4 Reasoning About Programs

In subsequent chapters, we will study in depth a family of program logics collectively called Dynamic Logic (DL). Before embarking on this task, we take the opportunity here to discuss program verification in general and introduce some key concepts on an informal level. Many of the ideas discussed in this chapter will be developed in more detail later on.

4.1 What are Programs?

For us, a *program* is a recipe written in a formal language for computing desired output data from given input data.

Example 4.1: The following program implements the Euclidean algorithm for calculating the greatest common divisor (gcd) of two integers. It takes as input a pair of integers in variables x and y and outputs their gcd in variable x:

while $y \neq 0$ **do**
  begin
    $z := x \bmod y$;
    $x := y$;
    $y := z$
  end

The value of the expression $x \bmod y$ is the (nonnegative) remainder obtained when dividing x by y using ordinary integer division.

Programs normally use *variables* to hold input and output values and intermediate results. Each variable can assume values from a specific *domain of computation*, which is a structure consisting of a set of data values along with certain distinguished constants, basic operations, and tests that can be performed on those values, as described in Section 3.4. In the program above, the domain of x, y, and z might be the integers $\mathbb{Z}$ along with basic operations including integer division with remainder and tests including $\neq$. In contrast with the usual use of variables in mathematics, a variable in a program normally assumes different values during the course of the computation. The value of a variable x may change whenever an assignment $x := t$ is performed with x on the left-hand side.

In order to make these notions precise, we will have to specify the programming language and its semantics in a mathematically rigorous way. In this text we will consider several programming languages with various properties, and each will be defined formally. In this chapter we give a brief introduction to some of these languages and the role they play in program verification.

4.2 States and Executions

As mentioned above, a program can change the values of variables as it runs. However, if we could freeze time at some instant during the execution of the program, we could presumably read the values of the variables at that instant, and that would give us an instantaneous snapshot of all information that we would need to determine how the computation would proceed from that point. This leads to the concept of a *state*—intuitively, an instantaneous description of reality.

Formally, we will define a *state* to be a function that assigns a value to each program variable. The value for variable x must belong to the domain associated with x. In logic, such a function is called a *valuation* (see Sections 3.3 and 3.4). At any given instant in time during its execution, the program is thought to be "in" some state, determined by the instantaneous values of all its variables. If an assignment statement is executed, say $x := 2$, then the state changes to a new state in which the new value of x is 2 and the values of all other variables are the same as they were before. We assume that this change takes place instantaneously; note that this is a mathematical abstraction, since in reality basic operations take some time to execute.

A typical state for the gcd program above is $(15, 27, 0, \ldots)$, where (say) the first, second, and third components of the sequence denote the values assigned to x, y, and z respectively. The ellipsis "..." refers to the values of the other variables, which we do not care about, since they do not occur in the program.

A program can be viewed as a transformation on states. Given an initial (input) state, the program will go through a series of intermediate states, perhaps eventually halting in a final (output) state. A sequence of states that can occur from the execution of a program α starting from a particular input state is called a *trace*. As a typical example of a trace for the program above, consider the initial state $(15, 27, 0)$ (we suppress the ellipsis). The program goes through the following sequence of states:

$(15, 27, 0),\ (15, 27, 15),\ (27, 27, 15),\ (27, 15, 15),\ (27, 15, 12),\ (15, 15, 12),$
$(15, 12, 12),\ (15, 12, 3),\ (12, 12, 3),\ (12, 3, 3),\ (12, 3, 0),\ (3, 3, 0),\ (3, 0, 0).$

The value of x in the last (output) state is 3, the gcd of 15 and 27.

The binary relation consisting of the set of all pairs of the form (input state, output state) that can occur from the execution of a program α, or in other words, the set of all first and last states of traces of α, is called the *input/output relation* of α. For example, the pair $((15, 27, 0), (3, 0, 0))$ is a member of the input/output relation of the gcd program above, as is the pair $((-6, -4, 303), (2, 0, 0))$. The values of other variables besides x, y, and z are not changed by the program. These values are therefore the same in the output state as in the input state. In this example, we may think of the variables x and y as the *input variables*, x as the *output variable*, and z as a *work variable*, although formally there is no distinction between any of the variables, including the ones not occurring in the program.

4.3 Programming Constructs

In subsequent sections we will consider a number of programming constructs. In this section we introduce some of these constructs and define a few general classes of languages built on them.

In general, programs are built inductively from *atomic programs* and *tests* using various *program operators*.

While Programs

A popular choice of programming language in the literature on DL is the family of deterministic **while** programs. This language is a natural abstraction of familiar imperative programming languages such as Pascal or C. Different versions can be defined depending on the choice of tests allowed and whether or not nondeterminism is permitted.

The language of **while** programs is defined inductively. There are atomic programs and atomic tests, as well as program constructs for forming compound programs from simpler ones.

In the propositional version of Dynamic Logic (PDL), atomic programs are simply letters $a, b, \ldots$ from some alphabet. Thus PDL abstracts away from the nature of the domain of computation and studies the pure interaction between programs and propositions. For the first-order versions of DL, atomic programs are *simple assignments* $x := t$, where x is a variable and t is a term. In addition, a *nondeterministic* or *wildcard assignment* $x :=?$ or *nondeterministic choice* construct may be allowed.

Tests can be *atomic tests*, which for propositional versions are simply proposi-

tional letters p, and for first-order versions are atomic formulas $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and p is an n-ary relation symbol in the signature of the domain of computation. In addition, we include the *constant tests* **1** and **0**. Boolean combinations of atomic tests are often allowed, although this adds no expressive power. These versions of DL are called *poor test*.

More complicated tests can also be included. These versions of DL are sometimes called *rich test*. In rich test versions, the families of programs and tests are defined by mutual induction.

Compound programs are formed from the atomic programs and tests by induction, using the *composition*, *conditional*, and *while* operators. Formally, if φ is a test and α and β are programs, then the following are programs:

- $\alpha\,;\,\beta$
- **if** φ **then** α **else** β
- **while** φ **do** α.

We can also parenthesize with **begin** ... **end** where necessary. The gcd program of Example 4.1 above is an example of a **while** program.

The semantics of these constructs is defined to correspond to the ordinary operational semantics familiar from common programming languages. We will give more detail about these programs in Sections 5.1 and 5.2.

Regular Programs

Regular programs are more general than **while** programs, but not by much. The advantage of regular programs is that they reduce the relatively more complicated **while** program operators to much simpler constructs. The deductive system becomes comparatively simpler too. They also incorporate a simple form of nondeterminism.

For a given set of atomic programs and tests, the set of *regular programs* is defined as follows:

(i) any atomic program is a program

(ii) if φ is a test, then $\varphi?$ is a program

(iii) if α and β are programs, then $\alpha\,;\,\beta$ is a program;

(iv) if α and β are programs, then $\alpha \cup \beta$ is a program;

(v) if α is a program, then α^* is a program.

These constructs have the following intuitive meaning:

(i) Atomic programs are basic and indivisible; they execute in a single step. They are called *atomic* because they cannot be decomposed further.

(ii) The program $\varphi?$ tests whether the property φ holds in the current state. If so, it continues without changing state. If not, it blocks without halting.

(iii) The operator ; is the *sequential composition* operator. The program $\alpha\,;\,\beta$ means, "Do α, then do β."

(iv) The operator $\cup$ is the *nondeterministic choice* operator. The program $\alpha \cup \beta$ means, "Nondeterministically choose one of α or β and execute it."

(v) The operator * is the *iteration* operator. The program α means, "Execute α some nondeterministically chosen finite number of times."

Keep in mind that these descriptions are meant only as intuitive aids. A formal semantics will be given in Section 5.2, in which programs will be interpreted as binary input/output relations and the programming constructs above as operators on binary relations.

The operators $\cup, ;, ^*$ may be familiar from automata and formal language theory (see Kozen (1997a)), where they are interpreted as operators on sets of strings over a finite alphabet. The language-theoretic and relation-theoretic semantics share much in common; in fact, they have the same equational theory, as shown in Kozen (1994a).

The operators of deterministic **while** programs can be defined in terms of the regular operators:

$$\textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta \stackrel{\text{def}}{=} \varphi? \,;\, \alpha \cup \neg\varphi? \,;\, \beta \tag{4.3.1}$$

$$\textbf{while } \varphi \textbf{ do } \alpha \stackrel{\text{def}}{=} (\varphi? \,;\, \alpha)^* \,;\, \neg\varphi? \tag{4.3.2}$$

The class of **while** programs is equivalent to the subclass of the regular programs in which the program operators $\cup$, ?, and * are constrained to appear only in these forms.

The definitions (4.3.1) and (4.3.2) may seem a bit mysterious at first, but we will be able to justify them after we have discussed binary relation semantics in Section 5.2.

Recursion

Recursion can appear in programming languages in several forms. We will study two such manifestations: *recursive calls* and *stacks*. We will show that under certain very general conditions, the two constructs can simulate each other. We will also show that recursive programs and **while** programs are equally expressive over the natural

numbers, whereas over arbitrary domains, **while** programs are strictly weaker. **While** programs correspond to what is often called *tail recursion* or *iteration*.

R.E. Programs

A *finite computation sequence* of a program α, or *seq* for short, is a finite-length string of atomic programs and tests representing a possible sequence of atomic steps that can occur in a halting execution of α. Seqs are denoted $\sigma, \tau, \ldots$. The set of all seqs of a program α is denoted $CS(\alpha)$. We use the word "possible" loosely—$CS(\alpha)$ is determined by the syntax of α alone. Because of tests that evaluate to false, $CS(\alpha)$ may contain seqs that are never executed under any interpretation.

The set $CS(\alpha)$ is a subset of A^*, where A is the set of atomic programs and tests occurring in α. For **while** programs, regular programs, or recursive programs, we can define the set $CS(\alpha)$ formally by induction on syntax. For example, for regular programs,

$$\begin{aligned}
CS(a) &\stackrel{\text{def}}{=} \{a\}, \quad a \text{ an atomic program or test} \\
CS(\mathbf{skip}) &\stackrel{\text{def}}{=} \{\varepsilon\} \\
CS(\mathbf{fail}) &\stackrel{\text{def}}{=} \varnothing \\
CS(\alpha\,;\,\beta) &\stackrel{\text{def}}{=} \{\sigma\,;\,\tau \mid \sigma \in CS(\alpha),\ \tau \in CS(\beta)\} \\
CS(\alpha \cup \beta) &\stackrel{\text{def}}{=} CS(\alpha) \cup CS(\beta) \\
CS(\alpha^*) &\stackrel{\text{def}}{=} CS(\alpha)^* \\
&= \bigcup_{n \geq 0} CS(\alpha^n),
\end{aligned}$$

where

$$\begin{aligned}
\alpha^0 &\stackrel{\text{def}}{=} \mathbf{skip} \\
\alpha^{n+1} &\stackrel{\text{def}}{=} \alpha^n\,;\,\alpha.
\end{aligned}$$

For example, if a is an atomic program and p an atomic formula, then the program

$$\mathbf{while}\ p\ \mathbf{do}\ a \quad = \quad (p?\,;\,a)^*\,;\,\neg p?$$

has as seqs all strings of the form

$$(p?\,;\,a)^n\,;\,\neg p? \quad = \quad \underbrace{p?;a;p?;a;\cdots;p?;a}_{n};\neg p?$$

for all $n \geq 0$. Note that each seq σ of a program α is itself a program, and

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

While programs and regular programs give rise to regular sets of seqs, and recursive programs give rise to context-free sets of seqs. Taking this a step further, we can define an *r.e. program* to be simply a recursively enumerable set of seqs. This is the most general programming language we will consider in the context of DL; it subsumes all the others in expressive power.

Nondeterminism

We should say a few words about the concept of *nondeterminism* and its role in the study of logics and languages, since this concept often presents difficulty the first time it is encountered.

In some programming languages we will consider, the traces of a program need not be uniquely determined by their start states. When this is possible, we say that the program is *nondeterministic*. A nondeterministic program can have both divergent and convergent traces starting from the same input state, and for such programs it does not make sense to say that the program halts on a certain input state or that it loops on a certain input state; there may be different computations starting from the same input state that do each.

There are several concrete ways nondeterminism can enter into programs. One construct is the *nondeterministic* or *wildcard assignment* $x := ?$. Intuitively, this operation assigns an arbitrary element of the domain to the variable x, but it is not determined which one.[1] Another source of nondeterminism is the unconstrained use of the choice operator $\cup$ in regular programs. A third source is the iteration operator * in regular programs. A fourth source is r.e. programs, which are just r.e. sets of seqs; initially, the seq to execute is chosen nondeterministically. For example, over $\mathbb{N}$, the r.e. program

$\{x := n \mid n \geq 0\}$

is equivalent to the regular program

$x := 0\,;\ (x := x + 1)^*$.

Nondeterministic programs provide no explicit mechanism for resolving the nondeterminism. That is, there is no way to determine which of many possible

1 This construct is often called *random assignment* in the literature. This terminology is misleading, because it has nothing at all to do with probability.

next steps will be taken from a given state. This is hardly realistic. So why study nondeterminism at all if it does not correspond to anything operational? One good answer is that nondeterminism is a valuable tool that helps us understand the expressiveness of programming language constructs. It is useful in situations in which we cannot necessarily predict the outcome of a particular choice, but we may know the range of possibilities. In reality, computations may depend on information that is out of the programmer's control, such as input from the user or actions of other processes in the system. Nondeterminism is useful in modeling such situations.

The importance of nondeterminism is not limited to logics of programs. Indeed, the most important open problem in the field of computational complexity theory, the $P{=}NP$ problem, is formulated in terms of nondeterminism.

4.4 Program Verification

Dynamic Logic and other program logics are meant to be useful tools for facilitating the process of producing correct programs. One need only look at the miasma of buggy software to understand the dire need for such tools. But before we can produce correct software, we need to know what it means for it to be correct. It is not good enough to have some vague idea of what is supposed to happen when a program is run or to observe it running on some collection of inputs. In order to apply formal verification tools, we must have a formal specification of correctness for the verification tools to work with.

In general, a *correctness specification* is a formal description of how the program is supposed to behave. A given program is *correct* with respect to a correctness specification if its behavior fulfills that specification. For the gcd program of Example 4.1, the correctness might be specified informally by the assertion

If the input values of x and y are positive integers c and d, respectively, then

(i) the output value of x is the gcd of c and d, and

(ii) the program halts.

Of course, in order to work with a formal verification system, these properties must be expressed formally in a language such as first-order logic.

The assertion (ii) is part of the correctness specification because programs do not necessarily halt, but may produce infinite traces for certain inputs. A finite trace, as for example the one produced by the gcd program above on input state (15,27,0), is called *halting*, *terminating*, or *convergent*. Infinite traces are called *looping* or *divergent*. For example, the program

while $x > 7$ **do** $x := x + 3$

loops on input state $(8, \ldots)$, producing the infinite trace

$(8, \ldots),\ (11, \ldots),\ (14, \ldots),\ \ldots$

For the purposes of this book, we will limit our attention to the behavior of a program that is manifested in its input/output relation. Dynamic Logic is not tailored to reasoning about program behavior manifested in intermediate states of a computation (although there are close relatives, such as Process Logic and Temporal Logic, that are). This is not to say that all interesting program behavior is captured by the input/output relation, and that other types of behavior are irrelevant or uninteresting. Indeed, the restriction to input/output relations is reasonable only when programs are supposed to halt after a finite time and yield output results. This approach will not be adequate for dealing with programs that normally are not supposed to halt, such as operating systems.

For programs that are supposed to halt, correctness criteria are traditionally given in the form of an *input/output specification* consisting of a formal relation between the input and output states that the program is supposed to maintain, along with a description of the set of input states on which the program is supposed to halt. The input/output relation of a program carries all the information necessary to determine whether the program is correct relative to such a specification. Dynamic Logic is well suited to this type of verification.

It is not always obvious what the correctness specification ought to be. Sometimes, producing a formal specification of correctness is as difficult as producing the program itself, since both must be written in a formal language. Moreover, specifications are as prone to bugs as programs. Why bother then? Why not just implement the program with some vague specification in mind?

There are several good reasons for taking the effort to produce formal specifications:

1. Often when implementing a large program from scratch, the programmer may have been given only a vague idea of what the finished product is supposed to do. This is especially true when producing software for a less technically inclined employer. There may be a rough informal description available, but the minor details are often left to the programmer. It is very often the case that a large part of the programming process consists of taking a vaguely specified problem and making it precise. The process of formulating the problem precisely can be considered a *definition* of what the program is supposed to do. And it is just good programming practice to have a very clear idea of what we want to do before we start doing it.

2. In the process of formulating the specification, several unforeseen cases may become apparent, for which it is not clear what the appropriate action of the program should be. This is especially true with error handling and other exceptional situations. Formulating a specification can define the action of the program in such situations and thereby tie up loose ends.

3. The process of formulating a rigorous specification can sometimes suggest ideas for implementation, because it forces us to isolate the issues that drive design decisions. When we know all the ways our data are going to be accessed, we are in a better position to choose the right data structures that optimize the tradeoffs between efficiency and generality.

4. The specification is often expressed in a language quite different from the programming language. The specification is *functional*—it tells *what* the program is supposed to do—as opposed to *imperative*—*how* to do it. It is often easier to specify the desired functionality independent of the details of how it will be implemented. For example, we can quite easily express what it means for a number x to be the gcd of y and z in first-order logic without even knowing how to compute it.

5. Verifying that a program meets its specification is a kind of sanity check. It allows us to give two solutions to the problem—once as a functional specification, and once as an algorithmic implementation—and lets us verify that the two are compatible. Any incompatibilities between the program and the specification are either bugs in the program, bugs in the specification, or both. The cycle of refining the specification, modifying the program to meet the specification, and reverifying until the process converges can lead to software in which we have much more confidence.

Partial and Total Correctness

Typically, a program is designed to implement some functionality. As mentioned above, that functionality can often be expressed formally in the form of an input/output specification. Concretely, such a specification consists of an *input condition* or *precondition* φ and an *output condition* or *postcondition* ψ. These are properties of the input state and the output state, respectively, expressed in some formal language such as the first-order language of the domain of computation. The program is supposed to halt in a state satisfying the output condition whenever the input state satisfies the input condition. We say that a program is *partially correct* with respect to a given input/output specification φ, ψ if, whenever the program is started in a state satisfying the input condition φ, then if and when it ever halts, it does so in a state satisfying the output condition ψ. The definition of partial

correctness does not stipulate that the program halts; this is what we mean by *partial*.

A program is *totally correct* with respect to an input/output specification φ, ψ if

- it is partially correct with respect to that specification; and
- it halts whenever it is started in a state satisfying the input condition φ.

The input/output specification imposes no requirements when the input state does not satisfy the input condition φ—the program might as well loop infinitely or erase memory. This is the "garbage in, garbage out" philosophy. If we really do care what the program does on some of those input states, then we had better rewrite the input condition to include them and say formally what we want to happen in those cases.

For example, in the gcd program of Example 4.1, the output condition ψ might be the condition (i) stating that the output value of x is the gcd of the input values of x and y. We can express this completely formally in the language of first-order number theory (we show how to do this later on). We may try to start off with the input specification $\varphi_0 = \mathbf{1}$ (*true*); that is, no restrictions on the input state at all. Unfortunately, if the initial value of y is 0 and x is negative, the final value of x will be the same as the initial value, thus negative. If we expect all gcds to be positive, this would be wrong. Another problematic situation arises when the initial values of x and y are both 0; in this case the gcd is not defined. Therefore, the program as written is not partially correct with respect to the specification φ_0, ψ.

We can remedy the situation by providing an input specification that rules out these troublesome input values. We can limit the input states to those in which x and y are both nonnegative and not both zero by taking the input specification

$$\varphi_1 \quad = \quad (x \geq 0 \wedge y > 0) \ \vee \ (x > 0 \wedge y \geq 0).$$

The gcd program of Example 4.1 above would be partially correct with respect to the specification φ_1, ψ. It is also totally correct, since the program halts on all inputs satisfying φ_1.

Perhaps we want to allow any input in which not both x and y are zero. In that case, we should use the input specification $\varphi_2 = \neg(x = 0 \wedge y = 0)$. But then the program of Example 4.1 is not partially correct with respect to φ_2, ψ; we must amend the program to produce the correct (positive) gcd on negative inputs.

Hoare Logic

A precursor to Dynamic Logic, and one of the first formal verification systems, was *Hoare Logic*, introduced by Hoare (1969). This is a system for proving partial correctness of deterministic **while** programs related to the invariant assertion method of Floyd (1967). Hoare Logic allows statements of the form

$$\{\varphi\}\,\alpha\,\{\psi\}, \tag{4.4.1}$$

which says that the program α is partially correct with respect to the input/output specification φ, ψ; that is, if α is started in an input state satisfying φ, then if and when it halts, it does so in a state satisfying ψ.

The deductive system for Hoare Logic consists of a small set of rules for deriving partial correctness assertions of the form (4.4.1) for compound programs inductively from similar assertions about their subprograms. There is one rule for each programming construct:

Assignment rule:

$$\{\varphi[x/e]\}\,x := e\,\{\varphi\},$$

where e is free for x in φ (see Section 3.4);

Composition rule:

$$\frac{\{\varphi\}\,\alpha\,\{\sigma\},\quad \{\sigma\}\,\beta\,\{\psi\}}{\{\varphi\}\,\alpha\,;\,\beta\,\{\psi\}}$$

Conditional rule:

$$\frac{\{\varphi\wedge\sigma\}\,\alpha\,\{\psi\},\quad \{\varphi\wedge\neg\sigma\}\,\beta\,\{\psi\}}{\{\varphi\}\ \textbf{if}\ \sigma\ \textbf{then}\ \alpha\ \textbf{else}\ \beta\,\{\psi\}}$$

While rule:

$$\frac{\{\varphi\wedge\sigma\}\,\alpha\,\{\varphi\}}{\{\varphi\}\ \textbf{while}\ \sigma\ \textbf{do}\ \alpha\,\{\varphi\wedge\neg\sigma\}}.$$

In addition, we include a rule

Weakening rule:

$$\frac{\varphi'\to\varphi,\quad \{\varphi\}\,\alpha\,\{\psi\},\quad \psi\to\psi'}{\{\varphi'\}\,\alpha\,\{\psi'\}}$$

that will allow us to incorporate the deductive apparatus of the underlying logic in which the pre- and postconditions are written.

We will see later on in Section 5.7 how these rules are subsumed by Dynamic Logic.

4.5 Exogenous and Endogenous Logics

There are two main approaches to modal logics of programs: the *exogenous* approach, exemplified by Dynamic Logic and its precursor Hoare Logic (Hoare (1969)), and the *endogenous* approach, exemplified by Temporal Logic and its precursor, the invariant assertions method of Floyd (1967). A logic is *exogenous* if its programs are explicit in the language. Syntactically, a Dynamic Logic program is a well-formed expression built inductively from primitive programs using a small set of program operators. Semantically, a program is interpreted as its input/output relation. The relation denoted by a compound program is determined by the relations denoted by its parts. This aspect of *compositionality* allows analysis by structural induction.

The importance of compositionality is discussed in van Emde Boas (1978). In Temporal Logic, the program is fixed and is considered part of the structure over which the logic is interpreted. The current location in the program during execution is stored in a special variable for that purpose, called the *program counter*, and is part of the state along with the values of the program variables. Instead of program operators, there are temporal operators that describe how the program variables, including the program counter, change with time. Thus Temporal Logic sacrifices compositionality for a less restricted formalism. We discuss Temporal Logic further in Section 17.2.

4.6 Bibliographical Notes

Systematic program verification originated with the work of Floyd (1967) and Hoare (1969). Hoare Logic was introduced in Hoare (1969); see Cousot (1990); Apt (1981); Apt and Olderog (1991) for surveys.

The *digital abstraction*, the view of computers as state transformers that operate by performing a sequence of discrete and instantaneous primitive steps, can be attributed to Turing (1936). Finite-state transition systems were defined formally by McCulloch and Pitts (1943). State-transition semantics is based on this idea and is quite prevalent in early work on program semantics and verification; see Hennessy and Plotkin (1979). The relational-algebraic approach taken here, in which programs are interpreted as binary input/output relations, was introduced in the context of DL by Pratt (1976).

The notions of partial and total correctness were present in the early work of Hoare (1969). Regular programs were introduced by Fischer and Ladner (1979) in the context of PDL. The concept of nondeterminism was introduced in the original paper of Turing (1936), although he did not develop the idea. Nondeterminism was further developed by Rabin and Scott (1959) in the context of finite automata.

Exercises

4.1. In this exercise we will illustrate the use of Hoare Logic by proving the correctness of the gcd program of Example 4.1 on p. 145. The program is of the form **while** σ **do** α, where σ is the test $y \neq 0$ and α is the program

$z := x \bmod y \,;\; x := y \,;\; y := z.$

We will use the precondition

$$\neg(x = 0 \wedge y = 0) \wedge x = x_0 \wedge y = y_0, \tag{4.6.1}$$

where x_0 and y_0 are new variables not appearing in the program. The purpose of x_0 and y_0 is to remember the initial values of x and y. The condition $\neg(x = 0 \wedge y = 0)$ ensures that the gcd exists.

The postcondition is

$$x \quad = \quad \gcd(x_0, y_0), \tag{4.6.2}$$

which says that the final value of the program variable x is the gcd of the values of x_0 and y_0. This can be expressed in the language of first order number theory if desired (see Exercise 3.27), although you do not need to do so for this exercise.

We assume that the variables range over $\mathbb{N}$, the natural numbers, thus we do not have to worry about fractional or negative values. The practical significance of this is that we may omit conditions such as $x \geq 0$ in our pre- and postconditions, since these are satisfied automatically by all elements of our domain of computation.

The partial correctness assertion that asserts the correctness of the gcd program is

$$\{\neg(x = 0 \wedge y = 0) \wedge x = x_0 \wedge y = y_0\}\ \textbf{while}\ \sigma\ \textbf{do}\ \alpha\ \{x = \gcd(x_0, y_0)\}.$$

(a) Let φ be the formula

$$\neg(x = 0 \wedge y = 0) \wedge \gcd(x, y) = \gcd(x_0, y_0).$$

This will be the invariant of our loop. Using informal number-theoretic arguments,

prove that the precondition (4.6.1) implies φ and that $\varphi \wedge \neg\sigma$ implies the postcondition (4.6.2). Conclude using the weakening rule of Hoare Logic that it suffices to establish the partial correctness assertion

$\{\varphi\}$ **while** σ **do** α $\{\varphi \wedge \neg\sigma\}$.

(b) By (a) and the **while** rule of Hoare Logic, it suffices to prove the partial correctness assertion

$\{\varphi \wedge \sigma\}\ \alpha\ \{\varphi\}$.

Prove this using a sequence of applications of the composition and weakening rules of Hoare Logic. You may use common number-theoretic facts such as

$$y \neq 0 \quad \rightarrow \quad \gcd(x, y) = \gcd(x \bmod y, y)$$

without proof.

4.2. When the domain of computation is the natural numbers $\mathbb{N}$, we can define a **for** loop construct. The syntax of the construct is **for** y **do** α, where y is a variable and α is a program. The intuitive operation of the **for** loop is as follows: upon entering the loop **for** y **do** α, the current (nonnegative integral) value of variable y is determined, and the program α is executed that many times. Assignment to the variable y within α does not change the number of times the loop is executed, nor does execution of α alone decrement y or change its value in any way except by explicit assignment.

(a) Show how to encode a **for** loop as a **while** loop. You may introduce new variables if necessary.

(b) Argue that **while** programs with **for** loops only but no **while** loops must always halt. (*Hint.* Use induction on the depth of nesting of **for** loops in the program.)

4.3. The **repeat-until** construct **repeat** α **until** φ is like the **while** loop, except that the body of the loop α is executed *before* the test φ (therefore is always executed at least once), and control exits the loop if the test is true. Show that in the presence of the other program operators, **repeat-until** and **while-do** are equivalent.

II PROPOSITIONAL DYNAMIC LOGIC

5 Propositional Dynamic Logic

Propositional Dynamic Logic (PDL) plays the same role in Dynamic Logic that classical propositional logic plays in classical predicate logic. It describes the properties of the interaction between programs and propositions that are independent of the domain of computation. Just as propositional logic is the appropriate place to begin the study of classical predicate logic, so too is PDL the appropriate place to begin our investigation of Dynamic Logic. Since PDL is a subsystem of first-order DL, we can be sure that all properties of PDL that we establish in Part II of the book will also be valid in first-order DL, which we will deal with in Part III.

Since there is no domain of computation in PDL, there can be no notion of assignment to a variable. Instead, primitive programs are interpreted as arbitrary binary relations on an abstract set of states K. Likewise, primitive assertions are just atomic propositions and are interpreted as arbitrary subsets of K. Other than this, no special structure is imposed.

This level of abstraction may at first appear too general to say anything of interest. On the contrary, it is a very natural level of abstraction at which many fundamental relationships between programs and propositions can be observed.

For example, consider the PDL formula

$$[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi. \tag{5.0.1}$$

The left-hand side asserts that the formula $\varphi \wedge \psi$ must hold after the execution of program α, and the right-hand side asserts that φ must hold after execution of α and so must ψ. The formula (5.0.1) asserts that these two statements are equivalent. This implies that to verify a conjunction of two postconditions, it suffices to verify each of them separately. The assertion (5.0.1) holds universally, regardless of the domain of computation and the nature of the particular α, φ, and ψ.

As another example, consider

$$[\alpha\,;\,\beta]\varphi \quad \leftrightarrow \quad [\alpha][\beta]\varphi. \tag{5.0.2}$$

The left-hand side asserts that after execution of the composite program $\alpha\,;\,\beta$, φ must hold. The right-hand side asserts that after execution of the program α, $[\beta]\varphi$ must hold, which in turn says that after execution of β, φ must hold. The formula (5.0.2) asserts the logical equivalence of these two statements. It holds regardless of the nature of α, β, and φ. Like (5.0.1), (5.0.2) can be used to simplify the verification of complicated programs.

As a final example, consider the assertion

$$[\alpha]p \quad \leftrightarrow \quad [\beta]p \tag{5.0.3}$$

where p is a primitive proposition symbol and α and β are programs. If this formula is true under all interpretations, then α and β are *equivalent* in the sense that they behave identically with respect to any property expressible in PDL or any formal system containing PDL as a subsystem. This is because the assertion will hold for any substitution instance of (5.0.3). For example, the two programs

$$\begin{aligned} \alpha &= \textbf{if } \varphi \textbf{ then } \gamma \textbf{ else } \delta \\ \beta &= \textbf{if } \neg\varphi \textbf{ then } \delta \textbf{ else } \gamma \end{aligned}$$

are equivalent in the sense of (5.0.3).

5.1 Syntax

Syntactically, PDL is a blend of three classical ingredients: propositional logic, modal logic, and the algebra of regular expressions. There are several versions of PDL, depending on the choice of program operators allowed. In this chapter we will introduce the basic version, called *regular* PDL. Variations of this basic version will be considered in later chapters.

The language of regular PDL has expressions of two sorts: *propositions* or *formulas* $\varphi, \psi, \ldots$ and *programs* $\alpha, \beta, \gamma, \ldots$. There are countably many *atomic symbols* of each sort. Atomic programs are denoted $a, b, c, \ldots$ and the set of all atomic programs is denoted Π_0. Atomic propositions are denoted $p, q, r, \ldots$ and the set of all atomic propositions is denoted Φ_0. The set of all programs is denoted Π and the set of all propositions is denoted Φ. Programs and propositions are built inductively from the atomic ones using the following operators:

Propositional operators:

$\rightarrow$	implication
$\mathbf{0}$	falsity

Program operators:

;	composition
$\cup$	choice
*	iteration

Mixed operators:

`[ ]`	necessity
?	test

The definition of programs and propositions is by mutual induction. All atomic programs are programs and all atomic propositions are propositions. If φ, ψ are propositions and α, β are programs, then

$\varphi \to \psi$	propositional implication
$\mathbf{0}$	propositional falsity
$[\alpha]\varphi$	program necessity

are propositions and

$\alpha\,;\,\beta$	sequential composition
$\alpha \cup \beta$	nondeterministic choice
α^*	iteration
$\varphi?$	test

are programs. In more formal terms, we define the set Π of all programs and the set Φ of all propositions to be the smallest sets such that

- $\Phi_0 \subseteq \Phi$
- $\Pi_0 \subseteq \Pi$
- if $\varphi, \psi \in \Phi$, then $\varphi \to \psi \in \Phi$ and $\mathbf{0} \in \Phi$
- if $\alpha, \beta \in \Pi$, then $\alpha;\beta$, $\alpha \cup \beta$, and $\alpha^* \in \Pi$
- if $\alpha \in \Pi$ and $\varphi \in \Phi$, then $[\alpha]\varphi \in \Phi$
- if $\varphi \in \Phi$ then $\varphi? \in \Pi$.

Note that the inductive definitions of programs Π and propositions Φ are intertwined and cannot be separated. The definition of propositions depends on the definition of programs because of the construct $[\alpha]\varphi$, and the definition of programs depends on the definition of propositions because of the construct $\varphi?$. Note also that we have allowed all formulas as tests. This is the *rich test* version of PDL.

Compound programs and propositions have the following intuitive meanings:

$[\alpha]\varphi$ "It is necessary that after executing α, φ is true."

$\alpha;\beta$ "Execute α, then execute β."

$\alpha \cup \beta$ "Choose either α or β nondeterministically and execute it."

α^* "Execute α a nondeterministically chosen finite number of times (zero or more)."

$\varphi?$ "Test φ; proceed if true, fail if false."

We avoid parentheses by assigning precedence to the operators: unary operators, including $[\alpha]$, bind tighter than binary ones, and ; binds tighter than $\cup$. Thus the expression

$$[\alpha;\beta^* \cup \gamma^*]\varphi \vee \psi$$

should be read

$$([(\alpha;(\beta^*)) \cup (\gamma^*)]\varphi) \vee \psi.$$

Of course, parentheses can always be used to enforce a particular parse of an expression or to enhance readability. Also, under the semantics to be given in the next section, the operators ; and $\cup$ will turn out to be associative, so we may write $\alpha\,;\,\beta\,;\,\gamma$ and $\alpha \cup \beta \cup \gamma$ without ambiguity. We often omit the symbol ; and write the composition $\alpha\,;\,\beta$ as $\alpha\beta$.

The primitive operators may at first seem rather unconventional. They are chosen for their mathematical simplicity. A number of more conventional constructs can be defined from them. The propositional operators $\wedge$, $\vee$, $\neg$, $\leftrightarrow$, and $\mathbf{1}$ are defined from $\rightarrow$ and $\mathbf{0}$ as in propositional logic (see Section 3.2).

The possibility operator $\langle\ \rangle$ is the modal dual of the necessity operator $[\ ]$ as described in Section 3.7. It is defined by

$$\langle\alpha\rangle\varphi \stackrel{\text{def}}{=} \neg[\alpha]\neg\varphi.$$

The propositions $[\alpha]\varphi$ and $\langle\alpha\rangle\varphi$ are read "box α φ" and "diamond α φ," respectively. The latter has the intuitive meaning, "There is a computation of α that terminates in a state satisfying φ."

One important difference between $\langle\ \rangle$ and $[\ ]$ is that $\langle\alpha\rangle\varphi$ implies that α terminates, whereas $[\alpha]\varphi$ does not. Indeed, the formula $[\alpha]\mathbf{0}$ asserts that no computation of α terminates, and the formula $[\alpha]\mathbf{1}$ is always true, regardless of α.

In addition, we define

$$\begin{aligned}
\textbf{skip} &\stackrel{\text{def}}{=} \mathbf{1}?\\
\textbf{fail} &\stackrel{\text{def}}{=} \mathbf{0}?\\
\textbf{if } \varphi_1 \to \alpha_1 \mid \cdots \mid \varphi_n \to \alpha_n \textbf{ fi} &\stackrel{\text{def}}{=} \varphi_1?;\alpha_1 \cup \cdots \cup \varphi_n?;\alpha_n\\
\textbf{do } \varphi_1 \to \alpha_1 \mid \cdots \mid \varphi_n \to \alpha_n \textbf{ od} &\stackrel{\text{def}}{=} (\varphi_1?;\alpha_1 \cup \cdots \cup \varphi_n?;\alpha_n)^*;(\neg\varphi_1 \wedge \cdots \wedge \neg\varphi_n)?\\
\textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta &\stackrel{\text{def}}{=} \textbf{if } \varphi \to \alpha \mid \neg\varphi \to \beta \textbf{ fi}\\
&= \varphi?;\alpha \cup \neg\varphi?;\beta\\
\textbf{while } \varphi \textbf{ do } \alpha &\stackrel{\text{def}}{=} \textbf{do } \varphi \to \alpha \textbf{ od}\\
&= (\varphi?;\alpha)^*;\neg\varphi?\\
\textbf{repeat } \alpha \textbf{ until } \varphi &\stackrel{\text{def}}{=} \alpha;\textbf{while } \neg\varphi \textbf{ do } \alpha\\
&= \alpha;(\neg\varphi?;\alpha)^*;\varphi?\\
\{\varphi\}\,\alpha\,\{\psi\} &\stackrel{\text{def}}{=} \varphi \to [\alpha]\psi.
\end{aligned}$$

The programs **skip** and **fail** are the program that does nothing (no-op) and the failing program, respectively. The ternary **if-then-else** operator and the binary **while-do** operator are the usual *conditional* and *while loop* constructs found in conventional programming languages. The constructs **if-|-fi** and **do-|-od** are the *alternative guarded command* and *iterative guarded command* constructs, respectively. The construct $\{\varphi\}\,\alpha\,\{\psi\}$ is the Hoare partial correctness assertion described in Section 4.4. We will argue later that the formal definitions of these operators given above correctly model their intuitive behavior.

