming that A has at least two elements. For the case when φ is of the form $\varphi_1 \to \varphi_2$, we use the divergence tests for the programs obtained by the induction hypothesis. For the case when φ is of the form $\forall z\ \varphi_1$, in addition to using the divergence test for the program corresponding to φ_1, we have to use $\textsc{Next}_m$ to search A; that is, the structure generated by the input. Finally, for the case when φ is of the form $[\alpha]\psi$, we find by the inductive hypothesis β_ψ such $\langle\beta_\psi\rangle\mathbf{1}$ and ψ are equivalent over all finite Herbrand-like interpretations. For β_ψ, we find γ_ψ such that $\langle\beta_\psi\rangle\mathbf{1}$ and $[\gamma]_\psi\mathbf{0}$ are equivalent over all finite Herbrand-like interpretations with at least two elements (we apply Lemma 15.8 here). Thus, it follows that φ and $[\alpha;\gamma_\psi]\mathbf{0}$ are equivalent over all finite Herbrand-like interpretations with at least two elements. Applying Lemma 15.8 again to the program $\alpha;\gamma_\psi$ yields the desired α_φ.

In order to extend the result to one element structures, we have to perform a test to see whether the structure is indeed of one element only. For this, denote by ψ the conjunction of the following formulas:

- $x_i = x_j$, for $0 \leq i, j \leq m$,
- $f(x_0, \ldots, x_0) = x_0$, where f ranges over all function symbols of Σ.

The next observation we need for the case of one element structures is that there are at most 2^k different isomorphism types of such structures, where k is the number of relation symbols in the vocabulary. Each such structure is uniquely determined by a conjunction of formulas of the form $r(x_0, \ldots, x_0)$ or $\neg r(x_0, \ldots, x_0)$, where r ranges over all relation symbols of Σ.

Now, given φ, let $\gamma_1, \ldots, \gamma_n$ be all the formulas that describe the one element structures in which φ holds. Let α' be the program found for φ in the first part of the proof; that is, the one that works correctly in structures containing at least two different elements. The program α we are looking for is:

if ψ
  then if $\gamma_1 \vee \cdots \vee \gamma_n$
    then skip
    else fail
  else α'

This completes the proof. ■

Observe that the above proof does not give an effective construction of the program α from φ. The reason is that in general there is no effective procedure to

determine whether a given formula of $\mathsf{DL}(K)$ holds in a one-element structure. For example, for an r.e. program α, it is undecidable whether α terminates in a given one-element interpretation.

We are now ready to present the main result of this section. It relates complexity classes and spectra to the expressive power of Dynamic Logic. This result proves to be a strong tool for establishing relative expressive power of several logics of programs. We will use it in a number of places in the present chapter.

THEOREM 15.11 (SPECTRAL THEOREM): Let Σ be a rich vocabulary. Let K_1 and K_2 be programming languages over Σ 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 downward closed under logarithmic space reductions. Let $SP(K_i) \approx C_i$ for $i = 1, 2$. The following statements are equivalent:

(i) $\mathsf{DL}(K_1) \le \mathsf{DL}(K_2)$;

(ii) $SP_m(K_1) \subseteq SP_m(K_2)$ for all $m \ge 0$;

(iii) $C_1 \subseteq C_2$;

(iv) $K_1 \preceq_T K_2$.

Proof For the implication (i) $\Longrightarrow$ (ii), consider any $m \ge 0$ and any $\alpha \in K_1$. It follows from (i) that there exists a formula φ of $\mathsf{DL}(K_2)$ such that $<\alpha>\mathbf{1}$ and φ are equivalent in all interpretations. By Proposition 15.10, there is a $\beta \in K_2$ such that

$$\mathfrak{A}, w \quad \vDash \quad <\beta>\mathbf{1} \ \leftrightarrow \ \varphi$$

holds for every finite Σ-stucture $\mathfrak{A}$ and every Herbrand-like m-state w. Thus $SP(\alpha) = SP(\beta)$, which proves (ii).

Now for (ii) $\Longrightarrow$ (iii). Consider any $X \in C_1$. By Lemma 13.10, there is a language $Y \subseteq H_0^L$ such that

$$X \quad \le_{\log} \quad Y \quad \le_{\log} \quad X. \tag{15.2.1}$$

Hence $Y \in C_1$, and since $SP(K_1)$ captures C_1, it follows that there exists $\alpha \in K_1$ such that $Y = SP_0(\alpha)$. By (ii), there is $\beta \in K_2$ such that $SP_0(\alpha) = SP_0(\beta)$, therefore $Y \in C_2$. Since C_2 is downward closed under log-space reductions, it follows from (15.2.1) that $X \in C_2$. This proves (iii).

For the proof of (iii) $\Longrightarrow$ (iv), consider any $\alpha \in K_1$. We describe a program $\beta \in K_2$ such that for all Σ-structures $\mathfrak{A}$ and states w, we have

$$\mathfrak{A}, w \vDash <\alpha>\mathbf{1} \quad \Longleftrightarrow \quad \mathfrak{A}, w \vDash <\beta>\mathbf{1}.$$

Let $FV(\alpha) \subseteq \{x_0, \ldots, x_m\}$ and let $\gamma \in K_2$ be such that $SP_m(\alpha) = SP_m(\gamma)$. Since K_1 is admissible, it follows that there is an r.e. program α' that is equivalent to α in all interpretations. Let $\beta' \in K_2$ be the program of Proposition 15.7, which has the same termination properties as α' in all infinite interpretations.

In the first phase of the simulation of α by β, the latter runs β' to find out whether α', and therefore α, has terminated. The simulation is performed under the assumption that the substructure $\mathfrak{A}'$ of $\mathfrak{A}$ generated by $\{w(x_0), \ldots, w(x_m)\}$ is infinite. Either the simulation succeeds with α terminating, in which case β terminates too, or else β' discovers that $\mathfrak{A}'$ is finite. The finiteness of $\mathfrak{A}'$ is discovered by finding out that the value of x_{m+1} returned by $\textsc{Next}_m$ equals the previous value of x_{m+1}. Having discovered this, β aborts the simulation and runs the program γ on the restored initial valuation of $x_0, \ldots, x_m$. If γ uses any variable x_n with $n > m$, then prior to running γ, β resets its value by the assignment $x_n := x_m$. Since $\mathfrak{A}'$ is finite, γ terminates iff α terminates. This proves $K_1 \preceq_T K_2$.

The implication (iv) $\Longrightarrow$ (i) is just Proposition 15.6. ∎

We conclude this section with an example of how the Spectral Theorem can be applied. We will see more applications of this theorem later in the book.

Theorem 15.12: Let Σ be a rich vocabulary. Then

(i) DL(stk) $\leq$ DL(array).

(ii) DL(stk) $\equiv$ DL(array) iff $P = \mathit{PSPACE}$.

Moreover, the same holds for deterministic regular programs with an algebraic stack and deterministic regular programs with arrays.

Proof The result follows immediately from Theorem 15.11, Proposition 15.9, Theorem 13.12, and Theorem 13.13. ∎

A similar result can be proved for poor vocabularies. The complexity classes change, though. This is treated in the exercises (Exercise 15.12).

We remark that part (i) of Theorem 15.12 can be proved directly by showing that (deterministic) regular programs with an algebraic stack are translatable into (deterministic) regular programs with arrays.

15.3 Bounded Nondeterminism

In this section we investigate the role that nondeterminism plays in the expressive power of logics of programs. As we shall see, the general conclusion is that for a programming language of sufficient computational power, nondeterminism does not increase the expressive power of the logic.

Regular Programs

We start our discussion of the role of nondeterminism with the basic case of regular programs. Recall that DL and DDL denote the logics of nondeterministic and deterministic regular programs, respectively.

For the purpose of this subsection, fix the vocabulary to consist of two unary function symbols f and g. Any given nonempty prefix-closed subset $A \subseteq \{0,1\}^*$ determines a structure $\mathfrak{A} = (A, f^{\mathfrak{A}}, g^{\mathfrak{A}})$, where

$$f^{\mathfrak{A}}(w) \quad = \quad \begin{cases} w \cdot 0, & \text{if } w \cdot 0 \in A \\ w, & \text{otherwise.} \end{cases}$$

In the above definition, $w \cdot 0$ denotes the result of concatenating 0 at the right end of word w. The definition of $g^{\mathfrak{A}}$ is similar with 1 replacing 0. Such structures are called *treelike structures*.

Throughout this subsection, we will be referring to the algebra $\mathfrak{A}$ by indicating its carrier A. This will not lead to confusion. In particular, we will be interested in the algebras $T_n = \{w \in \{0,1\}^* \mid |w| \leq n\}$ for $n \in \mathbb{N}$. The main combinatorial part of our proof demonstrates that the number of elements of T_n that a deterministic regular program can visit is at most polynomial in n. Thus, for sufficiently large n, there will be elements in T_n that are not visited during a computation starting from the root of T_n. This bound depends on the program—the larger the program, the larger n will be.

On the other hand, the following simple nondeterministic regular program visits all the elements of any T_n:

$\mathbf{while}\ x \neq y?\ \mathbf{do}\ (x := f(x) \cup x := g(x)).$

Thus, the formula

$$\varphi \quad = \quad \exists x\ \forall y\ \langle \mathbf{while}\ x \neq y?\ \mathbf{do}\ (x := f(x) \cup x := g(x))\rangle \mathbf{1} \tag{15.3.1}$$

states that there is an element from which every element of the domain is reachable by a finite number of applications of the operations f and g. It can be shown that this formula is equivalent to no formula of DDL.

For technical reasons, we represent **while** programs here as consisting of *labeled* statements. Thus, deterministic **while** programs contain the following three kinds of statements:

- $\ell: \ x_i := \xi(x_j)$, where $\xi(x_j)$ is either x_j, $f(x_j)$, or $g(x_j)$;
- $\ell:$ **halt**;
- $\ell:$ **if** $x_i = x_j$ **then** ℓ' **else** ℓ''.

The computational behavior of a program α in a structure $A \subseteq \{0,1\}^*$ is represented by a sequence of states $\pi = (\ell_1, a^1), \ldots, (\ell_i, a^i), \ldots$, where ℓ_i is a label of the statement to be executed at the ith step and a^i is the vector of current values stored in the registers of α. To represent a computation of α, π must satisfy the following properties:

- (ℓ_1, a^1) is the initial state; that is, ℓ_1 is the label of the first statement to be executed by α, and a^1 represents the input.
- To move from (ℓ_i, a^i) to (ℓ_{i+1}, a^{i+1}), the statement labeled ℓ_i is executed, which determines the next statement ℓ_{i+1}, and a^{i+1} is the vector of the new values after executing ℓ_i. If ℓ_i is a label of **halt**, then there is no next state.

By an *L-trace* of a computation π we mean the sequence $Ltr(\pi) = \ell_1, \ldots, \ell_n, \ldots$ of labels of the consecutive statements of π.

Let $Cmp(\alpha, A)$ denote the set of all computations of α in A. Call a computation π *terminating* if it is finite and the last pair of π contains the **halt** statement. Since we are dealing with deterministic programs, every nonterminating finite computation can be uniquely extended to a longer computation. The *length* of a computation is the number of pairs in it. Let $LtrCmp(\alpha, A, n)$ denote the set of all L-traces of computations of α in A whose length is at most n.

Let $L = \ell_1, \ell_2, \ldots$ be a sequence of labels. We define a *formal computation of α along L* as a sequence $t^0, t^1, \ldots$ of k-tuples of terms, where k is the number of registers of α. This sequence represents a history of values that are stored in registers, assuming that the computation followed the sequence L of labels. The values are terms. They depend on the input, which is represented by variables[1]

1 We do not make a clear distinction between registers of a program and variables. We usually think of registers as part of the computer on which the program is being executed, while variables are part of a formal language (usually they appear in terms) that is used to describe properties of a computation.

$x_1, \ldots, x_k$. Let $1 \le i \le k$ and $0 \le m < |L|$. We define t_i^m by induction on m:

$$t_i^0 \stackrel{\text{def}}{=} x_i$$
$$t_i^{m+1} \stackrel{\text{def}}{=} \begin{cases} \xi(t_j^m), & \text{if } \ell_m \text{ is a label of } x_i := \xi(x_j) \\ t_i^m, & \text{otherwise.} \end{cases}$$

In the above formula, we use the abbreviation $\xi(x)$ to denote one of $x, f(x)$ or $g(x)$.

Take any sequence $L = \ell_1, \ell_2, \ldots$ of labels and a formal computation $t^0, t^1, \ldots$ of α along L. For registers x_i and x_j of α, we say that x_i and x_j *witness a left turn* at the mth step of L and write $W_L(i,j) = m$ if $m > 0$ is the smallest number such that ℓ_{m-1} is a label of a statement **if** $x_p = x_q$ **then** ℓ_m **else** ℓ', the element t_p^m contains the variable x_i, and t_q^m contains the variable x_j (or conversely). If there is no such m, then we say that x_i and x_j do not witness a left turn, and in that case we let $W_L(i,j) = 0$.

The general form of a term is $\xi_1 \cdots \xi_m(x)$, where each ξ_i is either f or g. Taking into account the interpretation of function symbols in $\mathfrak{A}$, we can represent such a term by the word $xw_m \cdots w_1$, where $w_i \in \{0,1\}^*$ is 0 if ξ_i is f and to 1 if ξ_i is g. This representation of a term supports the intuition that, given a value for x as a word $u \in A$, the result of evaluating this term is obtained from u by traveling along the path $w = w_m \cdots w_1$. Of course, we apply here our convention that we follow the path w as long as we stay within the elements of A, i.e. the "true" result is $uw_n \cdots w_1$, where $w_n \cdots w_1$ is the longest prefix of w such that $uw_n \cdots w_1 \in A$.

Lemma 15.13: Let α be a deterministic **while** program, and let $\pi, \pi' \in$ *Cmp*(α, T_n) be computations with input values a and a', respectively. Let $L =$ *Ltr*(π) and $L' =$ *Ltr*(π') be the L-traces of the corresponding computations. Assume that

(i) $|L| = |L'|$,

(ii) For all $1 \le i, j \le k$, $W_L(i,j) = W_{L'}(i,j)$,

(iii) For all $1 \le i \le k$, $|a_i| = |a'_i|$.

Then $L = L'$.

Proof Let $L = \ell_1, \ell_2, \ldots$ and $L' = \ell'_1, \ell'_2, \ldots$. We prove by induction on $0 < m < |L|$ that $\ell_m = \ell'_m$ for all m.

For $m = 1$, this is obvious, since $\ell_1 = \ell'_1$ is the label of the start statement of α. Let $1 < m < |L|$ and assume that $\ell_r = \ell'_r$ for all $r < m$. Consider the statement labeled $\ell_{m-1} = \ell'_{m-1}$. If this is an assignment statement, then the next statement

is uniquely determined by α, hence $\ell_m = \ell'_m$.

Assume now that ℓ_{m-1} labels **if** $x_p = x_q$ **then** ℓ **else** ℓ' and $\ell_m = \ell$, $\ell'_m = \ell'$, $\ell \neq \ell'$. If there exist $1 \le i, j \le k$ such that $W_L(i,j) = m$, then $W_{L'}(i,j) = m$ and $\ell_m = \ell'_m$. So assume now that

$$W_L(i,j) \neq m, \quad 1 \le i, j \le k. \tag{15.3.2}$$

Consider a formal computation $t^0, t^1, \ldots, t^{m-1}$ of α along $\ell_1, \ldots, \ell_{m-1}$. Let $t_p^{m-1} = x_i w$ and $t_q^{m-1} = x_j w'$, for some $1 \le i, j \le k$, and let $w, w' \in T_n$. Thus, we have

$$T_n \models a_i w = a_j w' \tag{15.3.3}$$

$$T_n \models a'_i w \neq a'_j w'. \tag{15.3.4}$$

Let $m_0 = W_L(i,j)$. It follows from (15.3.3) that $m_0 > 0$, and by (15.3.2) we conclude that $m_0 < m$. It also follows from (15.3.3) that a_i is a prefix of a_j, or conversely, a_j is a prefix of a_i. Without loss of generality we may assume the former. Hence, for some $\xi \in \{0,1\}^*$, we have

$$T_n \models a_j = a_i \xi. \tag{15.3.5}$$

By (15.3.4) and (iii), we have

$$T_n \models a'_j \neq a'_i \xi. \tag{15.3.6}$$

Since at step m_0 both computations run through the "yes"-branch of some **if-then-else** statement, it follows that for some $u, u' \in \{0,1\}^*$ we have

$$T_n \models a_i = a_j u' \quad \text{and} \quad T_n \models a'_i u = a'_j u'. \tag{15.3.7}$$

Again, by (iii) and (15.3.7) it follows that there is a common $\xi' \in \{0,1\}^*$ such that

$$T_n \models a_j = a_i \xi' \quad \text{and} \quad a'_j = a'_i \xi'.$$

Thus, by (15.3.5) we have $\xi = \xi'$, which yields a contradiction with (15.3.6). This completes the proof. ■

LEMMA 15.14: Let α be a deterministic **while** program with k registers. Then for all $n, p \in \mathbb{N}$ we have

$$\#LtrCmp(\alpha, T_n, p) \quad \le \quad n^k p^{k^2}.$$

Proof It follows from Lemma 15.13 that an L-trace L of a given length $r \le p$ is

uniquely determined by the left-turn-witness function W_L and the length of the input data. The number of possible functions W_L is $r^{k^2} \leq p^{k^2}$, and the number of possible lengths of values for k input variables in T_n is n^k. Thus the total number of all L-traces of length at most p is no greater than $n^k \cdot p^{k^2}$. ■

Lemma 15.14 fails for nondeterministic programs. It holds for programs that are more powerful than **while** programs, though they still have to be deterministic.

For every $1 \leq i \leq k$, we define a function $G_i : \mathbb{N} \to \mathbb{N}$ as follows. For $n \in \mathbb{N}$, $G_i(n)$ is the maximum number $m \in \mathbb{N}$ such that there is a computation $\pi \in \mathit{Cmp}(\alpha, T_n)$ and an i-element set $B \subseteq T_n$ such that for m consecutive steps of π (not necessarily starting at the beginning of the computation), some registers of α store all the elements of B. Moreover, we require that no state in π repeats.

Observe now that the number of states of the program α is at most $2^{c \cdot n^k}$, where $c > 0$ depends on $|\alpha|$. Thus

$$G_i(n) \quad \leq \quad 2^{cn^k}$$

holds for all $1 \leq i \leq k$. We show that the G_i can in fact be bounded by a polynomial in n. Clearly, $G_k(n) \leq |\alpha|$ for all $n \in \mathbb{N}$.

LEMMA 15.15: For every $1 \leq i < k$ and $n \geq 1$,

$$G_i(n) \quad \leq \quad (n+1)G_{i+1}(n) + |\alpha|^{k+1} n^{k^3+k^2}.$$

Proof Take any $1 \leq i < k$ and $n \geq 1$. Let $B \subseteq T_n$ be an i-element set. Let $\pi \in \mathit{Cmp}(\alpha, T_n)$ be a computation without repeating states. Moreover, assume that starting from step $p \geq 1$, the values from B occur in every state after the pth state.

For any $q \geq 0$, let $V(B, q)$ be the set of values obtainable from B within q steps of π. The precise definition is by induction on q:

- $V(B, 0) = B$,
- $w \in V(B, q+1)$ iff either $w \in V(B, q)$ or there exist $r > p$, registers x_{j_1}, x_{j_2} of α, and a value $u \in V(B, q)$ such that $w = u \cdot 0$ or $w = u \cdot 1$, u occurs in the rth step of π in register x_{j_1}, and the rth statement of π is $x_{j_2} := f(x_{j_1})$ or $x_{j_2} := g(x_{j_1})$, depending on whether $w = u \cdot 0$ or $w = u \cdot 1$.

Take any state (ℓ, a) that occurs in π at position $q > p$. Let $m \leq n$, and assume that $q + (m+1)G_{i+1}(n) < |\pi|$. Let (ℓ', b) be a state that occurs in π at position

$q + (m+1)G_{i+1}(n)$. We prove the following property:

$$\text{For all } 1 \leq j \leq k,\ (|b_j| = m \Longrightarrow b_j \in V(B,m)). \tag{15.3.8}$$

The proof is by induction on $0 \leq m \leq n$. Let $m = 0$, and assume that $|b_j| = 0$. Since there is no way to set a register to value ε other than by assigning to it the contents of another register containing ε, it follows that ε must have been stored in registers throughout π. If $\varepsilon \notin B$, then $B \cup \{\varepsilon\}$ has been stored in states of π from the pth step on. There are more than $G_{i+1}(n)$ steps in π after the pth (since $p + G_{i+1}(n) < q + G_{i+1}(n) < |\pi|$), and we obtain a contradiction. Hence, $\varepsilon \in B$ and $b_j \in V(B,0)$.

For the induction step, let $0 < r \leq n$, and assume that (15.3.8) holds for all $m < r$. Assume that $q + (r+1)G_{i+1}(n) < |\pi|$ and let (ℓ', b) be a state that occurs in π in position $q + (r+1)G_{i+1}(n)$. Let $1 \leq j \leq k$ be such that $|b_j| = r$. If $b_j \notin B$, then b_j must have been created sometime after step $q + rG_{i+1}(n)$. Thus, there is a state (ℓ'', b') at a position later than $q + rG_{i+1}(n)$ such that the value b_j was obtained in a certain register x from a certain b'_{j_1} via an assignment of the form $x := f(x_{j_1})$ or $x := g(x_{j_1})$. Thus $|b'_{j_1}| = r - 1$. By the inductive hypothesis, we have $b'_{j_1} \in V(B, r-1)$, therefore $b_j \in V(B,r)$ as required. This proves (15.3.8).

It follows from (15.3.8) that all values occurring in π after step $p+(n+1)G_{i+1}(n)$ belong to $V(B,n)$. Thus, after $p + (n+1)G_{i+1}(n) + |\alpha| \cdot \#V(B,n)^k$ steps of π, at least one state must repeat. Therefore,

$$G_i(n) \quad \leq \quad (n+1)G_{i+1}(n) + |\alpha| \cdot \#V(B,n)^k.$$

By Lemma 15.14, we have that the number of possible L-traces of fragments of computations of α of length at most n is no greater than $|\alpha| n^k n^{k^2}$. Thus

$$\#V(B,n) \quad \leq \quad |\alpha| n^{k^2+k}.$$

From this we obtain

$$G_i(n) \quad \leq \quad (n+1)G_{i+1}(n) + |\alpha|^{k+1} n^{k^3+k^2},$$

which completes the proof. ■

Let $\mathbf{Moves}(\alpha, T_n)$ be the set of all words $w \in \{0,1\}^*$ such that there is a terminating computation $\pi \in \mathit{Cmp}(\alpha, T_n)$ and a variable x such that xw occurs in the formal computation along $\mathit{Ltr}(\pi)$. Thus, $\mathbf{Moves}(\alpha, T_n)$ is the set of all possible moves that α can perform on one of its inputs in a terminating computation. It turns out that this set is polynomially bounded.

PROPOSITION 15.16: For every deterministic **while**-program α there is a constant $c > 0$ such that

$$\#\mathbf{Moves}(\alpha, T_n) \quad \leq \quad (|\alpha|n)^{ck^5}.$$

Proof It follows from Lemma 15.15 that $G_0(n)$, the maximum number of steps α makes in T_n before terminating or repeating a state, is at most

$$k \cdot |\alpha|^{k+1}(n+1)^k n^{k^3+k^2} \quad \leq \quad (|\alpha|n)^{c'k^3}$$

for some $c' > 0$, which depends on α. Thus, by Lemma 15.14, the number of different L-traces of terminating computations in T_n is at most $(|\alpha|n)^{c''k^5}$ for some $c'' > 0$. Since an L-trace L of length p brings at most kp terms in the formal computation along L, it follows that

$$\#\mathbf{Moves}(\alpha, T_n) \quad \leq \quad k(|\alpha|n)^{c'k^3}(|\alpha|n)^{c''k^5} \quad \leq \quad (|\alpha|n)^{ck^5}$$

for a suitable $c > 0$. This completes the proof. ■

For a word $w \in \{0,1\}^*$, let

$$T^*(w) \quad \stackrel{\text{def}}{=} \quad \{w^n u \mid n \in \mathbb{N},\ u \in \{0,1\}^*,\ \text{and } |u| \leq |w|\}.$$

This set can be viewed as an infinite sequence of the trees $T_{|w|}$ connected along the path w.

PROPOSITION 15.17: Let α be a deterministic **while** program with k registers, and let $w \in \{0,1\}^*$ be a word of length $n \geq 2k$. If $w \notin \mathbf{Moves}(\alpha, T_n)$, then α unwinds in $T^*(w)$.

Proof Let α have registers $x_1, \ldots, x_k$, and choose n with $n \geq 2k$. We shall describe a deterministic **while** program β whose computation in T_n for a specially chosen input will simulate the computation of α in $T^*(w)$ for every w with $|w| = n$. In fact, β will not depend on w; the correctness of the simulation will follow from the choice of a suitable input for β. If we view $T^*(w)$ as consisting of an infinite number of copies of T_n, each connected along w to the next copy, then β will be doing the same in one block of T_n as α does in $T^*(w)$. Obviously, β has to remember when values stored in the registers of α enter the same block. The assumption that $w \notin \mathbf{Moves}(\alpha, T_n)$ implies that no value of α can be moved all the way along w.

The program β has k registers $x_1, \ldots, x_k$ that will hold the values of the registers of α truncated to a single T_n. It has two registers b and e, which will be initialized

to the root of T_n and the node w, respectively.[2] In addition, the program β has k registers $z_1, \ldots, z_k$, where z_i stores the name of the block in which α has the value stored in x_i. These names are represented by words of the form 0^m, where $1 \leq m \leq 2k$. The essential information, sufficient to carry out the simulation, is whether two variables store a value from the same block or from adjacent blocks. Two values that are at least one block apart are not accessible from each other.

For each statement in α of the form

$$\ell:\ x_i := \xi x_j, \quad \xi \in \{0, 1, \varepsilon\},$$

the program β will have the corresponding statement

$$\ell:\ x_i := \xi x_j;\ \textbf{if}\ x_i = e\ \textbf{then}\ z_i := 0 \cdot z_i; x_i := b\ \textbf{else}\ z_i := z_j.$$

Each statement of α of the form

$$\ell:\ \textbf{if}\ x_i = x_j\ \textbf{then}\ \ell'\ \textbf{else}\ \ell''$$

is replaced in β by

$$\ell:\ \textbf{if}\ x_i = x_j \wedge z_i = z_j\ \textbf{then}\ \ell'\ \textbf{else}\ \ell''.$$

Let us now take any $w \in \{0, 1\}^*$ with $|w| = n$. Every value $a \in T^*(w)$ can be uniquely represented as $a = w^m u$, where $m \geq 0$, $|u| \leq n$, and $u \neq w$. Given an initial valuation v for α in $T^*(w)$, where $v(x_i) = w^{m_i} u_i$ (with $|u_i| \leq n$ and $u_i \neq w$), we define an initial valuation $\overline{v}$ for β in T_n as follows:

$$\begin{aligned} \overline{v}(x_i) &= u_i \\ \overline{v}(b) &= \varepsilon \\ \overline{v}(e) &= w \\ \overline{v}(z_i) &= 0^p, \end{aligned}$$

where p in the above definition is the position of m_i in the set

$$\{m_j \mid j = 1, \cdots, k\} \quad \cup \quad \{m_j + 1 \mid j = 1, \cdots, k\},$$

counting from the smallest to the largest element starting from 1. The above enumeration of blocks takes into account whether two values are in the same block or in adjacent blocks or whether they are at least one full block apart. Now, if $w \notin \mathbf{Moves}(\alpha, T_n)$, then α terminates in $T^*(w)$ for the initial evaluation v iff β terminates in T_n for the corresponding evaluation $\overline{v}$. (An easy proof of this is left

2 We do not fix the word w at this stage—it will be introduced via a suitable valuation.

to the reader.) Morever, the simulation of α by β is faithful, in the sense that for every step of α there are at most 4 steps of β after which the above described correspondence $[v \mapsto \overline{v}]$ between valuations is maintained. Thus, α terminates in $T^*(w)$ for an input v iff it terminates in at most $|\beta| \cdot n^{2\cdot k+2}$ steps. Hence α unwinds in $T^*(w)$. ■

PROPOSITION 15.18: For every finite set $\{\alpha_1, \ldots, \alpha_p\}$ of deterministic **while** programs over the vocabulary containing two unary function symbols, there is a word $w \in \{0,1\}$ such that each α_i unwinds in $T^*(w)$.

Proof Take n sufficiently large that

$$\{0,1\}^n - \bigcup_{i=1}^{p} \mathbf{Moves}(\alpha_i, T_n) \quad \neq \quad \varnothing.$$

By Proposition 15.16, there is such an n. Then by Proposition 15.17, each α_i unwinds in $T^*(w)$, where $w \in \{0,1\}^n - \bigcup_{i=1}^{p} \mathbf{Moves}(\alpha_i, T_n)$. ■

Observe that the infinite structure $T^*(w)$ is constructed separately for each finite set $\{\alpha_1, \ldots, \alpha_p\}$ of deterministic **while** programs. That is, we do not construct one structure in which all deterministic **while** programs unwind. Still, it is enough for our purposes in this section. In fact, a stronger result can be shown.

THEOREM 15.19: There exists an infinite treelike structure $\mathfrak{A}$ in which all deterministic **while** programs unwind.

We do not prove this result here, since the proof is quite complicated and technical. The main idea is to build an infinite treelike structure $\mathfrak{A}$ as a limit of a sequence of finite treelike structures. This sequence is constructed inductively in such a way that if a deterministic **while** program tries to follow one of the very few infinite paths in $\mathfrak{A}$, then it must exhibit a periodic behavior. The interested reader is referred to Urzyczyn (1983b) for details.

We can now state the main result that separates the expressive power of deterministic and nondeterministic **while** programs.

THEOREM 15.20: For every vocabulary containing at least two unary function symbols or at least one function symbol of arity greater than one, DDL is strictly less expressive than DL; that is, DDL $<$ DL.

Proof For the vocabulary containing two unary function symbols, the theorem is an immediate corollary of Proposition 15.18 and Theorem 15.4. The case of a vocabulary containing a function symbol of arity greater than one is easily reducible to the former case. We leave the details as an exercise (Exercise 15.3). ■

It turns out that Theorem 15.20 cannot be extended to vocabularies containing just one unary function symbol without solving a well known open problem in complexity theory.

THEOREM 15.21: For every rich mono-unary vocabulary, the statement "DDL is strictly less expressive than DL" is equivalent to $LOGSPACE \neq NLOGSPACE$.

Proof This follows immediately from the Spectral Theorem (Theorem 15.11), Proposition 15.9 (ii), and Theorem 13.11. ■

Boolean Stacks

We now turn our attention to the discussion of the role non-determinism plays in the expressive power of regular programs with a Boolean stack. We will show that for a vocabulary containing at least two unary function symbols, nondeterminism increases the expressive power of DL over regular programs with a Boolean stack.

There are two known approaches to proving this result. These are similar in terms of the methods they use—they both construct an infinite treelike algebra in which deterministic regular programs with a Boolean stack unwind. This property is achieved by exhibiting a periodic behavior of deterministic regular programs with a Boolean stack. We sketch the main steps of both approaches, leaving out the technical details that prove the periodicity.

For the rest of this section, we let the vocabulary contain two unary function symbols. The main result of the section is the following.

THEOREM 15.22: For a vocabulary containing at least two unary function symbols or a function symbol of arity greater than two, $\mathsf{DL}(\text{dbstk}) < \mathsf{DL}(\text{bstk})$.

For the purpose of this section, we augment the deterministic **while** programs of Section 15.3 with instructions to manipulate the Boolean stack. Thus, a computation of a program α with a Boolean stack is a sequence of the form

$$(\ell_1, a^1, \sigma_1), \ldots, (\ell_i, a^i, \sigma_i), \ldots$$

where ℓ_i is a label of the statement to be executed at the ith step, a^i is a vector of

current values stored in the registers of α prior to the ith step and $\sigma_i \in \{0,1\}^*$ is the contents of the Boolean stack prior to the ith step. We do not assume here that ℓ_1 is the label of the first instruction of α, nor that σ_1 is empty.