5.2 Semantics

The semantics of PDL comes from the semantics for modal logic (see Section 3.7). The structures over which programs and propositions of PDL are interpreted are called *Kripke frames* in honor of Saul Kripke, the inventor of the formal semantics of modal logic. A *Kripke frame* is a pair

$$\mathfrak{K} \;=\; (K,\, \mathfrak{m}_{\mathfrak{K}}),$$

where K is a set of elements $u, v, w, \ldots$ called *states* and $\mathfrak{m}_{\mathfrak{K}}$ is a *meaning function* assigning a subset of K to each atomic proposition and a binary relation on K to

each atomic program. That is,

$$\begin{array}{lll} \mathfrak{m}_{\mathfrak{K}}(p) & \subseteq & K, \qquad p \in \Phi_0 \\ \mathfrak{m}_{\mathfrak{K}}(a) & \subseteq & K \times K, \quad a \in \Pi_0. \end{array}$$

We will extend the definition of the function $\mathfrak{m}_{\mathfrak{K}}$ by induction below to give a meaning to all elements of Π and Φ such that

$$\begin{array}{lll} \mathfrak{m}_{\mathfrak{K}}(\varphi) & \subseteq & K, \qquad \varphi \in \Phi \\ \mathfrak{m}_{\mathfrak{K}}(\alpha) & \subseteq & K \times K, \quad \alpha \in \Pi. \end{array}$$

Intuitively, we can think of the set $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ as the set of states *satisfying* the proposition φ in the model $\mathfrak{K}$, and we can think of the binary relation $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ as the set of input/output pairs of states of the program α.

Formally, the meanings $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ of $\varphi \in \Phi$ and $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ of $\alpha \in \Pi$ are defined by mutual induction on the structure of φ and α. The basis of the induction, which specifies the meanings of the atomic symbols $p \in \Phi_0$ and $a \in \Pi_0$, is already given in the specification of $\mathfrak{K}$. The meanings of compound propositions and programs are defined as follows.

$$\begin{array}{rcll} \mathfrak{m}_{\mathfrak{K}}(\varphi \to \psi) & \stackrel{\text{def}}{=} & (K - \mathfrak{m}_{\mathfrak{K}}(\varphi)) \cup \mathfrak{m}_{\mathfrak{K}}(\psi) & \\ \mathfrak{m}_{\mathfrak{K}}(\mathbf{0}) & \stackrel{\text{def}}{=} & \varnothing & \\ \mathfrak{m}_{\mathfrak{K}}(\,[\alpha]\,\varphi) & \stackrel{\text{def}}{=} & K - (\mathfrak{m}_{\mathfrak{K}}(\alpha) \circ (K - \mathfrak{m}_{\mathfrak{K}}(\varphi))) & \\ & = & \{u \mid \forall v \in K \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ then } v \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\} & \\ \mathfrak{m}_{\mathfrak{K}}(\alpha;\beta) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\beta) & (5.2.1) \\ & = & \{(u,v) \mid \exists w \in K \ (u,w) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta)\} & \\ \mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{K}}(\beta) & \\ \mathfrak{m}_{\mathfrak{K}}(\alpha^*) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha)^* \ = \ \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha)^n & (5.2.2) \\ \mathfrak{m}_{\mathfrak{K}}(\varphi?) & \stackrel{\text{def}}{=} & \{(u,u) \mid u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\}. & \end{array}$$

The operator $\circ$ in (5.2.1) is relational composition (Section 1.3). In (5.2.2), the first occurrence of * is the iteration symbol of PDL, and the second is the reflexive transitive closure operator on binary relations (Section 1.3). Thus (5.2.2) says that the program α^* is interpreted as the reflexive transitive closure of $\mathfrak{m}_{\mathfrak{K}}(\alpha)$.

We write $\mathfrak{K}, u \vDash \varphi$ and $u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$ interchangeably, and say that u *satisfies* φ in $\mathfrak{K}$, or that φ is *true* at state u in $\mathfrak{K}$. We may omit the $\mathfrak{K}$ and write $u \vDash \varphi$ when $\mathfrak{K}$ is understood. The notation $u \nvDash \varphi$ means that u does not satisfy φ, or in other words

that $u \notin \mathfrak{m}_{\mathfrak{K}}(\varphi)$. In this notation, we can restate the definition above equivalently as follows:

$$\begin{aligned}
u \vDash \varphi \to \psi &\iff^{\text{def}} u \vDash \varphi \text{ implies } u \vDash \psi \\
u &\nvDash \mathbf{0} \\
u \vDash [\alpha]\varphi &\iff^{\text{def}} \forall v \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ then } v \vDash \varphi \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha\beta) &\iff^{\text{def}} \exists w\ (u,w) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta) \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta) &\iff^{\text{def}} (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ or } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta) \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha^*) &\iff^{\text{def}} \exists n \geq 0\ \exists u_0, \ldots, u_n\ u = u_0,\ v = u_n, \\
&\qquad \text{and } (u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha),\ 0 \leq i \leq n-1 \\
(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\varphi?) &\iff^{\text{def}} u = v \text{ and } u \vDash \varphi.
\end{aligned}$$

The defined operators inherit their meanings from these definitions:

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{K}}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\varphi) \cup \mathfrak{m}_{\mathfrak{K}}(\psi) \\
\mathfrak{m}_{\mathfrak{K}}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\varphi) \cap \mathfrak{m}_{\mathfrak{K}}(\psi) \\
\mathfrak{m}_{\mathfrak{K}}(\neg\varphi) &\stackrel{\text{def}}{=} K - \mathfrak{m}_{\mathfrak{K}}(\varphi) \\
\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi) &\stackrel{\text{def}}{=} \{u \mid \exists v \in K\ (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } v \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\} \\
&= \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\varphi) \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{1}) &\stackrel{\text{def}}{=} K \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{skip}) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\mathbf{1}?) = \iota, \text{ the identity relation} \\
\mathfrak{m}_{\mathfrak{K}}(\mathbf{fail}) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\mathbf{0}?) = \varnothing.
\end{aligned}$$

In addition, the **if-then-else**, **while-do**, and guarded commands inherit their semantics from the above definitions, and the input/output relations given by the formal semantics capture their intuitive operational meanings. For example, the relation associated with the program **while** φ **do** α is the set of pairs (u,v) for which there exist states $u_0, u_1, \ldots, u_n$, $n \geq 0$, such that $u = u_0$, $v = u_n$, $u_i \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$ and $(u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$ for $0 \leq i < n$, and $u_n \notin \mathfrak{m}_{\mathfrak{K}}(\varphi)$. A thorough analysis will require more careful attention, so we defer further discussion until later.

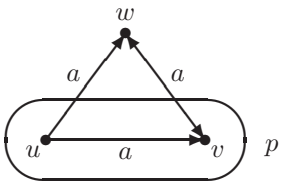
This version of PDL is usually called *regular* PDL and the elements of Π are called *regular programs* because of the primitive operators $\cup$, ;, and *, which are familiar from regular expressions. Programs can be viewed as regular expressions over the atomic programs and tests. In fact, it can be shown that if p is an atomic proposition symbol, then any two test-free programs α, β are equivalent as regular

expressions—that is, they represent the same regular set—if and only if the formula $\langle\alpha\rangle p \leftrightarrow \langle\beta\rangle p$ is valid (Exercise 5.13).

EXAMPLE 5.1: Let p be an atomic proposition, let a be an atomic program, and let $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame with

$$\begin{aligned} K &= \{u, v, w\} \\ \mathfrak{m}_{\mathfrak{K}}(p) &= \{u, v\} \\ \mathfrak{m}_{\mathfrak{K}}(a) &= \{(u, v), (u, w), (v, w), (w, v)\}. \end{aligned}$$

The following diagram illustrates $\mathfrak{K}$.



In this structure, $u \vDash \langle a\rangle\neg p \wedge \langle a\rangle p$, but $v \vDash [a]\neg p$ and $w \vDash [a]p$. Moreover, every state of $\mathfrak{K}$ satisfies the formula

$$\langle a^*\rangle[(aa)^*]p \wedge \langle a^*\rangle[(aa)^*]\neg p.$$

5.3 Computation Sequences

Let α be a program. Recall from Section 4.3 that a *finite computation sequence* of α is a finite-length string of atomic programs and tests representing a possible sequence of atomic steps that can occur in a halting execution of α. These strings are called *seqs* and are denoted $\sigma, \tau, \ldots$. The set of all such sequences is denoted $CS(\alpha)$. We use the word "possible" here loosely—$CS(\alpha)$ is determined by the syntax of α alone, and may contain strings that are never executed in any interpretation.

Formally, the set $CS(\alpha)$ is defined by induction on the structure of α:

$$\begin{aligned} CS(a) &\stackrel{\text{def}}{=} \{a\},\ a \text{ an atomic program} \\ CS(\varphi?) &\stackrel{\text{def}}{=} \{\varphi?\} \\ CS(\alpha;\beta) &\stackrel{\text{def}}{=} \{\gamma\delta \mid \gamma \in CS(\alpha),\ \delta \in CS(\beta)\} \\ CS(\alpha \cup \beta) &\stackrel{\text{def}}{=} CS(\alpha) \cup CS(\beta) \\ CS(\alpha^*) &\stackrel{\text{def}}{=} \bigcup_{n\geq 0} CS(\alpha^n) \end{aligned}$$

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

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

has as computation sequences all strings of the form

$$p?\, a\, p?\, a\, \cdots\, p?\, a\, \textbf{skip}\, \neg p?.$$

Note that each finite computation sequence β of a program α is itself a program, and $CS(\beta) = \{\beta\}$. Moreover, the following proposition is not difficult to prove by induction on the structure of α:

PROPOSITION 5.2:

$$\mathfrak{m}_{\mathfrak{K}}(\alpha) \quad = \quad \bigcup_{\sigma \in CS(\alpha)} \mathfrak{m}_{\mathfrak{K}}(\sigma).$$

Proof Exercise 5.1. ■

5.4 Satisfiability and Validity

The definitions of satisfiability and validity of propositions are identical to those of modal logic (see Section 3.7). Let $\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame and let φ be a proposition. We have defined in Section 5.2 what it means for $\mathfrak{K}, u \vDash \varphi$. If $\mathfrak{K}, u \vDash \varphi$ for some $u \in K$, we say that φ is *satisfiable* in $\mathfrak{K}$. If φ is satisfiable in some $\mathfrak{K}$, we say that φ is *satisfiable*.

If $\mathfrak{K}, u \vDash \varphi$ for all $u \in K$, we write $\mathfrak{K} \vDash \varphi$ and say that φ is *valid* in $\mathfrak{K}$. If $\mathfrak{K} \vDash \varphi$ for all Kripke frames $\mathfrak{K}$, we write $\vDash \varphi$ and say that φ is *valid*.

If Σ is a set of propositions, we write $\mathfrak{K} \vDash \Sigma$ if $\mathfrak{K} \vDash \varphi$ for all $\varphi \in \Sigma$. A proposition

ψ is said to be a *logical consequence* of Σ if $\mathfrak{K} \vDash \psi$ whenever $\mathfrak{K} \vDash \Sigma$, in which case we write $\Sigma \vDash \psi$. (Note that this is *not* the same as saying that $\mathfrak{K}, u \vDash \psi$ whenever $\mathfrak{K}, u \vDash \Sigma$.) We say that an inference rule

$$\frac{\varphi_1, \ldots, \varphi_n}{\varphi}$$

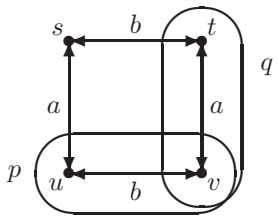
is *sound* if φ is a logical consequence of $\{\varphi_1, \ldots, \varphi_n\}$.

Satisfiability and validity are dual in the same sense that $\exists$ and $\forall$ are dual and `< >` and `[ ]` are dual: a proposition is valid (in $\mathfrak{K}$) if and only if its negation is not satisfiable (in $\mathfrak{K}$).

EXAMPLE 5.3: Let p, q be atomic propositions, let a, b be atomic programs, and let $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame with

$$\begin{aligned} K &= \{s, t, u, v\} \\ \mathfrak{m}_{\mathfrak{K}}(p) &= \{u, v\} \\ \mathfrak{m}_{\mathfrak{K}}(q) &= \{t, v\} \\ \mathfrak{m}_{\mathfrak{K}}(a) &= \{(t, v), (v, t), (s, u), (u, s)\} \\ \mathfrak{m}_{\mathfrak{K}}(b) &= \{(u, v), (v, u), (s, t), (t, s)\}. \end{aligned}$$

The following figure illustrates $\mathfrak{K}$.



The following formulas are valid in $\mathfrak{K}$.

$$\begin{aligned} p &\leftrightarrow \texttt{[}(ab^*a)^*\texttt{]}p \\ q &\leftrightarrow \texttt{[}(ba^*b)^*\texttt{]}q. \end{aligned}$$

Also, let α be the program

$$\alpha = (aa \cup bb \cup (ab \cup ba)(aa \cup bb)^*(ab \cup ba))^*. \tag{5.4.1}$$

Thinking of α as a regular expression, α generates all words over the alphabet $\{a, b\}$ with an even number of occurrences of each of a and b. It can be shown that for

any proposition φ, the proposition $\varphi \leftrightarrow [\alpha]\varphi$ is valid in $\mathfrak{K}$ (Exercise 5.5).

EXAMPLE 5.4: The formula

$$p \wedge [a^*]((p \to [a]\neg p) \wedge (\neg p \to [a]p)) \quad \leftrightarrow \quad [(aa)^*]p \wedge [a(aa)^*]\neg p \tag{5.4.2}$$

is valid. Both sides assert in different ways that p is alternately true and false along paths of execution of the atomic program a. See Exercise 5.6.

5.5 A Deductive System

The following list of axioms and rules constitutes a sound and complete Hilbert-style deductive system for PDL.

AXIOM SYSTEM 5.5:

(i) Axioms for propositional logic
(ii) $[\alpha](\varphi \to \psi) \quad \to \quad ([\alpha]\varphi \to [\alpha]\psi)$
(iii) $[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi$
(iv) $[\alpha \cup \beta]\varphi \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\beta]\varphi$
(v) $[\alpha\,;\,\beta]\varphi \quad \leftrightarrow \quad [\alpha][\beta]\varphi$
(vi) $[\psi?]\varphi \quad \leftrightarrow \quad (\psi \to \varphi)$
(vii) $\varphi \wedge [\alpha][\alpha^*]\varphi \quad \leftrightarrow \quad [\alpha^*]\varphi$
(viii) $\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi$ (induction axiom)

(MP) $\dfrac{\varphi, \quad \varphi \to \psi}{\psi}$

(GEN) $\dfrac{\varphi}{[\alpha]\varphi}$

The axioms (ii) and (iii) and the two rules of inference are not particular to PDL, but come from modal logic (see Section 3.7). The rules (MP) and (GEN) are called *modus ponens* and *(modal) generalization*, respectively.

Axiom (viii) is called the PDL *induction axiom*. Intuitively, (viii) says: "Suppose φ is true in the current state, and suppose that after any number of iterations of α, if φ is still true, then it will be true after one more iteration of α. Then φ will be true after any number of iterations of α." In other words, if φ is true initially, and if the truth of φ is preserved by the program α, then φ will be true after any number of iterations of α.

Notice the similarity of the formula (viii) to the usual induction axiom of Peano arithmetic:

$$\varphi(0) \wedge \forall n\ (\varphi(n) \to \varphi(n+1)) \quad \to \quad \forall n\ \varphi(n).$$

Here $\varphi(0)$ is the basis of the induction and $\forall n\ (\varphi(n) \to \varphi(n+1))$ is the induction step, from which the conclusion $\forall n\ \varphi(n)$ can be drawn. In the PDL axiom (viii), the basis is φ and the induction step is $[\alpha^*](\varphi \to [\alpha]\varphi)$, from which the conclusion $[\alpha^*]\varphi$ can be drawn.

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

The soundness of these axioms and rules over Kripke frames can be established by elementary arguments in relational algebra using the semantics of Section 5.2. We will do this in Section 5.6. We will prove the completeness of this system in Chapter 7.

5.6 Basic Properties

We establish some basic facts that follow from the definitions of Sections 5.1–5.5. Most of these results are in the form of valid formulas and rules of inference of PDL. In the course of proving these results, we will establish the soundness of the deductive system for PDL given in Section 5.5.

Properties Inherited from Modal Logic

We start with some properties that are not particular to PDL, but are valid in all modal systems. They are valid in PDL by virtue of the fact that PDL includes propositional modal logic. Theorems 5.6 and 5.7 were essentially proved in Section 3.7 (Theorems 3.70 and 3.71, respectively); these in turn were proved using the basic properties of relational composition given in Section 1.3. We restate them here in the framework of PDL.

THEOREM 5.6: The following are valid formulas of PDL:

(i) $\langle\alpha\rangle(\varphi \vee \psi) \quad \leftrightarrow \quad \langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi$

(ii) $[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi$

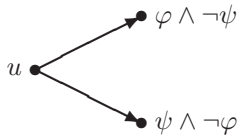
(iii) $\langle\alpha\rangle\varphi \wedge [\alpha]\psi \quad \to \quad \langle\alpha\rangle(\varphi \wedge \psi)$

(iv) $[\alpha](\varphi \to \psi) \quad \to \quad ([\alpha]\varphi \to [\alpha]\psi)$

(v) $\langle\alpha\rangle(\varphi \wedge \psi) \quad \rightarrow \quad \langle\alpha\rangle\varphi \wedge \langle\alpha\rangle\psi$
(vi) $[\alpha]\varphi \vee [\alpha]\psi \quad \rightarrow \quad [\alpha](\varphi \vee \psi)$
(vii) $\langle\alpha\rangle\mathbf{0} \quad \leftrightarrow \quad \mathbf{0}$
(viii) $[\alpha]\varphi \quad \leftrightarrow \quad \neg\langle\alpha\rangle\neg\varphi$.

Proof See Theorem 3.70. ■

The converses of Theorem 5.6(iii)–(vi) are not valid. For example, (iii) is violated in state u of the following Kripke frame:



One can construct similar counterexamples for the others (Exercise 5.10).

THEOREM 5.7: The following are sound rules of inference of PDL:

(i) Modal generalization (GEN):

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

(ii) Monotonicity of $\langle\alpha\rangle$:

$$\frac{\varphi \rightarrow \psi}{\langle\alpha\rangle\varphi \rightarrow \langle\alpha\rangle\psi}$$

(iii) Monotonicity of $[\alpha]$:

$$\frac{\varphi \rightarrow \psi}{[\alpha]\varphi \rightarrow [\alpha]\psi}$$

Proof See Theorem 3.71. ■

The properties expressed in Theorem 5.7(ii) and (iii) are quite useful. They say that the constructs $\langle\ \rangle$ and $[\ ]$ are *monotone* in their second argument with respect to the ordering of logical implication. Corollary 5.9 below asserts that these constructs are also monotone and antitone, respectively, in their first argument.

Properties of $\cup$, ;, and ?

In this section we develop the basic properties of the choice operator $\cup$, the composition operator ;, and the test operator ?.

THEOREM 5.8: The following are valid formulas of PDL:

(i) $\langle\alpha \cup \beta\rangle\varphi \quad \leftrightarrow \quad \langle\alpha\rangle\varphi \vee \langle\beta\rangle\varphi$
(ii) $[\alpha \cup \beta]\varphi \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\beta]\varphi$.

Proof For (i), we need to show that for any Kripke frame $\mathfrak{K}$,

$\mathfrak{m}_{\mathfrak{K}}(\langle\alpha \cup \beta\rangle\varphi) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi \vee \langle\beta\rangle\varphi)$.

By the semantics of PDL (Section 5.2), the left-hand side is equivalent to

$(\mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{K}}(\beta)) \circ \mathfrak{m}_{\mathfrak{K}}(\varphi)$

and the right-hand side is equivalent to

$(\mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\varphi)) \cup (\mathfrak{m}_{\mathfrak{K}}(\beta) \circ \mathfrak{m}_{\mathfrak{K}}(\varphi))$.

The equivalence of these two expressions follows from the fact that relational composition $\circ$ distributes over union $\cup$ (Lemma 1.1).

Statement (ii) follows from (i) by duality (Exercise 3.4). ■

Intuitively, Theorem 5.8(i) says that the program $\alpha \cup \beta$ can halt in a state satisfying φ iff either α or β can. Theorem 5.8(ii) says that any halting state of the program $\alpha \cup \beta$ must satisfy φ iff this is true for both α and β.

It follows that the box and diamond operators are monotone and antitone, respectively, in their first argument α:

COROLLARY 5.9: If $\mathfrak{m}_{\mathfrak{K}}(\alpha) \subseteq \mathfrak{m}_{\mathfrak{K}}(\beta)$, then for all φ,

(i) $\mathfrak{K} \models \langle\alpha\rangle\varphi \quad \rightarrow \quad \langle\beta\rangle\varphi$
(ii) $\mathfrak{K} \models [\beta]\varphi \quad \rightarrow \quad [\alpha]\varphi$.

Proof Equivalently, if $\mathfrak{m}_{\mathfrak{K}}(\alpha) \subseteq \mathfrak{m}_{\mathfrak{K}}(\beta)$, then for all φ,

(i) $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}(\langle\beta\rangle\varphi)$
(ii) $\mathfrak{m}_{\mathfrak{K}}([\beta]\varphi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}([\alpha]\varphi)$.

These statements follow from Theorem 5.8 by virtue of the fact that $\mathfrak{m}_{\mathfrak{K}}(\alpha) \subseteq \mathfrak{m}_{\mathfrak{K}}(\beta)$ iff $\mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{M}}(\beta) = \mathfrak{m}_{\mathfrak{K}}(\beta)$. We leave the details as an exercise (Exercise 5.11). ■

THEOREM 5.10: The following are valid formulas of PDL:

(i) $\langle\alpha\,;\,\beta\rangle\varphi \quad\leftrightarrow\quad \langle\alpha\rangle\langle\beta\rangle\varphi$
(ii) $[\alpha\,;\,\beta]\varphi \quad\leftrightarrow\quad [\alpha][\beta]\varphi.$

Proof We need to show that in all models $\mathfrak{K}$,

(i) $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\,;\,\beta\rangle\varphi) \;=\; \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\langle\beta\rangle\varphi)$
(ii) $\mathfrak{m}_{\mathfrak{K}}([\alpha\,;\,\beta]\varphi) \;=\; \mathfrak{m}_{\mathfrak{K}}([\alpha][\beta]\varphi).$

According to the semantics of PDL, statement (i) says

$$(\mathfrak{m}_{\mathfrak{K}}(\alpha)\circ\mathfrak{m}_{\mathfrak{K}}(\beta))\circ\mathfrak{m}_{\mathfrak{K}}(\varphi) \;=\; \mathfrak{m}_{\mathfrak{K}}(\alpha)\circ(\mathfrak{m}_{\mathfrak{K}}(\beta)\circ\mathfrak{m}_{\mathfrak{K}}(\varphi)).$$

This follows from the associativity of relational composition (Exercise 1.1). Statement (ii) follows from (i) by duality (Exercise 3.4). ■

THEOREM 5.11: The following are valid formulas of PDL:

(i) $\langle\varphi?\rangle\psi \quad\leftrightarrow\quad (\varphi\wedge\psi)$
(ii) $[\varphi?]\psi \quad\leftrightarrow\quad (\varphi\rightarrow\psi).$

Proof We need to show that in all models $\mathfrak{K}$,

(i) $\mathfrak{m}_{\mathfrak{K}}(\langle\varphi?\rangle\psi) \;=\; \mathfrak{m}_{\mathfrak{K}}(\varphi\wedge\psi)$
(ii) $\mathfrak{m}_{\mathfrak{K}}([\varphi?]\psi) \;=\; \mathfrak{m}_{\mathfrak{K}}(\varphi\rightarrow\psi).$

To show (i),

$$\begin{aligned}\mathfrak{m}_{\mathfrak{K}}(\langle\varphi?\rangle\psi) &= \{(u,u)\mid u\in\mathfrak{m}_{\mathfrak{K}}(\varphi)\}\circ\mathfrak{m}_{\mathfrak{K}}(\psi)\\ &= \{u\mid u\in\mathfrak{m}_{\mathfrak{K}}(\varphi)\}\cap\mathfrak{m}_{\mathfrak{K}}(\psi)\\ &= \mathfrak{m}_{\mathfrak{K}}(\varphi)\cap\mathfrak{m}_{\mathfrak{K}}(\psi)\\ &= \mathfrak{m}_{\mathfrak{K}}(\varphi\wedge\psi).\end{aligned}$$

Then (ii) follows from (i) by duality (Exercise 3.4). ■

The Converse Operator $^-$

The following properties deal with the converse operator $^-$ with semantics

$$\mathfrak{m}_{\mathfrak{K}}(\alpha^-) \;=\; \mathfrak{m}_{\mathfrak{K}}(\alpha)^- \;=\; \{(v,u)\mid(u,v)\in\mathfrak{m}_{\mathfrak{K}}(\alpha)\}.$$

Intuitively, the converse operator allows us to "run a program backwards;" semantically, the input/output relation of the program α^- is the output/input

relation of α. Although this is not always possible to realize in practice, it is nevertheless a useful expressive tool. For example, it gives us a convenient way to talk about *backtracking*, or rolling back a computation to a previous state.

THEOREM 5.12: For any programs α and β,

(i) $\mathfrak{m}_{\mathfrak{K}}((\alpha \cup \beta)^-) = \mathfrak{m}_{\mathfrak{K}}(\alpha^- \cup \beta^-)$
(ii) $\mathfrak{m}_{\mathfrak{K}}((\alpha\,;\,\beta)^-) = \mathfrak{m}_{\mathfrak{K}}(\beta^-\,;\,\alpha^-)$
(iii) $\mathfrak{m}_{\mathfrak{K}}(\varphi?^-) = \mathfrak{m}_{\mathfrak{K}}(\varphi?)$
(iv) $\mathfrak{m}_{\mathfrak{K}}(\alpha^{*-}) = \mathfrak{m}_{\mathfrak{K}}(\alpha^{-*})$
(v) $\mathfrak{m}_{\mathfrak{K}}(\alpha^{--}) = \mathfrak{m}_{\mathfrak{K}}(\alpha)$.

Proof All of these follow directly from the properties of binary relations (Section 1.3). For example, (i) follows from the fact that the converse operation $^-$ on binary relations commutes with set union $\cup$ (Exercise 1.5):

$$\begin{aligned}
\mathfrak{m}_{\mathfrak{K}}((\alpha \cup \beta)^-) &= \mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta)^- \\
&= (\mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{K}}(\beta))^- \\
&= \mathfrak{m}_{\mathfrak{K}}(\alpha)^- \cup \mathfrak{m}_{\mathfrak{K}}(\beta)^- \\
&= \mathfrak{m}_{\mathfrak{K}}(\alpha^-) \cup \mathfrak{m}_{\mathfrak{K}}(\beta^-) \\
&= \mathfrak{m}_{\mathfrak{K}}(\alpha^- \cup \beta^-).
\end{aligned}$$

Similarly, (ii) uses Exercise 1.6, (iii) follows from the fact that $\mathfrak{m}_{\mathfrak{K}}(\varphi?)$ is a subset of the identity relation ι and is therefore symmetric, (iv) uses Exercise 1.5, and (v) uses Exercise 1.7. ■

Theorem 5.12 can be used to transform any program containing occurrences of the operator $^-$ into an equivalent program in which all occurrences of $^-$ are applied to atomic programs only. The equivalent program is obtained by replacing any subprogram which looks like the left-hand side of one of Theorem 5.12(i)–(v) with the corresponding right-hand side. These rules are applied, moving occurrences of $^-$ inward, until they cannot be applied any more; that is, until all $^-$ are applied to primitive programs only. The resulting program is equivalent to the original.

Theorem 5.12 discusses the interaction of $^-$ with the other program operators. The interaction of $^-$ with the modal operations $\langle\alpha\rangle$ and $[\alpha]$ is described in the following theorem.

THEOREM 5.13: The following are valid formulas of PDL:

(i) $\varphi \rightarrow [\alpha]\langle\alpha^-\rangle\varphi$
(ii) $\varphi \rightarrow [\alpha^-]\langle\alpha\rangle\varphi$
(iii) $\langle\alpha\rangle[\alpha^-]\varphi \rightarrow \varphi$
(iv) $\langle\alpha^-\rangle[\alpha]\varphi \rightarrow \varphi$.

Proof We need to show that in any model $\mathfrak{K}$,

(i) $\mathfrak{m}_{\mathfrak{K}}(\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}([\alpha]\langle\alpha^-\rangle\varphi)$
(ii) $\mathfrak{m}_{\mathfrak{K}}(\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}([\alpha^-]\langle\alpha\rangle\varphi)$
(iii) $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle[\alpha^-]\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\varphi)$
(iv) $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^-\rangle[\alpha]\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\varphi)$.

To show (i), suppose $u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$. For any state v such that $(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$, $v \in \mathfrak{m}_{\mathfrak{K}}(\langle\alpha^-\rangle\varphi)$, thus $u \in \mathfrak{m}_{\mathfrak{M}}([\alpha]\langle\alpha^-\rangle\varphi)$. Statement (ii) follows immediately from (i) using Exercise 1.7, and (iii) and (iv) are dual to (i) and (ii). ■

Theorem 5.13 has a rather powerful consequence: in the presence of the converse operator $^-$, the operator $\langle\alpha\rangle$ is *continuous* on any Kripke frame $\mathfrak{K}$ (see Section 1.7) with respect to the partial order of implication. In PDL without $^-$, a Kripke frame can be constructed such that $\langle\alpha\rangle$ is not continuous (Exercise 5.12).

Let $\mathfrak{K}$ be any Kripke frame for PDL. Let $\mathfrak{m}_{\mathfrak{K}}(\Phi)$ be the set of interpretations of PDL propositions:

$$\mathfrak{m}_{\mathfrak{K}}(\Phi) \stackrel{\text{def}}{=} \{\mathfrak{m}_{\mathfrak{K}}(\varphi) \mid \varphi \in \Phi\}.$$

The set $\mathfrak{m}_{\mathfrak{K}}(\Phi)$ is partially ordered by inclusion $\subseteq$. Under this order, the supremum of any finite set $\{\mathfrak{m}_{\mathfrak{K}}(\varphi_1), \ldots, \mathfrak{m}_{\mathfrak{K}}(\varphi_n)\}$ always exists and is in $\mathfrak{m}_{\mathfrak{K}}(\Phi)$; it is

$$\mathfrak{m}_{\mathfrak{K}}(\varphi_1) \cup \cdots \cup \mathfrak{m}_{\mathfrak{K}}(\varphi_n) = \mathfrak{m}_{\mathfrak{K}}(\varphi_1 \vee \cdots \vee \varphi_n).$$

Moreover, $\langle\alpha\rangle$ always preserves suprema of finite sets:

$$\begin{aligned}\sup_{i=1}^{n} \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi_i) &= \mathfrak{m}_{\mathfrak{K}}(\bigvee_{i=1}^{n} \langle\alpha\rangle\varphi_i) \\ &= \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle \bigvee_{i=1}^{n} \varphi_i).\end{aligned}$$

This follows from $n-1$ applications of Theorem 5.6(i). However, if $A \subseteq \Phi$ is infinite, then $\sup_{\varphi\in A} \mathfrak{m}_{\mathfrak{K}}(\varphi)$ may not exist. Note that in general $\bigcup_{\varphi\in A} \mathfrak{m}_{\mathfrak{K}}(\varphi)$ is not the supremum, since it may not even be in $\mathfrak{m}_{\mathfrak{K}}(\Phi)$. Even if $\sup_{\varphi\in A} \mathfrak{m}_{\mathfrak{K}}(\varphi)$ does exist

(that is, if it is $\mathfrak{m}_{\mathfrak{K}}(\psi)$ for some $\psi \in \Phi$), it is not necessarily equal to $\bigcup_{\varphi \in A} \mathfrak{m}_{\mathfrak{K}}(\varphi)$.

The following theorem says that in the presence of $^-$, all existing suprema are preserved by the operator $\langle\alpha\rangle$.

THEOREM 5.14: In PDL with $^-$, the map $\varphi \mapsto \langle\alpha\rangle\varphi$ is continuous with respect to the order of logical implication. That is, if $\mathfrak{K}$ is a Kripke frame, A a (finite or infinite) set of formulas, and φ a formula such that

$$\mathfrak{m}_{\mathfrak{K}}(\varphi) \quad = \quad \sup_{\psi \in A} \mathfrak{m}_{\mathfrak{K}}(\psi),$$

then $\sup_{\psi \in A} \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\psi)$ exists and is equal to $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi)$.

Proof Since $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ is an upper bound for $\{\mathfrak{m}_{\mathfrak{K}}(\psi) \mid \psi \in A\}$, we have that

$$\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\psi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi)$$

for each $\psi \in A$ by the monotonicity of $\langle\alpha\rangle$ (Theorem 5.7(ii)), thus $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi)$ is an upper bound for $\{\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\psi) \mid \psi \in A\}$. To show it is the least upper bound, suppose ρ is any other upper bound; that is,

$$\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\psi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}(\rho)$$

for all $\psi \in A$. By the monotonicity of $[\alpha^-]$ (Theorem 5.7(iii)),

$$\mathfrak{m}_{\mathfrak{K}}([\alpha^-]\langle\alpha\rangle\psi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}([\alpha^-]\rho)$$

for all $\psi \in A$, and by Theorem 5.13(ii),

$$\mathfrak{m}_{\mathfrak{K}}(\psi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}([\alpha^-]\langle\alpha\rangle\psi)$$

for all $\psi \in A$, thus $\mathfrak{m}_{\mathfrak{K}}([\alpha^-]\rho)$ is an upper bound for $\{\mathfrak{m}_{\mathfrak{K}}(\psi) \mid \psi \in A\}$. Since $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ is the least upper bound,

$$\mathfrak{m}_{\mathfrak{K}}(\varphi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}([\alpha^-]\rho).$$

By the monotonicity of $\langle\alpha\rangle$ again,

$$\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle[\alpha^-]\rho),$$

and by Theorem 5.13(iii),

$$\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle[\alpha^-]\rho) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}(\rho),$$

therefore

$$\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi) \quad \subseteq \quad \mathfrak{m}_{\mathfrak{K}}(\rho).$$

Since $\mathfrak{m}_{\mathfrak{K}}(\rho)$ was an arbitrary upper bound for $\{\mathfrak{m}_{\mathfrak{K}}(\langle p\rangle\psi) \mid \psi \in A\}$, $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\varphi)$ must be the least upper bound. ■

The Iteration Operator *

The iteration operator * is interpreted as the reflexive transitive closure operator on binary relations. It is the means by which iteration is coded in PDL. This operator differs from the other operators in that it is infinitary in nature, as reflected by its semantics:

$$\mathfrak{m}_{\mathfrak{K}}(\alpha^*) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\alpha)^* \quad = \quad \bigcup_{n<\omega} \mathfrak{m}_{\mathfrak{K}}(\alpha)^n$$

(see Section 5.2). This introduces a level of complexity to PDL beyond the other operators. Because of it, PDL is not compact: the set

$$\{\langle\alpha^*\rangle\varphi\} \quad \cup \quad \{\neg\varphi,\ \neg\langle\alpha\rangle\varphi,\ \neg\langle\alpha^2\rangle\varphi,\ \ldots\} \tag{5.6.1}$$

is finitely satisfiable but not satisfiable. Because of this infinitary behavior, it is rather surprising that PDL should be decidable and that there should be a finitary complete axiomatization.

The properties of the * operator of PDL come directly from the properties of the reflexive transitive closure operator * on binary relations, as described in Section 1.3 and Exercises 1.12 and 1.13. In a nutshell, for any binary relation R, R^* is the $\subseteq$-least reflexive and transitive relation containing R.

THEOREM 5.15: The following are valid formulas of PDL:

(i) $[\alpha^*]\varphi \quad \rightarrow \quad \varphi$
(ii) $\varphi \quad \rightarrow \quad \langle\alpha^*\rangle\varphi$
(iii) $[\alpha^*]\varphi \quad \rightarrow \quad [\alpha]\varphi$
(iv) $\langle\alpha\rangle\varphi \quad \rightarrow \quad \langle\alpha^*\rangle\varphi$
(v) $[\alpha^*]\varphi \quad \leftrightarrow \quad [\alpha^*\alpha^*]\varphi$
(vi) $\langle\alpha^*\rangle\varphi \quad \leftrightarrow \quad \langle\alpha^*\alpha^*\rangle\varphi$
(vii) $[\alpha^*]\varphi \quad \leftrightarrow \quad [\alpha^{**}]\varphi$
(viii) $\langle\alpha^*\rangle\varphi \quad \leftrightarrow \quad \langle\alpha^{**}\rangle\varphi$
(ix) $[\alpha^*]\varphi \quad \leftrightarrow \quad \varphi \wedge [\alpha][\alpha^*]\varphi.$

(x) $\langle\alpha^*\rangle\varphi \quad \leftrightarrow \quad \varphi \vee \langle\alpha\rangle\langle\alpha^*\rangle\varphi$.
(xi) $[\alpha^*]\varphi \quad \leftrightarrow \quad \varphi \wedge [\alpha^*](\varphi \rightarrow [\alpha]\varphi)$.
(xii) $\langle\alpha^*\rangle\varphi \quad \leftrightarrow \quad \varphi \vee \langle\alpha^*\rangle(\neg\varphi \wedge \langle\alpha\rangle\varphi)$.