If for every $n \geq 0$ the number of **push** statements is greater than or equal to the number of **pop** statements during the first n steps

$$(\ell_1, a^1, \sigma_1), \ldots, (\ell_n, a^n, \sigma_n),$$

then such a computation will be called *legal*.

Let $\mathfrak{A}$ be a Σ-structure, let $r > 0$, and let α be a deterministic **while** program with a Boolean stack. A computation

$$(\ell_1, a^1, \sigma_1), \ldots, (\ell_i, a^i, \sigma_i), \ldots$$

of α in $\mathfrak{A}$ is said to be *strongly r-periodic* if there is $n < r$ such that for all $i \in \mathbb{N}$,

$$\ell_{n+i} = \ell_{n+r+i} \quad \text{and} \quad a^{n+i} = a^{n+r+i}.$$

A program α is said to be *uniformly periodic* in $\mathfrak{A}$ if for every $\sigma \in \{0,1\}^*$ there exists $r > 0$ such that for every label ℓ and every vector a of values, the computation that starts from (ℓ, a, σ) is strongly r-periodic.

Let $m \geq 2$. A computation

$$(\ell_1, a^1, \sigma_1), \ldots, (\ell_i, a^i, \sigma_i), \ldots$$

is said to be *upward periodic for m-periods* if there exist $r > 0$ and $n < r$ such that for all $0 \leq i < (m-1)r$,

$$\ell_{n+i} \quad = \quad \ell_{n+r+i},$$

and moreover, the computation

$$(\ell_n, a^n, \sigma_n), \ldots, (\ell_{n+r-1}, a^{n+r-1}, \sigma_{n+r-1})$$

is legal. Hence, the sequence of labels repeats for m times, and each of the m cycles is legal, i.e. it never inspects the contents of the Boolean stack with which the cycle started.

Adian Structures

Adian structures arise from the well known solution to the Burnside problem in group theory. In 1979, S. I. Adian proved that for every odd $n \geq 665$ there exists an infinite group G_n generated by two elements satisfying the identity $x^n = 1$, where 1 is the unit of the group (Adian (1979)).

Every such group G_n induces in a natural way a Σ-algebra $\mathfrak{G}_n = <G_n, f, g>$, where f and g are unary functions defined by

$$f(x) = ax, \qquad g(x) = bx,$$

where a and b are the generators of G_n.

Since in G_n we have $a^{-1} = a^{n-1}$ and $b^{-1} = b^{n-1}$, it follows that every term over G_n can be represented by a string in $\{a, b\}^*$, assuming that the unit 1 is represented by the empty string ε. Thus G_n induces an equivalence relation on $\{0, 1\}^*$: for $u, w \in \{0, 1\}^*$, $u \equiv w$ iff the terms obtained from u and w by replacing 0 with a and 1 with b are equal in G_n. The quotient $\{0, 1\}^*/\equiv$ can be viewed as an infinite directed graph in which every node is of out-degree 2. This graph is not a treelike structure, since it contains loops of length greater than 1. It might also be possible that for paths $u, w \in \{0, 1\}^*$ we have $0u \equiv 1w$.

Cyclicity of G_n implies periodic behavior of deterministic **while** programs with a Boolean stack. The reader interested in the details of the proof of the following result is referred to Stolboushkin (1983).

THEOREM 15.23: For every odd $n \geq 665$, any deterministic **while** program with a Boolean stack is uniformly periodic in $\mathfrak{G}_n$.

It follows immediately from Theorem 15.23 that every deterministic **while** program with a Boolean stack unwinds in $\mathfrak{G}_n$. On the other hand, the ordinary non-deterministic regular program

$$x := \varepsilon\,;\, x := g(x)^*$$

does not unwind in $\mathfrak{G}_n$. Hence, Theorem 15.22 follows immediately from Theorem 15.4.

Traps

The method of trapping programs from a given class $\mathcal{K}$ of programs consists of building a treelike structure $\mathfrak{A}$ satisfying the following two properties:

- Programs from $\mathcal{K}$, when computing in $\mathfrak{A}$, exhibit some form of limited periodic behavior.
- The structure $\mathfrak{A}$ contains only one infinite path, and this path has the property that there are very few repetitions of subwords on that path.

As a result of these two properties, no computation in $\mathfrak{A}$ can stay for a sufficiently long time on that infinite path. This yields the unwind property in $\mathfrak{A}$ of programs from $\mathcal{K}$. Of course, in this section we are only interested in deterministic regular programs with a Boolean stack.

Let $m \geq 2$, and let $\mathcal{T}$ be a class of treelike structures. We say that a program α *exhibits* m *repetitions* in $\mathcal{T}$ if there exists $n \in \mathbb{N}$ such that for every $\mathfrak{A} \in \mathcal{T}$, each legal fragment of length n of any computation of α in $\mathfrak{A}$ is upward periodic for m periods. We stress that the bound n is uniform for all structures in $\mathcal{T}$.

Let $\mathfrak{A}$ be a treelike structure and let $n \geq 0$. We say that level n is *incomplete* in $\mathfrak{A}$ if there is $w \in A$ such that $|w| = n$ and either $w0 \notin A$ or $w1 \notin A$. Otherwise, level n in $\mathfrak{A}$ is said to be *complete*.

The treelike structure $\mathfrak{A}$ is called *p-sparse* if every two incomplete levels in $\mathfrak{A}$ are separated by at least p complete levels.

The following theorem is the main tool used in establishing limited periodic behavior of deterministic **while** programs with a Boolean stack when the programs are run over certain treelike structures.

THEOREM 15.24: For every deterministic **while** program α with a Boolean stack and for every $m \geq 2$, there exists $p \in \mathbb{N}$ such that α exhibits m repetitions over the class of all p-sparse structures.

We are now ready to build a trap.

THEOREM 15.25: For every finite set $\{\alpha_1, \ldots, \alpha_n\}$ of deterministic **while** programs with a Boolean stack, there is an infinite treelike structure $\mathfrak{A}$ such that every α_i unwinds in $\mathfrak{A}$.

Proof Let W be an infinite cube-free string; that is, no finite non-empty string of the form uuu is a substring of W. It is known that such strings exist (see Salomaa (1981)). Let k be an upper bound on the number of registers used by each α_i. It is not hard to prove that there exists a number $r \in \mathbb{N}$ such that for every function $f : \{1, \ldots, k\} \to \{1, \ldots, k\}$, the rth power f^r of f is idempotent; that is, $f^r f^r = f^r$. We fix such an r and let $m = 4r$. We now apply Theorem 15.24 to m and to each α_i. Let $p_i \in \mathbb{N}$ be such that α_i exhibits m repetitions over the class of p_i-sparse structures. Clearly, we can choose a common p by taking the largest p_i.

We now cut W into pieces, each of length p; that is, $W = w_1 w_2 \cdots$, where

$|w_i| = p$ for all $i \geq 1$. Our trap structure is defined as follows.

$$A \stackrel{\text{def}}{=} \{u \in \{0,1\}^* \mid \exists j \geq 0\ \exists u' \in \{0,1\}^*\ u = w_1 w_2 \cdots w_j u' \text{ and } |u'| < p\}.$$

The set A can be viewed as sequence of blocks of full binary trees of depth p connected along the infinite path W. Since $\mathfrak{A}$ is p-sparse, it follows that every α_i exhibits m repetitions in $\mathfrak{A}$. Let $q \geq 0$ be such that every legal fragment of length q of any computation of α_i in $\mathfrak{A}$ is upward periodic for m-periods. Take any computation of α_i that starts with an empty stack and any initial valuation in $\mathfrak{A}$. Assume that the computation is of length at least q, and consider the first q steps in this computation. Thus, this fragment is upward periodic for m periods.

Consider the first period. After it has been completed, the value of any register, say x_j, depends on the value of a certain register $x_{j'}$ at the entering point of this period. That is, upon completing the first period, x_j is equal to $\xi x_{j'}$ for a certain $\xi \in \{0,1\}^*$. This gives rise to a function $f : \{1,\ldots,k\} \to \{1,\ldots,k\}$ whose value at j is $f(j) = j'$. Thus, after r periods, the contents of register x_j depends on the value stored in register $x_{f^r(j)}$ at the beginning of the first period. By the same argument, it follows that after $2r$ periods, the contents of x_j depends on the value stored in register $x_{f^r(j)}$ at the beginning of the $(r+1)^{\text{st}}$ period. The latter value depends on the contents stored in register $x_{f^r f^r(j)} = x_{f^r(j)}$ at the beginning of the first period. It follows that after $4r$ periods, the value stored in x_j is obtained from the value stored in $x_{f^r(j)}$ at the beginning of the first period by applying a term of the form $\xi_1 \xi_2 \xi_2 \xi_2$.

We have shown that after $4r$ periods, all values stored in registers of every α_i are outside the path W. Therefore, the computation cannot proceed to the next block, which implies that every program α_i unwinds in $\mathfrak{A}$. ■

We now derive Theorem 15.22 from Theorem 15.25 in exactly the same way as in the case of Adian structures and Theorem 15.23.

Algebraic Stacks and Beyond

It turns out that for programming languages that use sufficiently strong data types, nondeterminism does not increase the expressive power of Dynamic Logic.

Theorem 15.26: For every vocabulary,

(i) $\mathsf{DL}(\text{dstk}) \equiv \mathsf{DL}(\text{stk})$.
(ii) $\mathsf{DL}(\text{darray}) \equiv \mathsf{DL}(\text{array})$.

Proof Both parts follow immediately from the Spectral Theorem (Theorem 15.11), Proposition 15.9, and either Theorem 13.12 for case (i) or Theorem 13.13 for case (ii). ■

It can be shown that even though r.e. programs are not divergence closed, nondeterminism does not increase the expressive power. We leave this as an exercise (see Exercise 15.2).

15.4 Unbounded Memory

In this section we show that allowing unbounded memory increases the expressive power of the corresponding logic. However, this result depends on assumptions about the vocabulary Σ.

Recall from Section 11.2 that an r.e. program α has *bounded memory* if the set $CS(\alpha)$ contains only finitely many distinct variables from V, and if in addition the nesting of function symbols in terms that occur in seqs of $CS(\alpha)$ is bounded. This restriction implies that such a program can be simulated in all interpretations by a device that uses a fixed finite number of registers, say $x_1, \ldots, x_n$, and all its elementary steps consist of either performing a test of the form

$$r(x_{i_1}, \ldots, x_{i_m})?,$$

where r is an m-ary relation symbol of Σ, or executing a simple assignment of either of the following two forms:

$$x_i := f(x_{i_1}, \ldots, x_{i_k}) \qquad\qquad x_i := x_j.$$

In general, however, such a device may need a very powerful control (that of a Turing machine) to decide which elementary step to take next.

An example of a programming language with bounded memory is the class of regular programs with a Boolean stack. Indeed, the Boolean stack strengthens the control structure of a regular program without introducing extra registers for storing algebraic elements. We leave it as an exercise (Exercise 15.5) to prove that regular programs with a Boolean stack have bounded memory. On the other hand, regular programs with an algebraic stack or with arrays are programming languages with unbounded memory.

It turns out that the results on expressive power depend on assumptions on the vocabulary. Recall that a vocabulary Σ is *polyadic* if it contains a function symbol of arity greater than one. Vocabularies whose function symbols are all unary are called *monadic*. We begin our discussion with polyadic vocabularies and then move to the more difficult case of monadic ones.

Polyadic Vocabulary

We need some technical machinery for the proof of the main result of this section. We first discuss *pebble games* on dags, then exhibit a dag that is hard to pebble. The technical results will be used in the proof of Proposition 15.29.

A Pebble Game on Dags

Let $\mathcal{D} = (D, \rightarrow_{\mathcal{D}})$ be a dag, and let $n \geq 1$. We describe a game on $\mathcal{D}$ involving n pebbles. The game starts with some of the pebbles, perhaps all of them, placed on vertices of $\mathcal{D}$, at most one pebble on each vertex. A *move* consists of either removing pebbles from the graph or placing a free pebble on some vertex d. The latter move is allowed only if all direct predecessors of d (vertices c such that $c \rightarrow_{\mathcal{D}} d$) are pebbled, i.e., are occupied by pebbles. We also allow a pebble to be moved from a predecessor of d directly to d, provided all predecessors of d are pebbled.

The rules of the n-pebble game can be expressed more precisely by introducing the notion of an *n-configuration* and the relation of *succession* on the set of n-configurations. An *n-configuration* C is any subset of D of cardinality at most n. For n-configurations C and C', we say that C' *n-succeeds* C if either of the following two conditions hold:

(i) $C' \subseteq C$; or
(ii) for some d, $C' - C = \{d\}$ and $\{c \in D \mid c \rightarrow_{\mathcal{D}} d\} \subseteq C$.

A sequence of n-configurations $C_0, C_1, \ldots, C_m$ is called an *n-pebble game* if C_{i+1} n-succeeds C_i for $0 \leq i \leq m-1$.

The following lemma is useful for transforming an n-pebble game into an $(n-1)$-pebble game. It will be applied to a special dag constructed in the next section.

LEMMA 15.27: Let $\mathcal{D} = (D, \rightarrow_{\mathcal{D}})$ be a dag, and let $a \in D$. Define

$$A \stackrel{\text{def}}{=} \{d \mid a \rightarrow^*_{\mathcal{D}} d\},$$

where $\rightarrow^*_{\mathcal{D}}$ is the reflexive transitive closure of $\rightarrow_{\mathcal{D}}$. Let $C_0, \ldots, C_m$ be an n-pebble game in $\mathcal{D}$, $n \geq 2$. Suppose that for every $0 \leq i \leq m$, $A \cap C_i \neq \varnothing$. Then there is an $(n-1)$-pebble game $B_0, \ldots, B_m$ such that

$$\bigcup_{i=0}^{m} C_i \quad \subseteq \quad A \cup \bigcup_{i=0}^{m} B_i. \qquad (15.4.1)$$

Proof For each $0 \leq i \leq m$, let $B_i = C_i - A$. Surely, (15.4.1) holds. Since A and

C_i intersect, B_i is an $(n-1)$-configuration. It remains to show that B_{i+1} $(n-1)$-succeeds B_i for $0 \le i \le m-1$. In case (i), we have $C_{i+1} \subseteq C_i$, so that $B_{i+1} \subseteq B_i$ as well. In case (ii), we have $C_{i+1} - C_i = \{d\}$. Either $d \in A$, in which case $B_{i+1} \subseteq B_i$; or $d \notin A$, in which case $B_{i+1} - B_i = \{d\}$. However, if $d \notin A$, then no predecessor of d is in A, and since $\{c \mid c \to_{\mathcal{D}} d\} \subseteq C_i$, we must have $\{c \mid c \to_{\mathcal{D}} d\} \subseteq B_i$ too. ■

A Hard Dag

In this section we describe a dag that cannot be pebbled with finitely many pebbles. Let

$$\mathcal{A} \stackrel{\text{def}}{=} (\mathbb{N}, \to_{\mathcal{A}})$$

be a dag defined as follows.

$$\to_{\mathcal{A}} \stackrel{\text{def}}{=} \{(n, n+1) \mid n \in \mathbb{N}\} \cup \{(n, 2n+1) \mid n \in \mathbb{N}\} \cup \{(n, 2n+2) \mid n \in \mathbb{N}\}.$$

The dag $\mathcal{A}$ can be viewed as a union of the chain of successive natural numbers and the infinite binary tree that has 0 as its root and for each n has $2n+1$ as the left child and $2n+2$ as the right child. A parent of the node n is $\lfloor (n-1)/2 \rfloor$ (we will call it a *tree-parent* of n).

Observe that

$$n \to_{\mathcal{A}}^{*} m \iff n \le m.$$

Let $C \subseteq \mathbb{N}$ and $k \in \mathbb{N}$. We define the *k-neighborhood* of C, denoted $N(C, k)$, by

$$N(C, k) \stackrel{\text{def}}{=} \{j \in \mathbb{N} \mid (\exists i \in C \cup \{0\})\ i \le j \le i + k\}.$$

Finally, define a function $f : N \to N$ inductively, by

$$\begin{aligned} f(0) &\stackrel{\text{def}}{=} 0, \\ f(n+1) &\stackrel{\text{def}}{=} 4(n+1)(f(n)+1). \end{aligned}$$

The following lemma shows that $\mathcal{A}$ cannot be pebbled with finitely many pebbles.

Lemma 15.28: For every $n \ge 1$ and for every n-configuration C of $\mathcal{A}$, if $C, C_1, \ldots, C_r$ is an n-pebble game in $\mathcal{A}$, then $C_r \subseteq N(C, f(n))$.

Proof We prove the result by induction on n. For $n = 1$, the result is obvious.

For $n > 1$, assume that there is an n-configuration C of $\mathcal{A}$ and an n-pebble game $C, C_1, \ldots, C_r$ in $\mathcal{A}$ such that for some $k \in C_r$, $k \notin N(C, f(n))$. We will find an $(n-1)$-configuration and an $(n-1)$-pebble game that contradict the conclusion of the lemma.

Let $j \in C \cup \{0\}$ be the largest element such that $j < k$. It follows that $f(n) < k - j$. Let $m = \lceil (k-j+1)/2 \rceil + j + 1$, which is roughly the middle of the interval between j and k. Observe that this interval does not contain any node from C. In order to move a pebble from j to k, the n-game $C, C_1, \ldots, C_r$ must have moved at least one pebble through all the intermediate nodes. Let i_0 be the smallest number such that $m \in C_{i_0}$ and each configuration after C_{i_0} contains a node between m and k. In order to move a pebble through all these nodes, we must also move a pebble through the tree-parents of these nodes. Call these tree-parent nodes *red*.

Since the tree-parent of a node $i > 0$ is $\lfloor (i-1)/2 \rfloor$, it follows that all red nodes are smaller than or equal to $\lfloor (k-1)/2 \rfloor$. On the other hand we have

$$\lceil (k-j+1)/2 \rceil + j + 1 \quad \geq \quad \frac{k+j+3}{2} \quad > \quad \lfloor k/2 \rfloor,$$

thus $m > \lfloor k/2 \rfloor$, so every red node is smaller than m. We can now apply Lemma 15.27 to $\mathcal{A}$, the node m, and the n-pebble game $C_{i_0}, \ldots, C_r$. We obtain an $(n-1)$-pebble game $B_1, \ldots, B_p$ such that every red node is in $\bigcup_{i=1}^{p} B_i$. By the inductive hypothesis, we have

$$\# \bigcup_{i=1}^{p} B_i \quad \leq \quad \# N(B_1, f(n-1)) \quad \leq \quad n(f(n-1)+1). \tag{15.4.2}$$

On the other hand, the number of red nodes is half the number of nodes in the interval m through k; that is, it is at least $(k-j)/2$. We thus have

$$\frac{k-j}{2} \quad > \quad \frac{f(n)}{2} \quad = \quad 2n(f(n-1)+1).$$

Thus, the number of red nodes is larger than $n(f(n-1)+1)$, which contradicts (15.4.2). This completes the induction step. ■

The Unwind Property

We first define a structure

$$\mathfrak{A} \quad \stackrel{\text{def}}{=} \quad (\mathbb{N}, g, 0)$$

over the vocabulary that consists of one constant symbol 0 and one binary function symbol $g : \mathbb{N}^2 \to \mathbb{N}$ defined as follows:

$$g(m,n) \stackrel{\text{def}}{=} \begin{cases} n+1, & \text{if } n > 0 \text{ and } m = \lfloor (n-1)/2 \rfloor \\ 0, & \text{otherwise.} \end{cases}$$

We can now prove the main technical result of this section.

PROPOSITION 15.29: Every r.e. program with bounded memory unwinds in $\mathfrak{A}$.

Proof Let α be an r.e. program with bounded memory, and let $CS(\alpha) = \{\sigma_i \mid i \in \mathbb{N}\}$. Let $x_1, \ldots, x_n$ be all the variables that occur in seqs of $CS(\alpha)$. We claim that each seq $\sigma_i \in CS(\alpha)$ can be viewed as a simultaneous assignment

$$(x_1, \ldots, x_n) := (t_{1,i}, \ldots, t_{n,i}),$$

which is performed subject to the satisfiability of a quantifier-free condition (guard) φ_i. In other words, σ_i is equivalent to

$$\varphi_i? \,;\, (x_1, \ldots, x_n) := (t_{1,i}, \ldots, t_{n,i}).$$

This claim can be proved by a routine induction on the number of steps in σ_i, and we leave it to the reader.

From now on, we assume that the seqs in $CS(\alpha)$ are of the above form. Let $T(\alpha)$ be the least set of terms that contains all terms occurring in $CS(\alpha)$ and that is closed under subterms. For every $a_1, \ldots, a_n \in \mathbb{N}$, let

$$T^{\mathfrak{A}}(a_1, \ldots, a_n) \stackrel{\text{def}}{=} \{t^{\mathfrak{A}}(a_1, \ldots, a_n) \mid t \in T(\alpha)\}.$$

Observe that every element in $b \in T^{\mathfrak{A}}(a_1, \ldots, a_n)$ can be computed by simple assignments using only n variables. Hence b can be viewed as being reachable by an n-pebble game from the initial configuration $\{a_1, \ldots, a_n\}$. It follows from Lemma 15.28 that

$$T^{\mathfrak{A}}(a_1, \ldots, a_n) \subseteq N(\{a_1, \ldots, a_n\}, f(n)),$$

thus

$$\#T^{\mathfrak{A}}(a_1, \ldots, a_n) \leq (n+1) \cdot (f(n)+1). \tag{15.4.3}$$

We can conclude that the computation starting from any given input lies in the partial subalgebra of $\mathfrak{A}$ of cardinality $(n+1) \cdot (f(n)+1)$.

Since the number of pairwise non-isomorphic partial subalgebras of $\mathfrak{A}$ of

bounded cardinality is finite, it follows that there exists $m \geq 0$ such that α and $\sigma_1 \cup \cdots \cup \sigma_m$ represent the same input-output relation in $\mathfrak{A}$. To see this, suppose that we have two isomorphic partial subalgebras of $\mathfrak{A}$, say $(\mathfrak{B}_1, a_1, \ldots, a_n)$ and $(\mathfrak{B}_2, b_1, \ldots, b_n)$. Moreover, assume that the computation of α for input $a_1, \ldots, a_n$ lies in $\mathfrak{B}_1$, and similarly that the computation of α for input $b_1, \ldots, b_n$ lies in $\mathfrak{B}_2$. Then

$$\{i \in \mathbb{N} \mid \mathfrak{B}_1 \models \varphi_i(a_1, \ldots, a_n)\} \quad = \quad \{i \in \mathbb{N} \mid \mathfrak{B}_2 \models \varphi_i(b_1, \ldots, b_n)\}.$$

Let I denote this set. It follows from (15.4.3) that the set

$$\{(t_{1,i}^{\mathfrak{A}}(a_1, \ldots, a_n), \ldots, t_{n,i}^{\mathfrak{A}}(a_1, \ldots, a_n)) \mid i \in I\}$$

is finite. Let $m \in \mathbb{N}$ be such that

$$\begin{aligned} &\{(t_{1,i}^{\mathfrak{A}}(a_1, \ldots, a_n), \ldots, t_{n,i}^{\mathfrak{A}}(a_1, \ldots, a_n)) \mid i \in I\} \\ &\quad = \quad \{(t_{1,i}^{\mathfrak{A}}(a_1, \ldots, a_n), \ldots, t_{n,i}^{\mathfrak{A}}(a_1, \ldots, a_n)) \mid i \in I,\ i \leq m\}. \end{aligned}$$

It follows that the number m depends only on the isomorphism class of $(\mathfrak{B}_1, a_1, \ldots, a_n)$, not on the particular choice of this subalgebra. Since there are only finitely many isomorphism classes of bounded cardinality, it suffices to take the largest such m. Then

$$\alpha^{\mathfrak{A}} \quad = \quad \sigma_1^{\mathfrak{A}} \cup \cdots \cup \sigma_m^{\mathfrak{A}},$$

which completes the proof. ■

THEOREM 15.30: For every vocabulary containing at least one function symbol of arity greater than one, no DL over a programming language with bounded memory is reducible to any DL over a programming language that contains a program equivalent to NEXT_0.

Proof For a vocabulary containing a binary function symbol, the result follows immediately from Proposition 15.29, Theorem 15.4, and Proposition 15.3. The case of a vocabulary containing only function symbols of arity greater than two we leave as an exercise (Exercise 15.8). ■

THEOREM 15.31: For every vocabulary containing a function symbol of arity greater than one, DL(dbstk) $<$ DL(dstk) and DL(bstk) $<$ DL(stk).

Proof This is an immediate corollary of Theorem 15.30 and the fact that regular

programs with a Boolean stack have bounded memory (see Exercise 15.5). ■

Monadic Vocabulary

For monadic vocabularies the situation is much less clear. The method of pebbling, which is applicable to polyadic vocabularies, does not work for monadic vocabularies, since every term (viewed as a dag) can be pebbled with a single pebble. For this reason, formally speaking, the issue of unbounded memory in programs over a monadic vocabulary disappears. Nevertheless, it makes sense to compare the expressive power of regular programs with or without a Boolean stack and programs equipped with an algebraic stack.

It is not known whether $\mathsf{DL}(\text{reg}) < \mathsf{DL}(\text{stk})$ holds for monadic vocabularies. For deterministic regular programs, however, we have the following result.

Theorem 15.32: Let the vocabulary be rich and mono-unary. Then

$$\mathsf{DL}(\text{dreg}) \equiv \mathsf{DL}(\text{dstk}) \iff \mathit{LOGSPACE} = P.$$

Proof Since deterministic regular programs over mono-unary vocabularies are semi-universal and divergence closed (see Proposition 15.9), the result follows immediately from Theorem 15.11, Theorem 13.12 and Theorem 13.11. ■

The case of poor vocabularies is treated in Exercise 15.12.

For monadic vocabularies, the class of nondeterministic regular programs with a Boolean stack is computationally equivalent to the class of nondeterministic regular programs with an algebraic stack (see Exercise 15.11). Hence, we have:

Theorem 15.33: For all monadic vocabularies, $\mathsf{DL}(\text{bstk}) \equiv \mathsf{DL}(\text{stk})$.

For deterministic programs, the situation is slightly different.

Theorem 15.34:

(i) For all mono-unary vocabularies, $\mathsf{DL}(\text{dbstk}) \equiv \mathsf{DL}(\text{dstk})$.

(ii) For all monadic vocabularies containing at least two function symbols, $\mathsf{DL}(\text{dbstk}) < \mathsf{DL}(\text{dstk})$.

Proof Part (i) follows from Exercise 15.10. For part (ii), we observe that $\mathsf{DL}(\text{bstk}) \le \mathsf{DL}(\text{stk})$; hence, the result follows immediately from Theorem 15.22 and Theorem 15.26. ■

It is not known whether $\mathsf{DL}(\text{bstk}) < \mathsf{DL}(\text{stk})$ holds for monadic vocabularies. The case of poor vocabularies is treated in the exercises (Exercise 15.12).

15.5 The Power of a Boolean Stack

Regular programs with a Boolean stack are situated between pure regular programs and regular programs with an algebraic stack. We start our discussion by comparing the expressive power of regular programs with and without a Boolean stack. The only known definite answer to this problem is given in the following result, which covers the case of deterministic programs only.

THEOREM 15.35: If the vocabulary contains at least one function symbol of arity greater than one or at least two unary function symbols, then $\mathsf{DL}(\text{dreg}) < \mathsf{DL}(\text{dbstk})$.

Proof sketch. The main idea of the proof is as follows. We start with an infinite treelike structure $\mathfrak{A}$ in which all deterministic **while** programs unwind. Theorem 15.19 provides such structures. Next, we pick up an infinite path in $\mathfrak{A}$, cut it into finite pieces, and separate each two consecutive pieces u and w by inserting w^R in between them (the string w in reversed order). The hard part is to prove that all deterministic **while** programs still unwind in the transformed structure. However, it should be much clearer that there is a deterministic **while** program with a Boolean stack that can follow the entire infinite path; it simply stores on its stack the inserted strings and uses the stored string in order to find a way through the next piece of the infinite path. The technical details are rather complicated. The reader can consult Urzyczyn (1987) for the details. ■

It is not known whether Theorem 15.35 holds for nondeterministic programs, and neither is its statement known to be equivalent to any of the well known open problems in complexity theory. In contrast, it follows from Exercise 15.10 and from Theorem 15.32 that for rich mono-unary vocabularies the statement "$\mathsf{DL}(\text{dreg}) \equiv \mathsf{DL}(\text{dbstk})$" is equivalent to $LOGSPACE = P$. Hence, this problem cannot be solved without solving one of the major open problems in complexity theory.

The comparison of the expressive power of a Boolean stack and an algebraic stack is discussed in Theorem 15.31 for polyadic vocabularies and in Theorem 15.33 and Theorem 15.34 for monadic vocabularies.

15.6 Unbounded Nondeterminism

The wildcard assignment statement $x := ?$ discussed in Section 11.2 chooses an element of the domain of computation nondeterministically and assigns it to x. It is a device that represents *unbounded nondeterminism* as opposed to the binary nondeterminism of the nondeterministic choice construct $\cup$. The programming language of regular programs augmented with wildcard assignment is not an acceptable programming language, since a wildcard assignment can produce values that are outside the substructure generated by the input.

Our first result shows that wildcard assignment increases the expressive power in quite a substantial way; it cannot be simulated even by r.e. programs.

THEOREM 15.36: Let the vocabulary Σ contain two constants c_1, c_2, a binary predicate symbol p, the symbol $=$ for equality, and no other function or predicate symbols. There is a formula of DL(wild) that is equivalent to no formula of DL(r.e.), thus $\mathsf{DL}(\text{wild}) \not\leq \mathsf{DL}(\text{r.e.})$.

Proof Consider the DL(wild) formula

$$\varphi \quad \stackrel{\text{def}}{=} \quad \texttt{<}(x := c_1; z := ?; p(x,z)?; x := z)^*\texttt{>}\, x = c_2,$$

which is true in a structure $\mathfrak{A}$ iff (c_1, c_2) belongs to the transitive closure of p. Since the vocabulary contains no function symbols, it follows that every DL(r.e.) formula is equivalent to a first-order formula. It is well known (and in fact can be proved quite easily by the compactness of predicate logic) that there is no first-order formula capable of expressing the transitive closure of a binary relation. ■

It is not known whether any of the logics with unbounded memory are reducible to DL(wild). An interesting thing happens when both wildcard and array assignments are allowed. We show that in the resulting logic, it is possible to define the finiteness of (the domain of) a structure, but not in the logics with either of the additions removed. Thus, having both memory and nondeterminism unbounded provides more power than having either of them bounded.

THEOREM 15.37: Let vocabulary Σ contain only the symbol of equality. There is a formula of DL(array+wild) equivalent to no formula of either DL(array) or DL(wild).

Proof Let F be a unary function variable and consider the formula

$$\varphi \stackrel{\text{def}}{=} \langle\alpha\rangle\, \forall y\, \exists x\, \langle z := F(x)\rangle\, z = y,$$

where $\alpha = (x := ?; y := ?; F(x) := y)^*$. This program stores some elements in some locations of F. In a model $\mathfrak{A}$, the formula φ expresses the fact that we can store all elements of the domain in the variable F in a finite number of steps, thus the domain is finite. That finiteness cannot be expressed in DL(array) should be clear: since DL(array) is reducible over this vocabulary to first-order logic, another routine application of the compactness of predicate logic suffices.