Proof These properties follow immediately from the semantics of PDL (Section 5.2) and the properties of reflexive transitive closure (Exercises 1.7, 1.12, and 1.13). ■

Semantically, α^* is a reflexive and transitive relation containing α, and Theorem 5.15 captures this. That α^* is reflexive is captured in (ii); that it is transitive is captured in (vi); and that it contains α is captured in (iv). These three properties are captured by the single property (x).

Reflexive Transitive Closure and Induction

To prove properties of iteration, it is not enough to know that α^* is a reflexive and transitive relation containing α. So is the universal relation $K \times K$, and that is not very interesting. We also need some way of capturing the idea that α^* is the *least* reflexive and transitive relation containing α. There are several equivalent ways this can be done:

(RTC) The *reflexive transitive closure rule*:

$$\frac{(\varphi \vee \langle\alpha\rangle\psi) \rightarrow \psi}{\langle\alpha^*\rangle\varphi \rightarrow \psi}$$

(LI) The *loop invariance rule*:

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

(IND) The *induction axiom* (box form):

$$\varphi \wedge [\alpha^*](\varphi \rightarrow [\alpha]\varphi) \quad \rightarrow \quad [\alpha^*]\varphi$$

(IND) The *induction axiom* (diamond form):

$$\langle\alpha^*\rangle\varphi \quad \rightarrow \quad \varphi \vee \langle\alpha^*\rangle(\neg\varphi \wedge \langle\alpha\rangle\varphi)$$

The rule (RTC) is called the *reflexive transitive closure rule*. Its importance is best described in terms of its relationship to the valid PDL formula of Theorem 5.15(x). Observe that the right-to-left implication of this formula is obtained by substituting $\langle\alpha^*\rangle\varphi$ for R in the expression

$$\varphi \vee \langle\alpha\rangle R \quad \rightarrow \quad R. \tag{5.6.2}$$

Theorem 5.15(x) implies that $\langle\alpha^*\rangle\varphi$ is a solution of (5.6.2); that is, (5.6.2) is valid when $\langle\alpha^*\rangle\varphi$ is substituted for R. The rule (RTC) says that $\langle\alpha^*\rangle\varphi$ is the *least* such solution with respect to logical implication. That is, it is the least PDL-definable set of states that when substituted for R in (5.6.2) results in a valid formula.

The dual propositions labeled (IND) are jointly called the PDL *induction axiom*. Intuitively, the box form of (IND) says, "If φ is true initially, and if, after any number of iterations of the program α, the truth of φ is preserved by one more iteration of α, then φ will be true after any number of iterations of α." The diamond form of (IND) says, "If it is possible to reach a state satisfying φ in some number of iterations of α, then either φ is true now, or it is possible to reach a state in which φ is false but becomes true after one more iteration of α."

As mentioned in Section 5.5, the box form of (IND) bears a strong resemblance to the induction axiom of Peano arithmetic:

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

In Theorem 5.18 below, we argue that in the presence of the other axioms and rules of PDL, the rules (RTC), (LI), and (IND) are interderivable. First, however, we argue that the rule (RTC) is sound. The soundness of (LI) and (IND) will follow from Theorem 5.18.

THEOREM 5.16: The reflexive transitive closure rule (RTC) is sound.

Proof We need to show that in any model $\mathfrak{K}$, if $\mathfrak{m}_{\mathfrak{K}}(\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$ and $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\psi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$, then $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^*\rangle\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$. We show by induction on n that $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^n\rangle\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$. Certainly $\mathfrak{m}_{\mathfrak{K}}(\varphi) = \mathfrak{m}_{\mathfrak{K}}(\langle\mathbf{skip}\rangle\varphi)$, since $\mathfrak{m}_{\mathfrak{K}}(\mathbf{skip}) = \iota$, and ι is an identity for relational composition (Exercise 1.2). By definition, $\alpha^0 = \mathbf{skip}$, so $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^0\rangle\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$.

Now suppose $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^n\rangle\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$. Then

$$\begin{array}{lcll} \mathfrak{m}_{\mathfrak{K}}(\langle\alpha^{n+1}\rangle\varphi) & = & \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\langle\alpha^n\rangle\varphi) & \\ & \subseteq & \mathfrak{m}_{\mathfrak{K}}(\langle\alpha\rangle\psi) & \text{by the monotonicity of } \langle\alpha\rangle \\ & \subseteq & \mathfrak{m}_{\mathfrak{K}}(\psi) & \text{by assumption.} \end{array}$$

Thus for all n, $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^n\rangle\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$. Since $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^*\rangle\varphi) = \bigcup_{n<\omega} \mathfrak{m}_{\mathfrak{K}}(\langle\alpha^n\rangle\varphi)$, we have that $\mathfrak{m}_{\mathfrak{K}}(\langle\alpha^*\rangle\varphi) \subseteq \mathfrak{m}_{\mathfrak{K}}(\psi)$. ■

The deductive relationship between the induction axiom (IND), the reflexive transitive closure rule (RTC), and the rule of loop invariance (LI) is summed up in the following lemma and theorem. We emphasize that these results are purely proof-theoretic and independent of the semantics of Section 5.2.

LEMMA 5.17: The monotonicity rules of Theorem 5.7(ii) and (iii) are derivable in PDL without the induction axiom.

Proof This is really a theorem of modal logic. First we show that the rule of Theorem 5.7(iii) is derivable in PDL without induction. Assuming the premise $\varphi \to \psi$ and applying modal generalization, we obtain $[\alpha](\varphi \to \psi)$. The conclusion $[\alpha]\varphi \to [\alpha]\psi$ then follows from Axiom 5.5(ii) and modus ponens. The dual monotonicity rule, Theorem 5.7(ii), can be derived from (iii) by pure propositional reasoning. ■

THEOREM 5.18: In PDL without the induction axiom, the following axioms and rules are interderivable:

- the induction axiom (IND);
- the loop invariance rule (LI);
- the reflexive transitive closure rule (RTC).

Proof (IND) → (LI) Assume the premise of (LI):

$\varphi \to [\alpha]\varphi$.

By modal generalization,

$[\alpha^*](\varphi \to [\alpha]\varphi)$,

thus

$$\begin{aligned}\varphi &\to \varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)\\ &\to [\alpha^*]\varphi.\end{aligned}$$

The first implication is by propositional reasoning and the second is by (IND). By transitivity of implication (Example 3.7), we obtain

$$\varphi \to [\alpha^*]\varphi,$$

which is the conclusion of (LI).

(LI) → (RTC) Dualizing (RTC) by purely propositional reasoning, we obtain a rule

$$\frac{\psi \to \varphi \wedge [\alpha]\psi}{\psi \to [\alpha^*]\varphi} \tag{5.6.3}$$

equivalent to (RTC). It thus suffices to derive (5.6.3) from (LI). From the premise of (5.6.3), we obtain by propositional reasoning the two formulas

$$\psi \to \varphi \tag{5.6.4}$$

$$\psi \to [\alpha]\psi. \tag{5.6.5}$$

Applying (LI) to (5.6.5), we obtain

$$\psi \to [\alpha^*]\psi,$$

which by (5.6.4) and monotonicity (Lemma 5.17) gives

$$\psi \to [\alpha^*]\varphi.$$

This is the conclusion of (5.6.3).

(RTC) → (IND) By Axiom 5.5(iii) and (vii) and propositional reasoning, we have

$$\begin{aligned}&\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)\\ &\quad\to \varphi \wedge (\varphi \to [\alpha]\varphi) \wedge [\alpha][\alpha^*](\varphi \to [\alpha]\varphi)\\ &\quad\to \varphi \wedge [\alpha]\varphi \wedge [\alpha][\alpha^*](\varphi \to [\alpha]\varphi)\\ &\quad\to \varphi \wedge [\alpha](\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)).\end{aligned}$$

By transitivity of implication (Example 3.7),

$$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \to \varphi \wedge [\alpha](\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi)).$$

Applying (5.6.3), which we have argued is equivalent to (RTC), we obtain (IND):

$$\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi.$$

■

5.7 Encoding Hoare Logic

Recall that the Hoare partial correctness assertion $\{\varphi\}\,\alpha\,\{\psi\}$ is encoded as $\varphi \to [\alpha]\psi$ in PDL. The following theorem says that under this encoding, Dynamic Logic subsumes Hoare Logic.

THEOREM 5.19: The following rules of Hoare Logic are derivable in PDL:

(i) Composition rule:

$$\frac{\{\varphi\}\,\alpha\,\{\sigma\}, \quad \{\sigma\}\,\beta\,\{\psi\}}{\{\varphi\}\,\alpha\,;\,\beta\,\{\psi\}}$$

(ii) Conditional rule:

$$\frac{\{\varphi \wedge \sigma\}\,\alpha\,\{\psi\}, \quad \{\neg\varphi \wedge \sigma\}\,\beta\,\{\psi\}}{\{\sigma\}\,\textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta\,\{\psi\}}$$

(iii) **While** rule:

$$\frac{\{\varphi \wedge \psi\}\,\alpha\,\{\psi\}}{\{\psi\}\,\textbf{while } \varphi \textbf{ do } \alpha\,\{\neg\varphi \wedge \psi\}}$$

(iv) Weakening rule:

$$\frac{\varphi' \to \varphi, \quad \{\varphi\}\,\alpha\,\{\psi\}, \quad \psi \to \psi'}{\{\varphi'\}\,\alpha\,\{\psi'\}}$$

Proof We derive the **while** rule (iii) in PDL. The other Hoare rules are also derivable, and we leave them as exercises (Exercise 5.14).

Assuming the premise

$$\{\varphi \wedge \psi\}\,\alpha\,\{\psi\} \quad = \quad (\varphi \wedge \psi) \to [\alpha]\psi, \tag{5.7.1}$$

we wish to derive the conclusion

$$\{\psi\}\,\textbf{while } \varphi \textbf{ do } \alpha\,\{\neg\varphi \wedge \psi\} \quad = \quad \psi \to [(\varphi?;\alpha)^*;\neg\varphi?](\neg\varphi \wedge \psi). \tag{5.7.2}$$

Using propositional reasoning, (5.7.1) is equivalent to

$\psi \quad \rightarrow \quad (\varphi \rightarrow \texttt{[}\alpha\texttt{]}\psi)$,

which by Axioms 5.5(v) and (vi) is equivalent to

$\psi \quad \rightarrow \quad \texttt{[}\varphi?;\alpha\texttt{]}\psi$.

Applying the loop invariance rule (LI), we obtain

$\psi \quad \rightarrow \quad \texttt{[}(\varphi?;\alpha)^*\texttt{]}\psi$.

By the monotonicity of $\texttt{[}(\varphi?;\alpha)^*\texttt{]}$ (Lemma 5.17) and propositional reasoning,

$\psi \quad \rightarrow \quad \texttt{[}(\varphi?;\alpha)^*\texttt{]}(\neg\varphi \rightarrow (\neg\varphi \wedge \psi))$,

and by Axiom 5.5(vi), we obtain

$\psi \quad \rightarrow \quad \texttt{[}(\varphi?;\alpha)^*\texttt{]}\texttt{[}\neg\varphi?\texttt{]}(\neg\varphi \wedge \psi)$.

By Axiom 5.5(v), this is equivalent to the desired conclusion (5.7.2). ■

5.8 Bibliographical Notes

Burstall (1974) suggested using modal logic for reasoning about programs, but it was not until the work of Pratt (1976), prompted by a suggestion of R. Moore, that it was actually shown how to extend modal logic in a useful way by considering a separate modality for every program. The first research devoted to propositional reasoning about programs seems to be that of Fischer and Ladner (1977, 1979) on PDL. As mentioned in the Preface, the general use of logical systems for reasoning about programs was suggested by Engeler (1967).

Other semantics besides Kripke semantics have been studied; see Berman (1979); Nishimura (1979); Kozen (1979b); Trnkova and Reiterman (1980); Kozen (1980b); Pratt (1979b). Modal logic has many applications and a vast literature; good introductions can be found in Hughes and Cresswell (1968); Chellas (1980). Alternative and iterative guarded commands were studied in Gries (1981). Partial correctness assertions and the Hoare rules given in Section 5.7 were first formulated by Hoare (1969). Regular expressions, on which the regular program operators are based, were introduced by Kleene (1956). Their algebraic theory was further investigated by Conway (1971). They were first applied in the context of DL by Fischer and Ladner (1977, 1979). The axiomatization of PDL given in Axioms 5.5 was formulated by Segerberg (1977). Tests and converse were investigated by various

authors; see Peterson (1978); Berman (1978); Berman and Paterson (1981); Streett (1981, 1982); Vardi (1985b). Theorem 5.14 is due to Trnkova and Reiterman (1980).

Exercises

5.1. Prove Proposition 5.2.

5.2. A program α is said to be *semantically deterministic* in a Kripke frame $\mathfrak{K}$ if its traces are uniquely determined by their first states. Show that if α and β are semantically deterministic in a structure $\mathfrak{K}$, then so are **if** φ **then** α **else** β and **while** φ **do** α.

5.3. We say that two programs α and β are *equivalent* if they represent the same binary relation in all Kripke frames; that is, if $\mathfrak{m}_{\mathfrak{K}}(\alpha) = \mathfrak{m}_{\mathfrak{K}}(\beta)$ for all $\mathfrak{K}$. Let p be an atomic proposition not occurring in α or β. Prove that α and β are equivalent iff the PDL formula $\langle\alpha\rangle p \leftrightarrow \langle\beta\rangle p$ is valid.

5.4. Prove in PDL that the following pairs of programs are equivalent in the sense of Exercise 5.3. For (c), use the encodings (4.3.1) and (4.3.2) of Section 4.3 and reason in terms of the regular operators.

(a) $a(ba)^*$ $(ab)^*a$

(b) $(a \cup b)^*$ $(a^*b)^*a^*$

(c)
while b **do begin**
p;
while c **do** q
end

if b **then begin**
p;
while $b \vee c$ **do**
if c **then** q **else** p
end

5.5. Let α be the program (5.4.1) of Example 5.3. Show that for any proposition φ, the proposition $\varphi \leftrightarrow [\alpha]\varphi$ is valid in the model $\mathfrak{K}$ of that example.

5.6. Prove that the formula (5.4.2) of Example 5.4 is valid. Give a semantic

argument using the semantics of PDL given in Section 5.2, not the deductive system of Section 5.5.

5.7. Prove that the box and diamond forms of the PDL induction axiom are equivalent. See Section 5.6.

5.8. Prove that the following statements are valid:

$$\begin{aligned} \langle\alpha\rangle\varphi &\rightarrow \langle\alpha\alpha^-\alpha\rangle\varphi \\ \langle\alpha^*\alpha\rangle\varphi &\leftrightarrow \langle\alpha\alpha^*\rangle\varphi \\ \langle\alpha^*\rangle\varphi &\leftrightarrow \varphi \vee \langle\alpha\rangle\varphi \vee \langle\alpha\alpha\rangle\varphi \vee \cdots \vee \langle\alpha^{n-1}\rangle\varphi \vee \langle\alpha^n\alpha^*\rangle\varphi. \end{aligned}$$

5.9. Prove that the following are theorems of PDL. Use Axiom System 5.5, do not reason semantically.

(i) $\langle\alpha\rangle\varphi \wedge [\alpha]\psi \rightarrow \langle\alpha\rangle(\varphi \wedge \psi)$
(ii) $\langle\alpha\rangle(\varphi \vee \psi) \leftrightarrow \langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi$
(iii) $\langle\alpha \cup \beta\rangle\varphi \leftrightarrow \langle\alpha\rangle\varphi \vee \langle\beta\rangle\varphi$
(iv) $\langle\alpha\beta\rangle\varphi \leftrightarrow \langle\alpha\rangle\langle\beta\rangle\varphi$
(v) $\langle\psi?\rangle\varphi \leftrightarrow \psi \wedge \varphi$
(vi) $\langle\alpha^*\rangle\varphi \leftrightarrow \varphi \vee \langle\alpha\alpha^*\rangle\varphi$
(vii) $\langle\alpha^*\rangle\varphi \rightarrow \varphi \vee \langle\alpha^*\rangle(\neg\varphi \wedge \langle\alpha\rangle\varphi)$.
(viii) $\langle\alpha^*\rangle\varphi \leftrightarrow \varphi \vee \langle\alpha^*\rangle(\neg\varphi \wedge \langle\alpha\rangle\varphi)$.

In the presence of the converse operator $^-$,

(ix) $\langle\alpha\rangle[\alpha^-]\varphi \rightarrow \varphi$
(x) $\langle\alpha^-\rangle[\alpha]\varphi \rightarrow \varphi$.

5.10. Give counterexamples showing that the converses of Theorem 5.6(iv)–(vi) are not valid.

5.11. Supply the missing details in the proof of Corollary 5.9.

5.12. Show that Theorem 5.14 fails in PDL without the converse operator $^-$. Construct a Kripke model such that the operator $\langle a\rangle$ is not continuous.

5.13. Let Σ be a set of atomic programs and let Σ^* be the set of finite-length strings

over Σ. A *regular expression* over Σ is a PDL program over Σ with only operators $\cup$, *, and ;. A regular expression α denotes a set $L(\alpha)$ of strings in Σ^* as follows:

$$\begin{aligned} L(a) &\stackrel{\text{def}}{=} \{a\}, \quad a \in \Sigma \\ L(\alpha\beta) &\stackrel{\text{def}}{=} L(\alpha) \cdot L(\beta) \\ &= \{xy \mid x \in L(\alpha),\ y \in L(\beta)\} \\ L(\alpha \cup \beta) &\stackrel{\text{def}}{=} L(\alpha) \cup L(\beta) \\ L(\alpha^*) &\stackrel{\text{def}}{=} \bigcup_{n<\omega} L(\alpha)^n, \end{aligned}$$

where $L(\alpha)^0 = \{\varepsilon\}$, $L(\alpha^{n+1}) = L(\alpha^n) \cdot L(\alpha)$, and ε is the empty string. Let p be an atomic proposition. Prove that for any two regular expressions α, β, $L(\alpha) = L(\beta)$ iff $\langle\alpha\rangle p \leftrightarrow \langle\beta\rangle p$ is a theorem of PDL.

5.14. Prove that the composition, conditional, and weakening rules of Hoare Logic (Theorem 5.19(i), (ii), and (iv), respectively) are derivable in PDL.

6 Filtration and Decidability

In this chapter we will establish a *small model property* for PDL. This result and the technique used to prove it, called *filtration*, come directly from modal logic.

The small model property says that if φ is satisfiable, then it is satisfied at a state in a Kripke frame with no more than $2^{|\varphi|}$ states, where $|\varphi|$ is the number of symbols of φ. This immediately gives a naive decision procedure for the satisfiability problem for PDL: to determine whether φ is satisfiable, construct all Kripke frames with at most $2^{|\varphi|}$ states and check whether φ is satisfied at some state in one of them. Considering only interpretations of the primitive formulas and primitive programs appearing in φ, there are roughly $2^{2^{|\varphi|}}$ such models, so this algorithm is too inefficient to be practical. A more efficient algorithm will be given in Chapter 8.

6.1 The Fischer–Ladner Closure

Many proofs in simpler modal systems use induction on the well-founded subformula relation. In PDL, the situation is complicated by the simultaneous inductive definitions of programs and propositions and by the behavior of the * operator, which make the induction proofs somewhat tricky. Nevertheless, we can still use the well-founded subexpression relation in inductive proofs. Here an *expression* can be either a program or a proposition. Either one can be a subexpression of the other because of the mixed operators $[\,]$ and ?.

We start by defining two functions

$$\begin{array}{rcl} FL & : & \Phi \;\to\; 2^{\Phi} \\ FL^{\Box} & : & \{[\alpha]\varphi \mid \alpha \in \Psi,\ \varphi \in \Phi\} \;\to\; 2^{\Phi} \end{array}$$

by simultaneous induction. The set $FL(\varphi)$ is called the *Fischer–Ladner closure* of φ. The filtration construction of Lemma 6.3 uses the Fischer–Ladner closure of a given formula where the corresponding proof for propositional modal logic would use the set of subformulas.

The functions FL and $FL^{\Box}$ are defined inductively as follows:

(a) $FL(p) \stackrel{\text{def}}{=} \{p\}$, p an atomic proposition

(b) $FL(\varphi \to \psi) \stackrel{\text{def}}{=} \{\varphi \to \psi\} \cup FL(\varphi) \cup FL(\psi)$

(c) $FL(\mathbf{0}) \stackrel{\text{def}}{=} \{\mathbf{0}\}$

(d) $FL(\texttt{[}\alpha\texttt{]}\varphi) \stackrel{\text{def}}{=} FL^{\Box}(\texttt{[}\alpha\texttt{]}\varphi) \cup FL(\varphi)$

(e) $FL^{\Box}(\texttt{[}a\texttt{]}\varphi) \stackrel{\text{def}}{=} \{\texttt{[}a\texttt{]}\varphi\}$, a an atomic program

(f) $FL^{\Box}(\texttt{[}\alpha \cup \beta\texttt{]}\varphi) \stackrel{\text{def}}{=} \{\texttt{[}\alpha \cup \beta\texttt{]}\varphi\} \cup FL^{\Box}(\texttt{[}\alpha\texttt{]}\varphi) \cup FL^{\Box}(\texttt{[}\beta\texttt{]}\varphi)$

(g) $FL^{\Box}(\texttt{[}\alpha\,;\,\beta\texttt{]}\varphi) \stackrel{\text{def}}{=} \{\texttt{[}\alpha\,;\,\beta\texttt{]}\varphi\} \cup FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\beta\texttt{]}\varphi) \cup FL^{\Box}(\texttt{[}\beta\texttt{]}\varphi)$

(h) $FL^{\Box}(\texttt{[}\alpha^*\texttt{]}\varphi) \stackrel{\text{def}}{=} \{\texttt{[}\alpha^*\texttt{]}\varphi\} \cup FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\alpha^*\texttt{]}\varphi)$

(i) $FL^{\Box}(\texttt{[}\psi?\texttt{]}\varphi) \stackrel{\text{def}}{=} \{\texttt{[}\psi?\texttt{]}\varphi\} \cup FL(\psi)$.

This definition is apparently quite a bit more involved than for mere subexpressions. In fact, at first glance it may appear circular because of the rule (h). The auxiliary function $FL^{\Box}$ is introduced for the express purpose of avoiding any such circularity. It is defined only for formulas of the form $\texttt{[}\alpha\texttt{]}\varphi$ and intuitively produces those elements of $FL(\texttt{[}\alpha\texttt{]}\varphi)$ obtained by breaking down α and ignoring φ.

Even after convincing ourselves that the definition is noncircular, it may not be clear how the size of $FL(\varphi)$ depends on the length of φ. Indeed, the right-hand side of rule (h) involves a formula that is larger than the formula on the left-hand side. We will be able to establish a linear relationship by induction on the well-founded subexpression relation (Lemma 6.3).

First we show a kind of transitivity property of FL and $FL^{\Box}$ that will be useful in later arguments.

Lemma 6.1:

(i) If $\sigma \in FL(\varphi)$, then $FL(\sigma) \subseteq FL(\varphi)$.

(ii) If $\sigma \in FL^{\Box}(\texttt{[}\alpha\texttt{]}\varphi)$, then $FL(\sigma) \subseteq FL^{\Box}(\texttt{[}\alpha\texttt{]}\varphi) \cup FL(\varphi)$.

Proof We prove (i) and (ii) by simultaneous induction on the well-founded subexpression relation.

First we show (i), assuming by the induction hypothesis that (i) and (ii) hold for proper subexpressions of φ. There are four cases, depending on the form of φ: an atomic proposition p, $\varphi \to \psi$, $\mathbf{0}$, or $\texttt{[}\alpha\texttt{]}\varphi$. We argue the second and fourth cases explicitly and leave the first and third as exercises (Exercise 6.1).

If $\sigma \in FL(\varphi \to \psi)$, then by clause (b) in the definition of FL, either $\sigma = \varphi \to \psi$, $\sigma \in FL(\varphi)$, or $\sigma \in FL(\psi)$. In the first case, $FL(\sigma) = FL(\varphi \to \psi)$, and we are done. In the second and third cases, we have $FL(\sigma) \subseteq FL(\varphi)$ and $FL(\sigma) \subseteq FL(\psi)$, respectively, by the induction hypothesis (i). In either case, $FL(\sigma) \subseteq FL(\varphi \to \psi)$ by clause (b) in the definition of FL.

If $\sigma \in FL(\texttt{[}\alpha\texttt{]}\varphi)$, then by clause (d) in the definition of FL, either $\sigma \in$

$FL^{\Box}(\texttt{[}\alpha\texttt{]}\varphi)$ or $\sigma \in FL(\varphi)$. In the former case, $FL(\sigma) \subseteq FL^{\Box}(\texttt{[}\alpha\texttt{]}\varphi) \cup FL(\varphi)$ by the induction hypothesis (ii). (The induction hypothesis holds here because α is a proper subexpression of $\texttt{[}\alpha\texttt{]}\varphi$.) In the latter case, $FL(\sigma) \subseteq FL(\varphi)$ by the induction hypothesis (i). Thus in either case, $FL(\sigma) \subseteq FL(\texttt{[}\alpha\texttt{]}\varphi)$ by clause (d) in the definition of FL.

Now we show (ii), again assuming that (i) and (ii) hold for proper subexpressions. There are five cases, depending on the form of the program: an atomic program a, $\alpha \cup \beta$, $\alpha\,;\,\beta$, α^*, or $\psi?$. We argue the third and fourth cases explicitly, leaving the remaining three as exercises (Exercise 6.1).

If $\sigma \in FL^{\Box}(\texttt{[}\alpha\,;\,\beta\texttt{]}\varphi)$, then by clause (g) in the definition of $FL^{\Box}$, either

(A) $\sigma = \texttt{[}\alpha\,;\,\beta\texttt{]}\varphi$,
(B) $\sigma \in FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\beta\texttt{]}\varphi)$, or
(C) $\sigma \in FL^{\Box}(\texttt{[}\beta\texttt{]}\varphi)$.

In case (A), $FL(\sigma) = FL^{\Box}(\texttt{[}\alpha\,;\,\beta\texttt{]}\varphi) \cup FL(\varphi)$ by clause (d) in the definition of FL, and we are done. In case (B), we have

$$\begin{array}{lll} FL(\sigma) \subseteq & FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\beta\texttt{]}\varphi) \cup FL(\texttt{[}\beta\texttt{]}\varphi) & \text{by the induction hypothesis (ii)} \\ = & FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\beta\texttt{]}\varphi) \cup FL^{\Box}(\texttt{[}\beta\texttt{]}\varphi) \cup FL(\varphi) & \\ & & \text{by clause (d) in the definition of } FL \\ \subseteq & FL^{\Box}(\texttt{[}\alpha\,;\,\beta\texttt{]}\varphi) \cup FL(\varphi) & \text{by clause (g) in the definition of } FL^{\Box}. \end{array}$$

In case (C),

$$\begin{array}{lll} FL(\sigma) \subseteq & FL^{\Box}(\texttt{[}\beta\texttt{]}\varphi) \cup FL(\varphi) & \text{by the induction hypothesis (ii)} \\ \subseteq & FL^{\Box}(\texttt{[}\alpha\,;\,\beta\texttt{]}\varphi) \cup FL(\varphi) & \text{by clause (g) in the definition of } FL^{\Box}. \end{array}$$

If $\sigma \in FL^{\Box}(\texttt{[}\alpha^*\texttt{]}\varphi)$, then by clause (h) in the definition of $FL^{\Box}$, either $\sigma = \texttt{[}\alpha^*\texttt{]}\varphi$ or $\sigma \in FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\alpha^*\texttt{]}\varphi)$. In the former case, $FL(\sigma) = FL^{\Box}(\texttt{[}\alpha^*\texttt{]}\varphi) \cup FL(\varphi)$ by clause (d) in the definition of FL. In the latter case, we have

$$\begin{array}{ll} FL(\sigma) \subseteq & FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\alpha^*\texttt{]}\varphi) \cup FL(\texttt{[}\alpha^*\texttt{]}\varphi) \\ = & FL^{\Box}(\texttt{[}\alpha\texttt{]}\,\texttt{[}\alpha^*\texttt{]}\varphi) \cup FL^{\Box}(\texttt{[}\alpha^*\texttt{]}\varphi) \cup FL(\varphi) \\ \subseteq & FL^{\Box}(\texttt{[}\alpha^*\texttt{]}\varphi) \cup FL(\varphi) \end{array}$$

by the induction hypothesis (ii) and clauses (d) and (h) in the definition of FL and $FL^{\Box}$. ■

The following closure properties of FL are straightforward consequences of

Lemma 6.1.

LEMMA 6.2:

(i) If $[\alpha]\psi \in FL(\varphi)$, then $\psi \in FL(\varphi)$.
(ii) If $[\rho?]\psi \in FL(\varphi)$, then $\rho \in FL(\varphi)$.
(iii) If $[\alpha \cup \beta]\psi \in FL(\varphi)$, then $[\alpha]\psi \in FL(\varphi)$ and $[\beta]\psi \in FL(\varphi)$.
(iv) If $[\alpha\,;\,\beta]\psi \in FL(\varphi)$, then $[\alpha][\beta]\psi \in FL(\varphi)$ and $[\beta]\psi \in FL(\varphi)$.
(v) If $[\alpha^*]\psi \in FL(\varphi)$, then $[\alpha][\alpha^*]\psi \in FL(\varphi)$.

Proof Exercise 6.2. ■

The following lemma bounds the cardinality of $FL(\varphi)$ as a function of the length of φ. Recall that $\#A$ denotes the cardinality of a set A. Let $|\varphi|$ and $|\alpha|$ denote the length (number of symbols) of φ and α, respectively, excluding parentheses.

LEMMA 6.3:

(i) For any formula φ, $\#FL(\varphi) \;\leq\; |\varphi|$.
(ii) For any formula $[\alpha]\varphi$, $\#FL^{\Box}([\alpha]\varphi) \;\leq\; |\alpha|$.

Proof The proof is by simultaneous induction on the well-founded subexpression relation. First we show (i). If φ is an atomic formula p, then

$$\#FL(p) \;=\; 1 \;=\; |p|.$$

If φ is of the form $\psi \to \rho$, then

$$\begin{aligned} \#FL(\psi \to \rho) &\leq 1 + \#FL(\psi) + \#FL(\rho) \\ &\leq 1 + |\psi| + |\rho| \quad \text{by the induction hypothesis (i)} \\ &= |\psi \to \rho|. \end{aligned}$$

The argument for φ of the form $\mathbf{0}$ is easy. Finally, if φ is of the form $[\alpha]\psi$, then

$$\begin{aligned} \#FL([\alpha]\psi) &\leq \#FL^{\Box}([\alpha]\psi) + \#FL(\psi) \\ &\leq |\alpha| + |\psi| \quad \text{by the induction hypothesis (i) and (ii)} \\ &\leq |[\alpha]\psi|. \end{aligned}$$

Now we show (ii). If α is an atomic program a, then

$$\#FL^{\Box}([a]\varphi) \;=\; 1 \;=\; |a|.$$

If α is of the form $\beta \cup \gamma$, then

$$\begin{aligned} \#FL^{\Box}([\beta \cup \gamma]\varphi) &\leq 1 + \#FL^{\Box}([\beta]\varphi) + \#FL^{\Box}([\gamma]\varphi) \\ &\leq 1 + |\beta| + |\gamma| \\ &= |\beta \cup \gamma|. \end{aligned}$$

If α is of the form $\beta\,;\,\gamma$, then

$$\begin{aligned} \#FL^{\Box}([\beta\,;\,\gamma]\varphi) &\leq 1 + \#FL^{\Box}([\beta][\gamma]\varphi) + \#FL^{\Box}([\gamma]\varphi) \\ &\leq 1 + |\beta| + |\gamma| \\ &= |\beta\,;\,\gamma|. \end{aligned}$$

If α is of the form β^*, then

$$\begin{aligned} \#FL^{\Box}([\beta^*]\varphi) &\leq 1 + \#FL^{\Box}([\beta][\beta^*]\varphi) \\ &\leq 1 + |\beta| \\ &= |\beta^*|. \end{aligned}$$

Finally, if α is of the form $\psi?$, then

$$\begin{aligned} \#FL^{\Box}([\psi?]\varphi) &\leq 1 + \#FL(\psi) \\ &\leq 1 + |\psi| \quad \text{by the induction hypothesis (i)} \\ &= |\psi?|. \end{aligned}$$

■

6.2 Filtration and the Small Model Theorem

Given a PDL proposition φ and a Kripke frame $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$, we define a new frame $\mathfrak{K}/FL(\varphi) = (K/FL(\varphi),\ \mathfrak{m}_{\mathfrak{K}/FL(\varphi)})$, called the *filtration of* $\mathfrak{K}$ *by* $FL(\varphi)$, as follows. Define a binary relation $\equiv$ on states of $\mathfrak{K}$ by:

$$u \equiv v \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall \psi \in FL(\varphi)\ (u \in \mathfrak{m}_{\mathfrak{K}}(\psi) \Longleftrightarrow v \in \mathfrak{m}_{\mathfrak{K}}(\psi)).$$

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

$$\begin{aligned}
[u] &\stackrel{\text{def}}{=} \{v \mid v \equiv u\} \\
K/FL(\varphi) &\stackrel{\text{def}}{=} \{[u] \mid u \in K\} \\
\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(p) &\stackrel{\text{def}}{=} \{[u] \mid u \in \mathfrak{m}_{\mathfrak{K}}(p)\}, \quad p \text{ an atomic proposition} \\
\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(a) &\stackrel{\text{def}}{=} \{([u],[v]) \mid (u,v) \in \mathfrak{m}_{\mathfrak{K}}(a)\}, \quad a \text{ an atomic program.}
\end{aligned}$$

The map $\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}$ is extended inductively to compound propositions and programs as described in Section 5.2.

The following key lemma relates $\mathfrak{K}$ and $\mathfrak{K}/FL(\varphi)$. Most of the difficulty in the following lemma is in the correct formulation of the induction hypotheses in the statement of the lemma. Once this is done, the proof is a fairly straightforward induction on the well-founded subexpression relation.

Lemma 6.4 (Filtration Lemma): Let $\mathfrak{K}$ be a Kripke frame and let u, v be states of $\mathfrak{K}$.

(i) For all $\psi \in FL(\varphi)$, $u \in \mathfrak{m}_{\mathfrak{K}}(\psi)$ iff $[u] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\psi)$.

(ii) For all $\texttt{[}\alpha\texttt{]}\psi \in FL(\varphi)$,

(a)if $(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)$ then $([u],[v]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha)$;

(b)if $([u],[v]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha)$ and $u \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\alpha\texttt{]}\psi)$, then $v \in \mathfrak{m}_{\mathfrak{K}}(\psi)$.

Proof The proof is by simultaneous induction on the well-founded subexpression relation. We start with (i). There are four cases, depending on the form of ψ.

Case 1 For atomic propositions $p \in FL(\varphi)$, if $u \in \mathfrak{m}_{\mathfrak{K}}(p)$, then by definition of $\mathfrak{K}/FL(\varphi)$, $[u] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(p)$. Conversely, if $[u] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(p)$, then there exists a u' such that $u' \equiv u$ and $u' \in \mathfrak{m}_{\mathfrak{K}}(p)$. But then $u \in \mathfrak{m}_{\mathfrak{K}}(p)$ as well.

Case 2 If $\psi \to \rho \in FL(\varphi)$, then by Lemma 6.1, both $\psi \in FL(\varphi)$ and $\rho \in FL(\varphi)$. By the induction hypothesis, (i) holds for ψ and ρ, therefore

$$\begin{aligned}
s \in \mathfrak{m}_{\mathfrak{K}}(\psi \to \rho) &\iff s \in \mathfrak{m}_{\mathfrak{K}}(\psi) \Longrightarrow s \in \mathfrak{m}_{\mathfrak{K}}(\rho) \\
&\iff [s] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\psi) \Longrightarrow [s] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\rho) \\
&\iff [s] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\psi \to \rho).
\end{aligned}$$

Case 3 The case of **0** is easy. We leave the details as an exercise (Exercise 6.3).

Case 4 If $[\![\alpha]\!]\psi \in FL(\varphi)$, we use the induction hypothesis for α and ψ. By Lemma 6.2(i), $\psi \in FL(\varphi)$. By the induction hypothesis, (i) holds for ψ and (ii) holds for $[\![\alpha]\!]\psi$. Using the latter fact, we have

$$s \in \mathfrak{m}_{\mathfrak{K}}([\![\alpha]\!]\psi) \implies \forall t\ (([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha) \Longrightarrow t \in \mathfrak{m}_{\mathfrak{K}}(\psi)) \tag{6.2.1}$$

by clause (b) of (ii). Conversely,

$$\begin{aligned} &\forall t\ (([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha) \Longrightarrow t \in \mathfrak{m}_{\mathfrak{K}}(\psi)) \\ &\quad\Longrightarrow \forall t\ ((s,t) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \Longrightarrow t \in \mathfrak{m}_{\mathfrak{K}}(\psi)) \qquad (6.2.2)\\ &\quad\Longrightarrow s \in \mathfrak{m}_{\mathfrak{K}}([\![\alpha]\!]\psi) \end{aligned}$$

by clause (a) of (ii). Then

$$\begin{aligned} &s \in \mathfrak{m}_{\mathfrak{K}}([\![\alpha]\!]\psi) \\ &\quad\Longleftrightarrow \forall t\ (([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha) \Longrightarrow t \in \mathfrak{m}_{\mathfrak{K}}(\psi)) && \text{by (6.2.1) and (6.2.2)}\\ &\quad\Longleftrightarrow \forall t\ (([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\alpha) \Longrightarrow [t] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\psi)) && \text{by (i) for } \psi\\ &\quad\Longleftrightarrow [s] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}([\![\alpha]\!]\psi). \end{aligned}$$

This completes the proof of (i).