We show that over our vocabulary, DL(wild) is also reducible to first-order logic. For this it is enough to observe that for our simple vocabulary, every regular program with wildcard assignments unwinds in every structure. Given a regular program α with wildcard assignments, let $x_1, \ldots, x_k$ be all the variables occurring in α. Seqs in $CS(\alpha)$ are sequences of the following three kinds of atomic programs:

$$x_i := x_j \qquad x_i := ? \qquad \varphi?,$$

where $i, j \in \{1, \ldots, k\}$ and φ is a Boolean combination of atomic formulas of the form $x_i = x_j$. It is easy to show that for each seq $\sigma \in CS(\alpha)$, there is a program γ and a first-order formula ψ such that for every structure $\mathfrak{A}$,

$$\mathfrak{m}_{\mathfrak{A}}(\sigma) \quad = \quad \{(u, v) \in \mathfrak{m}_{\mathfrak{A}}(\gamma) \mid u \in \mathfrak{m}_{\mathfrak{A}}(\psi)\}.$$

The program γ uses only variables from $\{x_1, \ldots, x_k\}$, and it is a sequence of assignments (ordinary or wildcard) such that no variable on the left side of an assignment appears twice in γ. Moreover, ψ is a conjunction of formulas of the form $\exists x_{i_1} \ldots \exists x_{i_m}\ \varphi$, where each $x_{i_j} \in \{x_1, \ldots, x_k\}$ and φ is a Boolean combination of atomic formulas of the form $x_i = x_j$. Since there are only finitely many different γ and ψ satisfying the above conditions, it follows that that there are only finitely many semantically different seqs in $CS(\alpha)$, therefore α unwinds in all structures. ∎

15.7 Bibliographical Notes

Many of the results on relative expressiveness presented herein answer questions posed in Harel (1979). Similar uninterpreted research, comparing the expressive power of classes of programs (but detached from any surrounding logic) has taken place under the name *comparative schematology* quite extensively ever since Ianov (1960); see Greibach (1975) and Manna (1974).

The results of Section 15.1 are folklore. However, Kfoury (1985) contained the

first proposal to use the notion of the unwind property as a tool for separating the expressive power of logics of programs (Theorem 15.4). Kreczmar (1977) studied the unwind property over the fields of real and complex numbers as well as over Archimedian fields (the unwind property for deterministic **while** programs holds for each of these structures). A systematic study of the unwind property, mainly for regular programs, was carried out in the PhD thesis of Urzyczyn (1983c).

The material of Section 15.2, relating spectra of logics of programs to their relative expressive power, is due to Tiuryn and Urzyczyn. It started with Tiuryn and Urzyczyn (1983) (see Tiuryn and Urzyczyn (1988) for the full version). The general Spectral Theorem (Theorem 15.11) is from Tiuryn and Urzyczyn (1984). However, some of the definitions presented in our exposition are simpler than in the papers cited above. In particular, the notion of *admissibility* of a programming language has been simplified here, and an auxiliary notion of *termination subsumption* has been introduced. As a result, some of the proofs have become simpler too. In particular, our proof of the Spectral Theorem is simpler than that in Tiuryn and Urzyczyn (1984).

The main result of Section 15.3, Theorem 15.20, appears in Berman et al. (1982) and was proved independently in Stolboushkin and Taitslin (1983). These results extend in a substantial way an earlier and much simpler result for the case of regular programs without equality in the vocabulary, which appears in Halpern (1981). A simpler proof of the special case of the quantifier-free fragment of the logic of regular programs appears in Meyer and Winklmann (1982). The proof of Theorem 15.20 presented here is from Tiuryn (1989). Theorem 15.19 is due to Urzyczyn (1983b), and as a corollary it yields Theorem 15.20. Theorem 15.21 is from Tiuryn and Urzyczyn (1984).

Theorem 15.22 is from Stolboushkin (1983). The proof, as in the case of regular programs (see Stolboushkin and Taitslin (1983)), uses Adian's result from group theory (Adian (1979)). Theorem 15.23 is also from Stolboushkin (1983). The method of trapping programs is from Kfoury (1985). Theorems 15.24 and 15.25 are from Kfoury (1985). Observe that Theorem 15.25 is strong enough to yield Theorem 15.20. Theorem 15.26 is from Tiuryn and Urzyczyn (1983, 1988).

The main result of Section 15.4, Theorem 15.30, is from Erimbetov (1981) and was proved independently by Tiuryn (1981b) (see Tiuryn (1984) for the full version). Erimbetov (1981) contains a somewhat special case of this result, namely that $\mathsf{DL}(\text{dreg}) < \mathsf{DL}(\text{dstk})$. Both proofs applied similar methods: pebble games on finite trees. The proof given here is based on the idea presented in Kfoury (1983). In particular, Proposition 15.29 is from Kfoury (1983). However, the proof of this Proposition was further simplified by Kfoury and Stolboushkin (1997). We follow the latter proof in our exposition.

Theorem 15.35 is from Urzyczyn (1987). There is a different proof of this result, using Adian structures, which appears in Stolboushkin (1989). Exercise 15.11 is from Urzyczyn (1988), which also studies programs with Boolean arrays.

Wildcard assignments were considered in Harel et al. (1977) under the name *nondeterministic assignments*. Theorem 15.36 is from Meyer and Winklmann (1982). Theorem 15.37 is from Meyer and Parikh (1981).

In our exposition of the comparison of the expressive power of logics, we have made the assumption that programs use only quantifier-free first-order tests. It follows from the results of Urzyczyn (1986) that allowing full first-order tests in many cases results in increased expressive power. Urzyczyn (1986) also proves that adding array assignments to nondeterministic r.e. programs increases the expressive power of the logic. This should be contrasted with the result of Meyer and Tiuryn (1981, 1984) to the effect that for deterministic r.e. programs, array assignments do not increase expressive power.

Makowski (1980) considers a weaker notion of equivalence between logics common in investigations in abstract model theory, whereby models are extended with interpretations for additional predicate symbols. With this notion it is shown in Makowski (1980) that most of the versions of logics of programs treated here become equivalent.

Exercises

15.1. Show that program equivalence is not invariant under elementary equivalence of structures.

15.2. (Meyer and Tiuryn (1981, 1984)) Define the class of *deterministic r.e. programs* over a given vocabulary. Show that DL(r.e.) has the same expressive power as DL over deterministic r.e. programs. Notice that r.e. programs are not divergence-closed.

15.3. In Theorem 15.20, reduce the case of a vocabulary containing a function symbol of arity greater than one to the case of a vocabulary containing two unary function symbols.

15.4. Define super-atomic seqs as those that use only simple assignments in which the terms have depth at most one. Show that a term t has pebble complexity at most n iff there is a super-atomic seq with at most n variables that computes it.

15.5. Show that every nondeterministic **while** program with a Boolean stack has bounded memory.

15.6. Show that regular programs with an algebraic stack are translatable into regular programs with arrays. (*Hint*. Prove that for every regular program α with an algebraic stack, there is a polynomial $p(n)$ such that in every terminating computation of α over an n-element interpretation, the maximal depth of the stack is at most $p(n)$.)

15.7. Prove that regular programs with two algebraic stacks have the same computational power as arbitrary r.e. programs.

15.8. Prove Theorem 15.30 for vocabularies containing only symbols of arity greater than two.

15.9. Show that over a vocabulary containing no function symbols of arity greater than one all terms have pebble complexity one.

15.10. Show that over a mono-unary vocabulary, regular programs with a Boolean stack have the same computational power as regular programs with an algebraic stack. Show that the same result holds for deterministic programs. Conclude that the two version of DL over these programming languages are of equal expressive power.

15.11. Prove that over a monadic vocabulary, nondeterministic regular programs with a Boolean stack have the same computational power as nondeterministic regular programs with an algebraic stack.

15.12. Prove that for any poor vocabulary,

(a) $\mathsf{DL}(\text{stk}) \equiv \mathsf{DL}(\text{array})$ iff $DTIME(2^{O(n)}) = DSPACE(2^{O(n)})$;
(b) $\mathsf{DL}(\text{dreg}) \equiv \mathsf{DL}(\text{reg})$ iff $DSPACE(n) = NSPACE(n)$;
(c) $\mathsf{DL}(\text{dreg}) \equiv \mathsf{DL}(\text{dstk})$ iff $DSPACE(n) = DTIME(2^{O(n)})$.

MIT Press Math7X9/2010/08/25:15:15 Page 382

16 Variants of DL

In this section we consider some restrictions and extensions of DL. We are interested mainly in questions of comparative expressive power on the uninterpreted level. In arithmetical structures these questions usually become trivial, since it is difficult to go beyond the power of first-order arithmetic without allowing infinitely many distinct tests in programs (see Theorems 12.6 and 12.7). In regular DL this luxury is not present.

16.1 Algorithmic Logic

Algorithmic Logic (AL) is the predecessor of Dynamic Logic. The basic system was defined by Salwicki (1970) and generated an extensive amount of subsequent research carried out by a group of mathematicians working in Warsaw. Two surveys of the first few years of their work can be found in Banachowski et al. (1977) and Salwicki (1977).

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

$$\alpha\varphi \qquad \cup\,\alpha\varphi \qquad \cap\,\alpha\varphi$$

corresponding in our terminology to

$$<\alpha>\varphi \qquad <\alpha^*>\varphi \qquad \bigwedge_{n\in\omega} <\alpha^n>\varphi,$$

respectively, where α is a deterministic **while** program and φ is a quantifier-free first-order formula.

In Mirkowska (1980, 1981a,b), AL was extended to allow nondeterministic **while** programs and the constructs

$$\nabla\alpha\varphi \qquad \Delta\alpha\varphi$$

corresponding in our terminology to

$$<\alpha>\varphi \qquad \mathbf{halt}(\alpha) \land [\alpha]\varphi \land <\alpha>\varphi,$$

respectively. The latter asserts that all traces of α are finite and terminate in a state satisfying φ.

A feature present in AL but not in DL is the set of “dynamic terms” in addition to dynamic formulas. For a first-order term t and a deterministic **while** program

α, the meaning of the expression αt is the value of t after executing program α. If α does not halt, the meaning is undefined. Such terms can be systematically eliminated; for example, $P(x, \alpha t)$ is replaced by $\exists z\ (\langle\alpha\rangle(z = t) \wedge P(x, z))$.

The emphasis in the early research on AL was in obtaining infinitary completeness results (as in Section 14.1), developing normal forms for programs, investigating recursive procedures with parameters, and axiomatizing certain aspects of programming using formulas of AL. As an example of the latter, the algorithmic formula

$$(\textbf{while } s \neq \varepsilon \textbf{ do } s := \textbf{pop}(s))\mathbf{1}$$

can be viewed as an axiom connected with the data structure *stack*. One can then investigate the consequences of such axioms within AL, regarding them as properties of the corresponding data structures.

Complete infinitary deductive systems for first-order and propositional versions are given in Mirkowska (1980, 1981a,b). The infinitary completeness results for AL are usually proved by the algebraic methods of Rasiowa and Sikorski (1963).

Constable (1977), Constable and O'Donnell (1978) and Goldblatt (1982) present logics similar to AL and DL for reasoning about deterministic **while** programs.

16.2 Nonstandard Dynamic Logic

Nonstandard Dynamic Logic (NDL) was introduced by Andréka, Németi, and Sain in 1979. The reader is referred to Németi (1981) and Andréka et al. (1982a,b) 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 Andréka et al. (1982a,b) and further research in NDL is concentrated on proving properties of flowcharts, i.e., programs built up of assignments, conditionals and **go to** statements.

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 Σ be a first-order vocabulary. For the remainder of this section we fix a deterministic **while** program α over Σ 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 α, and let $\overline{y} = (y_1, \ldots, y_n)$ be a vector of n distinct individual variables disjoint from $\overline{z}$.

Since α is loop-free, it has only finitely many computation sequences. One can easily define a quantifier-free first-order formula θ_α with all free variable among $\overline{y}, \overline{z}$ that defines the input/output relation of α in all Σ-structures $\mathfrak{A}$ in the sense that the pair of states (u, v) is in $\mathfrak{m}_{\mathfrak{A}}(\alpha)$ if and only if

$$\mathfrak{A}, v[y_1/u(z_1), \ldots, y_n/u(z_n)] \quad \vDash \quad \theta_\alpha$$

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

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

$\overline{y} := \overline{z}$;
α;
while $\overline{z} \neq \overline{y}$ **do** $\overline{y} := \overline{z};\ \alpha$

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 α^+ executes α iteratively until α 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 assertions involving α^+ relative to a given first-order theory T over Σ. We denote provability in Hoare Logic by $\vdash_{\mathsf{HL}}$.

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

$$\frac{\varphi \to \chi,\ \chi[\overline{z}/\overline{y}] \wedge \theta_\alpha \to \chi,\ \chi[\overline{z}/\overline{y}] \wedge \theta_\alpha \wedge \overline{z} = \overline{y} \to \psi}{\varphi \to [\alpha^+]\psi}$$

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 $\mathfrak{I}$ 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 $\mathfrak{I}$:

- $x + 1 = y + 1 \to x = y$
- $x + 1 \neq 0$
- $x \neq 0 \to \exists y\ y + 1 = x$
- $x \neq x \underbrace{+1 + 1 + \cdots + 1}_{n}$, for any $n = 1, 2, \ldots$

Let $\mathfrak{A}$ be a Σ-structure and $\mathfrak{I}$ a time model. A function $\rho : \mathfrak{I} \to \mathfrak{A}^n$ is called a

run of α in $\mathfrak{A}$ if the following infinitary formulas are valid in $\mathfrak{A}$:

- $\bigwedge_{i\in\mathfrak{I}} \theta_\alpha[\overline{y}/\rho(i), \overline{z}/\rho(i+1)]$;
- for every first-order formula $\varphi(\overline{z})$ over Σ,

$$\varphi(\rho(0)) \wedge \bigwedge_{i\in\mathfrak{I}} (\varphi(\rho(i)) \to \varphi(\rho(i+1))) \quad \to \quad \bigwedge_{i\in\mathfrak{I}} \varphi(\rho(i)).$$

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

Finally, we say that a partial correctness formula $\varphi \to [\alpha^+]\psi$ *follows from* T *in nonstandard time semantics* and write $T \models_{\mathsf{NT}} \varphi \to [\alpha^+]\psi$ if for every model $\mathfrak{A}$ of T, time model $\mathfrak{I}$, and run ρ of α in $\mathfrak{A}$,

$$\mathfrak{A} \models \varphi[\overline{z}/\rho(0)] \quad \to \quad \bigwedge_{i\in\mathfrak{I}} (\rho(i) = \rho(i+1) \to \psi[\overline{z}/\rho(i)]).$$

The following theorem characterizes the power of Hoare Logic for programs of the form α^+ over nonstandard time models.

THEOREM 16.1: For every first-order theory T over Σ and first-order formulas φ, ψ over Σ, the following conditions are equivalent:

(i) $T \vdash_{\mathsf{HL}} \varphi \to [\alpha^+]\psi$;

(ii) $T \models_{\mathsf{NT}} \varphi \to [\alpha^+]\psi$.

Other proof methods have been characterized in the same spirit. The reader is referred to Makowski and Sain (1986) for more information on this issue and further references.

16.3 Well-Foundedness

As in Section 10.6 for PDL, we consider adding to DL assertions to the effect that programs can enter infinite computations. Here too, we shall be interested both in LDL and in RDL versions; i.e., those in which **halt** α and **wf** α, respectively, have been added inductively as new formulas for any program α. As mentioned there, the connection with the more common notation **repeat** α and **loop** α (from which

the L and R in the names LDL and RDL derive) is by:

$$\begin{aligned} \textbf{loop}\,\alpha &\overset{\text{def}}{\iff} \neg\textbf{halt}\,\alpha \\ \textbf{repeat}\,\alpha &\overset{\text{def}}{\iff} \neg\textbf{wf}\,\alpha. \end{aligned}$$

We now state some of the relevant results. The first concerns the addition of $\textbf{halt}\,\alpha$:

THEOREM 16.2:

$$\mathsf{LDL} \quad \equiv \quad \mathsf{DL}.$$

Proof sketch. In view of the equivalences (10.6.2)–(10.6.5) of Section 10.6, it suffices, for each regular program α, to find a DL formula φ_α such that

$$\vDash [\alpha^*]\textbf{halt}\,\alpha \quad \to \quad (\varphi_\alpha \leftrightarrow \textbf{wf}\,\alpha).$$

Given such φ_α, $\textbf{halt}\,(\alpha^*)$ is equivalent to $[\alpha^*]\textbf{halt}\,\alpha \wedge \varphi_\alpha$.

Consider the computation tree $T_\alpha(s)$ corresponding to the possible computations of α in state s. The tree is derived from α by identifying common prefixes of seqs. A node of $T_\alpha(s)$ is labeled with the state reached at that point. The tree $T_\alpha(s)$, it should be noted, is obtained from the syntactic tree T_α by truncating subtrees that are rooted below false tests. Then $s \vDash \textbf{halt}\,\alpha$ holds iff $T_\alpha(s)$ contains no infinite path.

For any program of the form α^*, consider the tree $S_\alpha(s)$ derived from $T_{\alpha^*}(s)$ by eliminating all states internal to executions of α. Thus t is an immediate descendant of t' in $S_\alpha(s)$ iff t' is reached from s by some execution of α^* and t is reached from t' by an additional execution of α.

If $s \vDash [\alpha^*]\textbf{halt}\,\alpha$, then by König's lemma $S_\alpha(s)$ is of finite outdegree. It can be shown that in this case $S_\alpha(s)$ has an infinite path iff either some state repeats along a path or there are infinitely many states t, each of which appears only within bounded depth in $S_\alpha(s)$ but for which there is a state appearing for the first time at depth greater than that of the last appearance of t.

This equivalent to "$S_\alpha(s)$ contains an infinite path" is then written in DL using the fact that a state is characterized by a finite tuple of values corresponding to the finitely many variables in α.

As an example of a typical portion of this definition, the following is a DL equivalent of the statement: "There is a state in $S_\alpha(s)$ appearing for the first time at depth greater than the greatest depth at which a given state $\overline{y}$ appears."

$$\exists\overline{z}\ (<\alpha^*>\overline{x} = \overline{z} \wedge [\overline{z}' := \overline{x}; (\alpha; \alpha[\overline{z}'/\overline{x}])^*; \overline{z}' = \overline{z}?; \alpha^*]\neg\overline{x} = \overline{y}).$$

Here $\overline{y}, \overline{z}$ and $\overline{z}'$ are n-tuples of new variables denoting states matching the n-tuple $\overline{x}$ of variables appearing in α. Assignments and tests are executed pointwise, as is the substitution $\alpha[\overline{z}'/\overline{x}]$, which replaces all occurrences of the variables in α with their $\overline{z}'$ counterparts. The inner program runs α simultaneously on $\overline{x}$ and $\overline{z}'$, reaches $\overline{z}$ and then continues running α on $\overline{x}$. The assertion is that $\overline{y}$ cannot be obtained in this manner. ■

In contrast to this, we have:

Theorem 16.3:

$$\mathsf{LDL} \quad < \quad \mathsf{RDL}.$$

Proof sketch. The result is proved by showing how to state in RDL that a binary function g is a well-order, where one first constrains the domain to be countable, with the unary f acting as a successor function starting at some "zero" constant c. The result then follows from the fact that well-order is not definable in $L_{\omega_1\omega}$ (see Keisler (1971)). ■

Turning to the validity problem for these extensions, clearly they cannot be any harder to decide than that of DL, which is Π^1_1-complete. However, the following result shows that detecting the absence of infinite computations of even simple uninterpreted programs is extremely hard.

Theorem 16.4: The validity problems for formulas of the form $\varphi \to \mathbf{wf}\,\alpha$ and formulas of the form $\varphi \to \mathbf{halt}\,\alpha$, for first-order φ and regular α, are both Π^1_1-complete. If α is constrained to have only first-order tests then the $\varphi \to \mathbf{wf}\,\alpha$ case remains Π^1_1-complete but the $\varphi \to \mathbf{halt}\,\alpha$ case is r.e.; that is, it is Σ^0_1-complete.

Proof sketch. That the problems are in Π^1_1 is easy. The Π^1_1-hardness results can be established by reductions from the recurring tiling problem of Proposition 2.22 similarly to the proof of Theorem 13.1. As for $\mathbf{halt}\,\alpha$ formulas with first-order tests in Σ^0_1, compactness and König's lemma are used. Details appear in Harel and Peleg (1985). ■

Axiomatizations of LDL and RDL are discussed in Harel (1984). We just mention here that the additions to Axiom System 14.12 of Chapter 14 that are used to obtain an arithmetically complete system for RDL are the axiom

$$[\alpha^*](\varphi \to <\alpha>\varphi) \quad \to \quad (\varphi \to \neg\mathbf{wf}\,\alpha)$$

and the inference rule

$$\frac{\varphi(n+1) \to [\alpha]\varphi(n),\ \neg\varphi(0)}{\varphi(n) \to \mathbf{wf}\,\alpha}$$

for first-order φ and n not occurring in α.

16.4 Dynamic Algebra

Dynamic algebra provides an abstract algebraic framework that relates to PDL as Boolean algebra relates to propositional logic. Dynamic algebra was introduced in Kozen (1980b) and Pratt (1979b) and studied by numerous authors; see Kozen (1979c,b, 1980a, 1981b); Pratt (1979a, 1980a, 1988); Németi (1980); Trnkova and Reiterman (1980). A survey of the main results appears in Kozen (1979a). A *dynamic algebra* is defined to be any two-sorted algebraic structure $(K, B, \cdot)$, where $B = (B, \to, 0)$ is a Boolean algebra, $K = (K, +, \cdot, {}^*, 0, 1)$ is a Kleene algebra (see Section 17.5), and $\cdot : K \times B \to B$ is a scalar multiplication satisfying algebraic constraints corresponding to the dual forms of the PDL axioms (Axioms 5.5). For example, all dynamic algebras satisfy the equations

$$\begin{aligned}
(\alpha\beta) \cdot \varphi &= \alpha \cdot (\beta \cdot \varphi) \\
\alpha \cdot 0 &= 0 \\
0 \cdot \varphi &= 0 \\
\alpha \cdot (\varphi \vee \psi) &= \alpha \cdot \varphi \vee \alpha \cdot \psi,
\end{aligned}$$

which correspond to the PDL validities

$$\begin{aligned}
<\alpha\,;\,\beta>\varphi &\leftrightarrow <\alpha><\beta>\varphi \\
<\alpha>\mathbf{0} &\leftrightarrow \mathbf{0} \\
<\mathbf{0}?>\varphi &\leftrightarrow \mathbf{0} \\
<\alpha>(\varphi \vee \psi) &\leftrightarrow <\alpha>\varphi \vee <\alpha>\psi,
\end{aligned}$$

respectively. The Boolean algebra B is an abstraction of the formulas of PDL and the Kleene algebra K is an abstraction of the programs.

Kleene algebra is of interest in its own right, and we defer a more detailed treatment until Section 17.5. In short, a Kleene algebra is an idempotent semiring under $+$, $\cdot$, 0, 1 satisfying certain axioms for * that say essentially that * behaves like the asterate operator on sets of strings or reflexive transitive closure on binary relations. There are finitary and infinitary axiomatizations of the essential

properties of * that are of quite different deductive strength. A Kleene algebra satisfying the stronger infinitary axiomatization is called **-continuous* (see Section 17.5).

The interaction of scalar multiplication with iteration can be axiomatized in a finitary or infinitary way. One can postulate

$$\alpha^* \cdot \varphi \quad \leq \quad \varphi \vee (\alpha^* \cdot (\neg\varphi \wedge (\alpha \cdot \varphi))) \tag{16.4.1}$$

corresponding to the diamond form of the PDL induction axiom (Axiom 5.5(viii)). Here $\varphi \leq \psi$ in B iff $\varphi \vee \psi = \psi$. Alternatively, one can postulate the stronger axiom of *-*continuity*:

$$\alpha^* \cdot \varphi \quad = \quad \sup_n (\alpha^n \cdot \varphi). \tag{16.4.2}$$

We can think of (16.4.2) as a conjunction of infinitely many axioms $\alpha^n \cdot \varphi \leq \alpha^* \cdot \varphi$, $n \geq 0$, and the infinitary Horn formula

$$(\bigwedge_{n \geq 0} \alpha^n \cdot \varphi \ \leq \ \psi) \quad \rightarrow \quad \alpha^* \cdot \varphi \ \leq \ \psi.$$

In the presence of the other axioms, (16.4.2) implies (16.4.1) (Kozen (1980b)), and is strictly stronger in the sense that there are dynamic algebras that are not *-continuous (Pratt (1979a)).

A standard Kripke frame $\mathfrak{K} = (U, \mathfrak{m}_{\mathfrak{K}})$ of PDL gives rise to a *-continuous dynamic algebra consisting of a Boolean algebra of subsets of U and a Kleene algebra of binary relations on U. Operators are interpreted as in PDL, including 0 as $\mathbf{0}$? (the empty program), 1 as $\mathbf{1}$? (the identity program), and $\alpha \cdot \varphi$ as $<\alpha>\varphi$. Nonstandard Kripke frames (see Section 6.3) also give rise to dynamic algebras, but not necessarily *-continuous ones. A dynamic algebra is *separable* if any pair of distinct Kleene elements can be distinguished by some Boolean element; that is, if $\alpha \neq \beta$, then there exists $\varphi \in B$ with $\alpha \cdot \varphi \neq \beta \cdot \varphi$.

Research directions in this area include the following.

- *Representation theory.* It is known that any separable dynamic algebra is isomorphic to some possibly nonstandard Kripke frame. Under certain conditions, "possibly nonstandard" can be replaced by "standard," but not in general, even for *-continuous algebras (Kozen (1980b, 1979c, 1980a)).
- *Algebraic methods in* PDL. The small model property (Theorem 6.5) and completeness (Theorem 7.6) for PDL can be established by purely algebraic considerations (Pratt (1980a)).

• *Comparative study of alternative axiomatizations of* *. For example, it is known that separable dynamic algebras can be distinguished from standard Kripke frames by a first-order formula, but even $L_{\omega_1\omega}$ cannot distinguish the latter from *-continuous separable dynamic algebras (Kozen (1981b)).

• *Equational theory of dynamic algebras.* Many seemingly unrelated models of computation share the same equational theory, namely that of dynamic algebras (Pratt (1979b,a)).

In addition, many interesting questions arise from the algebraic viewpoint, and interesting connections with topology, classical algebra, and model theory have been made (Kozen (1979b); Németi (1980)).

16.5 Probabilistic Programs

There is wide interest recently in programs that employ probabilistic moves such as coin tossing or random number draws and whose behavior is described probabilistically (for example, α is "correct" if it does what it is meant to do with probability 1). To give one well known example taken from Miller (1976) and Rabin (1980), there are fast probabilistic algorithms for checking primality of numbers but no known fast nonprobabilistic ones. Many synchronization problems including digital contract signing, guaranteeing mutual exclusion, etc. are often solved by probabilistic means.

This interest has prompted research into formal and informal methods for reasoning about probabilistic programs. It should be noted that such methods are also applicable for reasoning probabilistically about ordinary programs, for example, in average-case complexity analysis of a program, where inputs are regarded as coming from some set with a probability distribution.

Kozen (1981d) provided a formal semantics for probabilistic first-order **while** programs with a random assignment statement $x := ?$. Here the term "random" is quite appropriate (contrast with Section 11.2) as the statement essentially picks an element out of some fixed distribution over the domain D. This domain is assumed to be given with an appropriate set of measurable subsets. Programs are then interpreted as measurable functions on a certain measurable product space of copies of D.

In Feldman and Harel (1984) a probabilistic version of first-order Dynamic Logic, $Pr(\mathsf{DL})$, was investigated on the interpreted level. Kozen's semantics is extended as described below to a semantics for formulas that are closed under Boolean connectives and quantification over reals and integers and that employ

terms of the form $Fr(\varphi)$ for first-order φ. In addition, if α is a **while** program with nondeterministic assignments and φ is a formula, then $\{\alpha\}\varphi$ is a new formula.

The semantics assumes a domain D, say the reals, with a measure space consisting of an appropriate family of *measurable subsets* of D. The states $\mu, \nu, \ldots$ are then taken to be the positive measures on this measure space. Terms are interpreted as functions from states to real numbers, with $Fr(\varphi)$ in μ being the *frequency* (or simply, the *measure*) of φ in μ. Frequency is to positive measures as probability is to probability measures. The formula $\{\alpha\}\varphi$ is true in μ if φ is true in ν, the state (i.e., measure) that is the result of applying α to μ in Kozen's semantics. Thus $\{\alpha\}\varphi$ means "after α, φ" and is the construct analogous to $<\alpha>\varphi$ of DL.

For example, in $Pr(\mathsf{DL})$ one can write

$$Fr(1) = 1 \quad \rightarrow \quad \{\alpha\}Fr(1) \geq p$$

to mean, "α halts with probability at least p." The formula

$$\begin{aligned} Fr(1) = 1 \quad \rightarrow \quad & [i := 1; x := ?; \textbf{while } x > 1/2 \textbf{ do } (x := ?; i := i+1)] \\ & \forall n\ ((n \geq 1 \rightarrow Fr(i = n) = 2^{-n}) \wedge (n < 1 \rightarrow Fr(i = n) = 0)) \end{aligned}$$

is valid in all structures in which the distribution of the random variable used in $x := ?$ is a uniform distribution on the real interval $[0, 1]$.

An axiom system for $Pr(\mathsf{DL})$ was proved in Feldman and Harel (1984) to be complete relative to an extension of first-order analysis with integer variables, and for discrete probabilities first-order analysis with integer variables was shown to suffice.

Various propositional versions of probabilistic DL have also been proposed; see Reif (1980); Makowsky and Tiomkin (1980); Ramshaw (1981); Feldman (1984); Parikh and Mahoney (1983); Kozen (1985). In Ramshaw (1981), Ramshaw gave a Hoare-like logic, but observed that even the if-then-else rule was incomplete. Reif (1980) gave a logic that was not expressive enough to define if-then-else; moreover, the soundness of one of its proof rules was later called into question (Feldman and Harel (1984)). Makowsky and Tiomkin (1980) gave an infinitary system and proved completeness. Parikh and Mahoney (1983) studied the equational properties of probabilistic programs. Feldman (1984) gave a less expressive version of $Pr(\mathsf{DL})$, though still with quantifiers, and proved decidability by reduction to the first-order theory of $\mathbb{R}$ (Renegar (1991)). Kozen (1985) replaced the truth-functional propositional operators with analogous arithmetic ones, giving an arithmetical calculus closer in spirit to the semantics of Kozen (1981d). Three equivalent

semantics were given: a Markov transition semantics, a generalized operational semantics involving measure transformers, and a generalized predicate transformer semantics involving measurable function transformers. A small model property and *PSPACE* decision procedure over well-structured programs were given. A deductive calculus was proposed and its use demonstrated by calculating the expected running time of a random walk.

In a different direction, Lehmann and Shelah (1982) extend propositional temporal logic (Section 17.2) with an operator C for "certainly" where $C\varphi$ means essentially, "φ is true with probability 1." Actual numerical probabilities, like p or 2^{-n} in the examples above, are not expressible in this language. Nevertheless, the system can express many properties of interest, especially for finite state protocols that employ probabilistic choice, such as probabilistic solutions to such synchronization problems as mutual exclusion. In many such cases the probabilistic behavior of the program can be described without resorting to numerical values and is independent of the particular distribution used for the random choices.

For example, one can write

$$\mathbf{at}\, L_1 \quad \rightarrow \quad (\neg C \neg \bigcirc \mathbf{at}\, L_2 \wedge \neg C \neg \bigcirc \mathbf{at}\, L_3)$$

meaning "if execution is at label L_1, then it is possible (i.e., true with nonzero probability) to be at L_2 in the next step, and similarly for L_3." Three variants of the system, depending upon whether positive probabilities are bounded from below or not, and whether or not the number of possibilities is finite, are shown in Lehmann and Shelah (1982) to be decidable and complete with respect to finite effective axiomatizations that extend those of classical modal or temporal logic.

Probabilistic processes and model checking have recently become a popular topic of research; see Morgan et al. (1999); Segala and Lynch (1994); Hansson and Jonsson (1994); Jou and Smolka (1990); Pnueli and Zuck (1986, 1993); Baier and Kwiatkowska (1998); Huth and Kwiatkowska (1997); Blute et al. (1997). The relationship between all these formal approaches remains an interesting topic for further work.

16.6 Concurrency and Communication

As in Section 10.7 for PDL, we can add to DL the concurrency operator for programs, so that $\alpha \wedge \beta$ is a program, inductively, for any α and β. As in concurrent PDL, the meaning of a program is then a relation between states and sets of states.

It is not known whether the resulting logic, *concurrent* DL, is strictly more

expressive than DL, but this is known to be true if both logics are restricted to allow only quantifier-free first-order tests in the programs.

Also, the four axiom systems of Chapter 14 can be proved complete with the appropriate addition of the valid formulas of the concurrent versions of PDL.

16.7 Bibliographical Notes

Algorithmic logic was introduced by Salwicki (1970). Mirkowska (1980, 1981a,b) extended AL to allow nondeterministic **while** programs and studied the operators ∇ and Δ. Complete infinitary deductive systems for propositional and first-order versions were given by Mirkowska (1980, 1981a,b) using the algebraic methods of Rasiowa and Sikorski (1963). Surveys of early work in AL can be found in Banachowski et al. (1977); Salwicki (1977). Constable (1977), Constable and O'Donnell (1978) and Goldblatt (1982) presented logics similar to AL and DL for reasoning about deterministic **while** programs.

Nonstandard Dynamic Logic was introduced by Németi (1981) and Andréka et al. (1982a,b). Theorem 16.1 is due to Csirmaz (1985). See Makowski and Sain (1986) for more information and further references on NDL. Nonstandard semantics has also been studied at the propositional level; see Section 6.4.

The **halt** construct (actually its complement, **loop**) was introduced in Harel and Pratt (1978), and the **wf** construct (actually its complement, **repeat**) was investigated for PDL in Streett (1981, 1982). Theorem 16.2 is from Meyer and Winklmann (1982), Theorem 16.3 is from Harel and Peleg (1985), Theorem 16.4 is from Harel (1984), and the axiomatizations of LDL and PDL are discussed in Harel (1979, 1984).

Dynamic algebra was introduced in Kozen (1980b) and Pratt (1979b) and studied by numerous authors; see Kozen (1979c,b, 1980a, 1981b); Pratt (1979a, 1980a, 1988); Németi (1980); Trnkova and Reiterman (1980). A survey of the main results appears in Kozen (1979a).

The PhD thesis of Ramshaw (1981) contains an engaging introduction to the subject of probabilistic semantics and verification. Kozen (1981d) provided a formal semantics for probabilistic programs. The logic $Pr(\mathsf{DL})$ was presented in Feldman and Harel (1984), along with a deductive system that is complete for Kozen's semantics relative to an extension of first-order analysis. Various propositional versions of probabilistic DL have been proposed in Reif (1980); Makowsky and Tiomkin (1980); Feldman (1984); Parikh and Mahoney (1983); Kozen (1985). The temporal approach to probabilistic verification has been studied

in Lehmann and Shelah (1982); Hart et al. (1982); Courcoubetis and Yannakakis (1988); Vardi (1985a). Interest in the subject of probabilistic verification has undergone a recent revival; see Morgan et al. (1999); Segala and Lynch (1994); Hansson and Jonsson (1994); Jou and Smolka (1990); Baier and Kwiatkowska (1998); Huth and Kwiatkowska (1997); Blute et al. (1997).

Concurrent DL is defined in Peleg (1987b), in which the results mentioned in Section 16.6 are proved. Additional versions of this logic, which employ various mechanisms for communication among the concurrent parts of a program, are also considered in Peleg (1987c,a).

17 Other Approaches

In this chapter we describe some topics that are the subject of extensive past and present research and which are all closely related to Dynamic Logic. Our descriptions here are very brief and sketchy and are designed to provide the reader with only a most superficial idea of the essence of the topic, together with one or two central or expository references where details and further references can be found.

17.1 Logic of Effective Definitions

The Logic of Effective Definitions (LED), introduced by Tiuryn (1981a), 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 $\alpha = \beta$, where α and β are *effective definitional schemes* (the latter notion is due to Friedman (1971)):

if φ_1 **then** t_1
  else if φ_2 **then** t_2
    else if φ_3 **then** t_3
      else if . . .

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

Model theory and infinitary completeness of LED are treated in Tiuryn (1981a).

Effective definitional schemes in the definition of LED can be replaced by any programming language K, giving rise to various logical formalisms. The following result, which relates LED to other logics discussed here, is proved in Meyer and Tiuryn (1981, 1984).

THEOREM 17.1: For every signature L,

$$\mathsf{LED} \quad \equiv \quad \mathsf{DL}(\text{r.e.}).$$

17.2 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 (1977) and has since been developed by many authors in various forms. This topic is surveyed in depth in Emerson (1990) and Gabbay et al. (1994).

TL differs from DL chiefly in that it is *endogenous*; that is, 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** L, meaning "execution is currently at location L in the program." There are two competing semantics, giving rise to two different theories called *linear-time* and *branching-time* TL. In the former, a model is a linear sequence of program states representing an execution sequence of a deterministic program or a possible execution sequence of a nondeterministic or concurrent program. In the latter, a model is a tree of program states representing the space of all possible traces of a nondeterministic or concurrent program. Depending on the application and the semantics, different syntactic constructs can be chosen. The relative advantages of linear and branching time semantics are discussed in Lamport (1980); Emerson and Halpern (1986); Emerson and Lei (1987); Vardi (1998a).

Modal constructs used in TL include

$\Box\varphi$	"φ holds in all future states"
$\Diamond\varphi$	"φ holds in some future state"
$\bigcirc\varphi$	"φ holds in the next state"

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

"for all traces starting from the present state..."
"for some trace 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 classical program verification methods such as the *intermittent assertions method* are treated quite elegantly in this framework; we give an example of this below.

Temporal logic has been most successful in providing tools for proving properties of concurrent *finite state* protocols, such as solutions to the *dining philosophers* and *mutual exclusion* problems, which are popular abstract versions of synchronization and resource management problems in distributed systems.

The Inductive Assertions Method

In this section we give an example to illustrate the inductive assertions method. We will later give a more modern treatment using TL. For purposes of illustration, we use a programming language in which programs consist of a sequence of labeled statements. Statements may include simple assignments, conditional and unconditional **go to** statements, and **print** statements. For example, the following program computes $n!$.

Example 17.2:

L_0 : $x := 1$
L_1 : $y := 1$
L_2 : $y := y + 1$
L_3 : $x := x \cdot y$
L_4 : **if** $y \neq n$ **then go to** L_2
L_5 : **print** x

In this program, the variable n can be considered free; it is part of the input. Note that the program does not halt if $n = 1$. Suppose we wish to show that whenever the program halts, x will contain the value $n!$. Traditionally one establishes an *invariant*, which is a statement φ with the properties

(i) φ is true at the beginning,
(ii) φ is preserved throughout execution, and
(iii) φ implies the output condition.

In our case, the output condition is $x = n!$, and the appropriate invariant φ is

$$\begin{aligned} & \mathbf{at}\, L_1 \rightarrow x = 1 \\ \wedge\quad & \mathbf{at}\, L_2 \rightarrow x = y! \\ \wedge\quad & \mathbf{at}\, L_3 \rightarrow x = (y-1)! \\ \wedge\quad & \mathbf{at}\, L_4 \rightarrow x = y! \\ \wedge\quad & \mathbf{at}\, L_5 \rightarrow x = y! \wedge y = n \end{aligned} \tag{17.2.1}$$

where **at** L_i means the processor is about to execute statement L_i. Then (i) holds, because at the beginning of the program, **at** L_0 is true, therefore all five conjuncts are vacuously true. To show that (ii) holds, suppose we are at any point in the

program, say L_3, and φ holds. Then $x = (y-1)!$, since $\textbf{at}\, L_3 \rightarrow x = (y-1)!$ is a conjunct of φ. In the next step, we will be at L_4, and $x = y!$ will hold, since we will have just executed the statement $L_3 : x := x \cdot y$. Therefore $\textbf{at}\, L_4 \rightarrow x = y!$ will hold, and since $\textbf{at}\, L_4$ holds, all the other conjuncts will be vacuously true, so φ will hold. In this way we verify, for each possible location in the program, that φ is preserved after execution of one instruction. Finally, when we are about to execute L_5, φ ensures that x contains the desired result $n!$.

The Temporal Approach

To recast this development in the framework of Temporal Logic, note that we are arguing that a certain formula φ is preserved throughout time. If we define a *state* of the computation to be a pair (L_i, u) where L_i is the label of a statement and u is a valuation of the program variables, then we can consider the trace

$$\sigma \quad = \quad s_0 s_1 s_2 \cdots$$

of states that the program goes through during execution. Each state s_i contains all the information needed to determine whether φ is true at s_i. We write $s_i \vDash \varphi$ if the statement φ holds in the state s_i.

There is also a binary relation Next that tells which states can immediately follow a state. The relation Next depends on the program. For example, in the program of Example 17.2,

$$((L_2, x = 6, y = 14),\ (L_3, x = 6, y = 15)) \in \textsc{Next}.$$

In the sequence σ above, s_0 is the start state $(L_0,\ x = 0,\ y = 0)$ and s_{i+1} is the unique state such that $(s_i, s_{i+1}) \in \textsc{Next}$. In ordinary deterministic programs, each state has at most one Next-successor, but in concurrent or nondeterministic programs, there may be many possible Next-successors.

Define

$$\begin{aligned}
s \vDash \bigcirc\varphi &\overset{\text{def}}{\iff} \text{for all states } t \text{ such that } (s,t) \in \textsc{Next},\ t \vDash \varphi \\
s \vDash \Box\varphi &\overset{\text{def}}{\iff} \text{starting with } s, \text{ all future states satisfy } \varphi \\
&\iff \text{for all } t \text{ such that } (s,t) \in \textsc{Next}^*,\ t \vDash \varphi \\
&\qquad \text{where } \textsc{Next}^* \text{ is the reflexive transitive closure of } \textsc{Next} \\
s \vDash \Diamond\varphi &\overset{\text{def}}{\iff} s \vDash \neg\Box\neg\varphi.
\end{aligned}$$

In other words, $s \vDash \bigcirc\varphi$ if all Next-successors of s satisfy φ. In the trace σ, if s_{i+1} exists, then $s_i \vDash \bigcirc\varphi$ iff $s_{i+1} \vDash \varphi$. The formula $\bigcirc\varphi$ does not imply that a

NEXT-successor exists; however, the dual operator $\neg\bigcirc\neg$ can be used where this is desired:

$$s \vDash \neg\bigcirc\neg\varphi \iff \text{there exists } t \text{ such that } (s,t) \in \text{NEXT and } t \vDash \varphi.$$

In the trace σ, $s_i \vDash \Box\varphi$ iff $\forall j \geq i,\ s_j \vDash \varphi$.

To say that the statement φ of (17.2.1) is an *invariant* means that every s_i satisfies $\varphi \to \bigcirc\varphi$; that is, if $s_i \vDash \varphi$ then $s_{i+1} \vDash \varphi$. This is the same as saying

$$s_0 \quad \vDash \quad \Box(\varphi \to \bigcirc\varphi).$$

To say that φ holds at the beginning of execution is just

$$s_0 \quad \vDash \quad \varphi.$$

The principle of induction on $\mathbb{N}$ allows us to conclude that φ will be true in all reachable states; that is,

$$s_0 \quad \vDash \quad \Box\varphi.$$

We can immediately derive the correctness of the program, since (17.2.1) implies our desired output condition.

The induction principle of TL takes the form:

$$\varphi \wedge \Box(\varphi \to \bigcirc\varphi) \quad \to \quad \Box\varphi. \tag{17.2.2}$$

Note the similarity to the PDL induction axiom (Axiom 5.5(viii)):

$$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi.$$

This is a classical program verification method known as *inductive* or *invariant assertions*.

The operators $\Box$, $\bigcirc$, and $\Diamond$ are called *temporal operators* because they describe how the truth of the formula φ depends on time. The inductive or invariant assertions method is really an application of the temporal principle (17.2.2). The part $\Box(\varphi \to \bigcirc\varphi)$ of the formula φ of (17.2.2) says that φ is an *invariant*; that is, at all future points, if φ is true, then φ will be true after one more step of the program.

This method is useful for proving *invariant* or *safety properties*. These are properties that can be expressed as $\Box\varphi$; that is, properties that we wish to remain true throughout the computation. Examples of such properties are:

- *partial correctness*—see Example 17.2;
- *mutual exclusion*—no two processes are in their critical sections simultaneously;

• *clean execution*—for example, a stack never overflows, or we never divide by 0 at a particular division instruction;

• *freedom from deadlock*—it is never the case that all processes are simultaneously requesting resources held by another process.

Another very important class of properties that one would like to reason about are *eventuality* or *liveness properties*, which say that something will eventually become true. These are expressed using the $\diamond$ operator of TL. Examples are:

• *total correctness*—a program eventually halts and produces an output that is correct;

• *fairness* or *freedom from starvation*—if a process is waiting for a resource, it will eventually obtain access to it;

• *liveness of variables*—if a variable x is assigned a value through the execution of an assignment statement $x := t$, then that variable is used at some future point.

There are two historical methods of reasoning about eventualities. The first is called the *method of well-founded sets*; the second is called *intermittent assertions*.

Recall from Section 1.3 that a strict partial order $(A, <)$ is *well-founded* if every subset has a minimal element. This implies that there can be no infinite descending chains

$$a_0 \quad > \quad a_1 \quad > \quad a_2 \quad > \quad \cdots$$

in A. One way to prove that a program terminates is to find such a well-founded set $(A, <)$ and associate with each state s of the computation an element $a_s \in A$ such that if $(s, t) \in \textsc{Next}$ then $a_s > a_t$. Thus the program could not run forever through states $s_0, s_1, s_2, \ldots$, because then there would be an infinite descending chain

$$a_{s_0} \quad > \quad a_{s_1} \quad > \quad a_{s_2} \quad > \quad \cdots,$$

contradicting the assumption of well-foundedness. For example, in the program (17.2), if we start out with $n > 1$, then every time through the loop, y is incremented by 1, so progress is made toward $y = n$ which will cause the loop to exit at L_4. One can construct a well-founded order $<$ on states that models this forward progress. However, the expression describing it would be a rather lengthy and unnatural arithmetic combination of the values of n and y and label indices L_i, even for this very simple program.

A more natural method is the *intermittent assertions method*. This establishes

eventualities of the form $\psi \to \Diamond\varphi$ by applications of rules such as

$$\frac{\psi \to \Diamond\theta, \quad \theta \to \Diamond\varphi}{\psi \to \Diamond\varphi} \tag{17.2.3}$$

among others. This method may also use well-founded relations, although the well-founded relations one needs to construct are often simpler. For example, in the program of Example 17.2, total correctness is expressed by

$$\mathbf{at}\, L_0 \wedge n > 1 \quad \to \quad \Diamond(\mathbf{at}\, L_5 \wedge x = n!). \tag{17.2.4}$$

Using (17.2.3), we can prove

$$\mathbf{at}\, L_0 \wedge n > 1 \quad \to \quad \Diamond(\mathbf{at}\, L_4 \wedge y \leq n \wedge x = y!) \tag{17.2.5}$$

from the four statements

$$\begin{array}{rcl} \mathbf{at}\, L_0 \wedge n > 1 & \to & \bigcirc(\mathbf{at}\, L_1 \wedge n > 1 \wedge x = 1) \\ \mathbf{at}\, L_1 \wedge n > 1 \wedge x = 1 & \to & \bigcirc(\mathbf{at}\, L_2 \wedge n > 1 \wedge x = 1 \wedge y = 1) \\ \mathbf{at}\, L_2 \wedge n > 1 \wedge y = 1 & \to & \bigcirc(\mathbf{at}\, L_3 \wedge n > 1 \wedge x = 1 \wedge y = 2) \\ \mathbf{at}\, L_3 \wedge n > 1 \wedge x = 1 \wedge y = 2 & \to & \bigcirc(\mathbf{at}\, L_4 \wedge n > 1 \wedge x = 2 \wedge y = 2) \\ & \to & \bigcirc(\mathbf{at}\, L_4 \wedge y \leq n \wedge x = y!). \end{array}$$

Similarly, one can prove using (17.2.3) that for all values a,

$$\mathbf{at}\, L_4 \wedge y = a \wedge y < n \wedge x = y! \to \Diamond(\mathbf{at}\, L_4 \wedge y = a + 1 \wedge y \leq n \wedge x = y!) \tag{17.2.6}$$

by going through the loop once. This implies that every time through the loop, the value of $n - y$ decreases by 1. Thus we can use the well-founded relation $<$ on the natural numbers to get

$$\mathbf{at}\, L_4 \wedge y \leq n \wedge x = y! \quad \to \quad \Diamond(\mathbf{at}\, L_4 \wedge y = n \wedge x = y!) \tag{17.2.7}$$

from (17.2.6), using the principle

$$\exists m\ \psi(m) \wedge \forall m\ \Box(\psi(m+1) \to \Diamond\psi(m)) \quad \to \quad \Diamond\psi(0).$$

Finally, we observe that

$$\mathbf{at}\, L_4 \wedge y = n \wedge x = y! \to \bigcirc(\mathbf{at}\, L_5 \wedge x = n!),$$

so we achieve our proof of the total correctness assertion (17.2.4) by combining (17.2.5), (17.2.6), and (17.2.7) using (17.2.3).

Expressiveness

Recall

$$
\begin{aligned}
s \vDash \bigcirc\varphi &\overset{\text{def}}{\iff} \forall t\ (s,t) \in \textsc{Next} \to t \vDash \varphi\\
s \vDash \Box\varphi &\overset{\text{def}}{\iff} \forall t\ (s,t) \in \textsc{Next}^* \to t \vDash \varphi\\
s \vDash \Diamond\varphi &\overset{\text{def}}{\iff} s \vDash \neg\Box\neg\varphi\\
&\iff \exists t\ (s,t) \in \textsc{Next}^* \wedge t \vDash \varphi.
\end{aligned}
$$

Here are some interesting properties that can be expressed with $\bigcirc$, $\Diamond$, and $\Box$ over linear-time interpretations.

EXAMPLE 17.3:

(i) The trace consists of exactly one state:

$$\mathbf{halt} \overset{\text{def}}{\iff} \bigcirc\mathbf{0}$$

(ii) The trace is finite, that is, the computation eventually halts:

$$\mathbf{fin} \overset{\text{def}}{\iff} \Diamond\mathbf{halt}$$

(iii) The trace is infinite:

$$\mathbf{inf} \overset{\text{def}}{\iff} \neg\mathbf{fin}$$

(iv) The formula φ is true at infinitely many points along the trace (a formula is true at a state on a trace if the formula is satisfied by the suffix of the trace beginning at that state):

$$\mathbf{inf} \wedge \Box\Diamond\varphi$$

(v) The formula φ becomes true for the first time at some point, then remains true thereafter:

$$\Diamond\varphi \wedge \Box(\varphi \to \Box\varphi)$$

(vi) The trace has exactly one nonnull interval on which φ is true, and it is false elsewhere:

$$\Diamond\varphi \wedge \Box((\varphi \wedge \bigcirc\neg\varphi) \to \bigcirc\Box\neg\varphi)$$

(vii) The formula φ is true at each multiple of 4 but false elsewhere:

$$\varphi \wedge \Box(\varphi \rightarrow \bigcirc(\neg\varphi \wedge \bigcirc(\neg\varphi \wedge \bigcirc(\neg\varphi \wedge \bigcirc\varphi))))$$

The Until Operator

One useful operator that cannot be expressed is **until**. This is a binary operator written in infix (e.g., $\varphi\,\mathbf{until}\,\psi$). It says that there exists some future point t such that $t \vDash \psi$ and that all points strictly between the current state and t satisfy φ.

The operators $\bigcirc$, $\Diamond$, and $\Box$ can all be defined in terms of **until**:

$$\begin{aligned} \bigcirc\varphi &\iff \neg(\mathbf{0}\,\mathbf{until}\,\neg\varphi) \\ \Diamond\varphi &\iff \varphi \vee (\mathbf{1}\,\mathbf{until}\,\varphi) \\ \Box\varphi &\iff \varphi \wedge \neg(\mathbf{1}\,\mathbf{until}\,\neg\varphi) \end{aligned}$$

In the definition of $\bigcirc$, the subexpression $\mathbf{0}\,\mathbf{until}\,\neg\varphi$ says that some future point t satisfies $\neg\varphi$, but all points strictly between the current state and t satisfy $\mathbf{0}$ (*false*); but this can happen only if there are no intermediate states, that is, t is the next state. Thus $\mathbf{0}\,\mathbf{until}\,\neg\varphi$ says that there exists a NEXT-successor satisfying $\neg\varphi$. The definition of $\Diamond$ says that φ is true now or sometime in the future, and all intermediate points satisfy $\mathbf{1}$ (*true*).

It has been shown in Kamp (1968) and Gabbay et al. (1980) that the **until** operator is powerful enough to express anything that can be expressed in the first-order theory of $(\omega, <)$. It has also been shown in Wolper (1981, 1983) that there are very simple predicates that cannot be expressed by **until**; for example, "φ is true at every multiple of 4." Compare Example 17.3(vii) above; here, we do not say anything about whether φ is true at points that are not multiples of 4.

The **until** operator has been shown to be very useful in expressing non-input/output properties of programs such as: "If process p requests a resource before q does, then it will receive it before q does." Indeed, much of the research in TL has concentrated on providing useful methods for proving these and other kinds of properties (see Manna and Pnueli (1981); Gabbay et al. (1980)).

Concurrency and Nondeterminism

Unlike DL, TL can be applied to programs that are not normally supposed to halt, such as operating systems, because programs are interpreted as *traces* instead of pairs of states. Up to now we have only considered deterministic, single-process programs, so that for each state s, if $(s, t) \in$ NEXT then t is unique. There is no

reason however not to apply TL to *nondeterministic* and *concurrent (multiprocessor)* systems, although there is a slight problem with this, which we discuss below.

In the single-processor environment, a *state* is a pair (L_i, u), where L_i is the instruction the program is about to execute, and u is a valuation of the program variables. In a multiprocessor environment, say with n processors, a *state* is a tuple $(L_1, \ldots, L_n, u)$ where the ith process is just about to execute L_i. If s and t are states, then $(s, t) \in \textsc{Next}$ if t can be obtained from s by letting just one process p_i execute L_i while the other processes wait. Thus each s can have up to n possible next states. In a nondeterministic program, a statement

$$L_i \quad : \quad \textbf{go to } L_j \textbf{ or } L_k$$

can occur; to execute this statement, a process chooses nondeterministically to go to either L_j or L_k. Thus we can have two next states. In either of these situations, multiprocessing or nondeterminism, the computation is no longer a single trace, but many different traces are possible. We can assemble them all together to get a *computation tree* in which each node represents a state accessible from the start state.

As above, an *invariance property* is a property of the form $\Box\varphi$, which says that the property φ is preserved throughout time. Thus we should define

$$s \vDash \Box\varphi \quad \stackrel{\text{def}}{\iff} \quad t \vDash \varphi \;\text{ for every node } t \text{ in the tree below } s.$$

The problem is that the dual $\Diamond$ of the operator $\Box$ defined in this way does not really capture what we mean by *eventuality* or *liveness* properties. We would like to be able to say that *every* possible trace in the computation tree has a state satisfying φ. For instance, a nondeterministic program is *total* if there is no chance of an infinite trace out of the start state s; that is, every trace out of s satisfies $\Diamond\textbf{halt}$. The dual $\Diamond$ of $\Box$ as defined by $\Diamond\varphi = \neg\Box\neg\varphi$ does not really express this. It says instead

$$s \vDash \Diamond\varphi \quad \iff \quad \text{there is } \textit{some} \text{ node } t \text{ in the tree below } s \text{ such that } t \vDash \varphi.$$

This is not a very useful statement.

There have been several proposals to fix this. One way is to introduce a new modal operator A that says, “For all traces in the tree. . . ,” and then use $\Box$, $\Diamond$ in the sense of linear TL applied to the trace quantified by A. The dual of A is E, which says, “There exists a trace in the tree. . . .” Thus, in order to say that the computation tree starting from the current state satisfies a safety or invariance

property, we would write

$\mathsf{A}\Box\varphi$,

which says, "For all traces π out of the current state, π satisfies $\Box\varphi$," and to say that the tree satisfies an eventuality property, we would write

$\mathsf{A}\Diamond\varphi$,

which says, "For all traces π out of the current state, π satisfies $\Diamond\varphi$; that is, φ occurs somewhere along the trace π." The logic with the linear temporal operators augmented with the trace quantifiers A and E is known as CTL; see Emerson (1990); Emerson and Halpern (1986, 1985); Emerson and Lei (1987); Emerson and Sistla (1984).

An alternative approach that fits in well with PDL is to bring the programs α back into the language explicitly, only this time interpret programs as sets of traces instead of pairs of states. We could then write

$$[\alpha]\Diamond\varphi$$
$$[\alpha]\Box\varphi$$

which would mean, respectively, "For all traces π of program α, $\pi \vDash \Diamond\varphi$" and "For all traces π of α, $\pi \vDash \Box\varphi$," and these two statements would capture precisely our intuitive notion of *eventuality* and *invariance*. We discuss such a system, called *Process Logic*, below in Section 17.3.

Complexity and Deductive Completeness

A useful axiomatization of linear-time TL is given by the axioms

$$\begin{array}{rcl}
\Box(\varphi \to \psi) & \to & (\Box\varphi \to \Box\psi) \\
\Box(\varphi \wedge \psi) & \leftrightarrow & \Box\varphi \wedge \Box\psi \\
\Diamond\varphi & \leftrightarrow & \varphi \vee \bigcirc\Diamond\varphi \\
\bigcirc(\varphi \vee \psi) & \leftrightarrow & \bigcirc\varphi \vee \bigcirc\psi \\
\bigcirc(\varphi \wedge \psi) & \leftrightarrow & \bigcirc\varphi \wedge \bigcirc\psi \\
\varphi \wedge \Box(\varphi \to \bigcirc\varphi) & \to & \Box\varphi \\
\forall x\ \varphi(x) & \to & \varphi(t) \quad (t \text{ is free for } x \text{ in } \varphi) \\
\forall x\ \Box\varphi & \to & \Box\forall x\ \varphi
\end{array}$$

and rules

$$\frac{\varphi, \quad \varphi \to \psi}{\psi} \qquad \frac{\varphi}{\Box\varphi} \qquad \frac{\varphi}{\forall x\ \varphi}.$$

Compare the axioms of PDL (Axioms 5.5). The propositional fragment of this deductive system is complete for linear-time propositional TL, as shown in Gabbay et al. (1980).

Sistla and Clarke (1982) and Emerson and Halpern (1985) have shown that the validity problem for most versions of propositional TL is *PSPACE*-complete for linear structures and *EXPTIME*-complete for branching structures.

Embedding TL in DL

TL is subsumed by DL. To embed propositional TL into PDL, take an atomic program a to mean "one step of program p." In the linear model, the TL constructs $\bigcirc\varphi$, $\Box\varphi$, $\Diamond\varphi$, and $\varphi\ \mathbf{until}\ \psi$ are then expressed by $[a]\varphi$, $[a^*]\varphi$, $<a^*>\varphi$, and $<(a;\varphi?)^*;a>\psi$, respectively.

17.3 Process Logic

Dynamic Logic and Temporal Logic embody markedly different approaches to reasoning about programs. This dichotomy has prompted researchers to search for an appropriate process logic that combines the best features of both. An appropriate candidate should combine the ability to reason about programs compositionally with the ability to reason directly about the intermediate states encountered during the course of a computation.

Pratt (1979c), Parikh (1978b), Nishimura (1980), and Harel et al. (1982) all suggested increasingly more powerful propositional-level formalisms in which the basic idea is to interpret formulas in *traces* rather than in states. In particular, Harel et al. (1982) present a system called *Process Logic* (PL), which is essentially a union of TL and test-free regular PDL. That paper proves that the satisfiability problem is decidable and gives a complete finitary axiomatization.

We present here an extended version that includes tests. In order to interpret the **while** loop correctly, we also include an operator ω for infinite iteration. We allow only poor tests (see Section 10.2).

Syntactically, we have programs $\alpha, \beta, \ldots$ and propositions $\varphi, \psi, \ldots$ as in PDL. We have atomic symbols of each type and compound expressions built up from the operators $\to$, $\mathbf{0}$, ;, $\cup$, *, ? (applied to Boolean combinations of atomic formulas only), ω, and [ ]. In addition we have the temporal operators **first** and **until**. The

temporal operators are available for expressing and reasoning about trace properties, but programs are constructed compositionally as in PDL. Other operators are defined as in PDL (see Section 5.1) except for **skip**, which we handle specially below.

Semantically, both programs and propositions are interpreted as sets of traces. We start with a Kripke frame $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ as in Section 5.2, where K is a set of *states* $s, t, \ldots$ and the function $\mathfrak{m}_{\mathfrak{K}}$ interprets atomic formulas p as subsets of K and atomic programs a as binary relations on K.

A *trace* σ is a finite or infinite sequence of states

$$\sigma \quad = \quad s_0 s_1 s_2 \cdots$$

(repetitions allowed). A trace is of length n if it contains $n+1$ states; thus a single state constitutes a trace of length 0. The first state of a trace σ is denoted $\mathrm{first}(\sigma)$, and the last state (if it exists) is denoted $\mathrm{last}(\sigma)$. The state $\mathrm{last}(\sigma)$ exists iff σ is finite.

If $\sigma = s_0 s_1 \cdots s_k$ and $\tau = s_k s_{k+1} \cdots$ are traces, then the *fusion* of σ and τ is the trace

$$\sigma\tau \quad = \quad s_0 s_1 \cdots s_{k-1} s_k s_{k+1} \cdots$$

Note that s_k is written only once. The traces σ and τ cannot be fused unless σ is finite and $\mathrm{last}(\sigma) = \mathrm{first}(\tau)$. If σ is infinite, or if σ is finite but $\mathrm{last}(\sigma) \neq \mathrm{first}(\tau)$, then $\sigma\tau$ does not exist. A trace τ is a *suffix* of a trace ρ if there exists a finite trace σ such that $\rho = \sigma\tau$. It is a *proper suffix* if there exists such a σ of nonzero length. If A and B are sets of traces, we define

$$A \cdot B \quad \stackrel{\text{def}}{=} \quad \{\sigma\tau \mid \sigma \in A,\ \tau \in B\}$$
$$A \circ B \quad \stackrel{\text{def}}{=} \quad A \cdot B \ \cup \ \{\text{infinite traces in } A\}.$$

It is not hard to verify that $\cdot$ and $\circ$ are associative.

We define the interpretation of the temporal operators first. The definition is slightly different from that of Section 17.2, but the concept is similar.