For (ii), there are five cases, depending on the form of α.

Case 1 For an atomic program a, part (a) of (ii) is immediate from the definition of $\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(a)$. For part (b), if $([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(a)$, then by the definition of $\mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(a)$, there exist $s' \equiv s$ and $t' \equiv t$ such that $(s',t') \in \mathfrak{m}_{\mathfrak{K}}(a)$. If $s \in \mathfrak{m}_{\mathfrak{K}}([\![a]\!]\psi)$, then since $s' \equiv s$ and $[\![a]\!]\psi \in FL(\varphi)$, we have $s' \in \mathfrak{m}_{\mathfrak{K}}([\![a]\!]\psi)$ as well, thus $t' \in \mathfrak{m}_{\mathfrak{K}}(\psi)$ by the semantics of $[\![a]\!]$. But $\psi \in FL(\varphi)$ by Lemma 6.2(i), and since $t \equiv t'$, we have $t \in \mathfrak{m}_{\mathfrak{K}}(\psi)$.

Case 2 For a test $\rho?$, by Lemma 6.2(ii) we have $\rho \in FL(\varphi)$, thus (i) holds for ρ by the induction hypothesis. Part (a) of (ii) is immediate from this. For (b),

$$\begin{aligned} &([s],[s]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\rho?) \text{ and } s \in \mathfrak{m}_{\mathfrak{K}}([\![\rho?]\!]\psi)\\ &\quad\Longrightarrow [s] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\rho) \text{ and } s \in \mathfrak{m}_{\mathfrak{K}}(\rho \to \psi)\\ &\quad\Longrightarrow s \in \mathfrak{m}_{\mathfrak{K}}(\rho) \text{ and } s \in \mathfrak{m}_{\mathfrak{K}}(\rho \to \psi)\\ &\quad\Longrightarrow s \in \mathfrak{m}_{\mathfrak{K}}(\psi). \end{aligned}$$

Case 3 The case $\alpha = \beta \cup \gamma$ is left as an exercise (Exercise 6.3).

Case 4 For the case $\alpha = \beta\,;\,\gamma$, to show (a), we have by Lemma 6.2(iv) that $\texttt{[}\beta\texttt{]}\,\texttt{[}\gamma\texttt{]}\psi \in FL(\varphi)$ and $\texttt{[}\gamma\texttt{]}\psi \in FL(\varphi)$, so (a) holds for β and γ; then

$$
\begin{aligned}
&(s,t) \in \mathfrak{m}_{\mathfrak{K}}(\beta\,;\,\gamma)\\
&\quad\Longrightarrow\quad \exists u\ (s,u) \in \mathfrak{m}_{\mathfrak{K}}(\beta) \text{ and } (u,t) \in \mathfrak{m}_{\mathfrak{K}}(\gamma)\\
&\quad\Longrightarrow\quad \exists u\ ([s],[u]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta) \text{ and } ([u],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\gamma)\\
&\quad\Longrightarrow\quad ([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta\,;\,\gamma).
\end{aligned}
$$

To show (b), we have by the induction hypothesis that (b) holds for $\texttt{[}\beta\texttt{]}\,\texttt{[}\gamma\texttt{]}\psi$ and $\texttt{[}\gamma\texttt{]}\psi$. Then

$$
\begin{aligned}
&([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta\,;\,\gamma) \text{ and } s \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta\,;\,\gamma\texttt{]}\psi)\\
&\quad\Longrightarrow\quad \exists u\ ([s],[u]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta),\ ([u],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\gamma),\ \text{and } s \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta\texttt{]}\,\texttt{[}\gamma\texttt{]}\psi)\\
&\quad\Longrightarrow\quad \exists u\ ([u],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\gamma) \text{ and } u \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\gamma\texttt{]}\psi) \qquad \text{by (b) for } \texttt{[}\beta\texttt{]}\,\texttt{[}\gamma\texttt{]}\psi\\
&\quad\Longrightarrow\quad t \in \mathfrak{m}_{\mathfrak{K}}(\psi) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{by (b) for } \texttt{[}\gamma\texttt{]}\psi.
\end{aligned}
$$

Case 5 Finally, consider the case $\alpha = \beta^*$. By Lemma 6.2(v), $\texttt{[}\beta\texttt{]}\,\texttt{[}\beta^*\texttt{]}\psi \in FL(\varphi)$, so we can assume that (ii) holds for $\texttt{[}\beta\texttt{]}\,\texttt{[}\beta^*\texttt{]}\psi$. (The induction hypothesis holds because β is a proper subexpression of β^*.) By part (a) of (ii), if $(u,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta)$, then $([u],[v]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta)$. Therefore if $(s,t) \in \mathfrak{m}_{\mathfrak{K}}(\beta^*)$, then there exist $n \geq 0$ and $t_0,\ldots,t_n$ such that $s = t_0$, $(t_i,t_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\beta)$ for $0 \leq i < n$, and $t_n = t$. This implies that $([t_i],[t_{i+1}]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta)$ for $0 \leq i < n$, therefore $([s],[t]) = ([t_0],[t_n]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta^*)$. This establishes (a).

To show (b), suppose $([s],[t]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta^*)$ and $s \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta^*\texttt{]}\psi)$. Then there exist $t_0,\ldots,t_n$ such that $s = t_0$, $t = t_n$, and $([t_i],[t_{i+1}]) \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\beta)$ for $0 \leq i < n$. We have that $t_0 = s \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta^*\texttt{]}\psi)$ by assumption. Now suppose $t_i \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta^*\texttt{]}\psi)$, $i < n$. Then $t_i \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta\texttt{]}\,\texttt{[}\beta^*\texttt{]}\psi)$. By the induction hypothesis for $\texttt{[}\beta\texttt{]}\,\texttt{[}\beta^*\texttt{]}\psi \in FL(\varphi)$, $t_{i+1} \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta^*\texttt{]}\psi)$. Continuing for n steps, we get $t = t_n \in \mathfrak{m}_{\mathfrak{K}}(\texttt{[}\beta^*\texttt{]}\psi)$, therefore $t \in \mathfrak{m}_{\mathfrak{K}}(\psi)$, as desired. ■

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

THEOREM 6.5 (SMALL MODEL THEOREM): Let φ be a satisfiable formula of PDL. Then φ is satisfied in a Kripke frame with no more than $2^{|\varphi|}$ states.

Proof If φ is satisfiable, then there is a Kripke frame $\mathfrak{K}$ and state $u \in \mathfrak{K}$ with

$u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)$. Let $FL(\varphi)$ be the Fischer-Ladner closure of φ. By the filtration lemma (Lemma 6.4), $[u] \in \mathfrak{m}_{\mathfrak{K}/FL(\varphi)}(\varphi)$. Moreover, $\mathfrak{K}/FL(\varphi)$ has no more states than the number of truth assignments to formulas in $FL(\varphi)$, which by Lemma 6.3(i) is at most $2^{|\varphi|}$. ■

It follows immediately that the satisfiability problem for PDL is decidable, since there are only finitely many possible Kripke frames of size at most $2^{|\varphi|}$ to check, and there is a polynomial-time algorithm to check whether a given formula is satisfied at a given state in a given Kripke frame (Exercise 6.4). We will give a more efficient algorithm in Section 8.1.

6.3 Filtration over Nonstandard Models

In Chapter 7 we will prove the completeness of a deductive system for PDL. The proof will also make use of the filtration lemma (Lemma 6.4), but in a somewhat stronger form. We will show that it also holds for *nonstandard Kripke frames* (to be defined directly) as well as the standard Kripke frames defined in Section 5.2. The completeness theorem will be obtained by constructing a nonstandard Kripke frame from terms, as we did for propositional and first-order logic in Sections 3.2 and 3.4, and then applying the filtration technique to get a finite standard Kripke frame.

A *nonstandard Kripke frame* is any structure $\mathfrak{N} = (N, \mathfrak{m}_{\mathfrak{N}})$ that is a Kripke frame in the sense of Section 5.2 in every respect, except that $\mathfrak{m}_{\mathfrak{N}}(\alpha^*)$ need not be the reflexive transitive closure of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$, but only a reflexive, transitive binary relation containing $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ satisfying the PDL axioms for * (Axioms 5.5(vii) and (viii)). In other words, we rescind the definition

$$\mathfrak{m}_{\mathfrak{N}}(\alpha^*) \stackrel{\text{def}}{=} \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{N}}(\alpha)^n, \tag{6.3.1}$$

and replace it with the weaker requirement that $\mathfrak{m}_{\mathfrak{N}}(\alpha^*)$ be a reflexive, transitive binary relation containing $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ such that

$$\mathfrak{m}_{\mathfrak{N}}(\texttt{[}\alpha^*\texttt{]}\varphi) = \mathfrak{m}_{\mathfrak{N}}(\varphi \wedge \texttt{[}\alpha\,;\,\alpha^*\texttt{]}\varphi) \tag{6.3.2}$$

$$\mathfrak{m}_{\mathfrak{N}}(\texttt{[}\alpha^*\texttt{]}\varphi) = \mathfrak{m}_{\mathfrak{N}}(\varphi \wedge \texttt{[}\alpha^*\texttt{]}(\varphi \rightarrow \texttt{[}\alpha\texttt{]}\varphi)). \tag{6.3.3}$$

Otherwise, $\mathfrak{N}$ must satisfy all other requirements as given in Section 5.2. For

example, it must still satisfy the properties

$$\begin{aligned} \mathfrak{m}_{\mathfrak{N}}(\alpha\,;\,\beta) &= \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\beta) \\ \mathfrak{m}_{\mathfrak{N}}(\alpha^*) &\supseteq \bigcup_{n\geq 0} \mathfrak{m}_{\mathfrak{N}}(\alpha)^n. \end{aligned}$$

A nonstandard Kripke frame *standard* if it satisfies (6.3.1). According to our definition, all standard Kripke frames are nonstandard Kripke frames, since standard Kripke frames satisfy (6.3.2) and (6.3.3), but not necessarily vice-versa (Exercise 7.3).

It is easily checked that all the axioms and rules of PDL (Axiom System 5.5) are still sound over nonstandard Kripke frames. One consequence of this is that all theorems and rules derived in this system are valid for nonstandard frames as well as standard ones. In particular, we will use the results of Theorem 5.18 in the proof of Lemma 6.6 below.

Let $\mathfrak{N}$ be a nonstandard Kripke frame and let φ be a proposition. We can construct the finite standard Kripke frame $\mathfrak{N}/FL(\varphi)$ exactly as before, and $\mathfrak{N}/FL(\varphi)$ will have at most $2^{|\varphi|}$ states. Note that in $\mathfrak{N}/FL(\varphi)$, the semantics of α^* is defined in the standard way using (6.3.1).

The filtration lemma (Lemma 6.4) holds for nonstandard Kripke frames as well as standard ones:

Lemma 6.6 (Filtration for Nonstandard Models): Let $\mathfrak{N}$ be a nonstandard Kripke frame and let u, v be states of $\mathfrak{N}$.

(i) For all $\psi \in FL(\varphi)$, $u \in \mathfrak{m}_{\mathfrak{N}}(\psi)$ iff $[u] \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\psi)$.

(ii) For all $\texttt{[}\alpha\texttt{]}\psi \in FL(\varphi)$,

(a)if $(u, v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha)$ then $([u], [v]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha)$;

(b)if $([u], [v]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha)$ and $u \in \mathfrak{m}_{\mathfrak{N}}(\texttt{[}\alpha\texttt{]}\psi)$, then $v \in \mathfrak{m}_{\mathfrak{N}}(\psi)$.

Proof The argument is exactly the same as in the previous version for standard frames (Lemma 6.4) except for the cases involving *. Also, part (b) of (ii) for the case $\alpha = \beta^*$ uses only the fact that $\mathfrak{N}/FL(\varphi)$ is standard, not that $\mathfrak{N}$ is standard, so this argument will hold for the nonstandard case as well. Thus the only extra work we need to do for the nonstandard version is part (a) of (ii) for the case $\alpha = \beta^*$.

The proof for standard Kripke frames $\mathfrak{K}$ given in Lemma 6.4 depended on the fact that $\mathfrak{m}_{\mathfrak{K}}(\alpha^*)$ was the reflexive transitive closure of $\mathfrak{m}_{\mathfrak{K}}(\alpha)$. This does not hold in nonstandard Kripke frames in general, so we must depend on the weaker induction axiom.

For the nonstandard Kripke frame $\mathfrak{N}$, suppose $(u, v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha^*)$. We wish to show that $([u], [v]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha^*)$, or equivalently that $v \in E$, where

$$E \stackrel{\text{def}}{=} \{t \in \mathfrak{N} \mid ([u], [t]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha^*)\}.$$

There is a PDL formula ψ_E defining E in $\mathfrak{N}$; that is, $E = \mathfrak{m}_{\mathfrak{N}}(\psi_E)$. This is because E is a union of equivalence classes defined by truth assignments to the elements of $FL(\varphi)$. The formula ψ_E is a disjunction of conjunctive formulas $\psi_{[t]}$, one for each equivalence class $[t]$ contained in E. The conjunction $\psi_{[t]}$ includes either ρ or $\neg\rho$ for all $\rho \in FL(\varphi)$, depending on whether the truth assignment defining $[t]$ takes value **1** or **0** on ρ, respectively.

Now $u \in E$ since $([u], [u]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha^*)$. Also, E is closed under the action of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$; that is,

$$s \in E \text{ and } (s, t) \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \implies t \in E. \tag{6.3.4}$$

To see this, observe that if $s \in E$ and $(s, t) \in \mathfrak{m}_{\mathfrak{N}}(\alpha)$, then $([s], [t]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha)$ by the induction hypothesis (ii), and $([u], [s]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha^*)$ by the definition of E, therefore $([u], [t]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha^*)$. By the definition of E, $t \in E$.

These facts do not immediately imply that $v \in E$, since $\mathfrak{m}_{\mathfrak{N}}(\alpha^*)$ is not necessarily the reflexive transitive closure of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$. However, since $E = \mathfrak{m}_{\mathfrak{N}}(\psi_E)$, (6.3.4) is equivalent to

$$\mathfrak{N} \vDash \psi_E \to [\alpha]\psi_E.$$

Using the loop invariance rule (LI) of Section 5.6, we get

$$\mathfrak{N} \vDash \psi_E \to [\alpha^*]\psi_E.$$

By Theorem 5.18, (LI) is equivalent to the induction axiom (IND). (The proof of equivalence was obtained deductively, not semantically, therefore is valid for nonstandard models.) Now $(u, v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha^*)$ by assumption, and $u \in E$, therefore $v \in E$. By definition of E, $([u], [v]) \in \mathfrak{m}_{\mathfrak{N}/FL(\varphi)}(\alpha^*)$. ■

6.4 Bibliographical Notes

The filtration argument and the small model property for PDL are due to Fischer and Ladner (1977, 1979). Nonstandard Kripke frames for PDL were studied by Berman (1979, 1982), Parikh (1978a), Pratt (1979a, 1980a), and Kozen (1979c,b, 1980a,b, 1981b).

Exercises

6.1. Complete the proof of Lemma 6.1. For part (i), fill in the argument for the cases of an atomic proposition p and the constant proposition $\mathbf{0}$. For (ii), fill in the argument for the cases of an atomic program a and compound programs of the form $\beta \cup \gamma$ and $\varphi?$.

6.2. Prove Lemma 6.2.

6.3. Complete the proof of Lemma 6.4 by filling in the arguments for part (i), case 3 and part (ii), case 3.

6.4. Give a polynomial time algorithm to check whether a given PDL formula is satisfied at a given state in a given Kripke frame. Describe briefly the data structures you would use to represent the formula and the Kripke frame. Specify your algorithm at a high level and give a brief complexity analysis.

6.5. Prove that all finite nonstandard Kripke frames are standard.

7 Deductive Completeness

In Section 5.5 we gave a formal deductive system (Axiom System 5.5) for deducing properties of Kripke frames expressible in the language of PDL. For convenience, we collect the axioms and rules of inference here. To the right of each axiom or rule appears a reference to the proof of its soundness.

Axioms of PDL

(i) Axioms for propositional logic	Section 3.2
(ii) $[\alpha](\varphi \to \psi) \quad \to \quad ([\alpha]\varphi \to [\alpha]\psi)$	Theorem 5.6(iv)
(iii) $[\alpha](\varphi \wedge \psi) \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\alpha]\psi$	Theorem 5.6(ii)
(iv) $[\alpha \cup \beta]\varphi \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\beta]\varphi$	Theorem 5.8(ii)
(v) $[\alpha\,;\,\beta]\varphi \quad \leftrightarrow \quad [\alpha][\beta]\varphi$	Theorem 5.10(ii)
(vi) $[\psi?]\varphi \quad \leftrightarrow \quad (\psi \to \varphi)$	Theorem 5.11(ii)
(vii) $\varphi \wedge [\alpha][\alpha^*]\varphi \quad \leftrightarrow \quad [\alpha^*]\varphi$	Theorem 5.15(ix)
(viii) $\varphi \wedge [\alpha^*](\varphi \to [\alpha]\varphi) \quad \to \quad [\alpha^*]\varphi$	Theorem 5.15(xi)

In PDL with converse $^-$, we also include

(ix) $\varphi \quad \to \quad [\alpha]\langle\alpha^-\rangle\varphi$	Theorem 5.13(i)
(x) $\varphi \quad \to \quad [\alpha^-]\langle\alpha\rangle\varphi$	Theorem 5.13(ii)

Rules of Inference

(MP) $\dfrac{\varphi, \quad \varphi \to \psi}{\psi}$	Section 3.2
(GEN) $\dfrac{\varphi}{[\alpha]\varphi}$	Theorem 5.7(i).

We write $\vdash \varphi$ if the formula φ is provable in this deductive system. Recall from Section 3.1 that a formula φ is *consistent* if $\nvdash \neg\varphi$, that is, if it is not the case that $\vdash \neg\varphi$; that a finite set Σ of formulas is *consistent* if its conjunction $\bigwedge\Sigma$ is consistent; and that an infinite set of formulas is *consistent* if every finite subset is consistent.

7.1 Deductive Completeness

This deductive system is complete: all valid formulas are theorems. To prove this fact, we will use techniques from Section 3.2 to construct a nonstandard Kripke

frame from maximal consistent sets of formulas. Then we will use the filtration lemma for nonstandard models (Lemma 6.6) to collapse this nonstandard model to a finite standard model.

Since our deductive system contains propositional logic as a subsystem, the following lemma holds. The proof is similar to the proof of Lemma 3.10 for propositional logic.

LEMMA 7.1: Let Σ be a set of formulas of PDL. Then

(i) Σ is consistent iff either $\Sigma \cup \{\varphi\}$ is consistent or $\Sigma \cup \{\neg\varphi\}$ is consistent;
(ii) if Σ is consistent, then Σ is contained in a maximal consistent set.

In addition, if Σ is a maximal consistent set of formulas, then

(iii) Σ contains all theorems of PDL;
(iv) if $\varphi \in \Sigma$ and $\varphi \to \psi \in \Sigma$, then $\psi \in \Sigma$;
(v) $\varphi \vee \psi \in \Sigma$ iff $\varphi \in \Sigma$ or $\psi \in \Sigma$;
(vi) $\varphi \wedge \psi \in \Sigma$ iff $\varphi \in \Sigma$ and $\psi \in \Sigma$;
(vii) $\varphi \in \Sigma$ iff $\neg\varphi \notin \Sigma$;
(viii) $\mathbf{0} \notin \Sigma$.

Proof Exercise 7.1. ■

We also have an interesting lemma peculiar to PDL.

LEMMA 7.2: Let Σ and Γ be maximal consistent sets of formulas and let α be a program. The following two statements are equivalent:

(a) For all formulas ψ, if $\psi \in \Gamma$, then $\langle\alpha\rangle\psi \in \Sigma$.
(b) For all formulas ψ, if $[\alpha]\psi \in \Sigma$, then $\psi \in \Gamma$.

Proof (a) $\Longrightarrow$ (b):

$$
\begin{array}{lcll}
[\alpha]\psi \in \Sigma & \Longrightarrow & \langle\alpha\rangle\neg\psi \notin \Sigma & \text{by Lemma 7.1(vii)} \\
& \Longrightarrow & \neg\psi \notin \Gamma & \text{by (a)} \\
& \Longrightarrow & \psi \in \Gamma & \text{by Lemma 7.1(vii).}
\end{array}
$$

(b) $\Longrightarrow$ (a):

$$\begin{array}{lll} \psi \in \Gamma & \Longrightarrow & \neg\psi \notin \Gamma \qquad \text{by Lemma 7.1(vii)} \\ & \Longrightarrow & [\alpha]\neg\psi \notin \Sigma \qquad \text{by (b)} \\ & \Longrightarrow & \langle\alpha\rangle\psi \in \Sigma \qquad \text{by Lemma 7.1(vii).} \end{array}$$

■

Now we construct a nonstandard Kripke frame $\mathfrak{N} = (N,\ \mathfrak{m}_{\mathfrak{N}})$ as defined in Section 6.3. The states N will be the maximal consistent sets of formulas. We will write $s, t, u, \ldots$ for elements of N and call them *states*, but bear in mind that every $s \in N$ is a maximal consistent set of formulas, therefore it makes sense to write $\varphi \in s$.

Formally, let $\mathfrak{N} = (N,\ \mathfrak{m}_{\mathfrak{N}})$ be defined by:

$$\begin{array}{rcl} N & \stackrel{\text{def}}{=} & \{\text{maximal consistent sets of formulas of PDL}\} \\ \mathfrak{m}_{\mathfrak{N}}(\varphi) & \stackrel{\text{def}}{=} & \{s \mid \varphi \in s\} \\ \mathfrak{m}_{\mathfrak{N}}(\alpha) & \stackrel{\text{def}}{=} & \{(s,t) \mid \text{for all } \varphi, \text{ if } \varphi \in t, \text{ then } \langle\alpha\rangle\varphi \in s\} \\ & = & \{(s,t) \mid \text{for all } \varphi, \text{ if } [\alpha]\varphi \in s, \text{ then } \varphi \in t\}. \end{array}$$

The two definitions of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ are equivalent by Lemma 7.2. Note that the definitions of $\mathfrak{m}_{\mathfrak{N}}(\varphi)$ and $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ apply to *all* propositions φ and programs α, not just the atomic ones; thus the meaning of compound propositions and programs is not defined inductively from the meaning of the atomic ones as usual. However, $\mathfrak{m}_{\mathfrak{N}}(\alpha^*)$ will satisfy the axioms of *, and all the other operators will behave in $\mathfrak{N}$ as they do in standard models; that is, as if they were defined inductively. We will have to prove this in order to establish that $\mathfrak{N}$ is a nonstandard Kripke frame according to the definition of Section 6.3. We undertake that task now.

Lemma 7.3:

$$\begin{array}{lll} \text{(i)} & \mathfrak{m}_{\mathfrak{N}}(\varphi \to \psi) & = \quad (N - \mathfrak{m}_{\mathfrak{N}}(\varphi)) \cup \mathfrak{m}_{\mathfrak{N}}(\psi) \\ \text{(ii)} & \mathfrak{m}_{\mathfrak{N}}(\mathbf{0}) & = \quad \varnothing \\ \text{(iii)} & \mathfrak{m}_{\mathfrak{N}}([\alpha]\varphi) & = \quad N - \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ (N - \mathfrak{m}_{\mathfrak{N}}(\varphi)). \end{array}$$

Proof The equations (i) and (ii) follow from Lemma 7.1(iv) and (viii), respectively. It follows that $\mathfrak{m}_{\mathfrak{N}}(\neg\varphi) = N - \mathfrak{m}_{\mathfrak{N}}(\varphi)$; this is also a consequence of Lemma 7.1(vii).

For (iii), it suffices to show that

$$\mathfrak{m}_{\mathfrak{N}}(\langle\alpha\rangle\varphi) \quad = \quad \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\varphi).$$

We prove both inclusions separately.

$$\begin{aligned} s \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\varphi) \quad &\iff \quad \exists t\ (s,t) \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \text{ and } t \in \mathfrak{m}_{\mathfrak{N}}(\varphi) \\ &\iff \quad \exists t\ (\forall \psi \in t\ \langle\alpha\rangle\psi \in s) \text{ and } \varphi \in t \\ &\implies \quad \langle\alpha\rangle\varphi \in s \\ &\iff \quad s \in \mathfrak{m}_{\mathfrak{N}}(\langle\alpha\rangle\varphi). \end{aligned}$$

Conversely, suppose $s \in \mathfrak{m}_{\mathfrak{N}}(\langle\alpha\rangle\varphi)$; that is, $\langle\alpha\rangle\varphi \in s$. We would like to construct t such that $(s,t) \in \mathfrak{m}_{\mathfrak{N}}(\alpha)$ and $t \in \mathfrak{m}_{\mathfrak{N}}(\varphi)$. We first show that the set

$$\{\varphi\} \ \cup \ \{\psi \mid [\alpha]\psi \in s\} \tag{7.1.1}$$

is consistent. Let $\{\psi_1, \ldots, \psi_k\}$ be an arbitrary finite subset of $\{\psi \mid [\alpha]\psi \in s\}$. Then

$$\langle\alpha\rangle\varphi \wedge [\alpha]\psi_1 \wedge \cdots \wedge [\alpha]\psi_k \quad \in \quad s$$

by Lemma 7.1(vi), therefore

$$\langle\alpha\rangle(\varphi \wedge \psi_1 \wedge \cdots \wedge \psi_k) \quad \in \quad s$$

by Exercise 5.9(i) and Lemma 7.1(iii) and (iv). Since s is consistent, the formula

$$\langle\alpha\rangle(\varphi \wedge \psi_1 \wedge \cdots \wedge \psi_k)$$

is consistent, therefore so is the formula

$$\varphi \wedge \psi_1 \wedge \cdots \wedge \psi_k$$

by the rule (GEN). This says that the finite set $\{\varphi, \psi_1, \ldots, \psi_k\}$ is consistent. Since these elements were chosen arbitrarily from the set (7.1.1), that set is consistent.

As in the proof of Lemma 3.10, (7.1.1) extends to a maximal consistent set t, which is a state of $\mathfrak{N}$. Then $(s,t) \in \mathfrak{m}_{\mathfrak{N}}(\alpha)$ and $t \in \mathfrak{m}_{\mathfrak{N}}(\varphi)$ by the definition of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ and $\mathfrak{m}_{\mathfrak{N}}(\varphi)$, therefore $s \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\varphi)$. ■

Lemma 7.4:

(i) $\mathfrak{m}_{\mathfrak{N}}(\alpha \cup \beta) \quad = \quad \mathfrak{m}_{\mathfrak{N}}(\alpha) \cup \mathfrak{m}_{\mathfrak{N}}(\beta)$
(ii) $\mathfrak{m}_{\mathfrak{N}}(\alpha\,;\,\beta) \quad = \quad \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\beta)$
(iii) $\mathfrak{m}_{\mathfrak{N}}(\psi?) \quad = \quad \{(s,s) \mid s \in \mathfrak{m}_{\mathfrak{N}}(\psi)\}.$

In PDL with converse $^-$,

(iv) $\mathfrak{m}_{\mathfrak{N}}(\alpha^-) \;=\; \mathfrak{m}_{\mathfrak{N}}(\alpha)^-$.

Proof We argue (ii) and (iii) explicitly and leave the others as exercises (Exercise 7.2).

For the reverse inclusion $\supseteq$ of (ii),

$$\begin{aligned}
(u,v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\beta) &\iff \exists w\ (u,w) \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{N}}(\beta) \\
&\iff \exists w\ \forall \varphi \in v\ \langle\beta\rangle\varphi \in w \text{ and } \forall \psi \in w\ \langle\alpha\rangle\psi \in u \\
&\implies \forall \varphi \in v\ \langle\alpha\rangle\langle\beta\rangle\varphi \in u \\
&\iff \forall \varphi \in v\ \langle\alpha\,;\,\beta\rangle\varphi \in u \\
&\iff (u,v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha\,;\,\beta).
\end{aligned}$$

For the forward inclusion, suppose $(u,v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha\,;\,\beta)$. We claim that the set

$$\{\varphi \mid [\alpha]\varphi \in u\} \quad \cup \quad \{\langle\beta\rangle\psi \mid \psi \in v\} \tag{7.1.2}$$

is consistent. Let

$$\begin{aligned}
\{\varphi_1, \ldots, \varphi_k\} &\subseteq \{\varphi \mid [\alpha]\varphi \in u\} \\
\{\langle\beta\rangle\psi_1, \ldots, \langle\beta\rangle\psi_m\} &\subseteq \{\langle\beta\rangle\psi \mid \psi \in v\}
\end{aligned}$$

be arbitrarily chosen finite subsets, and let

$$\begin{aligned}
\varphi &= \varphi_1 \wedge \cdots \wedge \varphi_k, \\
\psi &= \psi_1 \wedge \cdots \wedge \psi_m.
\end{aligned}$$

Then $\psi \in v$ by Lemma 7.1(vi), and since $(u,v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha\,;\,\beta)$, we have by the definition of $\mathfrak{m}_{\mathfrak{N}}(\alpha\,;\,\beta)$ that $\langle\alpha\,;\,\beta\rangle\psi \in u$. Also $[\alpha]\varphi \in u$, since

$$[\alpha]\varphi \quad\leftrightarrow\quad [\alpha]\varphi_1 \wedge \cdots \wedge [\alpha]\varphi_k$$

is a theorem of PDL, and the right-hand side is in u by Lemma 7.1(vi). It follows that $[\alpha]\varphi \wedge \langle\alpha\rangle\langle\beta\rangle\psi \in u$. By Exercise 5.9(i), $\langle\alpha\rangle(\varphi \wedge \langle\beta\rangle\psi) \in u$, thus by (GEN), $\varphi \wedge \langle\beta\rangle\psi$ is consistent. But

$$\vdash\ \varphi \wedge \langle\beta\rangle\psi \quad\to\quad \varphi_1 \wedge \cdots \wedge \varphi_k \wedge \langle\beta\rangle\psi_1 \wedge \cdots \wedge \langle\beta\rangle\psi_m,$$

so the right-hand side of the implication is consistent. As this was the conjunction of an arbitrary finite subset of (7.1.2), (7.1.2) is consistent, thus extends to a maximal consistent set w. By the definition of $\mathfrak{m}_{\mathfrak{N}}(\alpha)$ and $\mathfrak{m}_{\mathfrak{N}}(\beta)$, $(u,w) \in \mathfrak{m}_{\mathfrak{N}}(\alpha)$ and

$(w,v) \in \mathfrak{m}_{\mathfrak{N}}(\beta)$, therefore $(u,v) \in \mathfrak{m}_{\mathfrak{N}}(\alpha) \circ \mathfrak{m}_{\mathfrak{N}}(\beta)$.

For (iii),

$$\begin{array}{lll}
(s,t) \in \mathfrak{m}_{\mathfrak{N}}(\psi?) & \iff \forall \varphi \in t \ \langle \psi? \rangle \varphi \in s & \text{definition of } \mathfrak{m}_{\mathfrak{N}}(\psi?) \\
& \iff \forall \varphi \in t \ \psi \wedge \varphi \in s & \text{Exercise 5.9(v)} \\
& \iff \forall \varphi \in t \ \psi \in s \text{ and } \varphi \in s & \text{Lemma 7.1(vi)} \\
& \iff t \subseteq s \text{ and } \psi \in s & \\
& \iff t = s \text{ and } \psi \in s & \text{since } t \text{ is maximal} \\
& \iff t = s \text{ and } s \in \mathfrak{m}_{\mathfrak{N}}(\psi). &
\end{array}$$

■

THEOREM 7.5: The structure $\mathfrak{N}$ is a nonstandard Kripke frame according to the definition of Section 6.3.

Proof By Lemmas 7.3 and 7.4, the operators $\rightarrow$, $\mathbf{0}$, $[\,]$, ;, $\cup$, $^-$, and ? behave in $\mathfrak{N}$ as in standard models. It remains to argue that the properties

$$\begin{array}{lll}
[\alpha^*]\varphi & \leftrightarrow & \varphi \wedge [\alpha \,;\, \alpha^*]\varphi \\
[\alpha^*]\varphi & \leftrightarrow & \varphi \wedge [\alpha^*](\varphi \rightarrow [\alpha]\varphi)
\end{array}$$

of the * operator hold at all states. But this is immediate, since both these properties are theorems of PDL (Exercise 5.9), thus by Lemma 7.1(iii) they must be in every maximal consistent set. This guarantees that $\mathfrak{N}$ satisfies conditions (6.3.2) and (6.3.3) in the definition of nonstandard Kripke frames. ■

The definition of the nonstandard Kripke frame $\mathfrak{N}$ is independent of any particular φ. It is a *universal model* in the sense that every consistent formula is satisfied at some state of $\mathfrak{N}$.

THEOREM 7.6 (COMPLETENESS OF PDL): If $\vDash \varphi$ then $\vdash \varphi$.

Proof Equivalently, we need to show that if φ is consistent, then it is satisfied in a standard Kripke frame. If φ is consistent, then by Lemma 7.1(ii), it is contained in a maximal consistent set u, which is a state of the nonstandard Kripke frame $\mathfrak{N}$ constructed above. By the filtration lemma for nonstandard models (Lemma 6.6), φ is satisfied at the state $[u]$ in the finite Kripke frame $\mathfrak{N}/FL(\varphi)$, which is a standard Kripke frame by definition. ■

7.2 Logical Consequences

In classical logics, a completeness theorem of the form of Theorem 7.6 can be adapted to handle the relation of logical consequence $\varphi \models \psi$ between formulas because of the deduction theorem, which says

$\varphi \vdash \psi \iff \vdash \varphi \to \psi$.

Unfortunately, the deduction theorem fails in PDL, as can be seen by taking $\psi = [a]p$ and $\varphi = p$. However, the following result allows Theorem 7.6, as well as Algorithm 8.2 of the next section, to be extended to handle the logical consequence relation:

THEOREM 7.7: Let φ and ψ be any PDL formulas. Then

$\varphi \models \psi \iff \models [(a_1 \cup \cdots \cup a_n)^*]\varphi \to \psi$,

where $a_1, \ldots, a_n$ are all atomic programs appearing in φ or ψ. Allowing infinitary conjunctions, if Σ is a set of formulas in which only finitely many atomic programs appear, then

$\Sigma \models \psi \iff \models \bigwedge\{[(a_1 \cup \cdots \cup a_n)^*]\varphi \mid \varphi \in \Sigma\} \to \psi$,

where $a_1, \ldots, a_n$ are all atomic programs appearing in Σ or ψ.

We leave the proof of Theorem 7.7 as an exercise (Exercise 7.4).

7.3 Bibliographical Notes

The axiomatization of PDL used here (Axiom System 5.5) was introduced by Segerberg (1977). Completeness was shown independently by Gabbay (1977) and Parikh (1978a). A short and easy-to-follow proof is given in Kozen and Parikh (1981). Completeness is also treated in Pratt (1978, 1980a); Berman (1979); Nishimura (1979). The completeness proof given here is from Kozen (1981a) and is based on the approach of Berman (1979); Pratt (1980a).

Exercises

7.1. Prove Lemma 7.1. (*Hint.* Study the proof of Lemma 3.10.)

7.2. Supply the missing proofs of parts (i) and (iv) of Lemma 7.4.

7.3. Prove that PDL is compact over nonstandard models; that is, every finitely satisfiable set of propositions is satisfiable in a nonstandard Kripke frame. Conclude that there exists a nonstandard Kripke frame that is not standard.

7.4. Prove Theorem 7.7.

8 Complexity of PDL

In this chapter we ask the question: how difficult is it to determine whether a given formula φ of PDL is satisfiable? This is known as the *satisfiability problem* for PDL.

8.1 A Deterministic Exponential-Time Algorithm

The small model theorem (Theorem 6.5) gives a naive deterministic algorithm for the satisfiability problem: construct all Kripke frames of at most $2^{|\varphi|}$ states and check whether φ is satisfied at any state in any of them. Although checking whether a given formula is satisfied in a given state of a given Kripke frame can be done quite efficiently (Exercise 6.4), the naive satisfiability algorithm is highly inefficient. For one thing, the models constructed are of exponential size in the length of the given formula; for another, there are $2^{2^{O(|\varphi|)}}$ of them. Thus the naive satisfiability algorithm takes double exponential time in the worst case.

Here we develop an algorithm that runs in deterministic single-exponential time. One cannot expect to get a much more efficient algorithm than this due to a corresponding lower bound (Corollary 8.6). In fact, the problem is deterministic exponential-time complete (Theorem 8.5).