For p an atomic proposition and σ a finite trace, define

$$\sigma \vDash p \quad \stackrel{\text{def}}{\iff} \quad \mathrm{last}(\sigma) \in \mathfrak{m}_{\mathfrak{K}}(p).$$

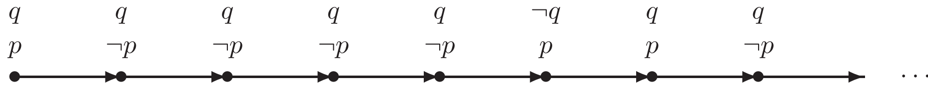
The right-hand side is given by the specification of the Kripke frame $\mathfrak{K}$. If σ is

infinite, or if σ is finite and $\mathrm{last}(\sigma) \notin \mathfrak{m}_{\mathfrak{K}}(p)$, then $\sigma \not\models p$. We also define

$$\sigma \models \mathbf{first}\,\varphi \overset{\text{def}}{\iff} \mathrm{first}(\sigma) \models \varphi$$

$$\sigma \models \varphi\,\mathbf{until}\,\psi \overset{\text{def}}{\iff} \text{there exists a proper suffix } \tau \text{ of } \sigma \text{ such that } \tau \models \psi, \text{ and for all proper suffixes } \rho \text{ of } \sigma \text{ such that } \tau \text{ is a proper suffix of } \rho,\ \rho \models \varphi.$$

The following trace satisfies $(\mathbf{first}\,(q \wedge \neg p))\,\mathbf{until}\,\mathbf{first}\,\neg q$:



As in Section 17.2, if we define

$$\begin{aligned}
\bigcirc\varphi &\overset{\text{def}}{\iff} \neg(\mathbf{0}\,\mathbf{until}\,\neg\varphi)\\
\Box\varphi &\overset{\text{def}}{\iff} \varphi \wedge \neg(\mathbf{1}\,\mathbf{until}\,\neg\varphi)\\
\Diamond\varphi &\overset{\text{def}}{\iff} \neg\Box\neg\varphi\\
&\iff \varphi \vee (\mathbf{1}\,\mathbf{until}\,\varphi),
\end{aligned}$$

then we get

$$\begin{aligned}
\sigma \models \bigcirc\varphi &\iff \text{the maximal proper suffix of } \sigma, \text{ if it exists, satisfies } \varphi,\\
\sigma \models \Box\varphi &\iff \text{all suffixes of } \sigma, \text{ proper or not, satisfy } \varphi,\\
\sigma \models \Diamond\varphi &\iff \text{there exists a suffix of } \sigma, \text{ proper or not, satisfying } \varphi.
\end{aligned}$$

Now we wish to extend the definition of $\mathfrak{m}_{\mathfrak{K}}$ to give meanings to programs. The extended meaning function $\mathfrak{m}_{\mathfrak{K}}$ will assign a set of traces to each program.

The meaning of an atomic program a is the binary relation $\mathfrak{m}_{\mathfrak{K}}(a)$ as determined by the frame $\mathfrak{K}$, considered as a set of traces of length one. We define

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{K}}(\beta)\\
\mathfrak{m}_{\mathfrak{K}}(\alpha\,;\,\beta) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\beta)\\
&= \mathfrak{m}_{\mathfrak{K}}(\alpha) \cdot \mathfrak{m}_{\mathfrak{K}}(\beta) \cup \{\text{infinite traces in } \mathfrak{m}_{\mathfrak{K}}(\alpha)\}\\
\mathfrak{m}_{\mathfrak{K}}(\alpha^*) &\overset{\text{def}}{=} \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n), \text{ where } \mathfrak{m}_{\mathfrak{K}}(\alpha^0) \overset{\text{def}}{=} K \text{ and } \mathfrak{m}_{\mathfrak{K}}(\alpha^{n+1}) \overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\alpha\alpha^n)\\
\mathfrak{m}_{\mathfrak{K}}(\alpha^\omega) &\overset{\text{def}}{=} \{\sigma_0\sigma_1\cdots \mid \sigma_n \in \mathfrak{m}_{\mathfrak{K}}(\alpha),\ n \geq 0\} \cup \{\text{infinite traces in } \mathfrak{m}_{\mathfrak{K}}(\alpha^*)\}\\
\mathfrak{m}_{\mathfrak{K}}(\varphi?) &\overset{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\varphi) \cap \{\text{traces of length } 0\}.
\end{aligned}$$

We do not define **skip** as **1**? as in PDL, but rather as the relation

$$\mathbf{skip} \stackrel{\text{def}}{=} \{(s,s) \mid s \in K\}.$$

The reason for including the ω operator is to model the **while** loop correctly. In PDL, we had

$$\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha \quad = \quad (\varphi?\,;\,\alpha)^*\,;\,\neg\varphi?$$

which was all right for binary relation semantics, since if the test φ never becomes false, there will be no output state. However, with trace semantics, such a computation would result in an infinite trace obtained by concatenating infinitely many finite α traces. This is given by α^ω and should be included in the semantics of the **while** loop. Thus for PL, we define

$$\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha \stackrel{\text{def}}{=} (\varphi?\,;\,\alpha)^*\,;\,\neg\varphi? \ \cup\ (\varphi?\,;\,\alpha)^\omega.$$

We would also like infinite traces in $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ included in $\mathfrak{m}_{\mathfrak{K}}(\alpha\,;\,\beta)$. Intuitively, such traces would result if α ran forever without terminating, thus they would also result from running $\alpha\,;\,\beta$.

For the semantics of the modal operator [ ], we define $\sigma \in \mathfrak{m}_{\mathfrak{K}}([\alpha]\varphi)$ iff either of the following two conditions holds:

(i) σ is finite, and for all traces $\tau \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$ such that $\sigma\tau$ exists, $\sigma\tau \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$; or

(ii) σ is infinite and $\sigma \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$.

Intuitively, either σ represents a finite computation and all extensions τ of σ obtained by running the program α satisfy φ; or σ is an infinite computation satisfying φ already.

The addition of clause (ii) takes care of the possibility that α does not halt. It causes the PDL axiom $[\alpha\,;\,\beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$ to be satisfied.

Axiomatization

Trace models satisfy (most of) the PDL axioms. As in Section 17.2, define

$$\begin{aligned}
\mathbf{halt} &\stackrel{\text{def}}{\iff} \bigcirc\mathbf{0}\\
\mathbf{fin} &\stackrel{\text{def}}{\iff} \Diamond\mathbf{halt}\\
\mathbf{inf} &\stackrel{\text{def}}{\iff} \neg\mathbf{fin},
\end{aligned}$$

which say that the trace is of length 0, of finite length, or of infinite length, respectively. Define two new operators $[\![\ ]\!]$ and $\ll\ \gg$:

$$[\![\alpha]\!]\varphi \quad \stackrel{\text{def}}{\iff} \quad \mathbf{fin} \to [\alpha]\varphi$$
$$\ll\alpha\gg\varphi \quad \stackrel{\text{def}}{\iff} \quad \neg[\![\alpha]\!]\neg\varphi \quad \iff \quad \mathbf{fin} \wedge <\alpha>\varphi.$$

Then

$$\mathfrak{m}_{\mathfrak{K}}([\![\alpha]\!]\varphi) \quad = \quad \{\sigma \mid \text{for all } \tau \in \mathfrak{m}_{\mathfrak{K}}(\alpha), \text{ if } \sigma\tau \text{ exists, then } \sigma\tau \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\}$$
$$\mathfrak{m}_{\mathfrak{K}}(\ll\alpha\gg\varphi) \quad = \quad \{\sigma \mid \text{there exists } \tau \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ such that } \sigma\tau \text{ exists and } \sigma\tau \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\}.$$

The operator $\ll\ \gg$ is just $<\ >$ restricted to finite traces.

By definition of $[\ ]$ and $<\ >$, the following are valid formulas of PL:

$$\begin{aligned} [\alpha]\varphi \quad &\leftrightarrow \quad (\mathbf{fin} \to [\![\alpha]\!]\varphi) \wedge (\mathbf{inf} \to \varphi) \\ &\leftrightarrow \quad (\mathbf{fin} \wedge [\![\alpha]\!]\varphi) \vee (\mathbf{inf} \wedge \varphi) \\ <\alpha>\varphi \quad &\leftrightarrow \quad (\mathbf{fin} \to \ll\alpha\gg\varphi) \wedge (\mathbf{inf} \to \varphi) \\ &\leftrightarrow \quad (\mathbf{fin} \wedge \ll\alpha\gg\varphi) \vee (\mathbf{inf} \wedge \varphi). \end{aligned}$$

First we show that the modal axioms

$$[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad ([\alpha]\varphi \wedge [\alpha]\psi) \qquad (17.3.1)$$
$$[\alpha](\varphi \to \psi) \quad \to \quad ([\alpha]\varphi \to [\alpha]\psi) \qquad (17.3.2)$$

are satisfied. To show (17.3.1), first observe that

$$[\![\alpha]\!](\varphi \wedge \psi) \quad \leftrightarrow \quad [\![\alpha]\!]\varphi \wedge [\![\alpha]\!]\psi$$

is valid. Then

$$\begin{aligned} [\alpha](\varphi \wedge \psi) \quad &\leftrightarrow \quad (\mathbf{fin} \to [\![\alpha]\!](\varphi \wedge \psi)) \wedge (\mathbf{inf} \to (\varphi \wedge \psi)) \\ &\leftrightarrow \quad (\mathbf{fin} \to ([\![\alpha]\!]\varphi \wedge [\![\alpha]\!]\psi)) \wedge (\mathbf{inf} \to (\varphi \wedge \psi)) \\ &\leftrightarrow \quad (\mathbf{fin} \to [\![\alpha]\!]\varphi) \wedge (\mathbf{fin} \to [\![\alpha]\!]\psi) \wedge (\mathbf{inf} \to \varphi) \wedge (\mathbf{inf} \to \psi) \\ &\leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi. \end{aligned}$$

To show (17.3.2), by propositional reasoning, it suffices to show

$$[\alpha](\varphi \to \psi) \wedge [\alpha]\varphi \quad \to \quad [\alpha]\psi.$$

First observe that

$$[\![\alpha]\!](\varphi \to \psi) \wedge [\![\alpha]\!]\varphi \quad \to \quad [\![\alpha]\!]\psi$$

is valid. Then

$$
\begin{aligned}
&[\alpha](\varphi \to \psi) \wedge [\alpha]\varphi \\
&\quad \leftrightarrow \quad (\mathbf{fin} \to [\![\alpha]\!](\varphi \to \psi)) \wedge (\mathbf{inf} \to (\varphi \to \psi)) \wedge (\mathbf{fin} \to [\![\alpha]\!]\varphi) \wedge (\mathbf{inf} \to \varphi) \\
&\quad \to \quad (\mathbf{fin} \to ([\![\alpha]\!]\varphi \to [\![\alpha]\!]\psi)) \wedge (\mathbf{inf} \to (\varphi \to \psi)) \wedge (\mathbf{fin} \to [\![\alpha]\!]\varphi) \wedge (\mathbf{inf} \to \varphi) \\
&\quad \leftrightarrow \quad (\mathbf{fin} \to ([\![\alpha]\!]\varphi \wedge ([\![\alpha]\!]\varphi \to [\![\alpha]\!]\psi))) \wedge (\mathbf{inf} \to (\varphi \wedge (\varphi \to \psi))) \\
&\quad \to \quad (\mathbf{fin} \to [\![\alpha]\!]\psi) \wedge (\mathbf{inf} \to \psi) \\
&\quad \leftrightarrow \quad [\alpha]\psi.
\end{aligned}
$$

The argument for the axiom

$$[\alpha \cup \beta]\varphi \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\beta]\varphi$$

is similar and uses the property

$$[\![\alpha \cup \beta]\!]\varphi \quad \leftrightarrow \quad [\![\alpha]\!]\varphi \wedge [\![\beta]\!]\varphi.$$

The axiom $[\alpha\,;\,\beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$ is obtained as follows. Suppose σ is finite. Arguing semantically, $\sigma \in \mathfrak{m}_{\mathfrak{K}}([\![\alpha\,;\,\beta]\!]\varphi)$ iff

- for all infinite α-traces τ such that $\sigma\tau$ exists, $\sigma\tau \vDash \varphi$; and
- for all finite α-traces τ such that $\sigma\tau$ exists, for all β-traces ρ such that $\sigma\tau\rho$ exists, $\sigma\tau\rho \vDash \varphi$.

Thus

$$
\begin{aligned}
[\![\alpha\,;\,\beta]\!]\varphi \quad &\leftrightarrow \quad [\![\alpha]\!](\mathbf{inf} \to \varphi) \wedge [\![\alpha]\!](\mathbf{fin} \to [\![\beta]\!]\varphi) \\
&\leftrightarrow \quad [\![\alpha]\!]((\mathbf{inf} \to \varphi) \wedge (\mathbf{fin} \to [\![\beta]\!]\varphi)) \\
&\leftrightarrow \quad [\![\alpha]\!][\beta]\varphi
\end{aligned}
$$

and

$$
\begin{aligned}
[\alpha\,;\,\beta]\varphi \quad &\leftrightarrow \quad (\mathbf{fin} \to [\![\alpha\,;\,\beta]\!]\varphi) \wedge (\mathbf{inf} \to \varphi) \\
&\leftrightarrow \quad (\mathbf{fin} \to [\![\alpha]\!][\beta]\varphi) \wedge (\mathbf{inf} \to \varphi) \\
&\leftrightarrow \quad (\mathbf{fin} \to [\![\alpha]\!][\beta]\varphi) \wedge (\mathbf{inf} \to [\beta]\varphi) \\
&\leftrightarrow \quad [\alpha][\beta]\varphi.
\end{aligned}
$$

The penultimate step uses the fact that φ and $[\beta]\varphi$ are equivalent for infinite traces.

The * operator is the same as in PDL. It can be shown that the two PDL axioms

$$\begin{aligned} \varphi \wedge [\alpha][\alpha^*]\varphi &\leftrightarrow [\alpha^*]\varphi \\ \varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) &\to [\alpha^*]\varphi \end{aligned}$$

hold by establishing that

$$\begin{aligned} \bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n) &= \mathfrak{m}_{\mathfrak{K}}(\alpha^0) \cup (\mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n)) \\ &= \mathfrak{m}_{\mathfrak{K}}(\alpha^0) \cup ((\bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha^n)) \circ \mathfrak{m}_{\mathfrak{K}}(\alpha)). \end{aligned}$$

The axiom for the test operator ? is not quite the same as in PDL. The PDL axiom $[\psi?]\varphi \leftrightarrow (\psi \to \varphi)$ is valid only for weak tests and finite traces. If either one of these restrictions is lifted, then the formula is no longer valid. Instead we postulate

$$(\mathbf{fin} \to ([\psi?]\varphi \leftrightarrow (\psi \to \varphi))) \quad \wedge \quad (\mathbf{inf} \to ([\psi?]\varphi \leftrightarrow \varphi)) \tag{17.3.3}$$

for weak tests only.

In our formulation, tests are instantaneous. One may argue that this interferes with the semantics of programs such as **while 1 do** φ?, which rightfully should generate an infinite trace but does not. This suggests an alternative approach in which tests would be interpreted as binary relations (traces of length one). However, the latter approach is even more problematic. For one thing, it is not clear how to axiomatize $[\psi?]\varphi$; certainly (17.3.3) is no longer valid. Since we can assert the length of a trace, our axiomatization would be encumbered by such irrelevancies as **length** $17 \to [\mathbf{1}?]$**length** 18. Worse, Boolean algebra would no longer be readily available, at least in any simple form. For example, $\varphi? \,;\, \varphi?$ and $\varphi?$ would no longer be equivalent. We thus prefer the formulation we have given. Note, however, that if we restrict programs to ordinary **while** programs in which φ? must occur in the test of a conditional or while statement, then pathelogical programs such as **while 1 do** φ? are disallowed, and all is well. The program **while 1 do skip** generates an infinite trace because of the redefinition of **skip**.

Finally, what can we say about ω? One property that is certain is

$$\begin{aligned} \mathfrak{m}_{\mathfrak{K}}(\alpha^\omega) &= \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\alpha^\omega) \\ &= \mathfrak{m}_{\mathfrak{K}}(\alpha^\omega) \circ \mathfrak{m}_{\mathfrak{K}}(\alpha), \end{aligned}$$

which leads to the axioms

$$\begin{aligned} [\alpha^\omega]\varphi &\leftrightarrow [\alpha\alpha^\omega]\varphi \\ &\leftrightarrow [\alpha^\omega\alpha]\varphi. \end{aligned}$$

One might expect the formula $[\alpha^\omega]\mathbf{inf}$ to be valid, but this is not the case. For example, $\mathfrak{m}_{\mathfrak{K}}(\mathbf{1}?^\omega)$ contains all and only traces of length 0. However, if any trace $\sigma \in \mathfrak{m}_{\mathfrak{K}}(\alpha^\omega)$ has a *state* satisfying φ—that is, if σ has a suffix satisfying $\mathbf{first}\,\varphi$—then some prefix of σ in $\mathfrak{m}_{\mathfrak{K}}(\alpha^*)$ also has this property. Thus

$$[\alpha^*]\Diamond\mathbf{first}\,\varphi \quad \rightarrow \quad [\alpha^\omega]\Diamond\mathbf{first}\,\varphi \tag{17.3.4}$$

is valid. We cannot replace $\mathbf{first}\,\varphi$ by an arbitrary property ψ; for instance, (17.3.4) does not necessarily hold for $\psi = \mathbf{inf}$.

As mentioned, the version of PL of Harel et al. (1982) is decidable (but, it seems, in nonelementary time only) and complete. It has also been shown that if we restrict the semantics to include only finite traces (not a necessary restriction for obtaining the results above), then PL is no more expressive than PDL. Translations of PL structures into PDL structures have also been investigated, making possible an elementary time decision procedure for deterministic PL; see Halpern (1982, 1983). An extension of PL in which **first** and **until** are replaced by regular operators on formulas has been shown to be decidable but nonelementary in Harel et al. (1982). This logic perhaps comes closer to the desired objective of a powerful decidable logic of traces with natural syntactic operators that is closed under attachment of regular programs to formulas.

First-order PL has not been properly investigated yet, perhaps because the "right" logic has not yet been agreed upon. It is also not quite clear yet whether the PL approach has pragmatic advantages over TL in reasoning about concurrent programs. The exact relationship of PL with the second order theory of n successors (see Rabin (1969)), to which the validity problem is reduced for obtaining decidability, seems also worthy of further study.

17.4 The μ-Calculus

The μ-calculus was suggested as a formalism for reasoning about programs in Scott and de Bakker (1969) and was further developed in Hitchcock and Park (1972), Park (1976), and de Bakker (1980).

The heart of the approach is μ, the *least fixpoint* operator, which captures the notions of iteration and recursion. The calculus was originally defined as a first-

order-level formalism, but propositional versions have become popular.

The μ operator binds relation variables. If $\varphi(X)$ is a logical expression with a free relation variable X, then the expression $\mu X.\varphi(X)$represents the least X such that $\varphi(X) = X$, if such an X exists. For example, the reflexive transitive closure R^* of a binary relation R is the least binary relation containing R and closed under reflexivity and transitivity; this would be expressed in the first-order μ-calculus as

$$R^* \stackrel{\text{def}}{=} \mu X(x,y).(x = y \vee \exists z\ (R(x,z) \wedge X(z,y))). \tag{17.4.1}$$

This should be read as, "the least binary relation $X(x,y)$ such that either $x = y$ or x is related by R to some z such that z and y are already related by X." This captures the usual fixpoint formulation of reflexive transitive closure (Section 1.7). The formula (17.4.1) can be regarded either as a recursive program computing R^* or as an inductively defined assertion that is true of a pair (x,y) iff that pair is in the reflexive transitive closure of R.

The existence of a least fixpoint is not guaranteed except under certain restrictions. Indeed, the formula $\neg X$ has no fixpoint, therefore $\mu X.\neg X$ does not exist. Typically, one restricts the application of the binding operator μX to formulas that are *positive* or *syntactically monotone* in X; that is, those formulas in which every free occurrence of X occurs in the scope of an even number of negations. This implies that the relation operator $X \mapsto \varphi(X)$ is (semantically) monotone in the sense of Section 1.7, which by the Knaster–Tarski theorem (Theorem 1.12) ensures the existence of a least fixpoint.

The first-order μ-calculus can define all sets definable by first-order induction and more. In particular, it can capture the input/output relation of any program built from any of the DL programming constructs we have discussed. Since the first-order μ-calculus also admits first-order quantification, it is easily seen to be as powerful as DL.

It was shown by Park (1976) that finiteness is not definable in the first-order μ-calculus with the monotonicity restriction, but well-foundedness is. Thus this version of the μ-calculus is independent of $L_{\omega_1^{\mathrm{ck}}\omega}$ (and hence of DL(r.e.)) in expressive power. Well-foundedness of a binary relation R can be written

$$\forall x\ (\mu X(x).\forall y\ (R(y,x) \rightarrow X(y))).$$

A more severe syntactic restriction on the binding operator μX is to allow its application only to formulas that are *syntactically continuous* in X; that is, those formulas in which X does not occur free in the scope of any negation or any universal quantifier. It can be shown that this syntactic restriction implies semantic

continuity (Section 1.7), so the least fixpoint is the union of $\varnothing$, $\varphi(\varnothing)$, $\varphi(\varphi(\varnothing)), \ldots$. As shown in Park (1976), this version is strictly weaker than $L_{\omega_1^{\mathrm{ck}}\omega}$.

In Pratt (1981a) and Kozen (1982, 1983), propositional versions of the μ-calculus were introduced. The latter version consists of propositional modal logic with a least fixpoint operator. It is the most powerful logic of its type, subsuming all known variants of PDL, game logic of Parikh (1983), various forms of temporal logic (see Section 17.2), and other seemingly stronger forms of the μ-calculus (Vardi and Wolper (1986c)). In the following presentation we focus on this version, since it has gained fairly widespread acceptance; see Kozen (1984); Kozen and Parikh (1983); Streett (1985a); Streett and Emerson (1984); Vardi and Wolper (1986c); Walukiewicz (1993, 1995, 2000); Stirling (1992); Mader (1997); Kaivola (1997).

The language of the propositional μ-calculus, also called the *modal μ-calculus*, is syntactically simpler than PDL. It consists of the usual propositional constructs $\rightarrow$ and $\mathbf{0}$, atomic modalities $[a]$, and the least fixpoint operator μ. A greatest fixpoint operator dual to μ can be defined:

$$\nu X.\varphi(X) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \neg\mu X.\neg\varphi(\neg X).$$

Variables are monadic, and the μ operator may be applied only to syntactically monotone formulas. As discussed above, this ensures monotonicity of the corresponding set operator. The language is interpreted over Kripke frames in which atomic propositions are interpreted as sets of states and atomic programs are interpreted as binary relations on states.

The propositional μ-calculus subsumes PDL. For example, the PDL formula $<a^*>\varphi$ for atomic a can be written $\mu X.(\varphi \vee <a>X)$. The formula $\mu X.<a>[a]X$, which expresses the existence of a forced win for the first player in a two-player game, and the formula $\mu X.[a]X$, which expresses well-foundedness and is equivalent to $\mathbf{wf}\, a$ (see Section 10.6), are both inexpressible in PDL, as shown in Streett (1981); Kozen (1981c). Niwinski (1984) has shown that even with the addition of the **halt** construct, PDL is strictly less expressive than the μ-calculus.

The propositional μ-calculus satisfies a finite model theorem, as first shown in Kozen (1988). Decidability results were obtained in Kozen and Parikh (1983); Vardi and Stockmeyer (1985); Vardi (1985b), culminating in a deterministic exponential-time algorithm of Emerson and Jutla (1988) based on an automata-theoretic lemma of Safra (1988). Since the μ-calculus subsumes PDL, it is *EXPTIME*-complete.

In Kozen (1982, 1983), an axiomatization of the propositional μ-calculus was proposed and conjectured to be complete. The axiomatization consists of the axioms

and rules of propositional modal logic, plus the axiom

$$\varphi[X/\mu X.\varphi] \quad \rightarrow \quad \mu X.\varphi$$

and rule

$$\frac{\varphi[X/\psi] \ \rightarrow \ \psi}{\mu X.\varphi \ \rightarrow \ \psi}$$

for μ. Completeness of this deductive system for a syntactically restricted subset of formulas was shown in Kozen (1982, 1983). Completeness for the full language was proved by Walukiewicz (1995, 2000). This was quickly followed by simpler alternative proofs by Ambler et al. (1995); Bonsangue and Kwiatkowska (1995); Hartonas (1998). Bradfield (1996) showed that the alternating μ/ν hierarchy (least/greatest fixpoints) is strict. An interesting open question is the complexity of *model checking*: does a given formula of the propositional μ-calculus hold in a given state of a given Kripke frame? Although some progress has been made (see Bhat and Cleaveland (1996); Cleaveland (1996); Emerson and Lei (1986); Sokolsky and Smolka (1994); Stirling and Walker (1989)), it is still unknown whether this problem has a polynomial-time algorithm.

The propositional μ-calculus has become a popular system for the specification and verification of properties of transition systems, where it has had some practical impact (Steffen et al. (1996)). Several recent papers on model checking work in this context; see Bhat and Cleaveland (1996); Cleaveland (1996); Emerson and Lei (1986); Sokolsky and Smolka (1994); Stirling and Walker (1989). A comprehensive introduction can be found in Stirling (1992).

17.5 Kleene Algebra

Kleene algebra (KA) is the algebra of regular expressions. It is named for S. C. Kleene (1909–1994), who among his many other achievements invented regular expressions and proved their equivalence to finite automata in Kleene (1956).

Kleene algebra has appeared in various guises and under many names in relational algebra (Ng (1984); Ng and Tarski (1977)), semantics and logics of programs (Kozen (1981b); Pratt (1988)), automata and formal language theory (Kuich (1987); Kuich and Salomaa (1986)), and the design and analysis of algorithms (Aho et al. (1975); Tarjan (1981); Mehlhorn (1984); Iwano and Steiglitz (1990); Kozen (1991b)). As discussed in Section 16.4, Kleene algebra plays a prominent role in dynamic algebra as an algebraic model of program behavior.

Beginning with the monograph of Conway (1971), many authors have con-

tributed over the years to the development of the algebraic theory; see Backhouse (1975); Krob (1991); Kleene (1956); Kuich and Salomaa (1986); Sakarovitch (1987); Kozen (1990); Bloom and Ésik (1992); Hopkins and Kozen (1999). See also Kozen (1996) for further references.

A *Kleene algebra* is an algebraic structure $(K, +, \cdot, ^*, 0, 1)$ satisfying the axioms

$$
\begin{aligned}
\alpha + (\beta + \gamma) &= (\alpha + \beta) + \gamma \\
\alpha + \beta &= \beta + \alpha \\
\alpha + 0 &= \alpha + \alpha = \alpha \\
\alpha(\beta\gamma) &= (\alpha\beta)\gamma \\
1\alpha &= \alpha 1 = \alpha \\
\alpha(\beta + \gamma) &= \alpha\beta + \alpha\gamma \\
(\alpha + \beta)\gamma &= \alpha\gamma + \beta\gamma \\
0\alpha &= \alpha 0 = 0 \\
1 + \alpha\alpha^* &= 1 + \alpha^*\alpha = \alpha^* && (17.5.1) \\
\beta + \alpha\gamma \le \gamma &\rightarrow \alpha^*\beta \le \gamma && (17.5.2) \\
\beta + \gamma\alpha \le \gamma &\rightarrow \beta\alpha^* \le \gamma && (17.5.3)
\end{aligned}
$$

where $\le$ refers to the natural partial order on K:

$$\alpha \le \beta \quad \stackrel{\text{def}}{\iff} \quad \alpha + \beta = \beta.$$

In short, a KA is an idempotent semiring under $+$, $\cdot$, 0, 1 satisfying (17.5.1)–(17.5.3) for *. The axioms (17.5.1)–(17.5.3) say essentially that * behaves like the asterate operator on sets of strings or reflexive transitive closure on binary relations. This particular axiomatization is from Kozen (1991a, 1994a), but there are other competing ones.

The axioms (17.5.2) and (17.5.3) correspond to the reflexive transitive closure rule (RTC) of PDL (Section 5.6). Instead, we might postulate the equivalent axioms

$$
\begin{aligned}
\alpha\gamma \le \gamma &\rightarrow \alpha^*\gamma \le \gamma && (17.5.4) \\
\gamma\alpha \le \gamma &\rightarrow \gamma\alpha^* \le \gamma, && (17.5.5)
\end{aligned}
$$

which correspond to the loop invariance rule (LI). The induction axiom (IND) is inexpressible in KA, since there is no negation.

A Kleene algebra is *-*continuous* if it satisfies the infinitary condition

$$\alpha\beta^*\gamma = \sup_{n\ge 0} \alpha\beta^n\gamma \qquad (17.5.6)$$

where

$$\beta^0 \stackrel{\text{def}}{=} 1 \qquad \beta^{n+1} \stackrel{\text{def}}{=} \beta\beta^n$$

and where the supremum is with respect to the natural order $\leq$. We can think of (17.5.6) as a conjunction of the infinitely many axioms $\alpha\beta^n\gamma \leq \alpha\beta^*\gamma$, $n \geq 0$, and the infinitary Horn formula

$$(\bigwedge_{n\geq 0} \alpha\beta^n\gamma \leq \delta) \quad \rightarrow \quad \alpha\beta^*\gamma \leq \delta.$$

In the presence of the other axioms, the *-continuity condition (17.5.6) implies (17.5.2)–(17.5.5) and is strictly stronger in the sense that there exist Kleene algebras that are not *-continuous (Kozen (1990)).

The fundamental motivating example of a Kleene algebra is the family of regular sets of strings over a finite alphabet, but other classes of structures share the same equational theory, notably the binary relations on a set. In fact it is the latter interpretation that makes Kleene algebra a suitable choice for modeling programs in dynamic algebras. Other more unusual interpretations are the min, + algebra used in shortest path algorithms (see Aho et al. (1975); Tarjan (1981); Mehlhorn (1984); Kozen (1991b)) and KAs of convex polyhedra used in computational geometry as described in Iwano and Steiglitz (1990).

Axiomatization of the equational theory of the regular sets is a central question going back to the original paper of Kleene (1956). A completeness theorem for relational algebras was given in an extended language by Ng (1984); Ng and Tarski (1977). Axiomatization is a central focus of the monograph of Conway (1971), but the bulk of his treatment is infinitary. Redko (1964) proved that there is no finite equational axiomatization. Schematic equational axiomatizations for the algebra of regular sets, necessarily representing infinitely many equations, have been given by Krob (1991) and Bloom and Ésik (1993). Salomaa (1966) gave two finitary complete axiomatizations that are sound for the regular sets but not sound in general over other standard interpretations, including relational interpretations. The axiomatization given above is a finitary universal Horn axiomatization that is sound and complete for the equational theory of standard relational and language-theoretic models, including the regular sets (Kozen (1991a, 1994a)). Other work on completeness appears in Krob (1991); Boffa (1990, 1995); Archangelsky (1992).

The literature contains a bewildering array of inequivalent definitions of Kleene algebras and related algebraic structures; see Conway (1971); Pratt (1988, 1990); Kozen (1981b, 1991a); Aho et al. (1975); Mehlhorn (1984); Kuich (1987); Kozen (1994b). As demonstrated in Kozen (1990), many of these are strongly related.

One important property shared by most of them is closure under the formation of $n \times n$ matrices. This was proved for the axiomatization of Section 16.4 in Kozen (1991a, 1994a), but the idea essentially goes back to Kleene (1956); Conway (1971); Backhouse (1975). This result gives rise to an algebraic treatment of finite automata in which the automata are represented by their transition matrices.

The equational theory of Kleene algebra is *PSPACE*-complete (Stockmeyer and Meyer (1973)); thus it is apparently less complex than PDL, which is *EXPTIME*-complete (Theorem 8.5), although the strict separation of the two complexity classes is still open.

Kleene Algebra with Tests

From a practical standpoint, many simple program manipulations such as loop unwinding and basic safety analysis do not require the full power of PDL, but can be carried out in a purely equational subsystem using the axioms of Kleene algebra. However, *tests* are an essential ingredient, since they are needed to model conventional programming constructs such as conditionals and **while** loops and to handle assertions. This motivates the definition of the following variant of KA introduced in Kozen (1996, 1997b).

A *Kleene algebra with tests* (KAT) is a Kleene algebra with an embedded Boolean subalgebra. Formally, it is a two-sorted algebra

$$(K,\ B,\ +,\ \cdot,\ {}^*,\ \bar{},\ 0,\ 1)$$

such that

- $(K,\ +,\ \cdot,\ {}^*,\ 0,\ 1)$ is a Kleene algebra
- $(B,\ +,\ \cdot,\ \bar{},\ 0,\ 1)$ is a Boolean algebra
- $B \subseteq K$.

The unary negation operator $\bar{}$ is defined only on B. Elements of B are called *tests* and are written $\varphi, \psi, \ldots$. Elements of K (including elements of B) are written $\alpha, \beta, \ldots$. In PDL, a test would be written $\varphi?$, but in KAT we dispense with the symbol ?.