The algorithm attempts to construct the small model

$$\mathfrak{M} \quad = \quad (M, \mathfrak{m}_{\mathfrak{M}}) \quad = \quad \mathfrak{N}/FL(\varphi)$$

described in the proof of Theorem 7.6 explicitly. Here $\mathfrak{N}$ is the universal nonstandard Kripke frame constructed in Section 7.1 and $\mathfrak{M}$ is the small model obtained by filtration with respect to φ. If φ is satisfiable, then it is consistent, by the soundness of Axiom System 5.5; then φ will be satisfied at some state u of $\mathfrak{N}$, hence also at the state $[u]$ of $\mathfrak{M}$. If φ is not satisfiable, then the attempt to construct a model will fail; in this case the algorithm will halt and report failure.

Our approach will be to start with a superset of the set of states of $\mathfrak{M}$, then repeatedly delete states when we discover some inconsistency. This will give a sequence of approximations

$$\mathfrak{M}_0 \quad \supseteq \quad \mathfrak{M}_1 \quad \supseteq \quad \mathfrak{M}_2 \quad \supseteq \quad \cdots$$

converging to $\mathfrak{M}$.

We start with the set M_0 of all subsets

$$u \quad \subseteq \quad FL(\varphi) \cup \{\neg\psi \mid \psi \in FL(\varphi)\}$$

such that for each $\psi \in FL(\varphi)$, exactly one of ψ or $\neg\psi$ is in u. (Alternatively, we could take M_0 to be the set of truth assignments to $FL(\varphi)$.) By Lemma 7.1(vii), each state s of $\mathfrak{N}$ determines a unique element of M_0, namely

$$u_s \stackrel{\text{def}}{=} s \cap (FL(\varphi) \cup \{\neg\psi \mid \psi \in FL(\varphi)\}).$$

Moreover, by the definition of the equivalence relation $\equiv$ of Section 6.2,

$$[s] = [t] \iff s \equiv t \iff u_s = u_t,$$

thus the map $s \mapsto u_s$ is well-defined on $\equiv$-equivalence classes and gives a one-to-one embedding $[s] \mapsto u_s : M \to M_0$. We henceforth identify the state $[s]$ of $\mathfrak{M}$ with its image u_s in M_0 under this embedding. This allows us to regard M as a subset of M_0. However, there are some elements of M_0 that do not correspond to any state of $\mathfrak{M}$, and these are the ones to be deleted.

Now we are left with the question: how do we distinguish the sets u_s from those not corresponding to any state of $\mathfrak{M}$? This question is answered in the following lemma.

LEMMA 8.1: Let $u \in M_0$. Then $u \in M$ if and only if u is consistent.

Proof By Lemma 6.6(i), every u_s is consistent, because it has a model: it is satisfied at the state $[s]$ of $\mathfrak{M}$.

Conversely, if $u \in M_0$ is consistent, then by Lemma 7.1(ii) it extends to a maximal consistent set $\widehat{u}$, which is a state of the nonstandard Kripke frame $\mathfrak{N}$; and by Lemma 6.6(i), $[\widehat{u}]$ is a state of $\mathfrak{M}$ satisfying u. ■

We now construct a sequence of structures $\mathfrak{M}_i = (M_i, \mathfrak{m}_{\mathfrak{M}_i})$, $i \geq 0$, approximating $\mathfrak{M}$. The domains M_i of these structures will be defined below and will satisfy

$$M_0 \supseteq M_1 \supseteq M_2 \supseteq \cdots$$

The interpretations of the atomic formulas and programs in $\mathfrak{M}_i$ will be defined in the same way for all i:

$$\mathfrak{m}_{\mathfrak{M}_i}(p) \stackrel{\text{def}}{=} \{u \in M_i \mid p \in u\} \tag{8.1.1}$$

$$\mathfrak{m}_{\mathfrak{M}_i}(a) \stackrel{\text{def}}{=} \{(u,v) \in M_i^2 \mid \text{for all } \texttt{[}a\texttt{]}\psi \in FL(\varphi), \text{ if } \texttt{[}a\texttt{]}\psi \in u, \text{ then } \psi \in v\} \tag{8.1.2}$$

The map $\mathfrak{m}_{\mathfrak{M}_i}$ extends inductively in the usual way to compound programs and propositions to determine the frame $\mathfrak{M}_i$.

Here is the algorithm for constructing the domains M_i of the frames $\mathfrak{M}_i$.

ALGORITHM 8.2:

Step 1 Construct M_0.

Step 2 For each $u \in M_0$, check whether u respects Axioms 5.5(i) and (iv)–(vii), all of which can be checked locally. For example, to check Axiom 5.5(iv), which says

$$[\alpha \cup \beta]\psi \quad \leftrightarrow \quad [\alpha]\psi \wedge [\beta]\psi,$$

check for any formula of the form $[\alpha \cup \beta]\psi \in FL(\varphi)$ that $[\alpha \cup \beta]\psi \in u$ if and only if both $[\alpha]\psi \in u$ and $[\beta]\psi \in u$. Let M_1 be the set of all $u \in M_0$ passing this test. The model $\mathfrak{M}_1$ is defined by (8.1.1) and (8.1.2) above.

Step 3 Repeat the following for $i = 1, 2, \ldots$ until no more states are deleted. Find a formula $[\alpha]\psi \in FL(\varphi)$ and a state $u \in M_i$ violating the property

$$(\forall v\ ((u, v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \Longrightarrow \psi \in v)) \quad \Longrightarrow \quad [\alpha]\psi \in u; \tag{8.1.3}$$

that is, such that $\neg[\alpha]\psi \in u$, but for no v such that $(u, v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha)$ is it the case that $\neg\psi \in v$. Pick such an $[\alpha]\psi$ and u for which $|\alpha|$ is minimum. Delete u from M_i to get M_{i+1}. ■

Step 3 can be justified intuitively as follows. To say that u violates the condition (8.1.3) says that u would like to go under α to some state satisfying $\neg\psi$, since u contains the formula $\neg[\alpha]\psi$, which is equivalent to $\langle\alpha\rangle\neg\psi$; but the left-hand side of (8.1.3) says that none of the states it currently goes to under α want to satisfy $\neg\psi$. This is evidence that u might not be in $\mathfrak{M}$, since in $\mathfrak{M}$ every state w satisfies every $\psi \in w$ by Lemma 6.6(i). But u may violate (8.1.3) not because $u \notin \mathfrak{M}$, but because there is some other state $v \notin \mathfrak{M}$ whose presence affects the truth of some subformula of $[\alpha]\psi$. This situation can be avoided by choosing $|\alpha|$ minimum.

The algorithm must terminate, since there are only finitely many states initially, and at least one state must be deleted in each iteration of step 3 in order to continue.

The correctness of this algorithm will follow from the following lemma. Note the similarity of this lemma to Lemmas 6.4 and 6.6.

LEMMA 8.3: Let $i \geq 0$, and assume that $M \subseteq M_i$. Let $\rho \in FL(\varphi)$ be such that every $[\alpha]\psi \in FL(\rho)$ and $u \in M_i$ satisfy (8.1.3).

(i) For all $\psi \in FL(\rho)$ and $u \in M_i$, $\psi \in u$ iff $u \in \mathfrak{m}_{\mathfrak{M}_i}(\psi)$.

(ii) For all $[\alpha]\psi \in FL(\rho)$ and $u, v \in M_i$,

(a)if $(u,v) \in \mathfrak{m}_{\mathfrak{M}}(\alpha)$, then $(u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha)$;

(b)if $(u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha)$ and $\texttt{[}\alpha\texttt{]}\psi \in u$, then $\psi \in v$.

Proof The proof is by simultaneous induction on the subterm relation.

(i) The basis for atomic p is by definition as given in (8.1.1). The induction steps for $\rightarrow$ and $\mathbf{0}$ are easy and are left as exercises (Exercise 8.1). For the case $\texttt{[}\alpha\texttt{]}\psi$,

$$\begin{array}{lll}
\texttt{[}\alpha\texttt{]}\psi \in u & & \\
\quad\Longrightarrow\quad \forall v\ (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \Longrightarrow \psi \in v & & \text{induction hypothesis (ii)(b)} \\
\quad\Longrightarrow\quad \forall v\ (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \Longrightarrow v \in \mathfrak{m}_{\mathfrak{M}_i}(\psi) & & \text{induction hypothesis (i)} \\
\quad\Longrightarrow\quad u \in \mathfrak{m}_{\mathfrak{M}_i}(\texttt{[}\alpha\texttt{]}\psi). & &
\end{array}$$

Conversely,

$$\begin{array}{lll}
u \in \mathfrak{m}_{\mathfrak{M}_i}(\texttt{[}\alpha\texttt{]}\psi) & & \\
\quad\Longrightarrow\quad \forall v\ (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \Longrightarrow v \in \mathfrak{m}_{\mathfrak{M}_i}(\psi) & & \\
\quad\Longrightarrow\quad \forall v\ (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \Longrightarrow \psi \in v & & \text{induction hypothesis (i)} \\
\quad\Longrightarrow\quad \texttt{[}\alpha\texttt{]}\psi \in u & & \text{by (8.1.3).}
\end{array}$$

(ii)(a) For the basis, let a be an atomic program.

$$\begin{array}{lll}
(u,v) \in \mathfrak{m}_{\mathfrak{M}}(a) & \Longrightarrow & \forall\psi\ (\texttt{[}a\texttt{]}\psi \in FL(\varphi) \text{ and } \texttt{[}a\texttt{]}\psi \in u) \Longrightarrow \psi \in v \\
& \Longrightarrow & (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(a) \quad \text{by (8.1.2).}
\end{array}$$

The case $\alpha \cup \beta$ is left as an exercise (Exercise 8.1).

For the case $\alpha\,;\,\beta$,

$$\begin{array}{lll}
(u,v) \in \mathfrak{m}_{\mathfrak{M}}(\alpha\,;\,\beta) & \Longleftrightarrow & \exists w \in M\ (u,w) \in \mathfrak{m}_{\mathfrak{M}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{M}}(\beta) \\
& \Longrightarrow & \exists w \in M_i\ (u,w) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\beta) \\
& \Longleftrightarrow & (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha\,;\,\beta).
\end{array}$$

The second step uses the induction hypothesis and the fact that $M \subseteq M_i$. The induction hypothesis holds for α and β because $\texttt{[}\alpha\texttt{]}\texttt{[}\beta\texttt{]}\psi \in FL(\rho)$ and $\texttt{[}\beta\texttt{]}\psi \in FL(\rho)$ by Lemma 6.2(iv).

The case α^* follows from this case by iteration, and is left as an exercise (Exercise 8.1).

For the case $\psi?$,

$$\begin{array}{lll}
(u,v) \in \mathfrak{m}_{\mathfrak{M}}(\psi?) & \iff & u = v \text{ and } u \in \mathfrak{m}_{\mathfrak{M}}(\psi) \\
& \implies & u = v \text{ and } \psi \in u \qquad \text{Lemma 6.6(i)} \\
& \iff & u = v \text{ and } u \in \mathfrak{m}_{\mathfrak{M}_i}(\psi) \qquad \text{induction hypothesis (i)} \\
& \iff & (u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\psi?).
\end{array}$$

(ii)(b) For the basis, let a be an atomic program. Then

$$(u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(a) \text{ and } \texttt{[}a\texttt{]}\psi \in u \implies \psi \in v \quad \text{by (8.1.2).}$$

The cases $\alpha \cup \beta$ and $\alpha\,;\,\beta$ are left as exercises (Exercise 8.1).

For the case α^*, suppose $(u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha^*)$ and $\texttt{[}\alpha^*\texttt{]}\psi \in u$. Then there exist $u_0, \ldots, u_n$, $n \geq 0$, such that $u = u_0$, $v = u_n$, and $(u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha)$, $0 \leq i \leq n-1$. Also $\texttt{[}\alpha\texttt{]}\texttt{[}\alpha^*\texttt{]}\psi \in u_0$, otherwise u_0 would have been deleted in step 2. By the induction hypothesis (ii)(b), $\texttt{[}\alpha^*\texttt{]}\psi \in u_1$. Continuing in this fashion, we can conclude after n steps of this argument that $\texttt{[}\alpha^*\texttt{]}\psi \in u_n = v$. Then $\psi \in v$, otherwise v would have been deleted in step 2.

Finally, for the case $\psi?$, if $(u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\psi?)$ and $\texttt{[}\psi?\texttt{]}\sigma \in u$, then $u = v$ and $u \in \mathfrak{m}_{\mathfrak{M}_i}(\psi)$. By the induction hypothesis (i), $\psi \in u$. Thus $\sigma \in u$, otherwise u would have been deleted in step 2. ■

Note that Lemma 8.3(ii)(a) actually holds of $\texttt{[}\alpha\texttt{]}\psi \in FL(\varphi)$ even if there exists $u \in M_i$ violating (8.1.3), provided $|\alpha|$ is minimum. This is because the condition regarding (8.1.3) in the statement of the lemma holds on strict subformulas of α, and that is all that is needed in the inductive proof of (ii)(a) for α. This says that no $u \in M$ is ever deleted in step 3, since for $u \in M$,

$$\begin{array}{lll}
\forall v \in M_i\ ((u,v) \in \mathfrak{m}_{\mathfrak{M}_i}(\alpha) \implies \psi \in v) & & \\
\implies & \forall v \in M\ ((u,v) \in \mathfrak{m}_{\mathfrak{M}}(\alpha) \implies \psi \in v) & \text{Lemma 8.3(ii)(a)} \\
\iff & \forall v \in M\ ((u,v) \in \mathfrak{m}_{\mathfrak{M}}(\alpha) \implies v \in \mathfrak{m}_{\mathfrak{M}}(\psi)) & \text{Lemma 6.6(i)} \\
\iff & u \in \mathfrak{m}_{\mathfrak{M}}(\texttt{[}\alpha\texttt{]}\psi) & \text{definition of } \mathfrak{m}_{\mathfrak{M}} \\
\implies & \texttt{[}\alpha\texttt{]}\psi \in u & \text{Lemma 6.6(i).}
\end{array}$$

Since every $u \in M$ passes the test of step 2 of the algorithm, and since no $u \in M$ is ever deleted in step 3, we have $M \subseteq M_i$ for all $i \geq 0$. Moreover, when the algorithm terminates with some model $\mathfrak{M}_n$, by Lemma 8.3(i), every $u \in M_n$ is satisfiable, since it is satisfied by the state u in the model $\mathfrak{M}_n$; thus $\mathfrak{M}_n = \mathfrak{M}$. We can now test the satisfiability of φ by checking whether $\varphi \in u$ for some $u \in M_n$.

Algorithm 8.2 can be programmed to run in exponential time without much difficulty. The efficiency can be further improved by observing that the minimal α in the $[\alpha]\psi$ violating (8.1.3) in step 3 must be either atomic or of the form β^* because of the preprocessing in step 2. This follows easily from Lemma 8.3. We have shown:

THEOREM 8.4: There is an exponential-time algorithm for deciding whether a given formula of PDL is satisfiable.

As previously noted, Theorem 7.7 allows this algorithm to be adapted to test whether one formula is a logical consequence of another.

8.2 A Lower Bound

In the previous section we gave an exponential-time algorithm for deciding satisfiability in PDL. Here we establish the corresponding lower bound.

THEOREM 8.5: The satisfiability problem for PDL is *EXPTIME*-complete.

Proof In light of Theorem 8.4, we need only show that PDL is *EXPTIME*-hard (see Section 2.3). We do this by constructing a formula of PDL whose models encode the computation of a given linear-space-bounded one-tape alternating Turing machine M on a given input x of length n over M's input alphabet. We show how to define a formula $\text{ACCEPTS}_{M,x}$ involving the single atomic program NEXT, atomic propositions SYMBOL_i^a and STATE_i^q for each symbol a in M's tape alphabet, q a state of M's finite control, and $0 \leq i \leq n$, and an atomic proposition ACCEPT. The formula $\text{ACCEPTS}_{M,x}$ will have the property that any satisfying Kripke frame encodes an accepting computation of M on x. In any such Kripke frame, states u will represent configurations of M occurring in the computation tree of M on input x; the truth values of SYMBOL_i^a and STATE_i^q at state u will give the tape contents, current state, and tape head position in the configuration corresponding to u. The truth value of the atomic proposition ACCEPT will be **1** at u iff the computation beginning in state u is an accepting computation according to the rules of alternating Turing machine acceptance (Section 2.1).

Let Γ be M's tape alphabet and Q the set of states. We assume without loss of generality that the machine is $2^{O(n)}$-time bounded. This can be enforced by requiring M to count each step it takes on a separate track and to shut off after

c^n steps, where c^n bounds the number of possible configurations of M on inputs of length n. There are at most $\Gamma^{n+2} \cdot Q \cdot (n+2)$ such configurations, and c can be chosen large enough that c^n bounds this number.

We also assume without loss of generality that the input is enclosed in left and right endmarkers $\vdash$ and $\dashv$, respectively, that these symbols are never overwritten, and that M is constrained never to move to the left of $\vdash$ nor to the right of $\dashv$.

Now we encode configurations as follows. The atomic proposition $\textsc{Symbol}_i^a$ says, "Tape cell i currently has symbol a written on it." The atomic proposition $\textsc{State}_i^q$ says, "The tape head is currently scanning tape cell i in state q." We also allow $\textsc{State}_i^\ell$ and $\textsc{State}_i^r$, where $\ell, r \notin Q$ are special annotations used to indicate that the tape head is currently scanning a cell somewhere to the left or right, respectively, of cell i.

- "Exactly one symbol occupies every tape cell."

$$\bigwedge_{0 \le i \le n+1} \bigvee_{a \in \Gamma} (\textsc{Symbol}_i^a \wedge \bigwedge_{\substack{b \in \Gamma \\ b \ne a}} \neg \textsc{Symbol}_i^b)$$

- "The symbols occupying the first and last tape cells are the endmarkers $\vdash$ and $\dashv$, respectively."

$$\textsc{Symbol}_0^\vdash \wedge \textsc{Symbol}_{n+1}^\dashv$$

- "The machine is in exactly one state scanning exactly one tape cell."

$$\bigvee_{0 \le i \le n+1} \bigvee_{q \in Q} \textsc{State}_i^q$$
$$\wedge \bigwedge_{0 \le i \le n+1} \bigvee_{q \in Q \cup \{\ell, r\}} (\textsc{State}_i^q \wedge \bigwedge_{\substack{p \in Q \cup \{\ell, r\} \\ p \ne q}} \neg \textsc{State}_i^p)$$
$$\wedge \bigwedge_{0 \le i \le n} \bigwedge_{q \in Q \cup \{\ell\}} (\textsc{State}_i^q \to \textsc{State}_{i+1}^\ell)$$
$$\wedge \bigwedge_{1 \le i \le n+1} \bigwedge_{q \in Q \cup \{r\}} (\textsc{State}_i^q \to \textsc{State}_{i-1}^r).$$

Let Config be the conjunction of these three formulas. Then $u \vDash \textsc{Config}$ iff u represents a configuration of M on an input of length n.

Now we can write down formulas that say that M moves correctly. Here we use the atomic program Next to represent the binary next-configuration relation. For each (state, tape symbol) pair (q, a), let $\Delta(q, a)$ be the set of all (state, tape symbol, direction) triples describing a possible action M can take when scanning symbol

a in state q. For example, if $(p, b, -1) \in \Delta(q, a)$, this means that when scanning a tape cell containing a in state q, M can print b on that tape cell, move its head one cell to the left, and enter state p.

• "If the tape head is not currently scanning cell i, then the symbol written on cell i does not change."

$$\bigwedge_{0\le i\le n+1} ((\textsc{State}_i^{\ell} \vee \textsc{State}_i^{r}) \to \bigwedge_{a\in\Gamma} (\textsc{Symbol}_i^{a} \to \texttt{[}\textsc{Next}\texttt{]}\textsc{Symbol}_i^{a}))$$

• "The machine moves according to its transition relation."

$$\bigwedge_{0\le i\le n+1} \bigwedge_{\substack{a\in\Gamma\\ q\in Q}} ((\textsc{Symbol}_i^{a} \wedge \textsc{State}_i^{q}) \to$$

$$(\bigwedge_{(p,b,d)\in\Delta(q,a)} \texttt{<}\textsc{Next}\texttt{>}(\textsc{Symbol}_i^{b} \wedge \textsc{State}_{i+d}^{p})) \tag{8.2.1}$$

$$\wedge \quad \texttt{[}\textsc{Next}\texttt{]}(\bigvee_{(p,b,d)\in\Delta(q,a)} (\textsc{Symbol}_i^{b} \wedge \textsc{State}_{i+d}^{p}))) \tag{8.2.2}$$

Note that when $\Delta(q, a) = \varnothing$, clause (8.2.1) reduces to $\mathbf{1}$ and clause (8.2.2) reduces to $\texttt{[}\textsc{Next}\texttt{]}\mathbf{0}$. This figures into the definition of acceptance below.

Let Move be the conjunction of these two formulas. Then $u \vDash \textsc{Move}$ if the configurations represented by states v such that (u, v) is in the relation denoted by Next are exactly the configurations that follow from the configuration represented by u in one step according to the transition relation of M.

We can describe the start configuration of the machine M on input x:

• "The machine is in its start state s with its tape head scanning the left endmarker, and $x = x_1 \cdots x_n$ is written on the tape."

$$\textsc{State}_0^{s} \wedge \bigwedge_{1\le i\le n} \textsc{Symbol}_i^{x_i}$$

Let this formula be called Start.

Finally, we can describe the condition of acceptance for alternating Turing machines. Let $U \subseteq Q$ be the set of universal states of M and let $E \subseteq Q$ be the set of existential states of M. Then $Q = U \cup E$ and $U \cap E = \varnothing$.

• "If q is an existential state, then q leads to acceptance if at least one of its

successor configurations leads to acceptance."

$$\bigwedge_{0\le i\le n+1}\ \bigwedge_{q\in E}(\textsc{State}_i^q \to (\textsc{Accept} \leftrightarrow \texttt{<}\textsc{Next}\texttt{>}\textsc{Accept})) \tag{8.2.3}$$

• "If q is a universal state, then q leads to acceptance if all its successor configurations lead to acceptance."

$$\bigwedge_{0\le i\le n+1}\ \bigwedge_{q\in U}(\textsc{State}_i^q \to (\textsc{Accept} \leftrightarrow \texttt{[}\textsc{Next}\texttt{]}\,\textsc{Accept})) \tag{8.2.4}$$

Let Acceptance denote the conjunction of these two formulas.

Recall from Section 2.1 that an *accept configuration* of M is a universal configuration with no next configuration and a *reject configuration* is an existential configuration with no next configuration. As observed above, when this occurs, clauses (8.2.1) and (8.2.2) reduce to **1** and $\texttt{[}\textsc{Next}\texttt{]}\mathbf{0}$, respectively. In conjunction with Acceptance, this implies that Accept is always true at accept configurations and always false at reject configurations.

Now let $\textsc{Accepts}_{M,x}$ be the formula

$$\textsc{Start} \wedge \texttt{[}\textsc{Next}^*\texttt{]}(\textsc{Config} \wedge \textsc{Move} \wedge \textsc{Acceptance}) \wedge \textsc{Accept}.$$

Then M accepts x if and only if $\textsc{Accepts}_{M,x}$ is satisfiable.

We have given an efficient reduction from the membership problem for linear-space alternating Turing machines to the problem of PDL satisfiability. For *EXPTIME*-hardness, we need to give a reduction from the membership problem for polynomial-space alternating Turing machines, but essentially the same construction works. The only differences are that instead of the bound n we use the bound n^k for some fixed constant k in the definition of the formulas, and the formula Start is modified to pad the input out to length n^k with blanks:

$$\textsc{State}_0^s \wedge \bigwedge_{1\le i\le n} \textsc{Symbol}_i^{x_i} \wedge \bigwedge_{n+1\le i\le n^k} \textsc{Symbol}_i^{\sqcup}.$$

Since the membership problem for alternating polynomial-space machines is *EXPTIME*-hard (Chandra et al. (1981)), so is the satisfiability problem for PDL. ■

Corollary 8.6: There is a constant $c > 1$ such that the satisfiability problem for PDL is not solvable in deterministic time $c^{n/\log n}$, where n is the size of the input formula.

Proof An analysis of the construction of $\textsc{Accepts}_{M,x}$ in the proof of Theorem

8.5 reveals that its length is bounded above by $an \log n$ for some constant a, where $n = |x|$, and the time to construct $\text{ACCEPTS}_{M,x}$ from x is at most polynomial in n. The number of symbols in Γ and states in Q are constants and contribute at most a constant factor to the length of the formula.

Now we can use the fact that the complexity class $DTIME(2^n)$ contains a set A not contained in any complexity class $DTIME(d^n)$ for any $d < 2$ (see Hopcroft and Ullman (1979)). Since $A \in DTIME(2^n)$, it is accepted by an alternating linear-space Turing machine M (Chandra et al. (1981); see Section 2.1). We can decide membership in A by converting a given input x to the formula $\text{ACCEPTS}_{M,x}$ using the reduction of Theorem 8.5, then deciding whether $\text{ACCEPTS}_{M,x}$ is satisfiable. Since $|\text{ACCEPTS}_{M,x}| \leq an \log n$, if the satisfiability problem is in $DTIME(c^{n/\log n})$ for some constant c, then we can decide membership in A in time

$$n^k + c^{an \log n / \log(an \log n)};$$

the term n^k is the time required to convert x to $\text{ACCEPTS}_{M,x}$, and the remaining term is the time required to decide the satisfiability of $\text{ACCEPTS}_{M,x}$. But, assuming $a \geq 1$,

$$\begin{aligned} n^k + c^{an \log n / \log(an \log n)} &\leq n^k + c^{an \log n / \log n} \\ &\leq n^k + c^{an}, \end{aligned}$$

which for $c < 2^{1/a}$ is asymptotically less than 2^n. This contradicts the choice of A. ■

It is interesting to compare the complexity of satisfiability in PDL with the complexity of satisfiability in propositional logic. In the latter, satisfiability is *NP*-complete; but at present it is not known whether the two complexity classes *EXPTIME* and *NP* differ. Thus, as far as current knowledge goes, the satisfiability problem is no easier in the worst case for propositional logic than for its far richer superset PDL.

8.3 Compactness and Logical Consequences

As we have seen, current knowledge does not permit a significant difference to be observed between the complexity of satisfiability in propositional logic and in PDL. However, there is one easily verified and important behavioral difference: propositional logic is *compact*, whereas PDL is not.

Compactness has significant implications regarding the relation of logical con-

sequence. If a propositional formula φ is a consequence of a set Γ of propositional formulas, then it is already a consequence of some finite subset of Γ; but this is not true in PDL.

Recall that we write $\Gamma \models \varphi$ and say that φ is a *logical consequence* of Γ if φ satisfied in any state of any Kripke frame $\mathfrak{K}$ all of whose states satisfy all the formulas of Γ. That is, if $\mathfrak{K} \models \Gamma$, then $\mathfrak{K} \models \varphi$.

An alternative intepretation of logical consequence, not equivalent to the above, is that in any Kripke frame, the formula φ holds in any state satisfying all formulas in Γ. Allowing infinite conjunctions, we might write this as $\models \bigwedge \Gamma \to \varphi$. This is not the same as $\Gamma \models \varphi$, since $\models \bigwedge \Gamma \to \varphi$ implies $\Gamma \models \varphi$, but not necessarily vice versa. A counterexample is provided by $\Gamma = \{p\}$ and $\varphi = [a]p$. However, if Γ contains only finitely many atomic programs, we can reduce the problem $\Gamma \models \varphi$ to the problem $\models \bigwedge \Gamma' \to \varphi$ for a related Γ', as shown in Theorem 7.7.

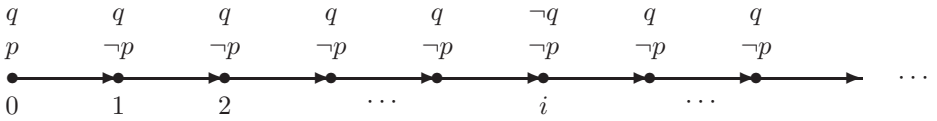
Under either interpretation, compactness fails:

THEOREM 8.7: There is an infinite set of formulas Γ and a formula φ such that $\models \bigwedge \Gamma \to \varphi$ (hence $\Gamma \models \varphi$), but for no proper subset $\Gamma' \subseteq \Gamma$ is it the case that $\Gamma' \models \varphi$ (hence neither is it the case that $\models \bigwedge \Gamma' \to \varphi$).

Proof Take

$$\varphi \stackrel{\text{def}}{=} p \to [a^*]q$$
$$\Gamma \stackrel{\text{def}}{=} \{p \to q,\ p \to [a]q,\ p \to [aa]q,\ \ldots,\ p \to [a^i]q,\ \ldots\}.$$

Then $\models \bigwedge \Gamma \to \varphi$. But for $\Gamma' \subsetneqq \Gamma$, say with $p \to [a^i]q \in \Gamma - \Gamma'$, consider a structure with states ω, atomic program a interpreted as the successor relation, p true only at 0, and q false only at i.



Then all formulas of Γ' are true in all states of this model, but φ is false in state 0. ■

As shown in Theorem 7.7, logical consequences $\Gamma \models \varphi$ for finite Γ are no more difficult to decide than validity of single formulas. But what if Γ is infinite? Here compactness is the key factor. If Γ is an r.e. set and the logic is compact, then the consequence problem is r.e.: to check whether $\Gamma \models \varphi$, the finite subsets of Γ can be

effectively enumerated, and checking $\Gamma \vDash \varphi$ for finite Γ is a decidable problem.

Since compactness fails in PDL, this observation does us no good, even when Γ is known to be recursively enumerable. However, the following result shows that the situation is much worse than we might expect: even if Γ is taken to be the set of substitution instances of a single formula of PDL, the consequence problem becomes very highly undecidable. This is a rather striking manifestation of PDL's lack of compactness.

Let φ be a given formula. The set S_φ of *substitution instances* of φ is the set of all formulas obtained by substituting a formula for each atomic proposition appearing in φ.

THEOREM 8.8: The problem of deciding whether $S_\varphi \vDash \psi$ is Π^1_1-complete. The problem is Π^1_1-hard even for a particular fixed φ.

Proof For the upper bound, it suffices to consider only countable models. The problem is to decide whether for all countable Kripke frames $\mathfrak{M}$, if $\mathfrak{M} \vDash S_\varphi$, then $\mathfrak{M} \vDash \psi$. The Kripke frame $\mathfrak{M}$ is first selected with universal second-order quantification, which determines the interpretation of the atomic program and proposition symbols. Once $\mathfrak{M}$ is selected, the check that $\mathfrak{M} \vDash S_\varphi \implies \mathfrak{M} \vDash \psi$ is first-order: either $\mathfrak{M}, u \vDash \psi$ at all states u of $\mathfrak{M}$, or there exists a substitution instance φ' of φ and a state u of $\mathfrak{M}$ such that $\mathfrak{M}, u \vDash \neg\varphi'$.

For the lower bound, we encode (the complement of) the tiling problem of Proposition 2.22. The fixed scheme φ is used to ensure that any model consists essentially of an $\omega \times \omega$ grid. Let NORTH and EAST be atomic programs and p an atomic proposition. Take φ to be the scheme

$$\begin{aligned}&\texttt{[}(\textsc{North} \cup \textsc{East})^*\texttt{]}(\texttt{<}\textsc{North}\texttt{>}\mathbf{1} \wedge \texttt{<}\textsc{East}\texttt{>}\mathbf{1}\\ &\quad \wedge (\texttt{<}\textsc{North}\texttt{>}p \to \texttt{[}\textsc{North}\texttt{]}p) \wedge (\texttt{<}\textsc{East}\texttt{>}p \to \texttt{[}\textsc{East}\texttt{]}p) \qquad (8.3.1)\\ &\quad \wedge (\texttt{<}\textsc{North}; \textsc{East}\texttt{>}p \to \texttt{[}\textsc{East}; \textsc{North}\texttt{]}p)).\end{aligned}$$

The first line of (8.3.1) says that from any reachable point, one can always continue the grid in either direction. The second line says that any two states reachable from any state under NORTH are indistinguishable by any PDL formula (note that any formula can be substituted for p), and similary for EAST. The third line is a commutativity condition; it says that any state reachable by going NORTH and then EAST is indistinguishable from any state reachable by going EAST and then NORTH. It follows by induction that if σ and τ are any seqs over atomic programs NORTH, EAST such that σ and τ contain the same number of occurrences of each atomic

program—that is, if σ and τ are permutations of each other—then any model of all substitution instances of (8.3.1) must also satisfy all substitution instances of the formula

$$\texttt{[}(\textsc{North} \cup \textsc{East})^*\texttt{]}(\texttt{<}\sigma\texttt{>}p \to \texttt{<}\tau\texttt{>}p)$$

(Exercise 8.3).

Now we will construct the formula ψ, which will be used in two ways:

(i) to describe the legal tilings of the grid with some given set T of tile types, and

(ii) to say that red occurs only finitely often.

For (i), we mimic the construction of Theorem 3.60. We use atomic propositions $\textsc{North}_c, \textsc{South}_c, \textsc{East}_c, \textsc{West}_c$ for each color c. For example, the proposition $\textsc{North}_{\text{blue}}$ says that the north edge of the tile is colored blue, and similarly for the other colors and directions. As in Theorem 3.60, for each tile type $A \in T$, one can construct a formula $\textsc{Tile}_A$ from these propositions that is true at a state iff the truth values of $\textsc{North}_c, \textsc{South}_c, \textsc{East}_c, \textsc{West}_c$ at that state describe a tile of type A. For example, the formula corresponding to the example given in Theorem 3.60 would be

$$\begin{aligned}
\textsc{Tile}_A \iff^{\text{def}} \quad & \textsc{North}_{\text{blue}} \wedge \bigwedge_{\substack{c \in C \\ c \neq \text{blue}}} \neg \textsc{North}_c \\
& \wedge \textsc{South}_{\text{black}} \wedge \bigwedge_{\substack{c \in C \\ c \neq \text{black}}} \neg \textsc{South}_c \\
& \wedge \textsc{East}_{\text{red}} \wedge \bigwedge_{\substack{c \in C \\ c \neq \text{red}}} \neg \textsc{East}_c \\
& \wedge \textsc{West}_{\text{green}} \wedge \bigwedge_{\substack{c \in C \\ c \neq \text{green}}} \neg \textsc{West}_c.
\end{aligned}$$

Let ψ_T be the conjunction

$$\begin{aligned}
&\texttt{[}(\textsc{North} \cup \textsc{East})^*\texttt{]} \bigvee_{A \in T} \textsc{Tile}_A \\
&\quad \wedge\ \texttt{[}\textsc{East}^*\texttt{]}\textsc{South}_{\text{blue}} \\
&\quad \wedge\ \texttt{[}\textsc{North}^*\texttt{]}\textsc{West}_{\text{blue}} \\
&\quad \wedge\ \texttt{[}(\textsc{North} \cup \textsc{East})^*\texttt{]} \bigwedge_{c \in C} (\textsc{East}_c \rightarrow \texttt{<}\textsc{East}\texttt{>}\textsc{West}_c) \\
&\quad \wedge\ \texttt{[}(\textsc{North} \cup \textsc{East})^*\texttt{]} \bigwedge_{c \in C} (\textsc{North}_c \rightarrow \texttt{<}\textsc{North}\texttt{>}\textsc{South}_c).
\end{aligned}$$

These correspond to the sentences (3.4.3)–(3.4.7) of Theorem 3.60. As in that theorem, any model of ψ_T must be a legal tiling.

Finally, we give a formula that says that red occurs only finitely often in the tiling:

$$\begin{aligned}
\textsc{red} &\iff^{\text{def}} \textsc{North}_{\text{red}} \vee \textsc{South}_{\text{red}} \vee \textsc{East}_{\text{red}} \vee \textsc{West}_{\text{red}} \\
\psi_{\text{red}} &\iff^{\text{def}} \texttt{<}\textsc{North}^*\texttt{>[}(\textsc{North} \cup \textsc{East})^*\texttt{]}\neg\textsc{red} \\
&\quad \wedge\ \texttt{<}\textsc{East}^*\texttt{>[}(\textsc{North} \cup \textsc{East})^*\texttt{]}\neg\textsc{red}.
\end{aligned}$$

Then all valid tilings use only finitely many tiles with a red edge iff

$$S_\varphi \ \models\ \psi_T \rightarrow \psi_{\text{red}}.$$

■

The proof of Theorem 8.8 can be refined so as to yield similar results for more restricted versions of PDL discussed in Chapter 10. Specifically, the result holds for SDPDL of Section 10.1 and $\mathsf{PDL}^{(0)}$ of Section 10.2. Of course, since the result is negative in nature, it holds for any extensions of these logics.

8.4 Bibliographical Notes

The exponential-time lower bound for PDL was established by Fischer and Ladner (1977, 1979) by showing how PDL formulas can encode computations of linear-space-bounded alternating Turing machines.

Deterministic exponential-time algorithms were first given in Pratt (1978, 1979b, 1980b). The algorithm given here is essentially from Pratt (1979b). The algorithm has been implemented by Pratt and reportedly works well on small formulas.

Theorem 8.8 showing that the problem of deciding whether $\Gamma \models \psi$, where Γ is a fixed r.e. set of PDL formulas, is Π^1_1-complete is due to Meyer et al. (1981).

Exercises

8.1. Supply the missing arguments in the proof of Lemma 8.3: part (i) for the cases $\rightarrow$ and $\mathbf{0}$, part (ii)(a) for the cases $\cup$ and *, and part (ii)(b) for the cases $\cup$ and ;.