This deceptively concise definition actually carries a lot of information. The operators $+, \cdot, 0, 1$ each play two roles: applied to arbitrary elements of K, they refer to nondeterministic choice, composition, fail, and skip, respectively; and applied to tests, they take on the additional meaning of Boolean disjunction, conjunction, falsity, and truth, respectively. These two usages do not conflict—for example, sequential testing of two tests is the same as testing their conjunction—and their coexistence admits considerable economy of expression.

For applications in program verification, the standard interpretation would be a Kleene algebra of binary relations on a set and the Boolean algebra of subsets of the identity relation. One could also consider trace models, in which the Kleene elements are sets of traces (sequences of states) and the Boolean elements are sets of states (traces of length 0). As with KA, one can form the algebra $n \times n$ matrices over a KAT (K, B); the Boolean elements of this structure are the diagonal matrices over B.

KAT can express conventional imperative programming constructs such as conditionals and while loops as in PDL. It can perform elementary program manipulation such as loop unwinding, constant propagation, and basic safety analysis in a purely equational manner. The applicability of KAT and related equational systems in practical program verification has been explored in Cohen (1994a,b,c); Kozen (1996); Kozen and Patron (2000).

There is a language-theoretic model that plays the same role in KAT that the regular sets play in KA, namely the algebra of regular sets of *guarded strings*, and a corresponding completeness result was obtained by Kozen and Smith (1996). Moreover, KAT is complete for the equational theory of relational models, as shown in Kozen and Smith (1996). Although less expressive than PDL, KAT is also apparently less difficult to decide: it is *PSPACE*-complete, the same as KA, as shown in Cohen et al. (1996).

In Kozen (1999a), it is shown that KAT subsumes propositional Hoare Logic in the following sense. The partial correctness assertion $\{\varphi\}\,\alpha\,\{\psi\}$ is encoded in KAT as the equation $\varphi\alpha\overline{\psi} = 0$, or equivalently $\varphi\alpha = \varphi\alpha\psi$. If a rule

$$\frac{\{\varphi_1\}\,\alpha_1\,\{\psi_1\},\ \ldots,\ \{\varphi_n\}\,\alpha_n\,\{\psi_n\}}{\{\varphi\}\,\alpha\,\{\psi\}}$$

is derivable in propositional Hoare Logic, then its translation, the universal Horn formula

$$\varphi_1\alpha_1\overline{\psi}_1 = 0 \land \cdots \land \varphi_n\alpha_n\overline{\psi}_n = 0 \quad \to \quad \varphi\alpha\overline{\psi} = 0,$$

is a theorem of KAT. For example, the **while** rule of Section 4.4 becomes

$$\sigma\varphi\alpha\overline{\varphi} = 0 \quad \to \quad \varphi(\sigma\alpha)^*\overline{\sigma}\,\overline{\overline{\sigma}\varphi} = 0.$$

More generally, all relationally valid Horn formulas of the form

$$\gamma_1 = 0 \land \cdots \land \gamma_n = 0 \quad \to \quad \alpha = \beta$$

are theorems of KAT (Kozen (1999a)).

Horn formulas are important from a practical standpoint. For example, com-

mutativity conditions are used to model the idea that the execution of certain instructions does not affect the result of certain tests. In light of this, the complexity of the universal Horn theory of KA and KAT are of interest. There are both positive and negative results. It is shown in Kozen (1997c) that for a Horn formula $\Phi \to \varphi$ over *-continuous Kleene algebras,

- if Φ contains only commutativity conditions $\alpha\beta = \beta\alpha$, the universal Horn theory is Π^0_1-complete;
- if Φ contains only monoid equations, the problem is Π^0_2-complete;
- for arbitrary finite sets of equations Φ, the problem is Π^1_1-complete.

On the other hand, commutativity assumptions of the form $\alpha\varphi = \varphi\alpha$, where φ is a test, and assumptions of the form $\gamma = 0$ can be eliminated without loss of efficiency, as shown in Cohen (1994a); Kozen and Smith (1996). Note that assumptions of this form are all we need to encode Hoare Logic as described above.

In typed Kleene algebra introduced in Kozen (1998, 1999b), elements have types $s \to t$. This allows Kleene algebras of nonsquare matrices, among other applications. It is shown in Kozen (1999b) that Hoare Logic is subsumed by the type calculus of typed KA augmented with a typecast or coercion rule for tests. Thus Hoare-style reasoning with partial correctness assertions reduces to typechecking in a relatively simple type system.

References

Abrahamson, K. (1980). *Decidability and expressiveness of logics of processes*. Ph. D. thesis, Univ. of Washington.

Adian, S. I. (1979). *The Burnside Problem and Identities in Groups*. Springer-Verlag.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1975). *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley.

Ambler, S., M. Kwiatkowska, and N. Measor (1995, November). Duality and the completeness of the modal μ-calculus. *Theor. Comput. Sci. 151*(1), 3–27.

Andréka, H., I. Németi, and I. Sain (1982a). A complete logic for reasoning about programs via nonstandard model theory, part I. *Theor. Comput. Sci. 17*, 193–212.

Andréka, H., I. Németi, and I. Sain (1982b). A complete logic for reasoning about programs via nonstandard model theory, part II. *Theor. Comput. Sci. 17*, 259–278.

Apt, K. R. (1981). Ten years of Hoare's logic: a survey—part I. *ACM Trans. Programming Languages and Systems 3*, 431–483.

Apt, K. R. and E.-R. Olderog (1991). *Verification of Sequential and Concurrent Programs*. Springer-Verlag.

Apt, K. R. and G. Plotkin (1986). Countable nondeterminism and random assignment. *J. Assoc. Comput. Mach. 33*, 724–767.

Archangelsky, K. V. (1992). A new finite complete solvable quasiequational calculus for algebra of regular languages. Manuscript, Kiev State University.

Arnold, A. (1997a). An initial semantics for the μ-calculus on trees and Rabin's complementation lemma. Technical report, University of Bordeaux.

Arnold, A. (1997b). The μ-calculus on trees and Rabin's complementation theorem. Technical report, University of Bordeaux.

Backhouse, R. C. (1975). *Closure Algorithms and the Star-Height Problem of Regular Languages*. Ph. D. thesis, Imperial College, London, U.K.

Backhouse, R. C. (1986). *Program Construction and Verification*. Prentice-Hall.

Baier, C. and M. Kwiatkowska (1998, April). On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters 66*(2), 71–79.

Banachowski, L., A. Kreczmar, G. Mirkowska, H. Rasiowa, and A. Salwicki (1977). An introduction to algorithmic logic: metamathematical investigations in the theory of programs. In Mazurkiewitz and Pawlak (Eds.), *Math. Found. Comput. Sci.*, pp. 7–99. Banach Center, Warsaw.

Barwise, J. (1975). *Admissible Sets and Structures*. North-Holland.

Bell, J. S. and A. B. Slomson (1971). *Models and Ultraproducts*. North Holland.

Ben-Ari, M., J. Y. Halpern, and A. Pnueli (1982). Deterministic propositional dynamic logic: finite models, complexity and completeness. *J. Comput. Syst. Sci. 25*, 402–417.

Berman, F. (1978). Expressiveness hierarchy for PDL with rich tests. Technical Report 78-11-01, Comput. Sci. Dept., Univ. of Washington.

Berman, F. (1979). A completeness technique for D-axiomatizable semantics. In *Proc. 11th Symp. Theory of Comput.*, pp. 160–166. ACM.

Berman, F. (1982). Semantics of looping programs in propositional dynamic logic. *Math. Syst. Theory 15*, 285–294.

Berman, F. and M. Paterson (1981). Propositional dynamic logic is weaker without tests. *Theor. Comput. Sci. 16*, 321–328.

Berman, P., J. Y. Halpern, and J. Tiuryn (1982). On the power of nondeterminism in dynamic logic. In Nielsen and Schmidt (Eds.), *Proc 9th Colloq. Automata Lang. Prog.*, Volume 140 of *Lect. Notes in Comput. Sci.*, pp. 48–60. Springer-Verlag.

Bhat, G. and R. Cleaveland (1996, March). Efficient local model checking for fragments of the modal μ-calculus. In T. Margaria and B. Steffen (Eds.), *Proc. Second Int. Workshop Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS'96), Volume 1055 of *Lect. Notes in Comput. Sci.*, pp. 107–112. Springer-Verlag.

Birkhoff, G. (1935). On the structure of abstract algebras. *Proc. Cambridge Phil. Soc. 31*, 433–454.

Birkhoff, G. (1973). *Lattice Theory* (third ed.). American Mathematical Society.

Bloom, S. L. and Z. Ésik (1992). Program correctness and matricial iteration theories. In *Proc. Mathematical Foundations of Programming Semantics, 7th Int. Conf.*, Volume 598 of *Lecture Notes in Computer Science*, pp. 457–476. Springer-Verlag.

Bloom, S. L. and Z. Ésik (1993). Equational axioms for regular sets. *Math. Struct. Comput. Sci. 3*, 1–24.

Blute, R., J. Desharnais, A. Edelat, and P. Panangaden (1997). Bisimulation for labeled Markov processes. In *Proc. 12th Symp. Logic in Comput. Sci.*, pp. 149–158. IEEE.

Boffa, M. (1990). Une remarque sur les systèmes complets d'identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications 24*(4), 419–423.

Boffa, M. (1995). Une condition impliquant toutes les identités rationnelles. *Informatique Théoretique et Applications/Theoretical Informatics and Applications 29*(6), 515–518.

Bonsangue, M. and M. Kwiatkowska (1995, August). Re-interpreting the modal μ-calculus. In A. Ponse, M. van Rijke, and Y. Venema (Eds.), *Modal Logic and Process Algebra*, pp. 65–83. CSLI Lecture Notes.

Boole, G. (1847). *The Mathematical Analysis of Logic.* MacMillan, Barclay and MacMillan, Cambridge.

Börger, E. (1984). Spectralproblem and completeness of logical decision problems. In G. H. E. Börger and D. Rödding (Eds.), *Logic and Machines: Decision Problems and Complexity, Proccedings*, Volume 171 of *Lect. Notes in Comput. Sci.*, pp. 333–356. Springer-Verlag.

Bradfield, J. C. (1996). The modal μ-calculus alternation hierarchy is strict. In U. Montanari and V. Sassone (Eds.), *Proc. CONCUR'96*, Volume 1119 of *Lect. Notes in Comput. Sci.*, pp. 233–246. Springer.

Burstall, R. M. (1974). Program proving as hand simulation with a little induction. *Information Processing*, 308–312.

Chandra, A., D. Kozen, and L. Stockmeyer (1981). Alternation. *J. Assoc. Comput. Mach. 28*(1), 114–133.

Chang, C. C. and H. J. Keisler (1973). *Model Theory.* North-Holland.

Chellas, B. F. (1980). *Modal Logic: An Introduction.* Cambridge University Press.

Clarke, E. M. (1979). Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. Assoc. Comput. Mach. 26*, 129–147.

Cleaveland, R. (1996, July). Efficient model checking via the equational μ-calculus. In *Proc. 11th Symp. Logic in Comput. Sci.*, pp. 304–312. IEEE.

Cohen, E. (1994a, April). Hypotheses in Kleene algebra. Available as ftp://ftp.bellcore.com/pub/ernie/research/homepage.html.

Cohen, E. (1994b). Lazy caching. Available as ftp://ftp.bellcore.com/pub/ernie/research/homepage.html.

Cohen, E. (1994c). Using Kleene algebra to reason about concurrency control. Available as ftp://ftp.bellcore.com/pub/ernie/research/homepage.html.

Cohen, E., D. Kozen, and F. Smith (1996, July). The complexity of Kleene algebra with tests. Technical Report 96-1598, Computer Science Department, Cornell University.

Constable, R. L. (1977, May). On the theory of programming logics. In *Proc. 9th Symp. Theory of Comput.*, pp. 269–285. ACM.

Constable, R. L. and M. O'Donnell (1978). *A Programming Logic.* Winthrop.

Conway, J. H. (1971). *Regular Algebra and Finite Machines*. London: Chapman and Hall.

Cook, S. A. (1971). The complexity of theorem proving procedures. In *Proc. Third Symp. Theory of Computing*, New York, pp. 151–158. Assoc. Comput. Mach.

Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM J. Comput. 7*, 70–80.

Courcoubetis, C. and M. Yannakakis (1988, October). Verifying temporal properties of finite-state probabilistic programs. In *Proc. 29th Symp. Foundations of Comput. Sci.*, pp. 338–345. IEEE.

Cousot, P. (1990). Methods and logics for proving programs. In J. van Leeuwen (Ed.), *Handbood of Theoretical Computer Science*, Volume B, pp. 841–993. Amsterdam: Elsevier.

Csirmaz, L. (1985). A completeness theorem for dynamic logic. *Notre Dame J. Formal Logic 26*, 51–60.

Davis, M. D., R. Sigal, and E. J. Weyuker (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press.

de Bakker, J. (1980). *Mathematical Theory of Program Correctness*. Prentice-Hall.

Ehrenfeucht, A. (1961). An application of games in the completeness problem for formalized theories. *Fund. Math. 49*, 129–141.

Emerson, E. A. (1985). Automata, tableax, and temporal logics. In R. Parikh (Ed.), *Proc. Workshop on Logics of Programs*, Volume 193 of *Lect. Notes in Comput. Sci.*, pp. 79–88. Springer-Verlag.

Emerson, E. A. (1990). Temporal and modal logic. In J. van Leeuwen (Ed.), *Handbook of theoretical computer science*, Volume B: formal models and semantics, pp. 995–1072. Elsevier.

Emerson, E. A. and J. Y. Halpern (1985). Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci. 30*(1), 1–24.

Emerson, E. A. and J. Y. Halpern (1986). "Sometimes" and "not never" revisited: on branching vs. linear time temporal logic. *J. ACM 33*(1), 151–178.

Emerson, E. A. and C. Jutla (1988, October). The complexity of tree automata and logics of programs. In *Proc. 29th Symp. Foundations of Comput. Sci.*, pp. 328–337. IEEE.

Emerson, E. A. and C. Jutla (1989, June). On simultaneously determinizing and complementing ω-automata. In *Proc. 4th Symp. Logic in Comput. Sci.* IEEE.

Emerson, E. A. and C.-L. Lei (1986, June). Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st Symp. Logic in Comput. Sci.*, pp. 267–278. IEEE.

Emerson, E. A. and C. L. Lei (1987). Modalities for model checking: branching time strikes back. *Sci. Comput. Programming 8*, 275–306.

Emerson, E. A. and P. A. Sistla (1984). Deciding full branching-time logic. *Infor. and Control 61*, 175–201.

Engeler, E. (1967). Algorithmic properties of structures. *Math. Syst. Theory 1*, 183–195.

Engelfriet, J. (1983). Iterated pushdown automata and complexity classes. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, Boston, Massachusetts, pp. 365–373.

Erimbetov, M. M. (1981). On the expressive power of programming logics. In *Proc. Alma-Ata Conf. Research in Theoretical Programming*, pp. 49–68. In Russian.

Feldman, Y. A. (1984). A decidable propositional dynamic logic with explicit probabilities. *Infor. and Control 63*, 11–38.

Feldman, Y. A. and D. Harel (1984). A probabilistic dynamic logic. *J. Comput. Syst. Sci. 28*, 193–215.

Ferman, A. and D. Harel (2000). In preparation.

Fischer, M. J. and R. E. Ladner (1977). Propositional modal logic of programs. In *Proc. 9th Symp. Theory of Comput.*, pp. 286–294. ACM.

Fischer, M. J. and R. E. Ladner (1979). Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci. 18*(2), 194–211.

Fischer, P. C. (1966). Turing machines with restricted memory access. *Information and Control 9*(4), 364–379.

Fischer, P. C., A. R. Meyer, and A. L. Rosenberg (1968). Counter machines and counter languages. *Math. Systems Theory 2*(3), 265–283.

Floyd, R. W. (1967). Assigning meanings to programs. In *Proc. Symp. Appl. Math.*, Volume 19, pp. 19–31. AMS.

Friedman, H. (1971). Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. In Gandy and Yates (Eds.), *Logic Colloq. 1969*, pp. 361–390. North-Holland.

Gabbay, D. (1977). Axiomatizations of logics of programs. Unpublished.

Gabbay, D., I. Hodkinson, and M. Reynolds (1994). *Temporal Logic: Mathematical Foundations and Computational Aspects*. Oxford University Press.

Gabbay, D., A. Pnueli, S. Shelah, and J. Stavi (1980). On the temporal analysis of fairness. In *Proc. 7th Symp. Princip. Prog. Lang.*, pp. 163–173. ACM.

Garey, M. R. and D. S. Johnson (1979). *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman.

Gödel, K. (1930). Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatsh. Math. Phys. 37*, 349–360.

Goldblatt, R. (1982). *Axiomatising the Logic of Computer Programming*, Volume 130 of *Lect. Notes in Comput. Sci.* Springer-Verlag.

Goldblatt, R. (1987). Logics of time and computation. Technical Report Lect. Notes 7, Center for the Study of Language and Information, Stanford Univ.

Graham, R., D. Knuth, and O. Patashnik (1989). *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley.

Grätzer, G. (1978). *Universal Algebra*. Springer-Verlag.

Greibach, S. (1975). *Theory of Program Structures: Schemes, Semantics, Verification*, Volume 36 of *Lecture Notes in Computer Science*. Springer Verlag.

Gries, D. (1981). *The Science of Programming*. Springer-Verlag.

Gries, D. and F. B. Schneider (1994). *A Logical Approach to Discrete Math*. Springer-Verlag. Third printing.

Gurevich, Y. (1983). Algebras of feasible functions. In *24-th IEEE Annual Symposium on Foundations of Computer Science*, pp. 210–214.

Halmos, P. R. (1960). *Naive Set Theory*. Van Nostrand.

Halpern, J. Y. (1981). On the expressive power of dynamic logic II. Technical Report TM-204, MIT/LCS.

Halpern, J. Y. (1982). Deterministic process logic is elementary. In *Proc. 23rd Symp. Found. Comput. Sci.*, pp. 204–216. IEEE.

Halpern, J. Y. (1983). Deterministic process logic is elementary. *Infor. and Control 57*(1), 56–89.

Halpern, J. Y. and J. H. Reif (1981). The propositional dynamic logic of deterministic, well-structured programs. In *Proc. 22nd Symp. Found. Comput. Sci.*, pp. 322–334. IEEE.

Halpern, J. Y. and J. H. Reif (1983). The propositional dynamic logic of deterministic, well-structured programs. *Theor. Comput. Sci. 27*, 127–165.

Hansson, H. and B. Jonsson (1994). A logic for reasoning about time and probability. *Formal Aspects of Computing 6*, 512–535.

Harel, D. (1979). *First-Order Dynamic Logic*, Volume 68 of *Lect. Notes in Comput. Sci.* Springer-Verlag.

Harel, D. (1984). Dynamic logic. In Gabbay and Guenthner (Eds.), *Handbook of Philosophical Logic*, Volume II: Extensions of Classical Logic, pp. 497–604. Reidel.

Harel, D. (1985). Recurring dominoes: Making the highly undecidable highly understandable. *Annals of Discrete Mathematics 24*, 51–72.

Harel, D. (1992). *Algorithmics: The Spirit of Computing* (second ed.). Addison-Wesley.

Harel, D. and D. Kozen (1984). A programming language for the inductive sets, and applications. *Information and Control 63*(1–2), 118–139.

Harel, D., D. Kozen, and R. Parikh (1982). Process logic: Expressiveness, decidability, completeness. *J. Comput. Syst. Sci. 25*(2), 144–170.

Harel, D., A. R. Meyer, and V. R. Pratt (1977). Computability and completeness in logics of programs. In *Proc. 9th Symp. Theory of Comput.*, pp. 261–268. ACM.

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

Harel, D. and D. Peleg (1985). More on looping vs. repeating in dynamic logic. *Information Processing Letters 20*, 87–90.

Harel, D., A. Pnueli, and J. Stavi (1983). Propositional dynamic logic of nonregular programs. *J. Comput. Syst. Sci. 26*, 222–243.

Harel, D., A. Pnueli, and M. Vardi (1982). Two dimensional temporal logic and PDL with intersection. Unpublished.

Harel, D. and V. R. Pratt (1978). Nondeterminism in logics of programs. In *Proc. 5th Symp. Princip. Prog. Lang.*, pp. 203–213. ACM.

Harel, D. and D. Raz (1993). Deciding properties of nonregular programs. *SIAM J. Comput. 22*, 857–874.

Harel, D. and D. Raz (1994). Deciding emptiness for stack automata on infinite trees. *Information and Computation 113*, 278–299.

Harel, D. and R. Sherman (1982). Looping vs. repeating in dynamic logic. *Infor. and Control 55*, 175–192.

Harel, D. and R. Sherman (1985). Propositional dynamic logic of flowcharts. *Infor. and Control 64*, 119–135.

Harel, D. and E. Singerman (1996). More on nonregular PDL: Finite models and Fibonacci-like programs. *Information and Computation 128*, 109–118.

Hart, S., M. Sharir, and A. Pnueli (1982). Termination of probabilistic concurrent programs. In *Proc. 9th Symp. Princip. Prog. Lang.*, pp. 1–6. ACM.

Hartmanis, J. and R. E. Stearns (1965). On the complexity of algorithms. *Trans. Amer. Math. Soc. 117*, 285–306.

Hartonas, C. (1998). Duality for modal μ-logics. *Theor. Comput. Sci. 202*(1–2), 193–222.

Henkin, L. (1949). The completeness of the first order functional calculus. *J. Symb. Logic 14*, 159–166.

Hennessy, M. C. B. and G. D. Plotkin (1979). Full abstraction for a simple programming language. In *Proc. Symp. Semantics of Algorithmic Languages*, Volume 74 of *Lecture Notes in Computer Science*, pp. 108–120. Springer-Verlag.

Hitchcock, P. and D. Park (1972). Induction rules and termination proofs. In M. Nivat (Ed.), *Int. Colloq. Automata Lang. Prog.*, pp. 225–251. North-Holland.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Comm. Assoc. Comput. Mach. 12*, 576–580, 583.

Hopcroft, J. E. and J. D. Ullman (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Hopkins, M. and D. Kozen (1999, July). Parikh's theorem in commutative Kleene algebra. In *Proc. Conf. Logic in Computer Science (LICS'99)*, pp. 394–401. IEEE.

Hughes, G. E. and M. J. Cresswell (1968). *An Introduction to Modal Logic*. Methuen.

Huth, M. and M. Kwiatkowska (1997). Quantitative analysis and model checking. In *Proc. 12th Symp. Logic in Comput. Sci.*, pp. 111–122. IEEE.

Ianov, Y. I. (1960). The logical schemes of algorithms. In *Problems of Cybernetics*, Volume 1, pp. 82–140. Pergamon Press.

Iwano, K. and K. Steiglitz (1990). A semiring on convex polygons and zero-sum cycle problems. *SIAM J. Comput. 19*(5), 883–901.

Jou, C. and S. Smolka (1990). Equivalences, congruences and complete axiomatizations for probabilistic processes. In *Proc. CONCUR'90*, Volume 458 of *Lecture Notes in Comput. Sci.*, pp. 367–383. Springer-Verlag.

Kaivola, R. (1997, April). *Using Automata to Characterise Fixed Point Temporal Logics*. Ph. D. thesis, University of Edinburgh. Report CST-135-97.

Kamp, H. W. (1968). *Tense logics and the theory of linear order*. Ph. D. thesis, UCLA.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher (Eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum Press.

Keisler, J. (1971). *Model Theory for Infinitary Logic*. North Holland.

Kfoury, A. (1983). Definability by programs in first-order structures. *Theoretical Computer Science 25*, 1–66.

Kfoury, A. and A. Stolboushkin (1997). An infinite pebble game and applications. *Information and Computation 136*, 53–66.

Kfoury, A. J. (1985). Definability by deterministic and nondeterministic programs with applications to first-order dynamic logic. *Infor. and Control 65*(2–3), 98–121.

Kleene, S. C. (1943). Recursive predicates and quantifiers. *Trans. Amer. Math. Soc. 53*, 41–74.

Kleene, S. C. (1952). *Introduction to Metamathematics*. D. van Nostrand.

Kleene, S. C. (1955). On the forms of the predicates in the theory of constructive ordinals (second paper). *Amer. J. Math. 77*, 405–428.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy (Eds.), *Automata Studies*, pp. 3–41. Princeton, N.J.: Princeton University Press.

Knijnenburg, P. M. W. (1988, November). On axiomatizations for propositional logics of programs. Technical Report RUU-CS-88-34, Rijksuniversiteit Utrecht.

Koren, T. and A. Pnueli (1983). There exist decidable context-free propositional dynamic logics. In *Proc. Symp. on Logics of Programs*, Volume 164 of *Lecture Notes in Computer Science*, pp. 290–312. Springer-Verlag.

Kowalczyk, W., D. Niwiński, and J. Tiuryn (1987). A generalization of Cook's auxiliary–pushdown–automata theorem. *Fundamenta Informaticae XII*, 497–506.

Kozen, D. (1979a). Dynamic algebra. In E. Engeler (Ed.), *Proc. Workshop on Logic of Programs*, Volume 125 of *Lecture Notes in Computer Science*, pp. 102–144. Springer-Verlag. chapter of *Propositional dynamic logics of programs: A survey* by Rohit Parikh.

Kozen, D. (1979b). On the duality of dynamic algebras and Kripke models. In E. Engeler (Ed.), *Proc. Workshop on Logic of Programs*, Volume 125 of *Lecture Notes in Computer Science*, pp. 1–11. Springer-Verlag.

Kozen, D. (1979c, October). On the representation of dynamic algebras. Technical Report RC7898, IBM Thomas J. Watson Research Center.

Kozen, D. (1980a, May). On the representation of dynamic algebras II. Technical Report RC8290, IBM Thomas J. Watson Research Center.

Kozen, D. (1980b, July). A representation theorem for models of *-free *PDL*. In *Proc. 7th Colloq. Automata, Languages, and Programming*, pp. 351–362. EATCS.

Kozen, D. (1981a). Logics of programs. Lecture notes, Aarhus University, Denmark.

Kozen, D. (1981b). On induction vs. *-continuity. In Kozen (Ed.), *Proc. Workshop on Logic of Programs*, Volume 131 of *Lecture Notes in Computer Science*, New York, pp. 167–176. Springer-Verlag.

Kozen, D. (1981c). On the expressiveness of μ. Manuscript.

Kozen, D. (1981d). Semantics of probabilistic programs. *J. Comput. Syst. Sci. 22*, 328–350.

Kozen, D. (1982, July). Results on the propositional μ-calculus. In *Proc. 9th Int. Colloq. Automata, Languages, and Programming*, Aarhus, Denmark, pp. 348–359. EATCS.

Kozen, D. (1983). Results on the propositional μ-calculus. *Theor. Comput. Sci. 27*, 333–354.

Kozen, D. (1984, May). A Ramsey theorem with infinitely many colors. In Lenstra, Lenstra, and van Emde Boas (Eds.), *Dopo Le Parole*, pp. 71–72. Amsterdam: University of Amsterdam.

Kozen, D. (1985, April). A probabilistic *PDL*. *J. Comput. Syst. Sci. 30*(2), 162–178.

Kozen, D. (1988). A finite model theorem for the propositional μ-calculus. *Studia Logica 47*(3), 233–241.

Kozen, D. (1990). On Kleene algebras and closed semirings. In Rovan (Ed.), *Proc. Math. Found. Comput. Sci.*, Volume 452 of *Lecture Notes in Computer Science*, Banska-Bystrica, Slovakia, pp. 26–47. Springer-Verlag.

Kozen, D. (1991a, July). A completeness theorem for Kleene algebras and the algebra of regular events. In *Proc. 6th Symp. Logic in Comput. Sci.*, Amsterdam, pp. 214–225. IEEE.

Kozen, D. (1991b). *The Design and Analysis of Algorithms*. New York: Springer-Verlag.

Kozen, D. (1994a, May). A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput. 110*(2), 366–390.

Kozen, D. (1994b). On action algebras. In J. van Eijck and A. Visser (Eds.), *Logic and Information Flow*, pp. 78–88. MIT Press.

Kozen, D. (1996, March). Kleene algebra with tests and commutativity conditions. In T. Margaria and B. Steffen (Eds.), *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, Volume 1055 of *Lecture Notes in Computer Science*, Passau, Germany, pp. 14–33. Springer-Verlag.

Kozen, D. (1997a). *Automata and Computability*. New York: Springer-Verlag.

Kozen, D. (1997b, May). Kleene algebra with tests. *Transactions on Programming Languages and Systems 19*(3), 427–443.

Kozen, D. (1997c, June). On the complexity of reasoning in Kleene algebra. In *Proc. 12th Symp. Logic in Comput. Sci.*, Los Alamitos, Ca., pp. 195–202. IEEE.

Kozen, D. (1998, March). Typed Kleene algebra. Technical Report 98-1669, Computer Science Department, Cornell University.

Kozen, D. (1999a, July). On Hoare logic and Kleene algebra with tests. In *Proc. Conf. Logic in Computer Science (LICS'99)*, pp. 167–172. IEEE.

Kozen, D. (1999b, July). On Hoare logic, Kleene algebra, and types. Technical Report 99-1760, Computer Science Department, Cornell University. Abstract in: Abstracts of 11th Int. Congress Logic, Methodology and Philosophy of Science, Ed. J. Cachro and K. Kijania-Placek, Krakow, Poland, August 1999, p. 15. To appear in: Proc. 11th Int. Congress Logic, Methodology and Philosophy of Science, ed. P. Gardenfors, K. Kijania-Placek and J. Wolenski, Kluwer.

Kozen, D. and R. Parikh (1981). An elementary proof of the completeness of *PDL*. *Theor. Comput. Sci. 14*(1), 113–118.

Kozen, D. and R. Parikh (1983). A decision procedure for the propositional μ-calculus. In Clarke and Kozen (Eds.), *Proc. Workshop on Logics of Programs*, Volume 164 of *Lecture Notes in Computer Science*, pp. 313–325. Springer-Verlag.

Kozen, D. and M.-C. Patron (2000, July). Certification of compiler optimizations using Kleene algebra with tests. In U. Furbach and M. Kerber (Eds.), *Proc. 1st Int. Conf. Computational Logic*, London. To appear.

Kozen, D. and F. Smith (1996, September). Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem (Eds.), *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, Volume 1258 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, pp. 244–259. Springer-Verlag.

Kozen, D. and J. Tiuryn (1990). Logics of programs. In van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Volume B, pp. 789–840. Amsterdam: North Holland.

Kreczmar, A. (1977). Programmability in fields. *Fundamenta Informaticae I*, 195–230.

Kripke, S. (1963). Semantic analysis of modal logic. *Zeitschr. f. math. Logik und Grundlagen d. Math. 9*, 67–96.

Krob, D. (1991, October). A complete system of B-rational identities. *Theoretical Computer Science 89*(2), 207–343.

Kuich, W. (1987). The Kleene and Parikh theorem in complete semirings. In T. Ottmann (Ed.), *Proc. 14th Colloq. Automata, Languages, and Programming*, Volume 267 of *Lecture Notes in Computer Science*, New York, pp. 212–225. EATCS: Springer-Verlag.

Kuich, W. and A. Salomaa (1986). *Semirings, Automata, and Languages*. Berlin: Springer-Verlag.

Ladner, R. E. (1977). Unpublished.

Lamport, L. (1980). "Sometime" is sometimes "not never". *Proc. 7th Symp. Princip. Prog. Lang.*, 174–185.

Lehmann, D. and S. Shelah (1982). Reasoning with time and chance. *Infor. and Control 53*(3), 165–198.

Lewis, H. R. and C. H. Papadimitriou (1981). *Elements of the Theory of Computation*. Prentice Hall.

Lipton, R. J. (1977). A necessary and sufficient condition for the existence of Hoare logics. In *Proc. 18th Symp. Found. Comput. Sci.*, pp. 1–6. IEEE.

Luckham, D. C., D. Park, and M. Paterson (1970). On formalized computer programs. *J. Comput. Syst. Sci. 4*, 220–249.

Mader, A. (1997, September). *Verification of Modal Properties Using Boolean Equation Systems*. Ph. D. thesis, Fakultt fr Informatik, Technische Universitt Mnchen.

Makowski, J. A. (1980). Measuring the expressive power of dynamic logics: an application of abstract model theory. In *Proc. 7th Int. Colloq. Automata Lang. Prog.*, Volume 80 of *Lect. Notes in Comput. Sci.*, pp. 409–421. Springer-Verlag.

Makowski, J. A. and I. Sain (1986). On the equivalence of weak second-order and nonstandard time semantics for various program verification systems. In *Proc. 1st Symp. Logic in Comput. Sci.*, pp. 293–300. IEEE.

Makowsky, J. A. and M. L. Tiomkin (1980). Probabilistic propositional dynamic logic. Manuscript.

Manna, Z. (1974). *Mathematical Theory of Computation*. McGraw-Hill.

Manna, Z. and A. Pnueli (1981). Verification of concurrent programs: temporal proof principles. In D. Kozen (Ed.), *Proc. Workshop on Logics of Programs*, Volume 131 of *Lect. Notes in Comput. Sci.*, pp. 200–252. Springer-Verlag.

Manna, Z. and A. Pnueli (1987, January). Specification and verification of concurrent programs by ∀-automata. In *Proc. 14th Symp. Principles of Programming Languages*, pp. 1–12. ACM.

McCulloch, W. S. and W. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics 5*, 115–143.

Mehlhorn, K. (1984). *Graph Algorithms and NP-Completeness*, Volume II of *Data Structures and Algorithms*. Springer-Verlag.

Meyer, A. R. and J. Y. Halpern (1982). Axiomatic definitions of programming languages: a theoretical assessment. *J. Assoc. Comput. Mach. 29*, 555–576.

Meyer, A. R. and R. Parikh (1981). Definability in dynamic logic. *J. Comput. Syst. Sci. 23*, 279–298.

Meyer, A. R., R. S. Streett, and G. Mirkowska (1981). The deducibility problem in propositional dynamic logic. In E. Engeler (Ed.), *Proc. Workshop Logic of Programs*, Volume 125 of *Lect. Notes in Comput. Sci.*, pp. 12–22. Springer-Verlag.

Meyer, A. R. and J. Tiuryn (1981). A note on equivalences among logics of programs. In D. Kozen (Ed.), *Proc. Workshop on Logics of Programs*, Volume 131 of *Lect. Notes in Comput. Sci.*, pp. 282–299. Springer-Verlag.

Meyer, A. R. and J. Tiuryn (1984). Equivalences among logics of programs. *Journal of Computer and Systems Science 29*, 160–170.

Meyer, A. R. and K. Winklmann (1982). Expressing program looping in regular dynamic logic. *Theor. Comput. Sci. 18*, 301–323.

Miller, G. L. (1976). Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci. 13*, 300–317.

Minsky, M. L. (1961). Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines. *Ann. Math. 74*(3), 437–455.

Mirkowska, G. (1971). On formalized systems of algorithmic logic. *Bull. Acad. Polon. Sci. Ser. Sci. Math. Astron. Phys. 19*, 421–428.

Mirkowska, G. (1980). Algorithmic logic with nondeterministic programs. *Fund. Informaticae III*, 45–64.

Mirkowska, G. (1981a). PAL—propositional algorithmic logic. In E. Engeler (Ed.), *Proc. Workshop Logic of Programs*, Volume 125 of *Lect. Notes in Comput. Sci.*, pp. 23–101. Springer-Verlag.

Mirkowska, G. (1981b). PAL—propositional algorithmic logic. *Fund. Informaticae IV*, 675–760.

Morgan, C., A. McIver, and K. Seidel (1999). Probabilistic predicate transformers. *ACM Trans. Programming Languages and Systems 8*(1), 1–30.

Moschovakis, Y. N. (1974). *Elementary Induction on Abstract Structures*. North-Holland.

Moschovakis, Y. N. (1980). *Descriptive Set Theory*. North-Holland.

Muller, D. E., A. Saoudi, and P. E. Schupp (1988, July). Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proc. 3rd Symp. Logic in Computer Science*, pp. 422–427. IEEE.

Németi, I. (1980). Every free algebra in the variety generated by the representable dynamic algebras is separable and representable. Manuscript.

Németi, I. (1981). Nonstandard dynamic logic. In D. Kozen (Ed.), *Proc. Workshop on Logics of Programs*, Volume 131 of *Lect. Notes in Comput. Sci.*, pp. 311–348. Springer-Verlag.

Ng, K. C. (1984). *Relation Algebras with Transitive Closure*. Ph. D. thesis, University of California, Berkeley.

Ng, K. C. and A. Tarski (1977). Relation algebras with transitive closure, abstract 742-02-09. *Notices Amer. Math. Soc. 24*, A29–A30.

Nishimura, H. (1979). Sequential method in propositional dynamic logic. *Acta Informatica 12*, 377–400.

Nishimura, H. (1980). Descriptively complete process logic. *Acta Informatica 14*, 359–369.

Niwinski, D. (1984). The propositional μ-calculus is more expressive than the propositional dynamic logic of looping. University of Warsaw.

Parikh, R. (1978a). The completeness of propositional dynamic logic. In *Proc. 7th Symp. on Math. Found. of Comput. Sci.*, Volume 64 of *Lect. Notes in Comput. Sci.*, pp. 403–415. Springer-Verlag.

Parikh, R. (1978b). A decidability result for second order process logic. In *Proc. 19th Symp. Found. Comput. Sci.*, pp. 177–183. IEEE.

Parikh, R. (1981). Propositional dynamic logics of programs: a survey. In E. Engeler (Ed.), *Proc. Workshop on Logics of Programs*, Volume 125 of *Lect. Notes in Comput. Sci.*, pp. 102–144. Springer-Verlag.

Parikh, R. (1983). Propositional game logic. In *Proc. 23rd IEEE Symp. Foundations of Computer Science*.

Parikh, R. and A. Mahoney (1983). A theory of probabilistic programs. In E. Clarke and D. Kozen (Eds.), *Proc. Workshop on Logics of Programs*, Volume 164 of *Lect. Notes in Comput. Sci.*, pp. 396–402. Springer-Verlag.

Park, D. (1976). Finiteness is μ-ineffable. *Theor. Comput. Sci. 3*, 173–181.

Paterson, M. S. and C. E. Hewitt (1970). Comparative schematology. In *Record Project MAC Conf. on Concurrent Systems and Parallel Computation*, pp. 119–128. ACM.

Pecuchet, J. P. (1986). On the complementation of Büchi automata. *Theor. Comput. Sci. 47*, 95–98.

Peleg, D. (1987a). Communication in concurrent dynamic logic. *J. Comput. Sys. Sci. 35*, 23–58.

Peleg, D. (1987b). Concurrent dynamic logic. *J. Assoc. Comput. Mach. 34*(2), 450–479.

Peleg, D. (1987c). Concurrent program schemes and their logics. *Theor. Comput. Sci. 55*, 1–45.

Peng, W. and S. P. Iyer (1995). A new type of pushdown-tree automata on infinite trees. *Int. J. of Found. of Comput. Sci. 6*(2), 169–186.

Peterson, G. L. (1978). The power of tests in propositional dynamic logic. Technical Report 47, Comput. Sci. Dept., Univ. of Rochester.

Pnueli, A. (1977). The temporal logic of programs. In *Proc. 18th Symp. Found. Comput. Sci.*, pp. 46–57. IEEE.

Pnueli, A. and L. D. Zuck (1986). Verification of multiprocess probabilistic protocols. *Distributed Computing 1*(1), 53–72.

Pnueli, A. and L. D. Zuck (1993, March). Probabilistic verification. *Information and Computation 103*(1), 1–29.

Post, E. (1943). Formal reductions of the general combinatorial decision problem. *Amer. J. Math. 65*, 197–215.

Post, E. (1944). Recursively enumerable sets of positive natural numbers and their decision problems. *Bull. Amer. Math. Soc. 50*, 284–316.

Pratt, V. (1988, June). Dynamic algebras as a well-behaved fragment of relation algebras. In D. Pigozzi (Ed.), *Proc. Conf. on Algebra and Computer Science*, Volume 425 of *Lecture Notes in Computer Science*, Ames, Iowa, pp. 77–110. Springer-Verlag.

Pratt, V. (1990, September). Action logic and pure induction. In J. van Eijck (Ed.), *Proc. Logics in AI: European Workshop JELIA '90*, Volume 478 of *Lecture Notes in Computer Science*, New York, pp. 97–120. Springer-Verlag.

Pratt, V. R. (1976). Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Symp. Found. Comput. Sci.*, pp. 109–121. IEEE.

Pratt, V. R. (1978). A practical decision method for propositional dynamic logic. In *Proc. 10th Symp. Theory of Comput.*, pp. 326–337. ACM.

Pratt, V. R. (1979a, July). Dynamic algebras: examples, constructions, applications. Technical Report TM-138, MIT/LCS.

Pratt, V. R. (1979b). Models of program logics. In *Proc. 20th Symp. Found. Comput. Sci.*, pp. 115–122. IEEE.

Pratt, V. R. (1979c). Process logic. In *Proc. 6th Symp. Princip. Prog. Lang.*, pp. 93–100. ACM.

Pratt, V. R. (1980a). Dynamic algebras and the nature of induction. In *Proc. 12th Symp. Theory of Comput.*, pp. 22–28. ACM.

Pratt, V. R. (1980b). A near-optimal method for reasoning about actions. *J. Comput. Syst. Sci. 20*(2), 231–254.

Pratt, V. R. (1981a). A decidable μ-calculus: preliminary report. In *Proc. 22nd Symp. Found. Comput. Sci.*, pp. 421–427. IEEE.

Pratt, V. R. (1981b). Using graphs to understand PDL. In D. Kozen (Ed.), *Proc. Workshop on Logics of Programs*, Volume 131 of *Lect. Notes in Comput. Sci.*, pp. 387–396. Springer-Verlag.

Rabin, M. O. (1969). Decidability of second order theories and automata on infinite trees. *Trans. Amer. Math. Soc. 141*, 1–35.

Rabin, M. O. (1980). Probabilistic algorithms for testing primality. *J. Number Theory 12*, 128–138.

Rabin, M. O. and D. S. Scott (1959). Finite automata and their decision problems. *IBM J. Res. Develop. 3*(2), 115–125.

Ramshaw, L. H. (1981). *Formalizing the analysis of algorithms*. Ph. D. thesis, Stanford Univ.

Rasiowa, H. and R. Sikorski (1963). *Mathematics of Metamathematics*. Polish Scientific Publishers, PWN.

Redko, V. N. (1964). On defining relations for the algebra of regular events. *Ukrain. Mat. Z. 16*, 120–126. In Russian.

Reif, J. (1980). Logics for probabilistic programming. In *Proc. 12th Symp. Theory of Comput.*, pp. 8–13. ACM.

Renegar, J. (1991). Computational complexity of solving real algebraic formulae. In *Proc. Int. Congress of Mathematicians*, pp. 1595–1606. Springer-Verlag.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc. 89*, 25–59.

Rice, H. G. (1956). On completely recursively enumerable classes and their key arrays. *J. Symbolic Logic 21*, 304–341.

Rogers, H. (1967). *Theory of Recursive Functions and Effective Computability*. McGraw-Hill.

Rogers, Jr., H. (1967). *Theory of Recursive Functions and Effective Computability*. McGraw-Hill.

Rosen, K. H. (1995). *Discrete Mathematics and Its Applications* (3rd ed.). McGraw-Hill.

Safra, S. (1988, October). On the complexity of ω-automata. In *Proc. 29th Symp. Foundations of Comput. Sci.*, pp. 319–327. IEEE.

Sakarovitch, J. (1987). Kleene's theorem revisited: A formal path from Kleene to Chomsky. In A. Kelemenova and J. Keleman (Eds.), *Trends, Techniques, and Problems in Theoretical Computer Science*, Volume 281 of *Lecture Notes in Computer Science*, New York, pp. 39–50. Springer-Verlag.

Salomaa, A. (1966, January). Two complete axiom systems for the algebra of regular events. *J. Assoc. Comput. Mach. 13*(1), 158–169.

Salomaa, A. (1981). *Jewels of Formal Language Theory*. Pitman Books Limited.

Salwicki, A. (1970). Formalized algorithmic languages. *Bull. Acad. Polon. Sci. Ser. Sci. Math. Astron. Phys. 18*, 227–232.

Salwicki, A. (1977). Algorithmic logic: a tool for investigations of programs. In Butts and Hintikka (Eds.), *Logic Foundations of Mathematics and Computability Theory*, pp. 281–295. Reidel.

Saudi, A. (1989). Pushdown automata on infinite trees and omega-Kleene closure of context-free tree sets. In *Proc. Math. Found. of Comput. Sci.*, Volume 379 of *Lecture Notes in Computer Science*, pp. 445–457. Springer-Verlag.

Sazonov, V. (1980). Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kibernetik 16*, 319–323.

Scott, D. S. and J. W. de Bakker (1969). A theory of programs. IBM Vienna.

Segala, R. and N. Lynch (1994). Probabilistic simulations for probabilistic processes. In *Proc. CONCUR'94*, Volume 836 of *Lecture Notes in Comput. Sci.*, pp. 481–496. Springer-Verlag.

Segerberg, K. (1977). A completeness theorem in the modal logic of programs (preliminary report). *Not. Amer. Math. Soc. 24*(6), A–552.

Shoenfield, J. R. (1967). *Mathematical Logic*. Addison-Wesley.

Sholz, H. (1952). Ein ungelöstes Problem in der symbolischen Logik. *The Journal of Symbolic Logic 17*, 160.

Sistla, A. P. and E. M. Clarke (1982). The complexity of propositional linear temporal logics. In *Proc. 14th Symp. Theory of Comput.*, pp. 159–168. ACM.

Sistla, A. P., M. Y. Vardi, and P. Wolper (1987). The complementation problem for Büchi automata with application to temporal logic. *Theor. Comput. Sci. 49*, 217–237.

Soare, R. I. (1987). *Recursively Enumerable Sets and Degrees*. Springer-Verlag.

Sokolsky, O. and S. Smolka (1994, June). Incremental model checking in the modal μ-calculus. In D. Dill (Ed.), *Proc. Conf. Computer Aided Verification*, Volume 818 of *Lect. Notes in Comput. Sci.*, pp. 352–363. Springer.

Steffen, B., T. Margaria, A. Classen, V. Braun, R. Nisius, and M. Reitenspiess (1996, March). A constraint oriented service environment. In T. Margaria and B. Steffen (Eds.), *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, Volume 1055 of *Lect. Notes in Comput. Sci.*, pp. 418–421. Springer.

Stirling, C. (1992). Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum (Eds.), *Handbook of Logic in Computer Science*, pp. 477–563. Clarendon Press.

Stirling, C. and D. Walker (1989, March). Local model checking in the modal μ-calculus. In *Proc. Int. Joint Conf. Theory and Practice of Software Develop. (TAPSOFT89)*, Volume 352 of *Lect. Notes in Comput. Sci.*, pp. 369–383. Springer.

Stockmeyer, L. J. and A. R. Meyer (1973). Word problems requiring exponential time. In *Proc. 5th Symp. Theory of Computing*, New York, pp. 1–9. ACM: ACM.

Stolboushkin, A. (1983). Regular dynamic logic is not interpretable in deterministic context-free dynamic logic. *Information and Computation 59*, 94–107.

Stolboushkin, A. (1989, June). Some complexity bounds for dynamic logic. In *Proc. 4th Symp. Logic in Comput. Sci.*, pp. 324–332. IEEE.

Stolboushkin, A. P. and M. A. Taitslin (1983). Deterministic dynamic logic is strictly weaker than dynamic logic. *Infor. and Control 57*, 48–55.

Stone, M. H. (1936). The representation theorem for Boolean algebra. *Trans. Amer. Math. Soc. 40*, 37–111.

Streett, R. (1985a). Fixpoints and program looping: reductions from the propositional μ-calculus into propositional dynamic logics of looping. In Parikh (Ed.), *Proc. Workshop on Logics of Programs 1985*, pp. 359–372. Springer. Lect. Notes in Comput. Sci. 193.

Streett, R. and E. A. Emerson (1984). The propositional μ-calculus is elementary. In *Proc. 11th Int. Colloq. on Automata Languages and Programming*, pp. 465–472. Springer. Lect. Notes in Comput. Sci. 172.

Streett, R. S. (1981). Propositional dynamic logic of looping and converse. In *Proc. 13th Symp. Theory of Comput.*, pp. 375–381. ACM.

Streett, R. S. (1982). Propositional dynamic logic of looping and converse is elementarily decidable. *Infor. and Control 54*, 121–141.

Streett, R. S. (1985b). 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*, Volume 193 of *Lect. Notes in Comput. Sci.*, pp. 359–372. Springer-Verlag.

Tarjan, R. E. (1981). A unified approach to path problems. *J. Assoc. Comput. Mach.*, 577–593.

Tarski, A. (1935). Die Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica 1*, 261–405.

Thiele, H. (1966). Wissenschaftstheoretische untersuchungen in algorithmischen sprachen. In

Theorie der Graphschemata-Kalkale Veb Deutscher Verlag der Wissenschaften. Berlin.

Thomas, W. (1997, May). Languages, automata, and logic. Technical Report 9607, Christian-Albrechts-Universität Kiel.

Tiuryn, J. (1981a). A survey of the logic of effective definitions. In E. Engeler (Ed.), *Proc. Workshop on Logics of Programs*, Volume 125 of *Lect. Notes in Comput. Sci.*, pp. 198–245. Springer-Verlag.

Tiuryn, J. (1981b). Unbounded program memory adds to the expressive power of first-order programming logics. In *Proc. 22nd Symp. Found. Comput. Sci.*, pp. 335–339. IEEE.

Tiuryn, J. (1984). Unbounded program memory adds to the expressive power of first-order programming logics. *Infor. and Control 60*, 12–35.

Tiuryn, J. (1986). 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.*, Volume 233 of *Lect. Notes in Comput. Sci.*, pp. 177–198. Springer-Verlag.

Tiuryn, J. (1989). A simplified proof of $\mathsf{DDL} < \mathsf{DL}$. *Information and Computation 81*, 1–12.

Tiuryn, J. and P. Urzyczyn (1983). Some relationships between logics of programs and complexity theory. In *Proc. 24th Symp. Found. Comput. Sci.*, pp. 180–184. IEEE.

Tiuryn, J. and P. Urzyczyn (1984). Remarks on comparing expressive power of logics of programs. In Chytil and Koubek (Eds.), *Proc. Math. Found. Comput. Sci.*, Volume 176 of *Lect. Notes in Comput. Sci.*, pp. 535–543. Springer-Verlag.

Tiuryn, J. and P. Urzyczyn (1988). Some relationships between logics of programs and complexity theory. *Theor. Comput. Sci. 60*, 83–108.

Trnkova, V. and J. Reiterman (1980). Dynamic algebras which are not Kripke structures. In *Proc. 9th Symp. on Math. Found. Comput. Sci.*, pp. 528–538.

Turing, A. M. (1936). On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc. 42*, 230–265. Erratum: Ibid., 43 (1937), pp. 544–546.

Urzyczyn, P. (1983a). A necessary and sufficient condition in order that a Herbrand interpretation be expressive relative to recursive programs. *Information and Control 56*, 212–219.

Urzyczyn, P. (1983b). Nontrivial definability by flowchart programs. *Infor. and Control 58*, 59–87.

Urzyczyn, P. (1983c). *The Unwind Property*. Ph. D. thesis, Warsaw University. In Polish.

Urzyczyn, P. (1986). "During" cannot be expressed by "after". *Journal of Computer and System Sciences 32*, 97–104.

Urzyczyn, P. (1987). Deterministic context-free dynamic logic is more expressive than deterministic dynamic logic of regular programs. *Fundamenta Informaticae 10*, 123–142.

Urzyczyn, P. (1988). Logics of programs with Boolean memory. *Fundamenta Informaticae XI*, 21–40.

Valiev, M. K. (1980). Decision complexity of variants of propositional dynamic logic. In *Proc. 9th Symp. Math. Found. Comput. Sci.*, Volume 88 of *Lect. Notes in Comput. Sci.*, pp. 656–664. Springer-Verlag.

van Dalen, D. (1994). *Logic and Structure* (Third ed.). Springer-Verlag.

van Emde Boas, P. (1978). The connection between modal logic and algorithmic logics. In *Symp. on Math. Found. of Comp. Sci.*, pp. 1–15.

Vardi, M. (1998a). Linear vs. branching time: a complexity-theoretic perspective. In *Proc. 13th Symp. Logic in Comput. Sci.*, pp. 394–405. IEEE.

Vardi, M. and P. Wolper (1986a). Automata-theoretic techniques for modal logics of programs. *J. Comput. Sys. Sci. 32*, 183–221.

Vardi, M. Y. (1985a, October). Automatic verification of probabilistic concurrent finite-state

programs. In *Proc. 26th Symp. Found. Comput. Sci.*, pp. 327–338. IEEE.

Vardi, M. Y. (1985b). The taming of the converse: reasoning about two-way computations. In R. Parikh (Ed.), *Proc. Workshop on Logics of Programs*, Volume 193 of *Lect. Notes in Comput. Sci.*, pp. 413–424. Springer-Verlag.

Vardi, M. Y. (1987, June). Verification of concurrent programs: the automata-theoretic framework. In *Proc. 2nd Symp. Logic in Comput. Sci.*, pp. 167–176. IEEE.

Vardi, M. Y. (1998b, July). Reasoning about the past with two-way automata. In *Proc. 25th Int. Colloq. Automata Lang. Prog.*, Volume 1443 of *Lect. Notes in Comput. Sci.*, pp. 628–641. Springer-Verlag.

Vardi, M. Y. and L. Stockmeyer (1985, May). Improved upper and lower bounds for modal logics of programs: preliminary report. In *Proc. 17th Symp. Theory of Comput.*, pp. 240–251. ACM.

Vardi, M. Y. and P. Wolper (1986b, June). An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. Logic in Computer Science*, pp. 332–344. IEEE.

Vardi, M. Y. and P. Wolper (1986c). Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci. 32*, 183–221.

Walukiewicz, I. (1993, June). Completeness result for the propositional μ-calculus. In *Proc. 8th IEEE Symp. Logic in Comput. Sci.*

Walukiewicz, I. (1995, June). Completeness of Kozen's axiomatisation of the propositional μ-calculus. In *Proc. 10th Symp. Logic in Comput. Sci.*, pp. 14–24. IEEE.

Walukiewicz, I. (2000, February–March). Completeness of Kozen's axiomatisation of the propositional μ-calculus. *Infor. and Comput. 157*(1–2), 142–182.

Wand, M. (1978). A new incompleteness result for Hoare's system. *J. Assoc. Comput. Mach. 25*, 168–175.

Whitehead, A. N. and B. Russell (1910–1913). *Principia Mathematica*. Cambridge University Press. Three volumes.

Wolper, P. (1981). Temporal logic can be more expressive. In *Proc. 22nd Symp. Foundations of Computer Science*, pp. 340–348. IEEE.

Wolper, P. (1983). Temporal logic can be more expressive. *Infor. and Control 56*, 72–99.

Notation and Abbreviations

$\mathbb{Z}$ integers . . . 3
$\mathbb{Q}$ rational numbers . . . 3
$\mathbb{R}$ real numbers . . . 3
$\mathbb{N}$ natural numbers . . . 3
ω finite ordinals . . . 3
$\Longrightarrow$ meta-implication . . . 3
$\rightarrow$ implication . . . 3
$\Longleftrightarrow$ meta-equivalence . . . 3
$\leftrightarrow$ equivalence . . . 3
iff if and only if . . . 3
$\stackrel{\text{def}}{=}$ definition . . . 3
$\stackrel{\text{def}}{\Longleftrightarrow}$ definition . . . 3
$|\sigma|$ length of a sequence . . . 3
A^* set of all finite strings over A . . . 3
ε empty string . . . 3
w^R reverse of a string . . . 3
$A, B, C, \ldots$ sets . . . 3
$\in$ set containment . . . 3
$\subseteq$ set inclusion, subset . . . 3
$\subset$ strict inclusion . . . 4
$\#A$ cardinality of a set . . . 4
2^A powerset . . . 4
$\varnothing$ empty set . . . 4
$A \cup B$ union . . . 4
$A \cap B$ intersection . . . 4
$\bigcup \mathcal{A}$ union . . . 4
$\bigcap \mathcal{A}$ intersection . . . 4
$B - A$ complement of A in B . . . 4
$\sim A$ complement . . . 4
$A \times B$ Cartesian product . . . 4
$\prod_{\alpha \in I} A_\alpha$ Cartesian product . . . 4
A^n Cartesian power . . . 4
π_β projection function . . . 4

ZFC Zermelo–Fraenkel set theory with choice 4
$P, Q, R, \ldots$ relations 5
$\varnothing$ empty relation 6
$R(a_1, \ldots, a_n)$ $(a_1, \ldots, a_n) \in R$ 6
$a\ R\ b$ $(a, b) \in R$ 6
$\circ$ relational composition 7
ι identity relation 7
R^n n-fold composition of a binary relation 7
$^-$ converse 7
R^+ transitive closure 8
R^* reflexive transitive closure 8
$[a]$ equivalence class 9
$f, g, h, \ldots$ functions 9
$f : A \to B$ function with domain A and range B 9
$A \to B$ function space 9
B^A function space 9
$\mapsto$ anonymous function specifier 9
$\restriction$ function restriction 10
$\circ$ function composition 10
f^{-1} inverse 10
$f[a/b]$ function patching 10
$\sup B$ supremum 11
WFI well-founded induction 12
$\alpha, \beta, \gamma, \ldots$ ordinals 14
Ord class of all ordinals 14
ZF Zermelo–Fraenkel set theory 16
$\tau^\dagger$ least prefixpoint operator 18
curry currying operator 25
$\vdash,\ \dashv$ endmarkers 27
$\sqcup$ blank symbol 28
δ transition function 28
$\alpha, \beta, \gamma, \ldots$ Turing machine configurations 29

$\xrightarrow[M,x]{1}$	next configuration relation . . . 29
$z[i/b]$	string replacement operator . . . 29
$\xrightarrow[M,x]{*}$	reflexive transitive closure of $\xrightarrow[M,x]{1}$ . . . 30
$L(M)$	strings accepted by a Turing machine . . . 30
δ	transition relation . . . 33
HP	halting problem . . . 37
MP	membership problem . . . 37
$DTIME(f(n))$	deterministic time complexity class . . . 39
$NTIME(f(n))$	nondeterministic time complexity class . . . 39
$ATIME(f(n))$	alternating time complexity class . . . 39
$DSPACE(f(n))$	deterministic space complexity class . . . 39
$NSPACE(f(n))$	nondeterministic space complexity class . . . 39
$ASPACE(f(n))$	alternating space complexity class . . . 39
$EXPTIME$	deterministic exponential time . . . 40
$NEXPTIME$	nondeterministic exponential time . . . 40
P	deterministic polynomial time . . . 40
NP	nondeterministic polynomial time . . . 40
$M[B]$	oracle Turing machine . . . 41
Σ_1^0	r.e. sets . . . 43
Π_1^0	co-r.e. sets . . . 43
Δ_1^0	recursive sets . . . 43
$\Sigma_n^0, \Pi_n^0, \Delta_n^0$	arithmetic hierarchy . . . 43
Π_1^1	second-order universal relations . . . 45
Δ_1^1	hyperarithmetic relations . . . 45
IND	programming language for inductive sets . . . 45
ω_1	least uncountable ordinal . . . 50
ω_1^{ck}	least nonrecursive ordinal . . . 50
ord	labeling of well-founded tree . . . 50
$\leq_{\mathrm{m}}$	many-one reducibility . . . 54
$\leq_{\mathrm{m}}^{\log}$	logspace reducibility . . . 54
$\leq_{\mathrm{m}}^{\mathrm{p}}$	polynomial-time reducibility . . . 54
$\vDash$	satisfiability relation . . . 69

Th Φ	logical consequences of a set of formulas	69
$\vdash$	provability relation	70
$\neg$	negation	70
$p, q, r, \ldots$	atomic propositions	71
$\wedge$	conjunction	71
$\vee$	disjunction	71
1	truth	71
0	falsity	71
$\varphi, \psi, \rho, \ldots$	propositional formulas	72
S, K	axioms of propositional logic	77
DN	double negation	77
MP	modus ponens	77
EFQ	e falso quodlibet	78
Σ	vocabulary	87
$a, b, c, \ldots$	constants	87
$=$	equality symbol	87
$x, y, \ldots$	individual variables	87
$s, t, \ldots$	terms	87
$T_\Sigma(X)$	set of terms	87
T_Σ	ground terms	87
$\mathfrak{A} = (A, \mathfrak{m}_\mathfrak{A})$	Σ-algebra	88
$\mathfrak{m}_\mathfrak{A}$	meaning function	88
$f^\mathfrak{A}$	meaning of a function in a structure	88
$\lvert\mathfrak{A}\rvert$	carrier of a structure	88
$T_\Sigma(X)$	term algebra	89
$u, v, w, \ldots$	valuations	90
$t^\mathfrak{A}$	meaning of a ground term in a structure	90
Mod Φ	models of a set of formulas	91
Th $\mathfrak{A}$	theory of a structure	91
Th $\mathcal{D}$	theory of a class of structures	91
$[a]$	congruence class	94
$a \equiv b \pmod{n}$	number theoretic congruence	95
$a \equiv b\ (I)$	congruence modulo an ideal	95

$\lhd$	normal subgroup	95
$\mathfrak{A}/\equiv$	quotient algebra	96
REF	reflexivity rule	99
SYM	symmetry rule	99
TRANS	transitivity rule	99
CONG	congruence rule	99
$\prod_{i\in I}\mathfrak{A}_i$	product algebra	100
$\mathbf{H}$	closure operator for homomorphic images	101
$\mathbf{S}$	closure operator for subalgebras	101
$\mathbf{P}$	closure operator for products	101
$\forall$	universal quantifier	102
$\exists$	existential quantifier	102
$p, q, r, \ldots$	predicate symbols	102
$\varphi, \psi, \rho, \ldots$	first-order formulas	103
$p(t_1, \ldots, t_n)$	atomic formula	103
$L_{\omega\omega}$	first-order predicate logic	103
$\varphi[x_1/t_1, \ldots, x_n/t_n]$	simultaneous substitution	105
$\varphi[x_i/t_i \mid 1 \le i \le n]$	simultaneous substitution	105
$\mathfrak{A} = (A,\ \mathfrak{m}_{\mathfrak{A}})$	relational structure	105
GEN	generalization rule	111
$\bigwedge_{\alpha\in A}\varphi_\alpha$	infinitary conjunction	120
$\bigvee_{\alpha\in A}\varphi_\alpha$	infinitary disjunction	120
$\Box$	modal necessity operator	127
$\Diamond$	modal possibility operator	127
$\mathfrak{K} = (K,\ R_{\mathfrak{K}},\ \mathfrak{m}_{\mathfrak{K}})$	Kripke frame for modal logic	127
GEN	modal generalization	130
$a, b, c, \ldots$	modalities	130
$\texttt{[}a\texttt{]}$	multimodal necessity operator	131
$\texttt{<}a\texttt{>}$	multimodal possibility operator	131
$\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$	Kripke frame for multimodal logic	131
first(σ)	first state of a path	132
last(σ)	last state of a path	132
$x := t$	assignment	145