8.2. Show how to encode the acceptance problem for linear-space alternating Turing machines in the validity problem for PDL. In other words, given such a machine M and an input x, show how to construct a PDL formula that is valid (true at all states in all Kripke frames) iff M accepts x. (*Hint.* Use the machinery constructed in Section 8.2.)

8.3. In the proof of Theorem 8.8, argue that if σ and τ are any seqs over atomic programs NORTH, EAST such that σ and τ contain the same number of occurrences of each atomic program—that is, if σ and τ are permutations of each other—then any model of all substitution instances of (8.3.1) must also satisfy all substitution instances of the formula

$[(\text{NORTH} \cup \text{EAST})^*](\langle\sigma\rangle p \rightarrow \langle\tau\rangle p)$.

9 Nonregular PDL

In this chapter we enrich the class of regular programs in PDL by introducing programs whose control structure requires more than a finite automaton. For example, the class of *context-free programs* requires a pushdown automaton (PDA), and moving up from regular to context-free programs is really going from iterative programs to ones with parameterless recursive procedures. Several questions arise when enriching the class of programs of PDL, such as whether the expressive power of the logic grows, and if so whether the resulting logics are still decidable. We first show that any nonregular program increases PDL's expressive power and that the validity problem for PDL with context-free programs is undecidable. The bulk of the chapter is then devoted to the difficult problem of trying to characterize the borderline between decidable and undecidable extensions. On the one hand, validity for PDL with the addition of even a single extremely simple nonregular program is shown to be already Π^1_1-complete; but on the other hand, when we add another equally simple program, the problem remains decidable. Besides these results, which pertain to very specific extensions, we discuss some broad decidability results that cover many languages, including some that are not even context-free. Since no similarly general undecidability results are known, we also address the weaker issue of whether nonregular extensions admit the finite model property and present a negative result that covers many cases.

9.1 Context-Free Programs

Consider the following self-explanatory program:

while p **do** a ; **now do** b **the same number of times** (9.1.1)

This program is meant to represent the following set of computation sequences:

$\{(p?\,;\,a)^i\,;\,\neg p?\,;\,b^i \mid i \geq 0\}$.

Viewed as a language over the alphabet $\{a, b, p, \neg p\}$, this set is not regular, thus cannot be programmed in PDL. However, it can be represented by the following parameterless recursive procedure:

proc V {
  if p **then** { a ; **call** V ; b }
    else return
  }

The set of computation sequences of this program is captured by the context-free grammar

$$V \quad \rightarrow \quad \neg p? \mid p?aVb.$$

We are thus led to the idea of allowing context-free programs inside the boxes and diamonds of PDL. From a pragmatic point of view, this amounts to extending the logic with the ability to reason about parameterless recursive procedures. The particular representation of the context-free programs is unimportant; we can use pushdown automata, context-free grammars, recursive procedures, or any other formalism that can be effectively translated into these.

In the rest of the chapter, a number of specific programs will be of interest, and we employ special abbreviations for them. For example, we define:

$$\begin{aligned} a^\Delta b a^\Delta &\stackrel{\text{def}}{=} \{a^i b a^i \mid i \geq 0\} \\ a^\Delta b^\Delta &\stackrel{\text{def}}{=} \{a^i b^i \mid i \geq 0\} \\ b^\Delta a^\Delta &\stackrel{\text{def}}{=} \{b^i a^i \mid i \geq 0\}. \end{aligned}$$

Note that $a^\Delta b^\Delta$ is really just a nondeterministic version of the program (9.1.1) in which there is simply no p to control the iteration. In fact, (9.1.1) could have been written in this notation as $(p?a)^\Delta \neg p? b^\Delta$.[1] In programming terms, we can compare the regular program $(ab)^*$ with the nonregular one $a^\Delta b^\Delta$ by observing that if a is "purchase a loaf of bread" and b is "pay \$1.00," then the former program captures the process of paying for each loaf when purchased, while the latter one captures the process of paying for them all at the end of the month.

9.2 Basic Results

We first show that enriching PDL with even a single arbitrary nonregular program increases expressive power.

1 It is noteworthy that the results of this chapter do not depend on nondeterminism. For example, the negative Theorem 9.6 holds for the deterministic version (9.1.1) too. Also, most of the results in the chapter involve nonregular programs over atomic programs only, but can be generalized to allow tests as well.

DEFINITION 9.1: If L is any language over atomic programs and tests, then $\mathsf{PDL} + L$ is defined exactly as PDL, but with the additional syntax rule stating that for any formula φ, the expression $\langle L\rangle\varphi$ is a new formula. The semantics of $\mathsf{PDL} + L$ is like that of PDL with the addition of the clause

$$\mathfrak{m}_{\mathfrak{K}}(L) \stackrel{\text{def}}{=} \bigcup_{\beta \in L} \mathfrak{m}_{\mathfrak{K}}(\beta).$$

Note that $\mathsf{PDL} + L$ does not allow L to be used as a formation rule for new programs or to be combined with other programs. It is added to the programming language as a single new stand-alone program only.

DEFINITION 9.2: If PDL_1 and PDL_2 are two extensions of PDL, we say that PDL_1 is *as expressive as* PDL_2 if for each formula φ of PDL_2 there is a formula ψ of PDL_1 such that $\vDash \varphi \leftrightarrow \psi$. If PDL_1 is as expressive as PDL_2 but PDL_2 is not as expressive as PDL_1, we say that PDL_1 is *strictly more expressive than* PDL_2.

Thus, one version of PDL is strictly more expressive than another if anything the latter can express the former can too, but there is something the former can express that the latter cannot.

A language is *test-free* if it is a subset of Π_0^*; that is, if its seqs contain no tests.

THEOREM 9.3: If L is any nonregular test-free language, then $\mathsf{PDL} + L$ is strictly more expressive than PDL.

Proof The result can be proved by embedding PDL into SkS, the monadic second-order theory of k successors (Rabin (1969)). It is possible to show that any set of nodes definable in SkS is regular, so that the addition of a nonregular predicate increases its expressive power.

A more direct proof can be obtained as follows. Fix a subset $\{a_0, \ldots, a_{k-1}\} \subseteq \Pi_0$. Define the Kripke frame $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ in which

$$\begin{aligned} K &\stackrel{\text{def}}{=} \{a_0, \ldots, a_{k-1}\}^* \\ \mathfrak{m}_{\mathfrak{K}}(a_i) &\stackrel{\text{def}}{=} \{(a_i x, x) \mid x \in \{a_0, \ldots, a_{k-1}\}^*\} \\ \mathfrak{m}_{\mathfrak{K}}(p) &\stackrel{\text{def}}{=} \{\varepsilon\}. \end{aligned}$$

The frame $\mathfrak{K}$ can be viewed as a complete k-ary tree in which p holds at the root only and each node has k offspring, one for each atomic program a_i, but with all

edges pointing upward. Thus, the only seq from the node $x \in \{a_0, \ldots, a_{k-1}\}^*$ that leads to a state satisfying p is x itself.

Now for any formula φ of PDL, the set $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ is the set of words over $\{a_0, \ldots, a_{k-1}\}$ describing paths in $\mathfrak{K}$ leading from states that satisfy φ to the root. It is easy to show by induction on the structure of φ that $\mathfrak{m}_{\mathfrak{K}}(\varphi)$ is a regular set over the alphabet $\{a_0, \ldots, a_{k-1}\}$ (Exercise 9.1). Since $\mathfrak{m}_{\mathfrak{K}}(\langle L\rangle p) = L$ is nonregular, $\langle L\rangle p$ cannot be equivalent to any PDL formula. ■

We can view the decidability of regular PDL as showing that propositional-level reasoning about iterative programs is computable. We now wish to know if the same is true for recursive procedures. We define *context-free* PDL to be PDL extended with context-free programs, where a *context-free program* is one whose seqs form a context-free language. The precise syntax is unimportant, but for definiteness we might take as programs the set of context-free grammars G over atomic programs and tests and define

$$\mathfrak{m}_{\mathfrak{K}}(G) \stackrel{\text{def}}{=} \bigcup_{\beta \in CS(G)} \mathfrak{m}_{\mathfrak{K}}(\beta),$$

where $CS(G)$ is the set of computation sequences generated by G as described in Section 4.3.

THEOREM 9.4: The validity problem for context-free PDL is undecidable.

Proof Consider the formula $\langle G\rangle p \leftrightarrow \langle G'\rangle p$ for context-free grammars G and G' and atomic p. It can be shown that if $CS(G)$ and $CS(G')$ are test-free, then this formula is valid iff $CS(G) = CS(G')$ (Exercise 9.2). This reduces the equivalence problem for context-free languages to the validity problem for context-free PDL. The equivalence problem for context-free languages is well known to be undecidable; see Hopcroft and Ullman (1979) or Kozen (1997a). ■

Theorem 9.4 leaves several interesting questions unanswered. What is the level of undecidability of context-free PDL? What happens if we want to add only a small number of specific nonregular programs? The first of these questions arises from the fact that the equivalence problem for context-free languages is co-r.e., or in the notation of the arithmetic hierarchy (Section 2.2), it is complete for Π^0_1. Hence, all Theorem 9.4 shows is that the validity problem for context-free PDL is Π^0_1-hard, while it might in fact be worse. The second question is far more general. We might be interested in reasoning only about deterministic or linear context-free

programs,[2] or we might be interested only in a few special context-free programs such as $a^\Delta ba^\Delta$ or $a^\Delta b^\Delta$. Perhaps PDL remains decidable when these programs are added. The general question is to determine the borderline between the decidable and the undecidable when it comes to enriching the class of programs allowed in PDL.

Interestingly, if we wish to consider such simple nonregular extensions as $\mathsf{PDL} + a^\Delta ba^\Delta$ or $\mathsf{PDL} + a^\Delta b^\Delta$, we will not be able to prove undecidability by the technique used for context-free PDL in Theorem 9.4, since standard problems that are undecidable for context-free languages, such as equivalence and inclusion, are decidable for classes containing the regular languages and the likes of $a^\Delta ba^\Delta$ and $a^\Delta b^\Delta$. Moreover, we cannot prove decidability by the technique used for PDL in Section 6.2, since logics like $\mathsf{PDL} + a^\Delta ba^\Delta$ and $\mathsf{PDL} + a^\Delta b^\Delta$ do not enjoy the finite model property, as we now show. Thus, if we want to determine the decidability status of such extensions, we will have to work harder.

THEOREM 9.5: There is a satisfiable formula in $\mathsf{PDL} + a^\Delta b^\Delta$ that is not satisfied in any finite structure.

Proof Let φ be the formula

$$p \;\wedge\; [a^*]\langle ab^*\rangle p \;\wedge\; [(a \cup b)^* ba]\mathbf{0} \;\wedge\; [a^* a][a^\Delta b^\Delta]\neg p \;\wedge\; [a^\Delta b^\Delta][b]\mathbf{0}.$$

Let $\mathfrak{K}_0$ be the infinite structure illustrated in Fig. 9.1 in which the only states satisfying p are the dark ones. It is easy to see that $\mathfrak{K}_0, u \vDash \varphi$. Now let $\mathfrak{K}$ be a finite structure with a state u such that $\mathfrak{K}, u \vDash \varphi$. Viewing $\mathfrak{K}$ as a finite graph, we associate paths with the sequences of atomic programs along them. Consider the set U of paths in $\mathfrak{K}$ leading from u to states satisfying p. The fact that $\mathfrak{K}$ is finite implies that U is a regular set of words. However, the third conjunct of φ eliminates from U paths that contain b followed by a, forcing U to be contained in a^*b^*; the fourth and fifth conjuncts force U to be a subset of $\{a^i b^i \mid i \geq 0\}$; and the first two conjuncts force U to contain a word in $a^i b^*$ for each $i \geq 0$. Consequently, U must be exactly $\{a^i b^i \mid i \geq 0\}$, contradicting regularity. ■

2 A *linear program* is one whose seqs are generated by a context-free grammar in which there is at most one nonterminal symbol on the right-hand side of each rule. This corresponds to a family of recursive procedures in which there is at most one recursive call in each procedure.

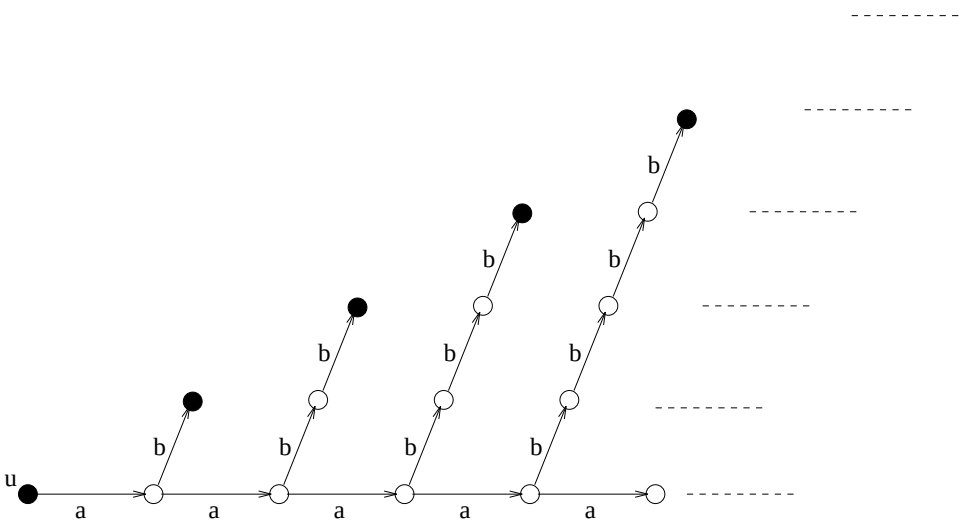


Figure 9.1

9.3 Undecidable Extensions

Two-Letter Programs

The proof of Theorem 9.5 can be modified easily to work for $\mathsf{PDL}+a^{\Delta}ba^{\Delta}$ (Exercise 9.3). However, for this extension the news is worse than mere undecidability:

THEOREM 9.6: The validity problem for $\mathsf{PDL} + a^{\Delta}ba^{\Delta}$ is Π^1_1-complete.

Proof To show that the problem is in Π^1_1, we use the Löwenheim–Skolem Theorem (Section 3.4) to write the notion of validity in the general form "For every countable structure. . . ," then observe that the question of whether a given formula is satisfied in a given countable structure is arithmetical.

To show that the problem is Π^1_1-hard, we reduce the tiling problem of Proposition 2.22 to it. Recall that in this tiling problem, we are given a set T of tile types, and we wish to know whether the $\omega \times \omega$ grid can be tiled so that the color red appears infinitely often.

We proceed as in the proof of the lower bound of Theorem 8.8. We use the same atomic propositions $\text{NORTH}_c, \text{SOUTH}_c, \text{EAST}_c, \text{WEST}_c$ for each color c. For example, NORTH_c says that the north edge of the current tile is colored c. As in Theorem 8.8, for each tile type $A \in T$, we construct a formula TILE_A from these propositions

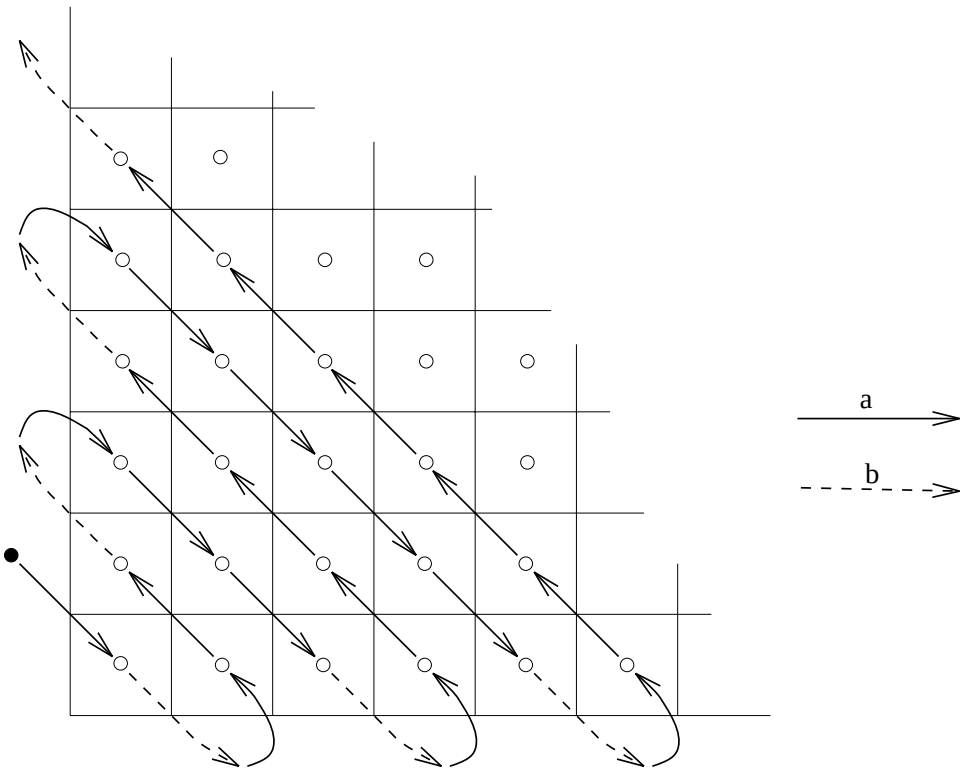


Figure 9.2

that is true at a state if the truth values of $\text{NORTH}_c, \text{SOUTH}_c, \text{EAST}_c, \text{WEST}_c$ at that state describe a tile of type A.

The construction of the grid must be different here, since the extension of PDL with the new program $a^{\Delta}ba^{\Delta}$ does not offer a direct way of setting up two atomic programs, such as NORTH and EAST, to correspond to the two directions on a grid, as was done in Theorems 3.60 and 8.8. The imaginary grid that we want to tile must be set up in a subtler manner.

Denoting $a^{\Delta}ba^{\Delta}$ by α and a^*ab by β, let φ_{snake} be the formula

$$\langle ab\rangle \mathbf{1} \quad \wedge \quad [\beta^*](\langle\beta\rangle \mathbf{1} \wedge [a^*a][\alpha][ab]\mathbf{0} \wedge [\alpha][aa]\mathbf{0}).$$

This formula forces the existence of an infinite path of the form $\sigma = aba^2ba^3ba^4b\cdots$. Fig. 9.2 shows how this path is to be imagined as snaking through $\omega \times \omega$, and the details of the proof are based on this correspondence.

We now have to state that the grid implicit in the path σ is tiled legally with tiles from T and that red occurs infinitely often. For this we use the formula RED from Theorem 8.8:

$$\text{RED} \iff^{\text{def}} \text{NORTH}_{\text{red}} \vee \text{SOUTH}_{\text{red}} \vee \text{EAST}_{\text{red}} \vee \text{WEST}_{\text{red}}.$$

We then construct the general formula φ_T as the conjunction of φ_{snake} and the following formulas:

$$\texttt{[}(a \cup b)^* a\texttt{]} \bigvee_{A \in T} \text{TILE}_A \tag{9.3.1}$$

$$\texttt{[}(\beta\beta)^* a^* a\texttt{]} \bigwedge_{c \in C} ((\text{EAST}_c \to \texttt{[}\alpha a\texttt{]}\,\text{WEST}_c) \wedge (\text{NORTH}_c \to \texttt{[}\alpha a a\texttt{]}\,\text{SOUTH}_c)) \tag{9.3.2}$$

$$\texttt{[}(\beta\beta)^* \beta a^* a\texttt{]} \bigwedge_{c \in C} ((\text{EAST}_c \to \texttt{[}\alpha a a\texttt{]}\,\text{WEST}_c) \wedge (\text{NORTH}_c \to \texttt{[}\alpha a\texttt{]}\,\text{SOUTH}_c)) \tag{9.3.3}$$

$$\texttt{[}\beta^*\texttt{]<}\beta^* a^* a\texttt{>}\text{RED}. \tag{9.3.4}$$

Clause (9.3.1) associates tiles from T with those points of σ that follow a's, which are exactly the points of $\omega \times \omega$. Clauses (9.3.2) and (9.3.3) force the matching of colors by using $a^\Delta b a^\Delta$ to reach the correct neighbor, coming from above or below depending on the parity of β's. Finally, clause (9.3.4) can be shown to force the recurrence of red. This is not straightforward. In the case of the consequence problem of Theorem 8.8, the ability to substitute arbitrary formulas for the atomic proposition p made it easy to enforce the uniformity of properties in the grid. In contrast, here the $\texttt{<}\beta^*\texttt{>}$ portion of the formula could be satisfied along different paths that branch off the main path σ. Nevertheless, a König-like argument can be used to show that indeed there is an infinite recurrence of red in the tiling along the chosen path σ (Exercise 9.4).

It follows that φ_T is satisfiable if and only if T can tile $\omega \times \omega$ with red recurring infinitely often. ■

The Π^1_1 result holds also for PDL extended with the two programs $a^\Delta b^\Delta$ and $b^\Delta a^\Delta$ (Exercise 9.5).

It is easy to show that the validity problem for context-free PDL in its entirety remains in Π^1_1. Together with the fact that $a^\Delta b a^\Delta$ is a context-free language, this yields an answer to the first question mentioned earlier: context-free PDL is Π^1_1-complete. As to the second question, Theorem 9.6 shows that the high undecidability phenomenon starts occurring even with the addition of one very simple nonregular program.

One-Letter Programs

We now turn to nonregular programs over a single letter. Consider the language of powers of 2:

$$a^{2^*} \stackrel{\text{def}}{=} \{a^{2^i} \mid i \geq 0\}.$$

Here we have:

THEOREM 9.7: The validity problem for $\mathsf{PDL} + a^{2^*}$ is undecidable.

Proof sketch. This proof is also carried out by a reduction from a tiling problem, but this time on a subset of the $\omega \times \omega$ grid. It makes essential use of simple properties of powers of 2.

The idea is to arrange the elements of the set $S = \{2^i + 2^j \mid i, j \geq 0\}$ in a grid as shown in Fig. 9.3. Elements of this set are reached by executing the new program a^{2^*} twice from the start state. The key observation in the proof has to do with the points that are reached when a^{2^*} is executed once more from a point u already in S. If u is not a power of two (that is, if $u = 2^i + 2^j$ for $i \neq j$), then the only points in S that can be reached by adding a third power of 2 to u are u's upper and right-hand neighbors in Fig. 9.3. If u is a power of 2 (that is, if $u = 2^i + 2^i$), then the points in S reached in this manner form an infinite set consisting of one row (finite) and one column (infinite) in the figure. A particularly delicate part of the proof involves setting things up so that the upper neighbor can be distinguished from the right-hand one. This is done by forcing a periodic marking of the grid with three diagonal stripes encoded by three new atomic programs. In this way, the two neighbors will always be associated with different detectable stripes. Exercise 9.6 asks for the details. ■

It is actually possible to prove this result for powers of any fixed $k \geq 2$. Thus PDL with the addition of any language of the form $\{a^{k^i} \mid i \geq 0\}$ for fixed $k \geq 2$ is undecidable. Another class of one-letter extensions that has been proven to be undecidable consists of Fibonacci-like sequences:

THEOREM 9.8: Let f_0, f_1 be arbitrary elements of $\mathbb{N}$ with $f_0 < f_1$, and let F be the sequence $f_0, f_1, f_2, \ldots$ generated by the recurrence $f_i = f_{i-1} + f_{i-2}$ for $i \geq 2$. Let $a^F \stackrel{\text{def}}{=} \{a^{f_i} \mid i \geq 0\}$. Then the validity problem for $\mathsf{PDL} + a^F$ is undecidable.

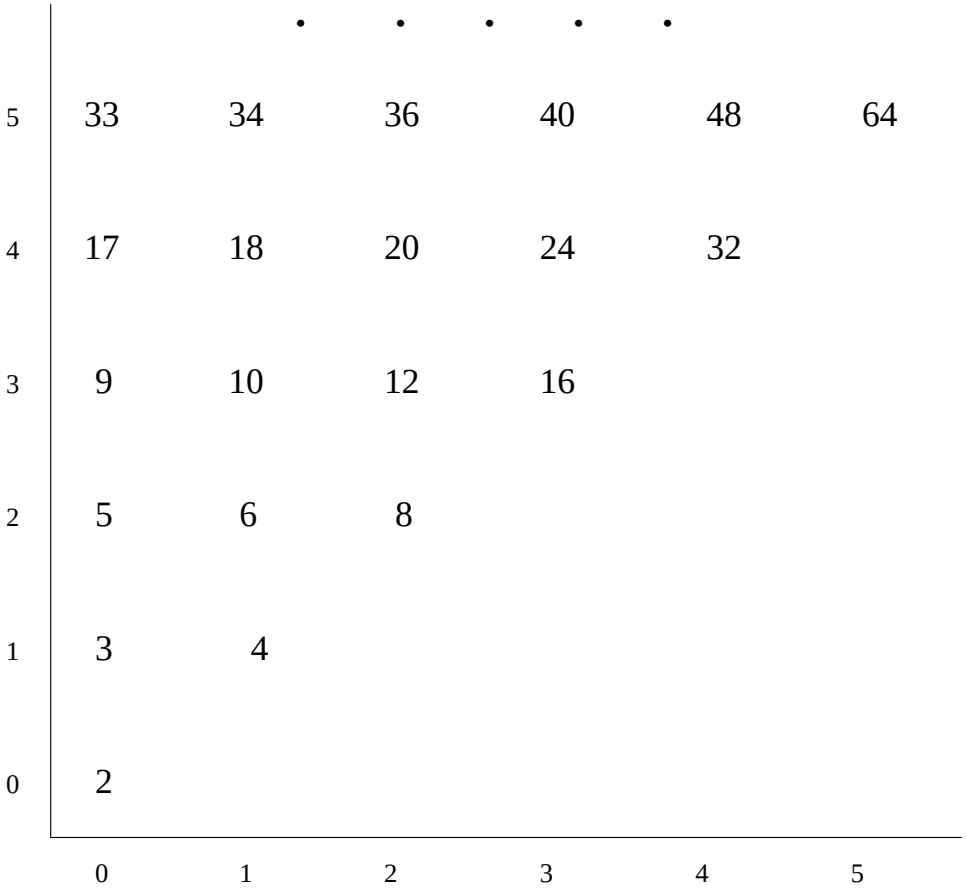


Figure 9.3

The proof of this result follows the general lines of the proof of Theorem 9.7, but is more complicated. It is based on a careful analysis of the properties of sums of elements of F.

In both these theorems, the fact that the sequences of a's in the programs grow exponentially is crucial to the proofs. Indeed, we know of no undecidability results for any one-letter extension in which the lengths of the sequences of a's grow

subexponentially. Particularly intriguing are the cases of squares and cubes:

$$a^{*^2} \stackrel{\text{def}}{=} \{a^{i^2} \mid i \geq 0\},$$
$$a^{*^3} \stackrel{\text{def}}{=} \{a^{i^3} \mid i \geq 0\}.$$

Are $\mathsf{PDL} + a^{*^2}$ and $\mathsf{PDL} + a^{*^3}$ undecidable?

In Section 9.5 we shall describe a decidability result for a slightly restricted version of the squares extension, which seems to indicate that the full unrestricted version $\mathsf{PDL} + a^{*^2}$ is decidable too. However, we conjecture that for cubes the problem is undecidable. Interestingly, several classical open problems in number theory reduce to instances of the validity problem for $\mathsf{PDL} + a^{*^3}$. For example, while no one knows whether every integer greater than 10000 is the sum of five cubes, the following formula is valid if and only if the answer is yes:

$$[(a^{*^3})^5]p \quad \rightarrow \quad [a^{10001}a^*]p.$$

(The 5-fold and 10001-fold iterations have to be written out in full, of course.) If $\mathsf{PDL} + a^{*^3}$ were decidable, then we could compute the answer in a simple manner, at least in principle.

9.4 Decidable Extensions

We now turn to positive results. In Theorem 9.5 we showed that $\mathsf{PDL} + a^\Delta b^\Delta$ does not have the finite model property. Nevertheless, we have the following:

THEOREM 9.9: The validity problem for $\mathsf{PDL} + a^\Delta b^\Delta$ is decidable.

When contrasted with Theorem 9.6, the decidability of $\mathsf{PDL} + a^\Delta b^\Delta$ is very surprising. We have two of the simplest nonregular languages—$a^\Delta b a^\Delta$ and $a^\Delta b^\Delta$—which are extremely similar, yet the addition of one to PDL yields high undecidability while the other leaves the logic decidable.

Theorem 9.9 was proved originally by showing that, although $\mathsf{PDL} + a^\Delta b^\Delta$ does not always admit finite models, it does admit finite *pushdown models*, in which transitions are labeled not only with atomic programs but also with push and pop instructions for a particular kind of stack. A close study of the proof (which relies heavily on the idiosyncrasies of the language $a^\Delta b^\Delta$) suggests that the decidability or undecidability has to do with the manner in which an automaton accepts the languages involved. For example, in the usual way of accepting $a^\Delta b a^\Delta$, a pushdown automaton (PDA) reading an a will carry out a push or a pop, depending upon its

location in the input word. However, in the standard way of accepting $a^\Delta b^\Delta$, the a's are always pushed and the b's are always popped, regardless of the location; the input symbol alone determines what the automaton does. More recent work, which we now set out to describe, has yielded a general decidability result that confirms this intuition. It is of special interest due to its generality, since it does not depend on specific programs.

DEFINITION 9.10: Let $M = (Q, \Sigma, \Gamma, q_0, z_0, \delta)$ be a PDA that accepts by empty stack. We say that M is *simple-minded* if, whenever $\delta(q, \sigma, \gamma) = (p, b)$, then for each q' and γ', either $\delta(q', \sigma, \gamma') = (p, b)$ or $\delta(q', \sigma, \gamma')$ is undefined. A context-free language is said to be *simple-minded* (a simple-minded CFL) if there exists a simple-minded PDA that accepts it.

In other words, the action of a simple-minded automaton is determined uniquely by the input symbol; the state and stack symbol are only used to help determine whether the machine halts (rejecting the input) or continues. Note that such an automaton is necessarily deterministic.

It is noteworthy that simple-minded PDAs accept a large fragment of the context-free languages, including $a^\Delta b^\Delta$ and $b^\Delta a^\Delta$, as well as all balanced parenthesis languages (Dyck sets) and many of their intersections with regular languages.

EXAMPLE 9.11: Let $M = (\{q_0, q\}, \Sigma, \Gamma, q_0, z_0, \delta)$ be a PDA, where $\Sigma = \{a, b\}$, $\Gamma = \{z, z_0\}$, and the transition function δ is given by:

$$\begin{aligned}
\delta(q_0, a, z_0) &= (q_0, \mathbf{pop}; \mathbf{push}(z)) \\
\delta(q_0, a, z) &= (q_0, \mathbf{push}(z)) \\
\delta(q_0, b, z) &= (q, \mathbf{pop}) \\
\delta(q, b, z) &= (q, \mathbf{pop}).
\end{aligned}$$

The function δ is undefined for all other possibilities. Since M accepts by empty stack, the language accepted is precisely $\{a^i b^i \mid i \geq 1\}$. The automaton M is simple-minded, since it always performs $\mathbf{push}(z)$ when the input is a and $\mathbf{pop}$ when the input is b.

EXAMPLE 9.12: Let $M = (\{q\}, \Sigma \cup \Sigma', \Gamma, q, z_0, \delta)$ be a PDA, where $\Sigma = \{[,]\}$, Σ' is some finite alphabet of interest disjoint from Σ, $\Gamma = \{[, z_0\}$, and the transition

function δ is given by:

$$\begin{aligned}\delta(q, [, z_0) &= (q, \mathbf{pop}; \mathbf{push}([\,]))\\ \delta(q, a, [\,) &= (q, \mathbf{sp}) \quad \text{for } a \in \Sigma'\\ \delta(q,], [\,) &= (q, \mathbf{pop}).\end{aligned}$$

Here **sp** stands for "stay put," and can be considered an abbreviation for $\mathbf{push}(\varepsilon)$. The function δ is undefined for all other possibilities. Since the automaton only accepts by empty stack, the language accepted by M is precisely the set of expressions over $\Sigma \cup \Sigma'$ beginning with [and ending with] in which the parentheses are balances. The automaton M is simple-minded, since it always performs $\mathbf{push}([\,)$ when the input is [, **pop** when the input is], and **sp** when the input is a letter from Σ'.

The main purpose of this entire section is to prove the following:

THEOREM 9.13: If L is accepted by a simple-minded PDA, then $\mathsf{PDL} + L$ is decidable.

First, however, we must discuss a certain class of models of PDL.

Tree Models

We first prove that $\mathsf{PDL} + L$ has the *tree model property*. Let ψ be a formula of $\mathsf{PDL} + L$ containing n distinct atomic programs, including those used in L. A *tree structure* for ψ in $\mathsf{PDL} + L$ is a Kripke frame $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ such that

- K is a nonempty prefix-closed subset of $[k]^*$, where $[k] = \{0, \ldots, k-1\}$ for some $k \geq 0$ a multiple of n;
- for all atomic programs a, $\mathfrak{m}_{\mathfrak{K}}(a) \subseteq \{(x, xi) \mid x \in [k]^*,\ i \in [k]\}$;
- if a, b are atomic programs and $a \neq b$, then $\mathfrak{m}_{\mathfrak{K}}(a) \cap \mathfrak{m}_{\mathfrak{K}}(b) = \varnothing$.

A tree structure $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ is a *tree model* for ψ if $\mathfrak{K}, \varepsilon \vDash \psi$, where ε is the null string, the root of the tree.

We now show that for any L, if a $\mathsf{PDL} + L$ formula ψ is satisfiable, then it has a tree model. To do this we first unwind any model of ψ into a tree as in Theorem 3.74, then use a construction similar to Exercise 3.42 to create a substructure in which every state has a finite number of successors. In order to proceed, we want to be able to refer to the Fischer–Ladner closure $FL(\psi)$ of a formula ψ in $\mathsf{PDL}+L$. The definition of Section 6.1 can be adopted as is, except that we take $FL^{\square}([L]\sigma) = \varnothing$

(we will not need it). Note however that if $[L]\sigma \in FL(\psi)$, then $\sigma \in FL(\psi)$ as well.

Now for the tree result:

PROPOSITION 9.14: A formula ψ in $\mathsf{PDL} + L$ is satisfiable iff it has a tree model.

Proof Suppose that $\mathfrak{K}, u \vDash \psi$ for some Kripke frame $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ and $u \in K$. Let C_i for $0 \leq i < 2^{|FL(\psi)|}$ be an enumeration of the subsets of $FL(\psi)$, let $k = n2^{|FL(\psi)|}$, and let

$$\mathbf{Th}_\psi(t) \stackrel{\text{def}}{=} \{\xi \in FL(\psi) \mid \mathfrak{K}, t \vDash \xi\}.$$

To show that ψ has a tree model, we first define a partial mapping $\rho : [k]^* \to 2^K$ by induction on the length of words in $[k]^*$:

$$\begin{aligned} \rho(\varepsilon) &\stackrel{\text{def}}{=} \{u\} \\ \rho(x(i+nj)) &\stackrel{\text{def}}{=} \{t \in K \mid \exists s \in \rho(x)\ (s,t) \in \mathfrak{m}_{\mathfrak{K}}(a_i) \text{ and } C_j = \mathbf{Th}_\psi(t)\} \end{aligned}$$

for all $0 \leq i < n$ and $0 \leq j < 2^{|FL(\psi)|}$. Note that if $\rho(x)$ is the empty set, then so is $\rho(xi)$.

We now define a Kripke frame $\mathfrak{K}' = (K', \mathfrak{m}_{\mathfrak{K}'})$ as follows:

$$\begin{aligned} K' &\stackrel{\text{def}}{=} \{x \mid \rho(x) \neq \varnothing\}, \\ \mathfrak{m}_{\mathfrak{K}'}(a_i) &\stackrel{\text{def}}{=} \{(x, x(i+nj)) \mid 0 \leq j < 2^{|FL(\psi)|},\ x(i+nj) \in K'\}, \\ \mathfrak{m}_{\mathfrak{K}'}(p) &\stackrel{\text{def}}{=} \{x \mid \exists t \in \rho(x)\ t \in \mathfrak{m}_{\mathfrak{K}}(p)\}. \end{aligned}$$

Note that $\mathfrak{m}_{\mathfrak{K}'}$ is well defined by the definitions of ρ and $\mathfrak{m}_{\mathfrak{K}}$. It is not difficult to show that $\mathfrak{K}'$ is a tree structure and that if $x \in K'$ and $\xi \in FL(\psi)$, then $\mathfrak{K}', x \vDash \xi$ iff $\mathfrak{K}, t \vDash \xi$ for some $t \in \rho(x)$. In particular, $\mathfrak{K}', \varepsilon \vDash \psi$.

The converse is immediate. ■

Let $CL(\psi)$ be the set of all formulas in $FL(\psi)$ and their negations. Applying the De Morgan laws and the PDL identities

$$\begin{aligned} \neg[\alpha]\varphi &\leftrightarrow \langle\alpha\rangle\neg\varphi \\ \neg\langle\alpha\rangle\varphi &\leftrightarrow [\alpha]\neg\varphi \\ \neg\neg\varphi &\leftrightarrow \varphi \end{aligned}$$

from left to right, we can assume without loss of generality that negations in formulas of $CL(\psi)$ are applied to atomic formulas only.

Let

$$CL^{\perp}(\psi) \stackrel{\text{def}}{=} CL(\psi) \cup \{\perp\}.$$

We would now like to embed the tree model $\mathfrak{K}' = (K', \mathfrak{m}_{\mathfrak{K}'})$ constructed above into a certain labeled complete k-ary tree. Every node in K' will be labeled by the formulas in $CL(\psi)$ that it satisfies, and all the nodes not in K' are labeled by the special symbol $\perp$. These trees satisfy some special properties, as we shall now see.

Definition 9.15: A *unique diamond path Hintikka tree* (or UDH tree for short) for a PDL$+L$ formula ψ with atomic programs $a_0, \ldots, a_{n-1}$ consists of a k-ary tree $[k]^*$ for some k a multiple of n and two labeling functions

$$T : [k]^* \rightarrow 2^{CL^{\perp}(\psi)}$$
$$\Phi : [k]^* \rightarrow CL^{\perp}(\psi)$$

such that $\psi \in T(\varepsilon)$; for all $x \in [k]^*$, $\Phi(x)$ is either a single diamond formula or the special symbol $\perp$; and

1. either $T(x) = \{\perp\}$ or $\perp \notin T(x)$, and in the latter case, $\xi \in T(x)$ iff $\neg\xi \notin T(x)$ for all $\xi \in FL(\psi)$;
2. if $\xi \rightarrow \sigma \in T(x)$ and $\xi \in T(x)$, then $\sigma \in T(x)$, and $\xi \wedge \sigma \in T(x)$ iff both $\xi \in T(x)$ and $\sigma \in T(x)$;
3. if $\langle\gamma\rangle\xi \in T(x)$, and
   (a) if γ is an atomic program a_i, then there exists j such that $i + nj < k$ and $\xi \in T(x(i + nj))$;
   (b) if $\gamma = \alpha\,;\,\beta$, then $\langle\alpha\rangle\langle\beta\rangle\xi \in T(x)$;
   (c) if $\gamma = \alpha \cup \beta$, then either $\langle\alpha\rangle\xi \in T(x)$ or $\langle\beta\rangle\xi \in T(x)$;
   (d) if $\gamma = \varphi?$, then both $\varphi \in T(x)$ and $\xi \in T(x)$;
   (e) if $\gamma = \alpha^*$, then there exists a word $w = w_1 \cdots w_m \in CS(\alpha^*)$ and $u_0, \ldots, u_m \in [k]^*$ such that $u_0 = x$, $\xi \in T(u_m)$, and for all $1 \leq i \leq m$, $\Phi(u_i) = \langle\alpha^*\rangle\xi$; moreover, if w_i is $\varphi?$, then $\varphi \in T(u_{i-1})$ and $u_i = u_{i-1}$, and if w_i is $a_j \in \Pi_0$, then $u_i = u_{i-1}r$, where $r = j + n\ell < k$ for some ℓ;
   (f) if $\gamma = L$, then there exists a word $w = w_1 \cdots w_m \in L$ and $u_0, \ldots, u_m \in [k]^*$ such that $u_0 = x$, $\xi \in T(u_m)$, and for all $1 \leq i \leq m$, $\Phi(u_i) = \langle L\rangle\xi$; moreover, if $w_i = a_j \in \Pi_0$, then $u_i = u_{i-1}r$, where $r = j + n\ell < k$ for some ℓ;
4. if $[\gamma]\xi \in T(x)$, and

(a)if γ is an atomic program a_j, then for all $r = j + n\ell < k$, if $T(xr) \neq \{\bot\}$ then $\xi \in T(xr)$;

(b)if $\gamma = \alpha\,;\,\beta$, then $\texttt{[}\alpha\texttt{]}\texttt{[}\beta\texttt{]}\xi \in T(x)$;

(c)if $\gamma = \alpha \cup \beta$, then both $\texttt{[}\alpha\texttt{]}\xi \in T(x)$ and $\texttt{[}\beta\texttt{]}\xi \in T(x)$;

(d)if $\gamma = \varphi?$ and if $\varphi \in T(x)$, then $\xi \in T(x)$;

(e)if $\gamma = \alpha^*$, then $\xi \in T(x)$ and $\texttt{[}\alpha\texttt{]}\texttt{[}\alpha^*\texttt{]}\xi \in T(x)$;

(f)if $\gamma = L$, then for all words $w = w_1 \cdots w_m \in L$ and $u_0, \ldots, u_m \in [k]^*$ such that $u_0 = x$ and for all $1 \leq i \leq m$, if $w_i = a_j \in \Pi_0$, then $u_i = u_{i-1}r$, where $r = j + n\ell < k$ for some ℓ, we have that either $T(u_m) = \{\bot\}$ or $\xi \in T(u_m)$.

PROPOSITION 9.16: A formula ψ in $\mathsf{PDL} + L$ has a UDH if and only if it has a model.

Proof Exercise 9.9. ■

Pushdown Automata on Infinite Trees

We now discuss pushdown automata on infinite trees. We show later that such an automaton accepts precisely the UDH's of some formula.

A *pushdown k-ary ω-tree automaton* (PTA) is a machine

$$M \quad = \quad (Q,\, \Sigma,\, \Gamma,\, q_0,\, z_0,\, \delta,\, F),$$

where Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite stack alphabet, $q_0 \in Q$ is the initial state, $z_0 \in \Gamma$ is the initial stack symbol, and $F \subseteq Q$ is the set of accepting states.

The transition function δ is of type

$$\delta : Q \times \Sigma \times \Gamma \quad \rightarrow \quad (2^{(Q \times B)^k} \cup 2^{Q \times B}),$$

where $B = \{\mathbf{pop}\} \cup \{\mathbf{push}(w) \mid w \in \Gamma^*\}$. The transition function reflects the fact that M works on trees with outdegree k that are labeled by Σ. The number of rules in δ is denoted by $|\delta|$.

A good informal way of viewing PTA's is as a pushdown machine that operates on an infinite tree of outdegree k. At each node u of the tree, the machine can read the input symbol $T(u)$ there. It can either stay at that node, performing some action on the stack and entering a new state as determined by an element of $Q \times B$; or it can split into k copies, each copy moving down to one of the k children of u, as determined by an element of $(Q \times B)^k$.

The set of *stack configurations* is $S = \{\gamma z_0 \mid \gamma \in \Gamma^*\}$. The top of the stack is to the left. The *initial stack configuration* is z_0. A *configuration* is a pair $(q, \gamma) \in Q \times S$. The *initial configuration* is (q_0, z_0). Let head : $S \to \Gamma$ be a function given by $\text{head}(z\gamma) = z$. This describes the letter on top of the stack. If the stack is empty, then head is undefined.

In order to capture the effect of δ on stack configurations, we define the partial function apply : $B \times S \to S$ that provides the new contents of the stack:

$$\begin{aligned} \text{apply}(\mathbf{pop}, z\gamma) &\stackrel{\text{def}}{=} \gamma; \\ \text{apply}(\mathbf{push}(w), \gamma) &\stackrel{\text{def}}{=} w\gamma. \end{aligned}$$

The latter includes the case $w = \varepsilon$, in which case the stack is unchanged; we abbreviate $\mathbf{push}(\varepsilon)$ by $\mathbf{sp}$.

The automaton M runs on complete labeled k-ary trees over Σ. That is, the input consists of the complete k-ary tree $[k]^*$ with labeling function

$$T : [k]^* \to \Sigma.$$

We denote the labeled tree by T. A *computation* of M on input T is a labeling

$$C : [k]^* \to (Q \times S)^+$$

of the nodes of T with sequences of configurations satisfying the following conditions. If $u \in [k]^*$, $T(u) = a \in \Sigma$, and $C(u) = ((p_0, \gamma_0), \ldots, (p_m, \gamma_m))$, then

- $(p_{i+1}, b_{i+1}) \in \delta(p_i, a, \text{head}(\gamma_i))$ and $\text{apply}(b_{i+1}, \gamma_i) = \gamma_{i+1}$ for $0 \le i < m$; and
- there exists $((r_0, b_0), \ldots, (r_{k-1}, b_{k-1})) \in \delta(p_m, a, \text{head}(\gamma_m))$ such that for all $0 \le j < k$, the first element of $C(uj)$ is $(r_j, \text{apply}(b_j, \gamma_m))$.

Intuitively, a computation is an inductive labeling of the nodes of the tree $[k]^*$ with configurations of the machine. The label of a node is the sequence of configurations that the machine goes through while visiting that node.

A computation C is said to be *Büchi accepting*, or just *accepting* for short, if the first configuration of $C(\varepsilon)$ is the start configuration (q_0, z_0) and every path in the tree contains infinitely many nodes u such that $q \in F$ for some $(q, \gamma) \in C(u)$. A tree T is *accepted* by M if there exists an accepting computation of M on T.

The *emptiness problem* is the problem of determining whether a given automaton M accepts some tree.

A PTA that uses only the symbol $\mathbf{sp}$ from B (that is, never **push**es nor **pop**s) is simply a Büchi k-ary ω-tree automaton as defined in Vardi and Wolper (1986a). Our definition is a simplified version of the more general definition of stack tree

automata from Harel and Raz (1994) and is similar to that appearing in Saudi (1989). If $k = 1$, the infinite trees become infinite sequences.

Our main result is the following.

THEOREM 9.17: The emptiness problem for PTA's is decidable.

The proof in Harel and Raz (1994) establishes decidability in 4-fold exponential time for STA's and in triple-exponential time for PTA's. A single-exponential-time algorithm for PTA's is given by Peng and Iyer (1995).

Decidability for Simple-Minded Languages

Given a simple-minded CFL L, we now describe the construction of a PTA A_ψ for each ψ in $\mathsf{PDL} + L$. This PTA will be shown to accept precisely the UDH trees of the formula ψ. The PTA A_ψ is a parallel composition of three machines. The first, called A_ℓ, is a tree automaton with no stack that tests the input tree for local consistency properties. The second component of A_ψ, called $A_\Box$, is a tree PDA that deals with box formulas that contain L. The third component, called $A_\Diamond$, is a tree PDA that deals with the diamond formulas of $CL(\psi)$.

Let $M_L = (Q, \Sigma, \Gamma, q_0, z_0, \rho)$ be a simple-minded PDA that accepts the language L, and let ψ be a formula in $\mathsf{PDL} + L$. Define the function $\Omega : \Sigma \times \Gamma \to \Gamma^*$ by: $\Omega(a, z) = w$ if there exist $p, q \in Q$ such that $\delta(p, a, z) = (q, w)$. Note that for a simple-minded PDA, Ω is a partial function.

The *local automaton* for ψ is

$$A_\ell \stackrel{\text{def}}{=} (2^{CL^\perp(\psi)},\ 2^{CL^\perp(\psi)},\ N_\psi,\ \delta,\ 2^{CL^\perp(\psi)}),$$

where:

- $CL^\perp(\psi) = CL(\psi) \cup \{\perp\}$;
- the starting set N_ψ consists of all sets s such that $\psi \in s$;
- $(s_0, \ldots, s_{k-1}) \in \delta(s, a)$ iff $s = a$, and
  - –either $s = \{\perp\}$ or $\perp \notin s$, and in the latter case, $\xi \in s$ iff $\neg\xi \notin s$;
  - –if $\xi \to \sigma \in s$ and $\xi \in s$, then $\sigma \in s$, and $\xi \wedge \sigma \in s$ iff both $\xi \in s$ and $\sigma \in s$;
  - –if $\langle\gamma\rangle\xi \in s$, then:
    - *if γ is an atomic program a_j, then there exists $r = j + n\ell < k$ for some ℓ such that $\xi \in s_r$;
    - *if $\gamma = \alpha\,;\,\beta$, then $\langle\alpha\rangle\langle\beta\rangle\xi \in s$;
    - *if $\gamma = \alpha \cup \beta$, then either $\langle\alpha\rangle\xi \in s$ or $\langle\beta\rangle\xi \in s$;

∗if $\gamma = \varphi?$ then both $\varphi \in s$ and $\xi \in s$;

–if $[\gamma]\xi \in s$, then:

∗if γ is an atomic program a_j, then for all $r = j + n\ell < k$, if $s_r \neq \{\bot\}$ then $\xi \in s$;

∗if $\gamma = \alpha\,;\,\beta$, then $[\alpha][\beta]\xi \in s$;

∗if $\gamma = \alpha \cup \beta$, then both $[\alpha]\xi \in s$ and $[\beta]\xi \in s$;

∗if $\gamma = \varphi?$ and $\varphi \in s$, then $\xi \in s$;

∗if $\gamma = \alpha^*$, then both $\xi \in s$ and $[\alpha][\alpha^*]\xi \in s$.

PROPOSITION 9.18: The automaton A_ℓ accepts precisely the trees that satisfy conditions 1, 2, 3(a)–(d), and 4(a)–(e) of Definition 9.15.

Proof A computation of an automaton M on an infinite tree $T : [k]^* \to \Sigma$ is an infinite tree $C : [k]^* \to Q'$, where Q' is the set of states of M. Clearly, if T satisfies conditions 1, 2, 3(a)–(d) and 4(a)–(e) of Definition 9.15, then T is also an accepting computation of A_ℓ on T.

Conversely, if C is an accepting computation of A_ℓ on some tree T, then C is itself an infinite tree over $2^{CL^\bot(\psi)}$ that satisfies the desired conditions. By the first rule of A_ℓ, for every node a we have $a = s$, hence $T = C$, and T satisfies conditions 1, 2, 3(a)–(d), and 4(a)–(e) of Definition 9.15. ■

The aim of the the next component of A_ψ is to check satisfaction of condition 4(f) of Definition 9.15, the condition that deals with box formulas containing the symbol L.

The *box automaton* for ψ is

$$A_\Box \stackrel{\text{def}}{=} (Q_\Box,\, 2^{CL^\bot(\psi)},\, \Gamma \times 2^{CL^\bot(\psi)},\, q_0,\, (z_0, \varnothing),\, \delta,\, Q_\Box),$$

where $Q_\Box = Q$ and δ is given by: $((p_0, w_0), \ldots, (p_{k-1}, w_{k-1})) \in \delta(q, \mathbf{a}, (z, \mathbf{s}))$ iff

1. either $a = \bot$ or $\mathbf{s} \subseteq \mathbf{a}$, and
2. for all $0 \leq j < n$ and for all $i = j + n\ell < k$ we have:

(a)if $\rho(q, a_j, z) = (q', \varepsilon)$, then $p_i = q'$ and $w_i = \varepsilon$,

(b)if $\rho(q, a_j, z) = (q', z)$, then $p_i = q'$ and $w_i = (z, \mathbf{s} \cup \mathbf{s}')$,

(c)if $\rho(q, a_j, z) = (q', zz')$, then $p_i = q'$ and $w_i = (z, \mathbf{s} \cup \mathbf{s}'), (z', \varnothing)$, and

(d)if $\rho(q, a_j, z)$ is undefined, then

i.if $\rho(q_0, a_j, z_0)$ is undefined, then $p_i = q_0$ and $w_i = (z_0, \varnothing)$;

ii.if $\rho(q_0, a_j, z_0) = (q', z_0)$, then $p_i = q'$ and $w_i = (z_0, \mathbf{s}')$; and

iii.if $\rho(q_0, a_j, z_0) = (q', z_0 z)$, then $p_i = q'$ and $w_i = (z_0, \mathbf{s}'), (z', \varnothing)$.

Here, if $\rho(q_0, a_j, z_0)$ is defined and $\texttt{[}L\texttt{]}\xi \in \mathbf{a}$, then $\xi \in \mathbf{s}'$, otherwise $\mathbf{s}' = \varnothing$.

In clause 1 we check whether old box promises that involve the language L are kept, while in clause 2 we put new such box promises on the stack to be checked later on. Note that the stack behavior of $A_\Box$ depends only on the path in the tree and not on the values of the tree nodes.

LEMMA 9.19: Let $x \in [k]^*$ and $T : [k]^* \to 2^{CL^\perp(\psi)}$, and let

$$C_\Box(x) \stackrel{\text{def}}{=} (q, (z_0, \mathbf{s}_0), \ldots, (z_m, \mathbf{s}_m)),$$

where $C_\Box$ is a computation of $A_\Box$ over T. Then for each $w = a_{j_1} \cdots a_{j_\ell} \in L$ and $r_m = j_m + n\ell_m < k$, the following two conditions hold:

- $C_\Box(x r_1 \cdots r_\ell) = (q', (z_0, \mathbf{s}_0), \ldots, (z_{m-1}, \mathbf{s}_{m-1}), (z_m, \mathbf{s}'_m))$;
- $\mathbf{s}'_m$ contains all formulas ξ for which $\texttt{[}L\texttt{]}\xi \in T(x)$.

Proof Define $\psi_m : Q \times (\Gamma \times 2^{CL^\perp(\psi)})^+ \to Q_\Box \times \Gamma^+$ by

$$\psi(q, (z_0, s_0) \cdots (z_m, s_m) \cdots (z_r, s_r)) \stackrel{\text{def}}{=} (q, z_m, \ldots, z_r).$$

Let $(q_0, \gamma_0) \cdots (q_\ell, \gamma_\ell)$ be a computation of M that accepts w. Since w is in L, for all $r_1 = j_1 + n\ell_1 < k$ we have that $\delta(q_0, a_j, z_0)$ is defined; hence by the definition of $\mathbf{s}'$ in $A_\Box$ we have that

$$C_\Box(x r_1) = (q', (z_0, \mathbf{s}_0), \ldots, (z_m, \mathbf{s}'_m), \gamma'),$$

where γ' may be empty and $\mathbf{s}'_m$ contains all formulas ξ such that $\texttt{[}L\texttt{]}\xi \in T(x)$.

We proceed by induction on i to prove that $\psi_m(C_\Box(x r_1 \cdots r_i)) = (q_i, \gamma_i)$ for all $1 \le i \le \ell$. The base case has just been established, and the general case follows immediately from the definition of $A_\Box$. For $i = \ell$, this proves the lemma. ■

PROPOSITION 9.20: The box automaton $A_\Box$ accepts precisely the trees that satisfy condition 4(f) of Definition 9.15.

Proof We must show that $A_\Box$ has an accepting computation over some tree T iff for all $x \in [k]^*$ the following holds: if $\texttt{[}L\texttt{]}\xi \in T(x)$, then for all $r_m = j_m + n\ell_m < k$, we have $\xi \in T(x r_1 \cdots r_\ell)$ or $T(x r_1 \cdots r_\ell) = \{\perp\}$.

($\Longrightarrow$) Suppose for a contradiction that there exist $x_0 \in [k]^*$, $\texttt{[}L\texttt{]}\xi \in T(x_0)$, and $w = a_{j_1}, \ldots, a_{j_\ell} \in L$ such that $T(xr_1 \cdots r_\ell) \neq \{\bot\}$ and $\xi \notin T(xr_1 \cdots r_\ell)$ for some $r_m = j_m + n\ell_m < k$. Let C be any computation of $A_\Box$. By Lemma 9.19, we know that $C(xr_1 \cdots r_\ell) = (q', (z_0, \mathbf{s}'_0) \cdots (z_m, \mathbf{s}'_m))$, and $\xi \in \mathbf{s}'_m$. This yields a contradiction to our assumption, since clause 1 in the definition of $A_\Box$ requires that $\mathbf{s} \subseteq \mathbf{a}$, which implies $\xi \in T(xr_1 \cdots r_\ell)$.

($\Longleftarrow$) If T satisfies the above condition and at each stage of the computation we add to $\mathbf{s}'$ exactly all ξ for which $\texttt{[}L\texttt{]}\xi \in T(x)$ when $\delta(q_0, a_j, z_0)$ is defined and add $\varnothing$ otherwise, we obtain an infinite computation of $A_\Box$ over T. This computation is accepting because $F_\Box = Q_\Box$. ■

The third component of A_ψ deals with diamond formulas. Note that unlike the box case, some diamond formulas are non-local in nature, thus cannot be handled by the local automaton. The special nature of UDH's is the key for the following construction, since it ensures that each diamond formula is satisfied along a unique path. All $A_\diamond$ must do is guess nondeterministically which successor lies on the appropriate path and check that there is indeed a finite path through that successor satisfying the diamond formula.

For technical reasons, we must define a finite automaton for each α such that $\texttt{<}\alpha^*\texttt{>}\xi \in CL(\psi)$ for some ξ. Define $\Sigma_\psi = \Pi \cup \{\varphi? \mid \varphi? \in CL(\psi)\}$, and let $M_\alpha = (Q_\alpha, \Sigma_\psi, q_{0\alpha}, \delta_\alpha, F_\alpha)$ be an automaton for $CS(\alpha)$.

The *diamond automaton* for ψ is

$$A_\diamond \stackrel{\text{def}}{=} (Q_\diamond, 2^{CL^\bot(\psi)}, \Gamma \times \{0,1\}, (1, \bot, \bot), (z_0, 0), \delta, F_\diamond),$$

where

- $Q_\diamond \stackrel{\text{def}}{=} \{0,1\} \times CL^\bot(\psi) \times (Q \cup \bigcup\{Q_\alpha \mid \texttt{<}\alpha^*\texttt{>}\xi \in CL(\psi) \text{ for some } \xi\})$. The first component is used to indicate acceptance, the second points to the diamond formula that is being verified or to $\bot$ if no such formula exists, and the third is used to simulate the computation of either M_L or M_α.
- $F_\diamond$ is the set of all triples in $Q_\diamond$ containing 1 in the first component or $\bot$ in the second.
- Define

$$\psi_M(a_j, z) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if} \quad \Omega(a_j, z) = \varepsilon \\ (z, 0) & \text{if} \quad \Omega(a_j, z) = z \\ (z, 0)(z', 1) & \text{if} \quad \Omega(a_j, z) = zz' \end{cases}$$

and

$$\psi_N(a_j, z) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if} \quad \Omega(a_j, z) = \varepsilon \\ (z, 1) & \text{if} \quad \Omega(a_j, z) = z \\ (z, 1)(z', 1) & \text{if} \quad \Omega(a_j, z) = zz'. \end{cases}$$

Then $((p_0, w_0), \ldots, (p_{k-1}, w_{k-1})) \in \delta((c, g, q), \mathbf{a}, (z, b))$ iff the following three conditions hold:

1. (a)for each $\langle\alpha\rangle^*\chi \in \mathbf{a}$, either $\chi \in \mathbf{a}$ or there exists $i = j + n\ell < k$ and a word $v = \varphi_1? \cdots \varphi_m?$ such that $\{\varphi_1, \ldots, \varphi_m\} \subseteq \mathbf{a}$ and $p_i = (c_i, \langle\alpha\rangle\chi, p)$, $p \in \delta_\alpha(q_{0\alpha}, va_j)$, and $w_i = \psi_M(a_j, z)$;

   (b)if $\langle L\rangle\chi \in \mathbf{a}$, then there exists $i = j + n\ell < k$ such that $p_i = (c_i, \langle L\rangle\chi, p)$, $p = \rho(q_{0\alpha}, a_j, z_0)$, and $w_i = \psi_N(a_j, z)$;

2. (a)if $\xi = \langle L\rangle\chi$, then either we are in an accepting state (that is, $c = 1$, $b = 0$, and $\chi \in \mathbf{a}$) or $c = 0$ and there exists $i = j + n\ell < k$ such that $p_i = (c_i, \langle L\rangle\chi, p)$, $p = \rho(q, a_j, z)$, $w_i = \psi_N(a_j, z)$, and if $w_i = \varepsilon$ then $b = 1$;

   (b)if $\xi = \langle\alpha\rangle\chi$, then there exists a word $v = \varphi_1? \cdots \varphi_m?$ such that $\{\varphi_1, \ldots, \varphi_m\} \subseteq \mathbf{a}$ and either we are in an accepting state (that is, $c = 1$, $\delta_\alpha(q, v) \in F_\alpha$, and $\chi \in \mathbf{a}$) or $c = 0$ and there exists $i = j + n\ell < k$ such that $p_i = (c_i, \langle\alpha\rangle\chi, p)$, $p \in \delta_\alpha(q_{0\alpha}, va_j)$, and $w_i = \psi_N(a_j, z)$;

3. for all $0 \leq j < n$ and $i = j + n\ell < k$, we have $w_i = \psi_N(a_j, z)$ or $w_i = \psi_M(a_j, z)$.

The idea here is much simpler than it might appear from the detailed construction. Condition 1 takes care of new diamond formulas. Each such formula is either satisfied in $\mathbf{a}$ or is written in the machine to be satisfied later. Condition 2 takes care of old promises which are either fulfilled or remain as promises in the machine. Condition 3 deals with the stack. We make sure that all stack operations coincide with those of M_L and use the extra bit on the stack to indicate the beginning of new simulations of M_L.

PROPOSITION 9.21: The automaton $A_\diamond$ accepts precisely the trees that satisfy both conditions 3(e) and 3(f) of Definition 9.15.

Proof Exercise 9.10. ■

LEMMA 9.22: There is a pushdown k-ary tree automaton A_ψ such that $L(A_\psi) = L(A_\ell) \cap L(A_\diamond) \cap L(A_\Box)$ and the size of A_ψ is at most $|A_\ell| \cdot |A_\diamond| \cdot |A_\Box|$.

Proof Define

$$A_\psi \stackrel{\text{def}}{=} (Q_\psi,\ 2^{CL^\perp(\psi)},\ \Gamma_\psi,\ q_{0\psi},\ z_{0\psi},\ \delta_\psi,\ F_\psi)$$

as follows:

$$\begin{aligned} Q_\psi &\stackrel{\text{def}}{=} Q_\ell \times Q_\Box \times Q_\Diamond \\ q_{0\psi} &\stackrel{\text{def}}{=} N_\psi \times q_{0\Box} \times q_{0\Diamond} \\ F_\psi &\stackrel{\text{def}}{=} Q_\ell \times Q_\Box \times F_\Diamond \\ \Gamma_\psi &\stackrel{\text{def}}{=} \Gamma_\Box \times \Gamma_\Diamond \\ z_{0\psi} &\stackrel{\text{def}}{=} z_{0\Box} \times z_{0\Diamond} \end{aligned}$$

and the transition function δ_ψ is the Cartesian product of the appropriate δ functions of the component automata.

Since all the states of both the local automaton and the box automaton are accepting states, and since we have taken the third component of A_ψ to be $F_\Diamond$, the accepted language is as required. Also, the size bound is immediate. We have only to show that this definition indeed describes a tree PDA; in other words, we have to show that the transition function δ_ψ is well defined. This is due to the simple-mindedness of the language L. More formally, for each $x \in [k]^*$ and each $i_m = j_m + n\ell_m < k$, the stack operations of $A_\Diamond$ are the same as the stack operations of $A_\Box$, since they both depend only on the letter a_{j_m}. ■

Lemma 9.22, together with the preceding results, yields:

PROPOSITION 9.23: Given a formula ψ in PDL + L, where L is a simple-minded CFL, one can construct a PTA A_ψ such that ψ has a model iff there is some tree T accepted by A_ψ.

Theorem 9.13 now follows.

Other Decidable Classes

Using techniques very similar to those of the previous proof, we can obtain another general decidability result involving languages accepted by deterministic stack automata. A stack automaton is a one-way PDA whose head can travel up and down the stack reading its contents, but can make changes only at the top of the stack. Stack automata can accept non-context-free languages such as $a^\Delta b^\Delta c^\Delta$ and its generalizations $a_1^\Delta a_2^\Delta \ldots a_n^\Delta$ for any n, as well as many variants thereof. It would

be nice to be able to prove decidability of PDL when augmented by any language accepted by such a machine, but this is not known. What has been proven, however, is that if each word in such a language is preceded by a new symbol to mark its beginning, then the enriched PDL is decidable:

THEOREM 9.24: Let $e \notin \Pi_0$, and let L be a language over Π_0 that is accepted by a deterministic stack automaton. If we let eL denote the language $\{eu \mid u \in L\}$, then $\mathsf{PDL} + eL$ is decidable.

While Theorems 9.13 and 9.24 are general and cover many languages, they do not prove decidability of $\mathsf{PDL} + a^{\Delta}b^{\Delta}c^{\Delta}$, which may be considered the simplest non-context-free extension of PDL. Nevertheless, the constructions used in the proofs of the two general results have been combined to yield:

THEOREM 9.25: $\mathsf{PDL} + a^{\Delta}b^{\Delta}c^{\Delta}$ is decidable.

9.5 More on One-Letter Programs

A Decidable Case

The results of the previous section provide sufficient conditions for an extension of PDL with a nonregular language to remain decidable.[3] If we consider one-letter languages, none of these results apply. Theorem 9.13 involves context-free languages, and by Parikh's theorem (see Kozen (1997a)) nonregular one-letter languages cannot be context-free; Theorem 9.24 involves adding a new letter to each word, and therefore does not apply to one-letter languages; and Theorem 9.25 talks about a specific three-letter language. The only negative results on one-letter extensions are those of Theorems 9.7 and Theorem 9.8, in which the words grow exponentially. We have no negative results for languages with subexponential growth. However, we do have a recent positive result, which we now describe.

The aim was to prove that the squares extension, $\mathsf{PDL} + a^{*^2}$, is decidable. The basic idea is to take advantage of the fact that the *difference sequence* of the squares language is linear and is in fact very simple: $(n+1)^2 - n^2 = 2n+1$. We exploit this in a construction similar in ways to that of Section 9.4, but using stack automata instead of pushdown automata. For technical reasons, the proof as it stands at the time of writing falls short of being applicable to the full $\mathsf{PDL} + a^{*^2}$. Accordingly, we

3 There are other decidable extensions that do not satisfy these conditions, so we do not have a tight characterization.

have had to restrict somewhat the context in which the squares language appears in formulas. Here is a definition of a restricted version of PDL + a^{*^2}, which we call Restricted-PDL + a^{*^2}.

Denote by L the squares language a^{*^2}. It is easy to see that $L^* = a^*$. Also, for any infinite regular language α over the alphabet $\{a\}$, the concatenation $L\alpha$ is regular (Exercise 9.14).

Now, given a formula φ, we say that φ is *clean* if L does not appear in φ. We say that L *appears simply* in φ (or in a program α) if all its appearances are either alone (that is, as the sole program within a box or diamond) or concatenated with a finite language over $\{a\}$ and then combined as a union with some regular language over $\{a\}$, as for example $Laa \cup (aa)^*$. A *nice box formula* is a formula of the form $[\alpha]\varphi$, where φ is clean and L appears simply in α. A regular expression α is said to be *unrestricted* if $\alpha \subseteq \{a, L\}^*$.

We now define inductively the set of formulas Φ in our PDL extension Restricted-PDL + a^{*^2}:

- $p, \neg p \in \Phi$ for all atomic propositions p;
- $[\alpha]\varphi \in \Phi$ whenever $\varphi \in \Phi$ and at least one of the following holds:
  - –both α and φ are clean,
  - –$[\alpha]\varphi$ is a nice box-formula,
  - –α is clean and φ is a nice box-formula;
- $<\alpha>\varphi \in \Phi$ whenever $\varphi \in \Phi$ and α is unrestricted;
- $\varphi \vee \psi \in \Phi$ whenever $\varphi, \psi \in \Phi$;
- $\varphi \wedge \psi \in \Phi$ whenever $\varphi, \psi \in \Phi$ and at least one of the following holds:
  - –either φ or ψ is clean,
  - –φ and ψ are nice box-formulas.

We now have:

THEOREM 9.26: Restricted-PDL + a^{*^2} is decidable.

Cases with no Finite Model Property

As explained, we know of no undecidabile extension of PDL with a polynomially growing language, although we conjecture that the cubes extension is undecidable. Since the decidability status of such extensions seems hard to determine, we now address a weaker notion: the presence or absence of a finite model property. The

technique used in Theorem 9.5 to show that $\mathsf{PDL} + a^{\Delta}b^{\Delta}$ violates the finite model property uses the two-letter comb-like model of Fig. 9.1, thus does not work for one-letter alphabets. Nevertheless, we now prove a general result leading to many one-letter extensions that violate the finite model property. In particular, the theorem will yield the following:

PROPOSITION 9.27 (SQUARES AND CUBES): The logics $\mathsf{PDL}+a^{*^2}$ and $\mathsf{PDL}+a^{*^3}$ do not have the finite model property.

Let us now prepare for the theorem.

DEFINITION 9.28: For a program β over Π_0 with $a \in \Pi_0$, we let $\mathrm{n}(\beta)$ denote the set $\{i \mid a^i \in CS(\beta)\}$. For $S \subseteq \mathbb{N}$, we let a^S denote the set $\{a^i \mid i \in S\}$; hence $\mathrm{n}(a^S) = S$.

THEOREM 9.29: Let $S \subseteq \mathbb{N}$. Suppose that for some program β in $\mathsf{PDL}+a^S$ with $CS(\beta) \subseteq a^*$, the following conditions are satisfied:

(i) there exists n_0 such that for all $x \geq n_0$ and $i \in \mathrm{n}(\beta)$,

$x \in S \implies x + i \notin S$;

(ii) for every $\ell, m > 0$, there exists $x, y \in S$ with $x > y \geq \ell$ and $d \in \mathrm{n}(\beta)$ such that $(x - y) \equiv d \pmod{m}$.

Then $\mathsf{PDL} + a^S$ does not have the finite model property.

Proof Every infinite path in a finite model must "close up" in a circular fashion. Thus, formulas satisfied along such a path must exhibit some kind of periodicity. Let S and β satisfy the conditions of the theorem. We use the nonperiodic nature of the set S given in property (i) in the statement of the theorem to construct a satisfiable formula φ in $\mathsf{PDL} + a^S$ that has no finite model.

Let φ be the conjunction of the following three formulas:

$\varphi_1 \stackrel{\text{def}}{=} \texttt{[}a^*\texttt{]<}a\texttt{>}\mathbf{1}$

$\varphi_2 \stackrel{\text{def}}{=} \texttt{[}a^S\texttt{]}p$

$\varphi_3 \stackrel{\text{def}}{=} \texttt{[}a^{n_0}\texttt{][}a^*\texttt{]}(p \to \texttt{[}\beta\texttt{]}\neg p)$.

Here n_0 is the constant from (i) and a^{n_0} is written out in full.

To show that φ is satisfiable, take the infinite model consisting of a sequence of

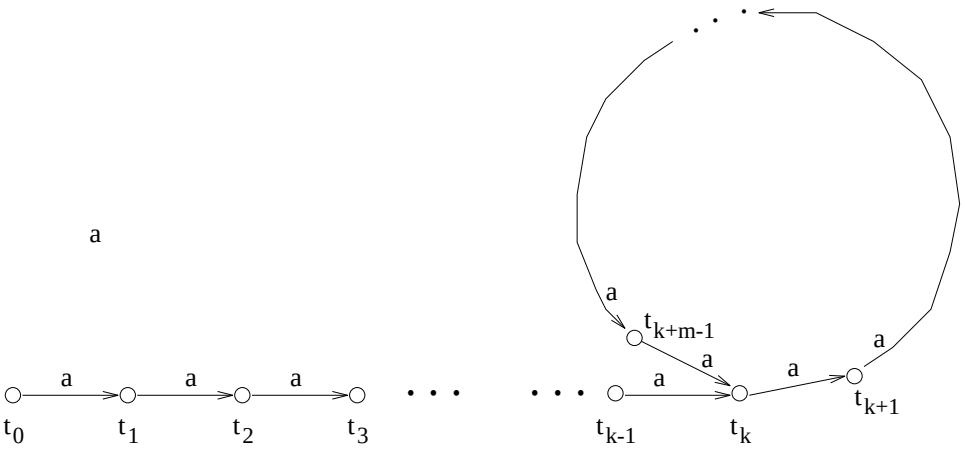


Figure 9.4

states $t_0, t_1, \ldots$ connected in order by the atomic program a. Assign p true in t_i iff $i \in S$. Then $t_0 \vDash \varphi$, since (i) guarantees that φ_3 holds in t_0.

We now show that φ does not have a finite model. Suppose $\mathfrak{K}, u_0 \vDash \varphi$ for some finite model $\mathfrak{K}$. By φ_1 and the finiteness of $\mathfrak{K}$, there must be a path in $\mathfrak{K}$ of the form shown in Fig. 9.4, where m denotes the size of the cycle. For every $z \in \mathbb{N}$, let z' be the remainder of $(z - k)$ when divided by m. Note that for $z \geq k$, the state $t_{k+z'}$ can be reached from t_0 by executing the program a^z.

By property (ii) in the statement of the theorem, we can find $x, y \in S$ and $d \in \mathrm{n}(\beta)$ such that

$$x > y > \max(n_0, k) \quad \text{and} \quad (x - y) \equiv d \pmod{m}.$$

That φ_2 holds at t_0 implies that $t_{k+y'} \vDash p$ and $t_{k+x'} \vDash p$. Since $y > n_0$, it follows from φ_3 that $t_{k+(y+d)'} \vDash \neg p$. However, $(x - y) \equiv d \pmod{m}$ implies that $(y + d)' = x'$, so that $t_{k+x'} \vDash \neg p$, which is a contradiction. ■

It is sometimes useful to replace condition (ii) of Theorem 9.29 with a weaker condition, call it (ii′), in which the consequent does not have to hold for every modulus m, but only for every $m >= m_0$ for some fixed m_0 (Exercise 9.12).

Now to some corollaries of the theorem. First, we prove the "squares" part of Proposition 9.27.