$\alpha\,;\,\beta$	sequential composition 148
$\varphi?$	test 148
$\alpha \cup \beta$	nondeterministic choice 148
α^*	iteration 148
$\sigma, \tau, \ldots$	seqs 150
$CS(\alpha)$	computation sequences of a program 150
$\{\varphi\}\ \alpha\ \{\psi\}$	partial correctness assertion 156
$\varphi, \psi, \ldots$	propositions 164
$\alpha, \beta, \gamma, \ldots$	programs 164
$a, b, c, \ldots$	atomic programs 164
Π_0	set of atomic programs 164
$p, q, r, \ldots$	atomic propositions 164
Φ_0	set of atomic propositions 164
Π	set of programs 164
Φ	set of propositions 164
$[\alpha]$	DL box operator 165
$<\alpha>$	DL diamond operator 166
skip	null program 167
fail	failing program 167
$\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$	Kripke frame for PDL 167
$\mathfrak{m}_{\mathfrak{K}}$	meaning function 167
$u, v, w, \ldots$	states 167
RTC	reflexive transitive closure rule 182
LI	loop invariance rule 182
IND	induction axiom 182
$FL(\varphi)$	Fischer–Ladner closure 191
$FL^{\square}(\varphi)$	Fischer–Ladner closure auxiliary function 191
$\mathfrak{K}/FL(\varphi)$	filtration of a Kripke frame 195
PDA	pushdown automaton 227
$a^{\Delta}ba^{\Delta}$	a nonregular program 228
$\mathsf{PDL} + L$	extension of PDL with a nonregular program 229
$\mathsf{S}k\mathsf{S}$	monadic second-order theory of k successors 229
CFL	context-free language 238

UDH	unique diamond path Hintikka tree . . . 241
PTA	pushdown tree automaton . . . 242
A_ℓ	local automaton . . . 244
$A_\Box$	box automaton . . . 245
$A_\Diamond$	diamond automaton . . . 247
DWP	deterministic **while** programs . . . 259
WP	nondeterministic **while** programs . . . 260
DPDL	deterministic PDL . . . 260
SPDL	strict PDL . . . 260
SDPDL	strict deterministic PDL . . . 260
$\Phi^{(i)}$	programs with nesting of tests at most i . . . 264
PDL$^{(i)}$	PDL with programs in $\Phi^{(i)}$ . . . 264
PDL$^{(0)}$	test-free PDL . . . 264
APDL	automata PDL . . . 267
$-\alpha$	complement of a program . . . 268
$\alpha \cap \beta$	intersection of programs . . . 268
IPDL	PDL with intersection . . . 269
$^-$	converse operator . . . 270
$TC(\varphi, \alpha, \psi)$	total correctness assertion . . . 271
μ	least fixed point operator . . . 271
wf	well-foundedness predicate . . . 272
halt	halt predicate . . . 272
loop	loop operator . . . 272
repeat, Δ	repeat operator . . . 272
RPDL	PDL with well-foundedness predicate . . . 272
LPDL	PDL with halt predicate . . . 272
CRPDL	RPDL with converse . . . 274
CLPDL	LPDL with converse . . . 274
$\wedge$	concurrency operator . . . 277
$F(t_1, \ldots, t_n) := t$	array assignment . . . 288
push(t)	push instruction . . . 289
pop(y)	pop instruction . . . 289
$x := ?$	wildcard assignment . . . 290

MIT Press Math7X9/2010/08/25:15:15 Page 446

Notation	Meaning	Page
w_a	constant valuation	293
DL(r.e.)	DL with r.e. programs	297
DL(array)	DL with array assignments	297
DL(stk)	DL with algebraic stack	297
DL(bstk)	DL with Boolean stack	297
DL(wild)	DL with wildcard assignment	297
DL(dreg)	DL with **while** programs	297
$\mathsf{DL}_1 \leq \mathsf{DL}_2$	no more expressive than	304
$\mathsf{DL}_1 < \mathsf{DL}_2$	strictly less expressive than	304
$\mathsf{DL}_1 \equiv \mathsf{DL}_2$	equivalent in expressive power	304
DL(rich-test r.e.)	rich test DL of r.e. programs	304
$L_{\omega_1^{\mathrm{ck}}\omega}$	constructive infinitary logic	305
$\mathsf{DL}_1 \leq_{\mathbb{N}} \mathsf{DL}_2$	relative expressiveness over $\mathbb{N}$	308
$C_{\mathfrak{A}}$	natural chain in a structure	318
$\mathfrak{A}_w$	expansion of a structure by constants	318
$\textsc{Next}_m$	program computing a natural chain	318
S_n	collection of n-element structures	319
$\ulcorner\mathfrak{A}\urcorner$	code of a structure	320
$SP_m(\alpha)$	m^{th} spectrum of a program	320
$SP(K)$	spectrum of a class of programs	320
H_m^{Σ}	language of codes of structures	321
$SP(K) \approx C$	spectrum $SP(K)$ captures complexity class C	322
APDA	auxiliary pushdown automaton	324
$\vdash_{S1}$	provability in DL Axiom System 14.2	328
$\varphi[x/t]$	substitution into DL formulas	329
$\vdash_{S2}$	provability in DL Axiom System 14.6	331
$\vdash_{S3}$	provability in DL Axiom System 14.9	335
$\vdash_{S4}$	provability in DL Axiom System 14.12	337
$FV(\alpha)$	free variables of an abstract program	344
$K_1 \leq K_2$	translatability	347
$K_1 \preceq_T K_2$	termination subsumption	348
T_n	full binary tree of depth n	355
$Ltr(\pi)$	L-trace of a computation	356

$Cmp(\alpha, A)$	computations of a program in a set 356
$LtrCmp(\alpha, A, n)$	L-traces of length at most n 356
$\mathfrak{G}_n$	structure arising from an Adian group 366
$N(C, k)$	k-neighborhood of a subset of $\mathbb{N}$ 371
AL	Algorithmic Logic 383
NDL	Nonstandard Dynamic Logic 384
$\vdash_{\mathsf{HL}}$	provability in Hoare Logic 385
$\mathfrak{I}$	time model 385
$\models_{\mathsf{NT}}$	satisfiability in nonstandard time semantics 386
LDL	DL with halt predicate 386
RDL	DL with well-foundedness predicate 386
$Pr(\mathsf{DL})$	probabilistic DL 391
LED	Logic of Effective Definitions 397
TL	Temporal Logic 398
$\Box\varphi$	box operator of temporal logic 398
$\Diamond\varphi$	diamond operator of temporal logic 398
$\bigcirc\varphi$	nexttime operator of temporal logic 398
at L_i	at statement 399
NEXT	next relation of TL 400
fin	finiteness predicate of TL 404
inf	infiniteness predicate of TL 404
until	until operator of TL 405
A	temporal operator "for all traces" 406
E	temporal operator "there exists a trace" 406
PL	Process Logic 408
first	temporal operator of PL 408
until	temporal operator of PL 408
first(σ)	first state in σ 409
last(σ)	last state in σ 409
$[\![\ ]\!]$	trace operator of PL 412
$\ll\ \gg$	trace operator of PL 412
$\mu X.\varphi(X)$	least fixpoint of $\varphi(X)$ 416
KA	Kleene algebra 418

KAT	Kleene algebra with tests 421
$\overline{b}$	negation of a Boolean element in KAT 421

Index

* operator,
  See iteration operator
*-continuity, 419
*-continuous Kleene algebra,
  See Kleene algebra
*-continuous dynamic algebra,
  See dynamic algebra

$\mathfrak{A}$-validity, 297
Abelian group, 92
accept configuration, 35
acceptable structure, 311
acceptance, 28, 30, 35, 36
accepting subtree, 35
accessibility relation, 127
acyclic, 13
Adian structure, 365, 380
admissibility, 347, 379
AEXPSPACE, 40
AEXPTIME, 40
AL,
  See Algorithmic Logic
algebra
  dynamic,
  See dynamic algebra
  Kleene,
  See Kleene algebra
  Σ-, 88
  term, 89
algebraic stack, 290
Algorithmic Logic, 383–384, 394
ALOGSPACE, 39
α-recursion theory, 64
alternating Turing machine,
  See Turing machine
analytic hierarchy, 45
and-configuration, 35
annihilator, 22
anti-monotone, 26
antichain, 11
antisymmetric, 6
APSPACE, 40
APTIME, 39
argument of a function, 9
arithmetic hierarchy, 42–45
arithmetical
  completeness, 334, 335, 338, 341
  structure, 308, 311, 334
arity, 5, 283, 288
array, 288
  assignment, 288
  variable, 288
    nullary, 292
as expressive as, 229, 304
assignment
  array, 288
  nondeterministic,
  See wildcard
  random, 290
  rule, 156
  simple, 147, 283, 284
  wildcard, xiii, 288, 290, 377, 380
associativity, 83, 166
asterate, 3, 98
ATIME, 39
atom, 82
atomic
  formula, 284
  program, 147, 283, 284
  symbol, 164
  test, 147
automata PDL, 267
automaton
  auxiliary pushdown, 324, 351
  box, 245
  counter, 63
  diamond, 247
  finite, 131, 266
  local, 244
  ω-, 266
  pushdown, 227
  pushdown k-ary ω-tree, 242
auxiliary pushdown automaton,
  See automaton
axiom, 69
  of choice, 16
  of regularity, 16
  scheme, 71
axiomatization
  DL, 327–341
  equational logic, 99
  equational theory of regular sets, 420
  infinitary logic, 122
  μ-calculus, 417
  PDL, 173–174, 203
  PL, 411
  predicate logic, 111
    with equality, 115
  propositional logic, 77, 82

bijective, 10
binary, 6
  function symbol, 86
  nondeterminism, 377
  relation, 6–8, 420
Birkhoff's theorem, 140
Boolean algebra, 82, 86, 93, 136, 138
  of sets, 137, 138
Boolean satisfiability,
  See satisfiability, propositional

bound occurrence of a variable, 104
bounded memory, 287, 369
box
  automaton, 245
  operator, 165, 398
branching-time TL, 398
Büchi acceptance, 243

canonical homomorphism, 96
Cantor's set theory, 5
capture
  of a complexity class by a spectrum, 322
  of a variable by a quantifier, 104
cardinality, 4
  of $FL(\varphi)$, 194
carrier, 88, 105, 291
Cartesian
  power, 4
  product, 4
chain, 16
  of sets, 17
change of bound variable, 109
choice
  axiom of, 16
  operator, xiv, 164
class, 5
closed, 18
  formula, 105
closure
  congruence, 95, 99
  Fischer–Ladner, 191–195, 267
  of a variety under homomorphic images, 93
  operator, 19
  ordinal, 21, 51
  universal, 105
CLPDL, 274
co-r.e., 42
  in B, 41
coarsest common refinement, 9
coding of finite structures, 318, 320
coinductive, 50
commutativity, 83
compactness, 122, 142, 181, 210, 220, 303
  first-order, 115–116
  propositional, 81–82
comparative schematology, 347, 378
complete disjunctive normal form, 83
complete lattice, 13
completeness, 303
  DL, 341
  equational logic, 99–100
  first-order, 112–115
    with equality, 115
  for a complexity class, 57
  for termination assertions, 341
  infinitary logic, 124–126
  LED, 397
  μ-calculus, 418
  of a deductive system, 70
  of a set of connectives, 75, 135
  PDL, 203–209
  propositional logic, 79–81
  relative, 341
  TL, 407
complexity, 38–40
  class, 57
  of DL, 313
  of DL, 313–324
  of infinitary logic, 126–127
  of PDL, 211–224
  of spectra, 321
composition, 148, 175
  functional, 10, 54
  operator, 164
  relational, 7, 168
  rule, 156, 186
  sequential, xiv
compositionality, 157
comprehension, 5
computability, 27
  relative, 40–41
computation, 356, 364
  formal, 356
  history, 44
  legal, 365
  on an infinite tree, 243
  sequence, 150, 170
  strongly r-periodic, 365
  terminating, 356
  upward periodic, 365
computational complexity, 64
conclusion, 70
concurrent DL, 393
concurrent PDL, 277
concurrent systems, 406
conditional, 148, 167
  rule, 156, 186
configuration, 29, 243
  n-, 370
congruence, 94
  class, 94
  closure,
  See closure
connective, 71
$coNP$, 57
  -completeness, 57
  -hardness, 57
consequence

deductive, 70
logical, 69, 91
consistency, 70, 138, 174, 203
constant, 86, 284
test, 148
constructive $L_{\omega_1\omega}$, 305
context-free
DL, 298
language
simple-minded, 238
PDL, 230
program, 227, 230
set of seqs, 151
continuity, 11, 17, 417
*- (star-),
See *-continuity
of <> in presence of converse, 179
contraposition, 136
converse, 7, 10, 177, 203, 270
Cook's theorem, 57
correctness
partial,
See partial correctness
specification, 152
total,
See total correctness
countable, 16
ordinal,
See ordinal
countably infinite, 16
counter automaton,
See automaton
counter machine,
See Turing machine
CRPDL, 274
currying, 25, 142

dag, 13
De Morgan law, 83, 137
infinitary, 123, 143
decidability, 37, 42
of PDL, 191, 199
of propositional logic, 75
deduction theorem, 79, 209
first-order, 111–112
infinitary, 124
deductive consequence, 70
deductive system, 67
Δ^0_1, 43
Δ^0_n, 43
Δ^1_1, 45, 51
ΔPDL, 272
dense, 6, 120
descriptive set theory, 64
deterministic
Kripke frame, 259
semantically, 188, 259
while program, 147
diagonalization, 38, 63
diamond
automaton, 247
operator, 166, 398
difference sequence, 250
directed graph, 13
disjunctive normal form, 83
distributivity, 83
infinitary, 123, 143
divergence-closed, 350
DL,
See Dynamic Logic
concurrent, 393
DN, 77
domain, 88, 105
of a function, 9
of computation, 145, 283, 291
double negation, 77, 83, 136
double-exponential time, 39
DPDL, 260
DSPACE, 39
DTIME, 39
duality, 135–136, 166, 172
duplicator, 119
DWP, 259
dyadic, 6
dynamic
formula, 383
term, 383
dynamic algebra, 389–391
*-continuous, 390
separable, 390
Dynamic Logic, 133
axiomatization, 329
basic, 284
context-free, 298
poor test, 148, 284
probabilistic, 391
rich test, 148, 285, 286
of r.e. programs, 304

edge, 13
effective definitional scheme, 397
EFQ, 78
Ehrenfeucht–Fraïssé games, 119–120
emptiness problem
for PTA, 243
empty
relation, 6
sequence, 3

set, 4
string, 3
endogenous, 157, 398
enumeration machine, 63
epimorphism, 90
equal expressive power, 304
equality symbol, 284
equation, 88
equational logic, 86–102
axiomatization, 99
equational theory, 91
equationally defined class,
See variety
equivalence
class, 9
of Kripke frames, 132
of logics, 304, 380
relation, 6, 8–9
eventuality, 402
excluded middle, 136
exogenous, 157
expanded vocabulary, 318
exponential time, 39
expressive structure, 334
expressiveness
of DL, 353
relative, 343, 378
over $\mathbb{N}$, 308
EXPSPACE, 40
EXPTIME, 40

fairness, 290
filter, 138
filtration, 191, 195–201, 273
for nonstandard models, 199–201, 204
finitary, 17
finite
automaton, 131, 266
branching, 65
intersection property, 81
model property,
See small model property
model theorem,
See small model theorem
satisfiability, 81, 115
variant, 293
first-order
logic,
See predicate logic
spectrum, 325
test, 380
vocabulary, 283
Fischer–Ladner closure, 191–195, 267
fixpoint, 18

forced win, 119
formal computation, 356
formula, 283
atomic, 284
DL, 286, 297
dynamic, 383
first-order, 103
Horn, 88
positive in a variable, 416
free, 97
algebra, 97–99
Boolean algebra, 137
commutative ring, 98
for a variable in a formula, 104
monoid, 98
occurrence of a variable, 104
in DL, 329
variable, 105
vector space, 99
function, 9–10
patching, 10, 105, 106, 292
projection, 4
Skolem, 142
symbol, 283
functional composition,
See composition
fusion of traces, 409

Galois connection, 25
game, 48
generalization rule, 111, 173, 203
generate, 89
generating set, 89
graph, 13
directed, 13
greatest lower bound,
See infimum
ground term, 87, 103
guarded command, 167, 187
guess and verify, 34, 40

halt, 30
halt, 272, 386
halting problem, 37, 55
for IND programs, 65
over finite interpretations, 322
undecidability of, 63
hard dag, 371
hardness, 57
Herbrand-like state, 317, 349
Hilbert system, 69
Hoare Logic, 133, 156, 186
homomorphic image, 90
homomorphism, 76, 89

canonical, 96
Horn formula, 88, 140
infinitary, 140
HSP theorem, 100–102
hyperarithmetic relation, 50
hyperelementary relation, 50, 51

ideal
of a Boolean algebra, 138
of a commutative ring, 95
idempotence, 83
infinitary, 143
identity relation, 7, 88
image of a function, 10
IND, 45–51, 64
independence, 16
individual, 88
variable, 284, 286
induction
axiom
PDL, 173, 182, 183, 201
Peano arithmetic, 174, 183
principle, 12, 13
for temporal logic, 401
transfinite, 14
structural, 12, 157
transfinite, 12, 15–16, 117
well-founded, 12–13
inductive
assertions method, 399, 401
definability, 45–53, 64
relation, 49, 51
infimum, 13
infinitary completeness
for DL, 341
infinitary logic, 120–127
infinite descending chain, 24
infix, 87
initial state, 293, 304, 317
injective, 10
input variable, 147
input/output
pair, 168, 291
relation, 147, 169, 287, 293
specification, 153, 154
intermittent assertions method, 398, 402
interpretation of temporal operators in PL, 409
interpreted reasoning, 307, 333
intuitionistic propositional logic, 79, 136
invariant, 399, 401
invariant assertions method, 157, 401
inverse, 10
IPDL,
See PDL with intersection
irreflexive, 6
isomorphism, 90
local, 119
iteration operator, xiv, 164, 181, 390

join, 11

K, 77
k-counter machine,
See Turing machine
k-fold exponential time, 39
k-neighborhood, 371
KA,
See Kleene algebra
KAT,
See Kleene algebra with tests
kernel, 90
Kleene algebra, 389, 418
*-continuous, 390, 419
typed, 423
with tests, 421
Kleene's theorem, 51, 63, 64
Knaster–Tarski theorem, 20–22, 37, 416
König's lemma, 61, 65, 387, 388
Kripke frame, 127, 167, 291, 292
nonstandard, 199, 204, 205, 210, 211

LOGSPACE, 39
L-trace, 356
language, 67–68
first-order DL, 283
lattice, 13, 92
complete, 13
LDL, 386
leaf, 50
least fixpoint, 49, 415, 416
least upper bound, 11
LED, 397
legal computation, 365
lexicographic order, 23
limit ordinal,
See ordinal
linear
order,
See total order
recurrence, 255, 257
linear-time TL, 398
literal, 82
liveness property, 402
local
automaton, 244
isomorphism, 119
logarithmic space, 39

logic, 67
Logic of Effective Definitions, 397
logical consequence, 69, 91, 106, 172
  in PDL, 209, 216, 220–224
logical equivalence, 106, 163
$L_{\omega_1\omega}$, 120, 142, 305
  constructive, 305
$L_{\omega_1^{\mathrm{ck}}\omega}$, 120, 142, 304, 305
$L_{\omega\omega}$,
  See predicate logic
loop, 30
  -free program, 385
  invariance rule, 182, 184, 201
loop, 272, 386
Löwenheim–Skolem theorem, 116–117, 302
  downward, 116, 122, 126
  upward, 116, 122, 142, 346
lower bound
  for PDL, 216–220
LPDL, 272

m-state, 317
many-one reduction, 53
maximal consistent set, 204
meaning,
  See semantics
  function, 167, 291
meet, 13
membership problem, 37, 55
meta-equivalence, 3
meta-implication, 3, 73
method of well-founded sets, 402
min,+ algebra, 420
modal logic, xiv, 127–134, 164, 167, 191
modal μ-calculus,
  See μ-calculus
modality, 130
model, 68, 106
  nonstandard, 384
model checking, 199, 202, 211
  for the μ-calculus, 418
model theory, 68
modus ponens, 77, 111, 173, 203
monadic, 6
mono-unary vocabulary, 319
monoid, 92
  free, 98
monomorphism, 90
monotone, 11, 17
  Boolean formula, 135
monotonicity, 416
MP, 77
m^{th} spectrum, 320
μ operator, 415
μ-calculus, 271, 415, 417
multimodal logic, 130–132
multiprocessor systems, 406

n-ary
  function symbol, 86
  relation, 6
n-configuration, 370
n-pebble game, 120, 370
natural chain, 318, 325
natural deduction, 69
NDL, 384, 394
necessity, 127
neighborhood, 371
NEXPSPACE, 40
NEXPTIME, 40
next configuration relation, 29
nexttime operator, 398
NLOGSPACE, 39
nondeterminism, 63, 151, 158
  binary, 377
  unbounded, 377
nondeterministic
  assignment,
  See wildcard assignment
  choice, xiv, 175
  program, 133
  Turing machine,
  See Turing machine
  while program, 285
nonstandard
  Kripke frame, 199, 204, 205, 210, 211
  model, 384
Nonstandard DL, 384–386, 394
normal subgroup, 95
not-configuration, 36
not-state, 36
NP, 40, 57
  -completeness, 57, 220
  -hardness, 57
NPSPACE, 39
NPTIME, 39
NTIME, 39
nullary, 6
  array variable, 292
  function symbol, 86
number theory, 103, 140
  second-order, 45

occurrence, 104
ω-automaton, 266
ω_1^{ck}, 50
one-to-one, 10
  correspondence, 10

onto, 10
or-configuration, 35
oracle, 41
  Turing machine,
  See Turing machine
ord, 50, 124
ordinal, 14
  countable, 50
  limit, 14, 15
  recursive, 45
  successor, 14, 15
  transfinite, 13–15
output variable, 147

P, 40
p-sparse, 367
$P{=}NP$ problem, 40, 76
pairing function, 142
parameterless recursion, 227, 290
parentheses, 72, 166
partial
  correctness, 154
    assertion, 133, 167, 187, 313, 316, 325, 385
  order, 6, 10–12
    strict, 6, 11
partition, 9
path, 50, 132
PDL, 163–277
  automata, 267
  concurrent, 277
  poor test, 263
  regular, 164
  rich test, 165, 263
  test-free, 264
  with intersection, 269
$\mathsf{PDL}^{(0)}$, 224
Peano arithmetic
  induction axiom of, 174
pebble game, 120, 370
Peirce's law, 136
Π^0_1, 43
Π^0_n, 43
Π^1_1, 45, 51, 126
  -completeness, 222
PL,
  See Process Logic
polyadic, 369
polynomial, 98
  space, 39
  time, 39
poor
  test, 148, 263, 284
  vocabulary, 319
positive, 49
possibility, 127
postcondition, 154
postfix, 87
precedence, 72, 103, 166
precondition, 154
predicate logic, 102–119
predicate symbol, 283
prefix, 87
prefixpoint, 18
premise, 70
prenex form, 45, 109
preorder, 6, 10
probabilistic program, 391–393
Process Logic, 408
product, 100
program, 145, 283, 287
  atomic, 147, 283, 284
  DL, 284
  loop-free, 385
  operator, 147
  probabilistic, 391–393
  r.e., 287, 296
  regular, 148, 169, 285
  schematology, 311
  scheme, 302
  simulation, 347
  uniformly periodic, 365
  variable, 286
  while, 149
  with Boolean arrays, 380
programming language
  semi-universal, 349
projection function, 4
proof, 70
proper class,
  See class
proposition, 72
propositional
  formula, 72
  logic, 71–86
    intuitionistic, 136
  operators, 71
  satisfiability,
  See satisfiability
Propositional Dynamic Logic,
  See PDL
$PSPACE$, 39
$PTIME$, 39
pushdown
  k-ary ω-tree automaton, 242
  automaton, 227
  store,
  See stack

quantifier, 102
  depth, 119
quasiorder,
  See preorder
quasivariety, 140
quotient
  algebra, 96
  construction, 96–97

Ramsey's theorem, 24
random assignment,
  See assignment
range of a function, 9
RDL, 386
r.e., 30, 42, 63
  in B, 41
  program, 287, 296
reasoning
  interpreted, 307, 333
  uninterpreted, 301, 327
recursion, 149, 289
  parameterless, 227, 290
recursive, 30, 42
  call, 149
  function theory, 63
  in B, 41
  ordinal, 45, 50–51
  tree, 50–51
recursively enumerable,
  See r.e.
reducibility, 54
  relation, 53–56, 63
reductio ad absurdum, 136
reduction
  many-one, 53
refinement, 6, 9
reflexive, 6
reflexive transitive closure, 8, 20, 47, 182, 183, 200
refutable, 70
regular
  expression, 164, 169, 190
  program, 148, 169, 285
    with arrays, 288
    with Boolean stack, 364
    with stack, 289
  set, 170, 420
reject configuration, 35
rejection, 28, 30
relation, 5
  binary, 6–8
  empty, 6
  hyperarithmetic, 50
  hyperelementary, 50, 51
  next configuration, 29
  reducibility, 53–56, 63
  symbol, 283
  universal, 45
  well-founded,
  See well-founded
relational
  composition,
  See composition
  structure, 105
relative
  completeness, 341
  computability, 40–41
  expressiveness, 343, 378
    over $\mathbb{N}$, 308
repeat, 272, 386
representation by sets, 76–77
resolution, 69
rich
  test, 148, 165, 263, 285, 286
  vocabulary, 319
ring, 92
  commutative, 92
RPDL, 272
rule of inference, 69
run, 386
Russell's paradox, 5

S, 77
safety property, 401
satisfaction
  equational logic, 90
  first-order, 106
  PDL, 168
  propositional, 74
  relation, 106
  TL, 400
satisfiability
  algorithm for PDL, 191, 213
  Boolean,
  See satisfiability, propositional
  DL, 297, 298
  finite, 81
  modal logic, 128
  PDL, 171, 191, 211
  propositional, 57, 74, 76, 129, 220
scalar multiplication, 390
schematology, 302, 347, 378
scope, 104
SDPDL, 224, 260
second-order number theory, 45
Segerberg axioms,
  See axiomatization, PDL
semantic determinacy, 188, 259

semantics, 67, 68
  abstract programs, 344
  DL, 291–298
  equational logic, 88
  infinitary logic, 120
  modal logic, 127
  multimodal logic, 131
  PDL, 167–170
  predicate logic, 105–109
    with equality, 115
  propositional logic, 73–74
semi-universal, 349, 350
semigroup, 92
semilattice, 13, 92
sentence, 69, 105
separable dynamic algebra,
  See dynamic algebra
seq, 150, 170, 287
sequent, 69
sequential composition,
  See composition
set operator, 16–22
Σ-algebra, 88
Σ^0_1, 43
Σ^0_n, 43
signature,
  See vocabulary
simple assignment,
  See assignment
simple-minded
  context-free language, 238, 256
  pushdown automaton, 238
simulation, 37, 347
Skolem function, 142
Skolemization, 142
SkS, 229
small model
  property, 191, 198, 227
  theorem, 198, 211
soundness, 70
  equational logic, 99–100
  modal logic, 128
  PDL, 172, 174
SPDL, 260
specification
  correctness, 152
  input/output, 153, 154
spectral
  complexity, 317, 321, 322
  theorem, 353, 379
spectrum, 379
  first-order, 325
  m^{th}, 320
  of a formula, 320
  second-order, 325
spoiler, 119
stack, 149, 288, 289
  algebraic, 290
  automaton, 249
  Boolean, 290
  configuration, 243
  higher-order, 325
  operation, 289
standard Kripke frame, 390
* operator,
  See iteration operator
*-continuity, 419
*-continuous dynamic algebra,
  See dynamic algebra
*-continuous Kleene algebra,
  See Kleene algebra
start configuration, 29
state, 127, 146, 167, 291, 293, 400
  Herbrand-like, 317, 349
  initial, 293, 304, 317
  m-, 317
static logic, 304
stored-program computer, 37
strict partial order, 6, 11
strictly more expressive than, 229, 304
strongly r-periodic, 365
structural induction,
  See induction
structure, 68
  acceptable, 311
  Adian, 365
  arithmetical, 308
  expressive, 334
  p-sparse, 367
  relational, 105
  treelike, 261, 262, 355, 367
subalgebra, 89
  generated by, 89
subexpression relation, 191
substitution, 90
  in DL formulas, 329
  instance, 90, 222
  operator, 105, 106
  rule, 84, 138
succession, 370
successor ordinal,
  See ordinal
supremum, 11
surjective, 10
symbol
  atomic, 164
  constant, 284
  equality, 284

function, 283
predicate, 283
relation, 102, 283
symmetric, 6
syntactic
continuity, 416
interpretation, 89
monotonicity, 416
syntax, 67

tableau, 69
tail recursion, 150
tautology, 74, 129
infinitary, 142
Temporal Logic, 133, 157, 398
branching-time, 133
linear-time, 133
temporal operators, 401
interpretation in PL, 409
term, 87, 103
algebra, 89
dynamic, 383
ground, 87, 103
termination, 166, 356
assertion, 327
properties of finite interpretations, 351
subsumption, 348, 379
ternary, 6
function symbol, 86
test, 147, 175
-free, 264
atomic, 147
first-order, 380
operator, xiv, 165, 284
poor,
See poor test
rich,
See rich test
theorem, 70
theory, 69
tile, 58
tiling problem, 58–63, 117, 126, 222
time model, 385
time-sharing, 42, 53
TL,
See temporal logic
topology, 81
total, 30
correctness, 155, 271, 327
assertion, 313
order, 6, 11
trace, 146
in PL, 409
L-, 356
quantifier, 406
transfinite
induction,
See induction
ordinal,
See ordinal
transition function, 28
transitive, 6
closure, 8, 19
set, 14
transitivity, 192
of implication, 78
of reductions, 54
translatability, 347
trap, 366
tree, 50
model, 143, 239
model property, 239
structure, 239
well-founded, 50
tree-parent, 371
treelike structure, 132, 261, 262, 355, 367
truth, 106
assignment, 74
table, 134
value, 73, 74
Turing machine, 27–37, 63
alternating, 34–37, 46, 216, 225
with negation, 36–37
deterministic, 28
nondeterministic, 33–34
oracle, 40–42
universal, 37, 63
with k counters, 32–33
with two stacks, 31–32
typed Kleene algebra,
See Kleene algebra

UDH tree, 241
ultrafilter, 138
unary, 6
function symbol, 86
unbounded nondeterminism, 377
undecidability, 37, 42
of predicate logic, 117–119
of the halting problem, 63
uniform
periodicity, 365
simulation, 37
uninterpreted reasoning, 301, 327
unique diamond path Hintikka tree,
See UDH tree
universal
closure,

See closure
formula, 141
model, 208, 211
relation, 7, 45
Turing machine,
See Turing machine
universality problem, 62
universe, 127
until operator, 405
unwind property, 344, 379
unwinding, 132
upper bound, 11
least, 11
upper semilattice, 13
upward periodic, 365
use vs. mention, 73

validity, 69
$\mathfrak{A}$-, 297
DL, 297, 298, 313
equational logic, 91
first-order, 106
modal logic, 128
PDL, 171
propositional, 74
valuation, 90, 105, 146, 283, 291, 292
value of a function, 9
variable, 145
array, 288
individual, 87, 284, 286
program, 286
work, 147
variety, 92
vector space, 93
verification conditions, 333
vertex, 13
vocabulary, 86
expanded, 318
first-order, 102, 283
monadic, 369
mono-unary, 319
polyadic, 369
poor, 319
rich, 319

weakening rule, 156, 186
well order, 6
well ordering principle, 16
well partial order, 12
well quasiorder, 12
well-founded, 6, 11, 121, 271, 402
induction,
See induction
relation, 12, 48
tree, 50
well-foundedness, 386–389
wf, 272, 386
while
loop, 167, 260
operator, 148
program, 149, 259
deterministic, 285
nondeterministic, 285
with arrays, 288
with stack, 289
rule, 156, 186
wildcard assignment,
See assignment
work variable, 147
world, 127
WP, 260

Zermelo's theorem, 16
Zermelo–Fraenkel set theory, 4, 16
ZF, 16
ZFC, 4, 16
Zorn's lemma, 16, 138