Proof Let $S_{\mathrm{sqr}} = \{i^2 \mid i \in \mathbb{N}\}$. To satisfy Theorem 9.29(i), take $n_0 = 1$ and $\beta = a$;

thus $\mathrm{n}(\beta) = \{1\}$. As for property (ii) of that theorem, given $\ell, m > 0$, let $d = 1$ and choose $y = (qm)^2 > \ell$ and $x = (qm+1)^2$. Then $x, y \in S_{\mathrm{sqr}}$, $x > y \geq \ell$, and $x - y = (qm+1)^2 - (qm)^2 \equiv d \pmod{m}$. ■

In fact, all polynomials of degree 2 or more exhibit the same property:

PROPOSITION 9.30 (POLYNOMIALS): For every polynomial of the form

$$p(n) \quad = \quad c_i n^i + c_{i-1} n^{i-1} + \cdots + c_0 \in \mathbb{Z}[n]$$

with $i \geq 2$ and positive leading coefficient $c_i > 0$, let $S_p = \{p(m) \mid m \in \mathbb{N}\} \cap \mathbb{N}$. Then $\mathsf{PDL} + a^{S_p}$ does not have the finite model property.

Proof To satisfy the conditions of Theorem 9.29, choose j_0 such that $p(j_0) - c_0 > 0$. Take β such that $\mathrm{n}(\beta) = \{p(j_0) - c_0\}$. Find some n_0 such that each $x \geq n_0$ will satisfy $p(x+1) - p(x) > p(j_0) - c_0$. This takes care of property (i) of the theorem.

Now, given $\ell, m > 0$, for $d = p(j_0) - c_0$, $y = p(qm) > \ell$, and $x = p(q'm + j_0) > y$, we have $x - y = p(q'm + j_0) - p(qm) \equiv p(j_0) - c_0 \pmod{m}$. ■

PROPOSITION 9.31 (SUMS OF PRIMES): Let p_i be the i^{th} prime (with $p_1 = 2$), and define

$$S_{\mathrm{sop}} \quad \stackrel{\mathrm{def}}{=} \quad \{\sum_{i=1}^{n} p_i \mid n \geq 1\}.$$

Then $\mathsf{PDL} + a^{S_{\mathrm{sop}}}$ does not have the finite model property.

Proof Clearly, property (i) of Theorem 9.29 holds with $n_0 = 3$ and $\beta = a$. To see that (ii) holds, we use a well known theorem of Dirichlet to the effect that there are infinitely many primes in the arithmetic progression $s + jt$, $j \geq 0$, if and only if $\gcd(s,t) = 1$. Given $\ell, m > 0$, find some i_0 such that $p_{i_0 - 1} > \ell$ and $p_{i_0} \equiv 1 \pmod{m}$. The existence of such a p_{i_0} follows from Dirichlet's theorem applied to the arithmetic progression $1 + jm$, $j \geq 0$.

Now let $d = 1$, $y = \sum_{i=1}^{i_0 - 1} p_i$, and $x = \sum_{i=1}^{i_0} p_i$. Then $x, y \in S_{\mathrm{sop}}$, $x > y \geq \ell$, and $x - y = p_{i_0} \equiv d \pmod{m}$. ■

PROPOSITION 9.32 (FACTORIALS): Let $S_{\mathrm{fac}} \stackrel{\mathrm{def}}{=} \{n! \mid n \in \mathbb{N}\}$. Then $\mathsf{PDL} + a^{S_{\mathrm{fac}}}$ does not have the finite model property.

Proof Exercise 9.11. ■

Since undecidable extensions of PDL cannot satisfy the finite model property, there is no need to prove that the powers of a fixed k or the Fibonacci numbers violate the finite model property.

The finite model property fails for any sufficiently fast-growing integer linear recurrence, not just the Fibonacci sequence, although we do not know whether these extensions also render PDL undecidable. A k^{th}*-order integer linear recurrence* is an inductively defined sequence

$$\ell_n \stackrel{\text{def}}{=} c_1\ell_{n-1} + \cdots + c_k\ell_{n-k} + c_0, \quad n \geq k, \tag{9.5.1}$$

where $k \geq 1$, $c_0, \ldots, c_k \in \mathbb{N}$, $c_k \neq 0$, and $\ell_0, \ldots, \ell_{k-1} \in \mathbb{N}$ are given.

Proposition 9.33 (linear recurrences): Let $S_{\text{lr}} = \{\ell_n \mid n \geq 0\}$ be the set defined inductively by (9.5.1). The following conditions are equivalent:

(i) $a^{S_{\text{lr}}}$ is nonregular;
(ii) PDL $+ a^{S_{\text{lr}}}$ does not have the finite model property;
(iii) not all $\ell_0, \ldots, \ell_{k-1}$ are zero and $\sum_{i=1}^{k} c_i > 1$.

Proof Exercise 9.13. ■

9.6 Bibliographical Notes

The main issues discussed in this chapter—the computational difficulty of the validity problem for nonregular PDL and the borderline between the decidable and undecidable—were raised in Harel et al. (1983). The fact that any nonregular program adds expressive power to PDL, Theorem 9.3, first appeared explicitly in Harel and Singerman (1996). Theorem 9.4 on the undecidability of context-free PDL was observed by Ladner (1977).

Theorems 9.5 and 9.6 are from Harel et al. (1983), but the proof of Theorem 9.6 using tiling is taken from Harel (1985). The existence of a primitive recursive one-letter extension of PDL that is undecidable was shown already in Harel et al. (1983), but undecidability for the particular case of a^{2^*}, Theorem 9.7, is from Harel and Paterson (1984). Theorem 9.8 is from Harel and Singerman (1996).

As to decidable extensions, Theorem 9.9 was proved in Koren and Pnueli (1983). The more general results of Section 9.4, namely Theorems 9.13, 9.24, and 9.25, are from Harel and Raz (1993), as is the notion of a simple-minded PDA. The decidability of emptiness for pushdown and stack automata on trees that is needed

for the proofs of these (Section 9.4) is from Harel and Raz (1994). A better bound on the complexity of the emptiness results can be found in Peng and Iyer (1995).

Theorem 9.29 is from Harel and Singerman (1996) and Theorem 9.26 is from Ferman and Harel (2000).

Exercises

9.1. Complete the proof of Theorem 9.3.

9.2. Consider the formula $\langle G\rangle p \leftrightarrow \langle G'\rangle p$ for context-free grammars G and G' over atomic programs $\{a_0, \ldots, a_{k-1}\}$. Show that this formula is valid iff $CS(G) = CS(G')$, where $CS(G)$ is the language over $\{a_0, \ldots, a_{k-1}\}$ generated by G.

9.3. Modify the proof of Theorem 9.5 to show that $\mathsf{PDL} + a^\Delta b a^\Delta$ does not have the finite model property.

9.4. Complete the proof of Theorem 9.6 by showing why (9.3.4) forces the recurrence of red.

9.5. Prove that PDL with the addition of both $a^\Delta b^\Delta$ and $b^\Delta a^\Delta$ is Π^1_1-complete.

9.6. Complete the proof of Theorem 9.7.

9.7. Show that $a^i b^{2i}$ is a simple-minded CFL.

9.8. Extend Example 9.12 to show that the language of balanced strings of parentheses over an alphabet with $k > 1$ pairs of different parentheses is simple-minded.

9.9. Prove Proposition 9.16.

9.10. Prove Proposition 9.21.

9.11. Prove Proposition 9.32.

9.12. Show that Theorem 9.29 still holds when condition (ii) is replaced by the weaker condition

(ii′) there exists an m_0 such that for every $m > m_0$ and $\ell > 0$, there exists $x, y \in S$

with $x > y \geq \ell$ and $d \in \mathrm{n}(\beta)$ such that $(x - y) \equiv d \pmod{m}$.

9.13. Show that the terms ℓ_n of a k^{th}-order integer linear recurrence of the form (9.5.1) grow either linearly or exponentially, and that condition (iii) of Proposition 9.33 is necessary and sufficient for exponential growth. Use this fact to prove the proposition. (*Hint.* Use Exercise 9.12.)

9.14. Prove that for any language L over the alphabet $\{a\}$ and any infinite regular language α over $\{a\}$, the concatenation language $L\alpha$ is regular.

10 Other Variants of PDL

A number of interesting variants are obtained by extending or restricting the standard version of PDL in various ways. In this section we describe some of these variants and review some of the known results concerning relative expressive power, complexity, and proof theory. These investigations are aimed at revealing the power of such programming features as recursion, testing, concurrency, and nondeterminism when reasoning on a propositional level.

The extensions and restrictions we consider are varied. One can require that programs be deterministic (Section 10.1), that tests not appear or be simple (Section 10.2), or that programs be expressed by finite automata (Section 10.3). We studied nonregular programs in Chapter 9; one can also augment the language of regular programs by adding operators for converse, intersection, or complementation (Sections 10.4 and 10.5), or the ability to assert that a program cannot execute forever (Section 10.6), or a form of concurrency and communication (Section 10.7).

Wherever appropriate, questions of expressiveness, complexity, and axiomatic completeness are addressed anew.

10.1 Deterministic PDL and While Programs

Nondeterminism arises in PDL in two ways:

- atomic programs can be interpreted in a structure as (not necessarily single-valued) binary relations on states; and
- the programming constructs $\alpha \cup \beta$ and α^* involve nondeterministic choice.

Many modern programming languages have facilities for concurrency and distributed computation, certain aspects of which can be modeled by nondeterminism. Nevertheless, the majority of programs written in practice are still deterministic. In this section we investigate the effect of eliminating either one or both of these sources of nondeterminism from PDL.

A program α is said to be (*semantically*) *deterministic* in a Kripke frame $\mathfrak{K}$ if its traces are uniquely determined by their first states. If α is an atomic program a, this is equivalent to the requirement that $\mathfrak{m}_{\mathfrak{K}}(a)$ be a partial function; that is, if both (s,t) and $(s,t') \in \mathfrak{m}_{\mathfrak{K}}(a)$, then $t = t'$. A *deterministic Kripke frame* $\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$ is one in which all atomic a are semantically deterministic.

The class of *deterministic* **while** *programs*, denoted DWP, is the class of programs in which

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

• tests in the conditional test and **while** loop are purely propositional; that is, there is no occurrence of the $\langle\ \rangle$ or $[\ ]$ operators.

The class of *nondeterministic* **while** *programs*, denoted WP, is the same, except unconstrained use of the nondeterministic choice construct $\cup$ is allowed. It is easily shown that if α and β are semantically deterministic in $\mathfrak{K}$, then so are **if** φ **then** α **else** β and **while** φ **do** α (Exercise 5.2).

By restricting either the syntax or the semantics or both, we obtain the following logics:

• DPDL (deterministic PDL), which is syntactically identical to PDL, but interpreted over deterministic structures only;

• SPDL (strict PDL), in which only deterministic **while** programs are allowed; and

• SDPDL (strict deterministic PDL), in which both restrictions are in force.

Validity and satisfiability in DPDL and SDPDL are defined just as in PDL, but with respect to deterministic structures only. If φ is valid in PDL, then φ is also valid in DPDL, but not conversely: the formula

$$\langle a\rangle\varphi \quad\rightarrow\quad [a]\varphi \tag{10.1.1}$$

is valid in DPDL but not in PDL. Also, SPDL and SDPDL are strictly less expressive than PDL or DPDL, since the formula

$$\langle (a \cup b)^*\rangle\varphi \tag{10.1.2}$$

is not expressible in SPDL, as shown in Halpern and Reif (1983).

THEOREM 10.1: If the axiom scheme

$$\langle a\rangle\varphi \quad\rightarrow\quad [a]\varphi, \quad a \in \Pi_0 \tag{10.1.3}$$

is added to Axiom System 5.5, then the resulting system is sound and complete for DPDL.

Proof sketch. The extended system is certainly sound, since (10.1.3) is a straightforward consequence of semantic determinacy.

Completeness can be shown by modifying the construction of Section 7.1 with some special provisions for determinancy. For example, in the construction leading

up to Lemma 7.3, we defined a nonstandard Kripke frame $\mathfrak{N}$ whose states were maximal consistent sets of formulas such that

$$\begin{aligned} \mathfrak{m}_{\mathfrak{N}}(a) &\stackrel{\text{def}}{=} \{(s,t) \mid \forall\varphi\ \varphi \in t \to \langle a\rangle\varphi \in s\} \\ &= \{(s,t) \mid \forall\varphi\ [a]\varphi \in s \to \varphi \in t\}. \end{aligned}$$

The structure $\mathfrak{N}$ produced in this way need not be deterministic, but it can be "unwound" into a treelike deterministic structure which satisfies the given satisfiable formula. ■

The proof sketched above also yields:

THEOREM 10.2: Validity in DPDL is deterministic exponential-time complete.

Proof sketch. The upper bound is shown in Ben-Ari et al. (1982). For the lower bound, a formula φ is valid in PDL iff φ' is valid in DPDL, where φ' is obtained from φ by replacing all atomic programs a by ab^* for some new atomic program b. The possibility of reaching many new states via a from some state s in PDL is modeled in DPDL by the possibility of executing b many times from the single state reached via a from s. The result follows from the linearity of this transformation. ■

Now we turn to SPDL, in which atomic programs can be nondeterministic but can be composed into larger programs only with deterministic constructs.

THEOREM 10.3: Validity in SPDL is deterministic exponential-time complete.

Proof sketch. Since we have restricted the syntax only, the upper bound carries over directly from PDL. For the lower bound, a formula φ of PDL is valid iff φ' is valid in SPDL, where φ' involves new nondeterministic atomic programs acting as "switches" for deciding when the tests that control the determinism of **if-then-else** and **while-do** statements are true. For example, the nondeterministic program α^* can be simulated in SPDL by the program $b; \textbf{while}\ p\ \textbf{do}\ (\alpha; b)$. ■

The final version of interest is SDPDL, in which both the syntactic restrictions of SPDL and the semantic ones of DPDL are adopted. Note that the crucial [NEXT*] that appears in the simulation of the alternating Turing machine in Section 8.2 can no longer be written as it is, because we do not have the use of the * construct, and it apparently cannot be simulated with nondeterministic atomic programs as above either. Indeed, the exponential-time lower bound fails here, and we have:

THEOREM 10.4: The validity problem for SDPDL is complete in polynomial space.

Proof sketch. For the upper bound, the following two nontrivial properties of formulas of SDPDL are instrumental:

(i) if φ is satisfiable, then it is satisfiable in a treelike structure with only polynomially many nodes at each level. (In Theorem 10.5 a counterexample for DPDL and PDL is given.)

(ii) if φ is satisfied in a treelike structure $\mathfrak{A}$, then $\mathfrak{A}$ can be collapsed into a finite structure by "bending" certain edges back to ancestors, resulting in a treelike structure with back edges of depth at most exponential in $|\varphi|$ that has no nested or crossing backedges.

The polynomial-space procedure attempts to construct a treelike model for a given formula by carrying out a depth-first search of potential structures, deciding nondeterministically whether or not to split nodes and whether or not to bend edges backwards. The size of the stack for such a procedure can be made to be polynomial in the size of the formula, since we have a treelike object of exponential depth but only polynomial width, hence exponential size. Savitch's theorem is then invoked to eliminate the nondeterminism while remaining in polynomial space.

For the lower bound, we proceed as in the proof of the lower bound for PDL in Theorem 8.5. Given a polynomial space-bounded one-tape deterministic Turing machine M accepting a set $L(M)$, a formula φ_x of SDPDL is constructed for each word x that simulates the computation of M on x. The formula φ_x will be polynomial-time computable and satisfiable iff $x \in K$. Since we do not have the program NEXT*, the entire formula constructed in the proof of Theorem 8.5 must be restructured, and will now take on the form

$$\langle \textbf{while } \neg\sigma \textbf{ do } (\text{NEXT}; \psi?)\rangle \mathbf{1},$$

where σ describes an accepting configuration of M and ψ verifies that configurations and transitions behave correctly. These parts of the formula can be constructed similarly to those used in the proof of Theorem 8.5. ■

The question of relative power of expression is of interest here. Is DPDL $<$ PDL? Is SDPDL $<$ DPDL? The first of these questions is inappropriate, since the syntax of both languages is the same but they are interpreted over different classes of structures. Considering the second, we have:

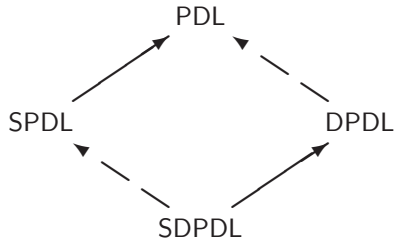
THEOREM 10.5: SDPDL $<$ DPDL and SPDL $<$ PDL.

Proof The DPDL formula

$$[(a \cup b)^*](\langle a\rangle \mathbf{1} \wedge \langle b\rangle \mathbf{1}) \tag{10.1.4}$$

is satisfied in the full infinite binary tree (with a modeled, say, by left transitions and b by right ones), but in no tree structure with polynomially many nodes at each level. This contradicts property (i) of SDPDL in the proof of Theorem 10.4. The argument goes through even if (10.1.4) is thought of as a PDL formula and is compared with SPDL. ■

In summary, we have the following diagram describing the relations of expressiveness between these logics. The solid arrows indicate added expressive power and broken ones a difference in semantics. The validity problem is exponential-time complete for all but SDPDL, for which it is *PSPACE*-complete. Straightforward variants of Axiom System 5.5 are complete for all versions.



10.2 Restricted Tests

Tests $\varphi?$ in PDL are defined for arbitrary propositions φ. This is sometimes called *rich test* PDL. Rich tests give substantially more power than one would normally find in a conventional programming language. For example, if φ is the formula $[\alpha]\psi$, the test $\varphi?$ in effect allows a program to pause during the computation and ask the question: "Had we run program α now, would ψ have been true upon termination?" without actually running α. For example, the formula $[([\alpha]p)?;\alpha]p$ is valid. In general, however, this kind of question would be undecidable.

A more realistic model would allow tests with Boolean combinations of atomic formulas only. This is called *poor-test* PDL.

To refine this distinction somewhat, we introduce a hierarchy of subsets of Φ determined by the depth of nesting of tests. We then establish that each level of the hierarchy is strictly more expressive than all lower levels.

Let $\Phi^{(0)}$ be the subset of Φ in which programs contain no tests. This actually means that programs are regular expressions over the set Π_0 of atomic programs. Now let $\Phi^{(i+1)}$ be the subset of Φ in which programs can contain tests $\varphi?$ only for $\varphi \in \Phi^{(i)}$. The logic restricted to formulas $\Phi^{(i)}$ is called $\mathsf{PDL}^{(i)}$. Clearly, $\Phi = \bigcup_i \Phi^{(i)}$, and we can also write $\mathsf{PDL} = \bigcup_i \mathsf{PDL}^{(i)}$. The logic $\mathsf{PDL}^{(0)}$ is sometimes called *test-free* PDL.

The language fragment $\mathsf{PDL}^{(1)}$ can test test-free formulas of PDL and these themselves can contain test-free programs. Poor-test PDL, which can test only Boolean combinations of atomic formulas, fits in between $\mathsf{PDL}^{(0)}$ and $\mathsf{PDL}^{(1)}$ (we might call it $\mathsf{PDL}^{(0.5)}$).

Since the lower bound proof of Theorem 8.5 does not make use of tests at all, the exponential time lower bound carries over even to $\mathsf{PDL}^{(0)}$, the weakest version considered here, and of course the upper bound of Section 8.1 holds too. Also, omitting axiom (vi) from Axiom System 5.5 yields a complete axiom system for $\mathsf{PDL}^{(0)}$.

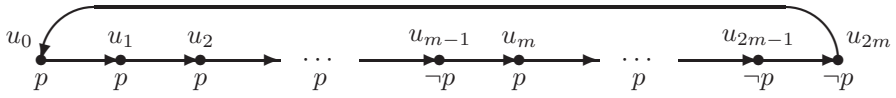
The question we now ask is whether even atomic tests add to the expressive power of PDL. The answer is affirmative.

THEOREM 10.6: $\mathsf{PDL}^{(0)} <$ poor-test PDL.

Proof sketch. Axioms 5.5(iv), (v), and (vi) enable one to eliminate from formulas all tests that do not appear under a * operator. Consequently, a proof of the theorem will have to make use of iterating a test. Let φ be the the poor-test PDL formula

$$\varphi \stackrel{\text{def}}{=} \langle (p?a)^*(\neg p)?a\rangle p$$

for atomic a and p. Consider the structure $\mathfrak{A}_m$ illustrated in the following figure, where arrows indicate a-transitions.



For $0 \le k < m$, $\mathfrak{A}_m, u_k \vDash \varphi$, but $\mathfrak{A}_m, u_{k+m} \nvDash \varphi$.

The rest of the proof is devoted to formalizing the following intuition. Without the ability to test p inside a loop, it is impossible to tell in general whether the current state belongs to the left- or right-hand portion of the structure, since it is always possible to proceed and find oneself eventually in the other portion.

To that end, it becomes necessary to see to it that a test-free a^* or $(a^i)^*$ program cannot distinguish between these possibilities. The constants m and k are therefore chosen carefully, taking into account the eventual periodicity of one-letter regular sets. Specifically, it can be shown that for any test-free formula ψ there are m and k such that $\mathfrak{A}_m, u_k \vDash \psi$ iff $\mathfrak{A}_m, u_{k+m} \vDash \psi$, hence φ cannot be equivalent to any formula of $\mathsf{PDL}^{(0)}$. ■

Theorem 10.6 can be generalized to

THEOREM 10.7: For every $i \geq 0$, $\mathsf{PDL}^{(i)} < \mathsf{PDL}^{(i+1)}$.

Proof sketch. The proof is very similar in nature to the previous one. In particular, let φ_0 be the φ of the previous proof with a_0 replacing a, and let φ_{j+1} be φ with a_{j+1} replacing a and φ_j replacing the atomic formula p. Clearly, $\varphi_i \in \Phi^{(i+1)} - \Phi^{(i)}$.

The idea is to build an elaborate multi-layered version of the structure $\mathfrak{A}_m$ described above, in which states satisfying p or $\neg p$ in $\mathfrak{A}_m$ now have transitions leading down to appropriate distinct points in lower levels of the structure. The lowest level is identical to $\mathfrak{A}_m$. The intuition is that descending a level in the structure corresponds to nesting a test in the formula. The argument that depth of nesting $i+1$ is required to distinguish between appropriately chosen states u_k and u_{k+m} is more involved but similar. ■

The proofs of these results make no essential use of nondeterminism and can be easily seen to hold for the deterministic versions of PDL from Section 10.1 (similarly refined according to test depth).

COROLLARY 10.8: For every $i \geq 0$, we have

$$\begin{aligned} \mathsf{DPDL}^{(i)} &< \mathsf{DPDL}^{(i+1)}, \\ \mathsf{SPDL}^{(i)} &< \mathsf{SPDL}^{(i+1)}, \\ \mathsf{SDPDL}^{(i)} &< \mathsf{SDPDL}^{(i+1)}. \end{aligned}$$

In fact, it seems that the ability to test is in a sense independent of the ability to branch nondeterministically. The proof of Theorem 10.5 uses no tests and therefore actually yields a stronger result:

THEOREM 10.9: There is a formula of $\mathsf{DPDL}^{(0)}$ (respectively, $\mathsf{PDL}^{(0)}$) that is not expressible in SDPDL (respectively SPDL).

We thus have the following situation: for nondeterministic structures,

$$\begin{array}{ccccccccc} \mathsf{SPDL}^{(0)} & < & \mathsf{SPDL}^{(1)} & < & \cdots & < & \mathsf{SPDL}, \\ \mathsf{PDL}^{(0)} & < & \mathsf{PDL}^{(1)} & < & \cdots & < & \mathsf{PDL}, \end{array}$$

and for deterministic structures,

$$\begin{array}{ccccccccc} \mathsf{SDPDL}^{(0)} & < & \mathsf{SDPDL}^{(1)} & < & \cdots & < & \mathsf{SDPDL} \\ \mathsf{DPDL}^{(0)} & < & \mathsf{DPDL}^{(1)} & < & \cdots & < & \mathsf{DPDL}. \end{array}$$

10.3 Representation by Automata

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

Since finite automata are exponentially more succinct in general, the upper bound of Section 8.1 could conceivably fail if finite automata were allowed as programs. Moreover, we must also rework the deductive system of Section 5.5.

However, it turns out that the completeness and exponential-time decidability results of PDL are not sensitive to the representation and still go through in the presence of finite automata as programs, provided the deductive system of Section 5.5 and the techniques of Chapter 7 and Section 8.1 are suitably modified, as shown in Pratt (1979b, 1981b) and Harel and Sherman (1985).

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

This connection has prompted renewed inquiry into the complexity of automata on infinite objects, with considerable success. See Courcoubetis and Yannakakis (1988); Emerson (1985); Emerson and Jutla (1988); Emerson and Sistla (1984); Manna and Pnueli (1987); Muller et al. (1988); Pecuchet (1986); Safra (1988); Sistla et al. (1987); Streett (1982); Vardi (1985a,b, 1987); Vardi and Stockmeyer (1985); Vardi and Wolper (1986c,b); Arnold (1997a,b); and Thomas (1997). Especially noteworthy in this area is the result of Safra (1988) involving the complexity

of converting a nondeterministic automaton on infinite strings into an equivalent deterministic one. This result has already had a significant impact on the complexity of decision procedures for several logics of programs; see Courcoubetis and Yannakakis (1988); Emerson and Jutla (1988, 1989); and Safra (1988).

We assume that nondeterministic finite automata are given in the form

$$M \quad = \quad (n,\, i,\, j,\, \delta), \tag{10.3.1}$$

where $\overline{n} = \{0, \ldots, n-1\}$ is the set of states, $i, j \in \overline{n}$ are the start and final states respectively, and δ assigns a subset of $\Pi_0 \cup \{\varphi? \mid \varphi \in \Phi\}$ to each pair of states. Intuitively, when visiting state ℓ and seeing symbol a, the automaton may move to state k if $a \in \delta(\ell, k)$.

The fact that the automata (10.3.1) have only one accept state is without loss of generality. If M is an arbitrary nondeterministic finite automaton with accept states F, then the set accepted by M is the union of the sets accepted by M_k for $k \in F$, where M_k is identical to M except that it has unique accept state k. A desired formula $\texttt{[}M\texttt{]}\varphi$ can be written as a conjunction

$$\bigwedge_{k \in F} \texttt{[}M_k\texttt{]}\varphi$$

with at most quadratic growth.

We now obtain a new logic APDL (*automata* PDL) by defining Φ and Π inductively using the clauses for Φ from Section 5.1 and letting $\Pi = \Pi_0 \cup \{\varphi? \mid \varphi \in \Phi\} \cup F$, where F is the set of automata of the form (10.3.1).

Exponential time decidability and completeness can be proved by adapting and generalizing the techniques used in Chapter 7 and Section 8.1 for PDL. We shall not supply full details here, except to make a couple of comments that will help give the reader the flavor of the adaptations needed.

There is an analogue $AFL(\varphi)$ of the Fischer–Ladner closure $FL(\varphi)$ of a formula φ defined in Section 6.1. The inductive clauses for $\alpha\,;\,\beta$, $\alpha \cup \beta$, and α^* are replaced by:

- if $\texttt{[}n, i, j, \delta\texttt{]}\psi \in AFL(\varphi)$, then for every $k \in \overline{n}$ and $\alpha \in \delta(i, k)$,

$\texttt{[}\alpha\texttt{]}\,\texttt{[}n, k, j, \delta\texttt{]}\psi \in AFL(\varphi)$;

- in addition, if $i = j$, then $\psi \in AFL(\varphi)$.

Axioms 5.5(iv), (v), and (vii) are replaced by:

$$[n,i,j,\delta]\varphi \quad\leftrightarrow\quad \bigwedge_{\substack{k\in\overline{n}\\ \alpha\in\delta(i,k)}} [\alpha][n,k,j,\delta]\varphi, \quad i\neq j \tag{10.3.2}$$

$$[n,i,i,\delta]\varphi \quad\leftrightarrow\quad \varphi\wedge \bigwedge_{\substack{k\in\overline{n}\\ \alpha\in\delta(i,k)}} [\alpha][n,k,i,\delta]\varphi. \tag{10.3.3}$$

The induction axiom 5.5(viii) becomes

$$(\bigwedge_{k\in\overline{n}} [n,i,k,\delta](\varphi_k \;\rightarrow\; \bigwedge_{\substack{m\in\overline{n}\\ \alpha\in\delta(k,m)}} [\alpha]\varphi_m)) \;\rightarrow\; (\varphi_i \;\rightarrow\; [n,i,j,\delta]\varphi_j). \tag{10.3.4}$$

These and other similar changes can be used to prove:

THEOREM 10.10: Validity in APDL is decidable in exponential time.

THEOREM 10.11: The axiom system described above is complete for APDL.

10.4 Complementation and Intersection

In previous sections we exploited the fact that programs in PDL are regular expressions, hence denote sets of computations recognizable by finite automata. Consequently, those operations on programs that do not lead outside the class of regular sets, such as the shuffle operator $\alpha \parallel \beta$ (of importance in reasoning about concurrent programs) need not be added explicitly to PDL. Thus the intersection of programs and the complement of a program are expressible in PDL by virtue of these operations being regular operations.

However, this is so only when the operations are regarded as being applied to the languages denoted by the programs, so that for example the intersection of α and β contains all execution sequences of atomic programs and tests contained in both. In this section we are interested in a more refined notion of such operations. Specifically, we consider the complementation and intersection of the binary relations on states denoted by programs. Let $-\alpha$ and $\alpha\cap\beta$ stand for new programs with semantics

$$\begin{aligned} \mathfrak{m}_{\mathfrak{K}}(-\alpha) &\stackrel{\text{def}}{=} (K\times K) - \mathfrak{m}_{\mathfrak{K}}(\alpha)\\ \mathfrak{m}_{\mathfrak{K}}(\alpha\cap\beta) &\stackrel{\text{def}}{=} \mathfrak{m}_{\mathfrak{K}}(\alpha)\cap\mathfrak{m}_{\mathfrak{K}}(\beta). \end{aligned}$$

It is clear that $\alpha \cap \beta$ can be defined as $-(-\alpha \cup -\beta)$, so we might have considered adding complementation only. However, for this case we have the following immediate result.

THEOREM 10.12: The validity problem for PDL with the complementation operator is undecidable.

Proof The result follows from the known undecidability of the equivalence problem for the algebra of binary relations with complementation. ■

However, it is of interest to consider the logic IPDL, defined as PDL with $\alpha \cap \beta$ in Π for each $\alpha, \beta \in \Pi$. The corresponding equivalence problem for binary relations is not known to be undecidable and can be shown to be no higher than Π^0_1 in the arithmetic hierarchy. This should be contrasted with Theorem 10.14 below. First, we establish the following.

THEOREM 10.13: There is a satisfiable formula of IPDL that has no finite model.

Proof sketch. Take α to be

$$[a^*](\langle a\rangle \mathbf{1} \wedge [a^* a \cap \mathbf{1}?]\mathbf{0}).$$

Satisfiability is seen to hold in an infinite a-path. The second conjunct, however, states that non-empty portions of a-paths do not bend backwards; therefore no two states on such an infinite path can be identical. ■

The following result is the strongest available. It concerns the version IDPDL of IPDL in which structures are deterministic.

THEOREM 10.14: The validity problem for IDPDL (hence also for DPDL with complementation of programs) is Π^1_1-complete.

Proof sketch. We reduce the recurring tiling problem of Proposition 2.22 to the satisfiability of formulas in IDPDL. First we construct a formula that forces its models to contain a (possibly cyclic) two-dimensional grid. This is done using atomic programs NORTH and EAST as follows:

$$[(\textsc{North} \cup \textsc{East})^*](\langle(\textsc{North}; \textsc{East}) \cap (\textsc{East}; \textsc{North})\rangle \mathbf{1}.$$

The proof then continues along the lines of the proof of Theorem 9.6. ■

It is interesting to observe that the techniques used in proving Theorem 10.14 do not seem to apply to the nondeterministic cases. It is not known at present whether IPDL is decidable, although it would be very surprising if were.

10.5 Converse

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

$$\mathfrak{m}_{\mathfrak{K}}(\alpha^-) \stackrel{\text{def}}{=} \{(s,t) \mid (t,s) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)\}.$$

PDL with converse is called CPDL.

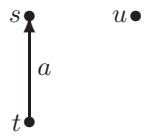
The following identities, proved valid in Theorem 5.12, allow us to assume without loss of generality that the converse operator is applied to atomic programs only.

$$\begin{aligned}(\alpha\,;\,\beta)^- &\leftrightarrow \beta^-\,;\,\alpha^-\\ (\alpha \cup \beta)^- &\leftrightarrow \alpha^- \cup \beta^-\\ \alpha^{*-} &\leftrightarrow \alpha^{-*}.\end{aligned}$$

The converse operator strictly increases the expressive power of PDL, since the formula $\langle\alpha^-\rangle\mathbf{1}$ is not expressible without it.

THEOREM 10.15: PDL $<$ CPDL.

Proof Consider the structure described in the following figure:



In this structure, $s \models \langle a^-\rangle\mathbf{1}$ but $u \not\models \langle a^-\rangle\mathbf{1}$. On the other hand, it can be shown by induction on the structure of formulas that if s and u agree on all atomic formulas, then no formula of PDL can distinguish between the two. ■

More interestingly, the presence of the converse operator implies that the operator $\langle\alpha\rangle$ is *continuous* in the sense that if A is any (possibly infinite) family of formulas possessing a join $\bigvee A$, then $\bigvee \langle\alpha\rangle A$ exists and is logically equivalent to $\langle\alpha\rangle \bigvee A$ (Theorem 5.14). In the absence of the converse operator, one can construct nonstandard models for which this fails (Exercise 5.12).

The completeness and exponential time decidability results of Chapter 7 and Section 8.1 can be extended to CPDL provided the following two axioms are added:

$$\varphi \rightarrow [\alpha]\langle\alpha^-\rangle\varphi$$
$$\varphi \rightarrow [\alpha^-]\langle\alpha\rangle\varphi.$$

The filtration lemma (Lemma 6.4) still holds in the presence of $^-$, as does the finite model property.

10.6 Well-Foundedness and Total Correctness

If α is a deterministic program, the formula $\varphi \rightarrow \langle\alpha\rangle\psi$ asserts the total correctness of α with respect to pre- and postconditions φ and ψ, respectively. For *nondeterministic* programs, however, this formula does not express the right notion of total correctness. It asserts that φ implies that *there exists* a halting computation sequence of α yielding ψ, whereas we would really like to assert that φ implies that *all* computation sequences of α terminate and yield ψ. Let us denote the latter property by

$$TC(\varphi, \alpha, \psi).$$

Unfortunately, this is not expressible in PDL.

The problem is intimately connected with the notion of *well-foundedness*. A program α is said to be *well-founded* at a state u_0 if there exists no infinite sequence of states $u_0, u_1, u_2, \ldots$ with $(u_i, u_{i+1}) \in \mathfrak{m}_{\mathfrak{R}}(\alpha)$ for all $i \geq 0$. This property is not expressible in PDL either, as we will see.

Several very powerful logics have been proposed to deal with this situation. The most powerful is perhaps the propositional μ-calculus, which is essentially propositional modal logic augmented with a least fixpoint operator μ. Using this operator, one can express any property that can be formulated as the least fixpoint of a monotone transformation on sets of states defined by the PDL operators. For example, the well-foundedness of a program α is expressed

$$\mu X.[\alpha]X \tag{10.6.1}$$

in this logic. We will discuss the propositional μ-calculus in more detail in Section 17.4.

Two somewhat weaker ways of capturing well-foundedness without resorting to the full μ-calculus have been studied. One is to add to PDL an explicit predicate

wf for well-foundedness:

$$\mathfrak{m}_{\mathfrak{K}}(\mathbf{wf}\,\alpha) \stackrel{\text{def}}{=} \{s_0 \mid \neg\exists s_1, s_2, \ldots\ \forall i \geq 0\ (s_i, s_{i+1}) \in \mathfrak{m}_{\mathfrak{K}}(\alpha)\}.$$

Another is to add an explicit predicate **halt**, which asserts that all computations of its argument α terminate. The predicate **halt** can be defined inductively from **wf** as follows:

$$\mathbf{halt}\,a \iff^{\text{def}} \mathbf{1}, \quad a \text{ an atomic program or test,} \tag{10.6.2}$$

$$\mathbf{halt}\,\alpha;\beta \iff^{\text{def}} \mathbf{halt}\,\alpha \wedge [\alpha]\,\mathbf{halt}\,\beta, \tag{10.6.3}$$

$$\mathbf{halt}\,\alpha \cup \beta \iff^{\text{def}} \mathbf{halt}\,\alpha \wedge \mathbf{halt}\,\beta, \tag{10.6.4}$$

$$\mathbf{halt}\,\alpha^* \iff^{\text{def}} \mathbf{wf}\,\alpha \wedge [\alpha^*]\,\mathbf{halt}\,\alpha. \tag{10.6.5}$$

These constructs have been investigated in Harel and Pratt (1978), Harel and Sherman (1982), Niwinski (1984), and Streett (1981, 1982, 1985b) under the various names **loop**, **repeat**, and Δ. The predicates **loop** and **repeat** are just the complements of **halt** and **wf**, respectively:

$$\mathbf{loop}\,\alpha \iff^{\text{def}} \neg\mathbf{halt}\,\alpha$$

$$\mathbf{repeat}\,\alpha \iff^{\text{def}} \neg\mathbf{wf}\,\alpha.$$

Clause (10.6.5) is equivalent to the assertion

$$\mathbf{loop}\,\alpha^* \iff^{\text{def}} \mathbf{repeat}\,\alpha \vee \langle\alpha^*\rangle\mathbf{loop}\,\alpha.$$

It asserts that a nonhalting computation of α^* consists of either an infinite sequence of halting computations of α or a finite sequence of halting computations of α followed by a nonhalting computation of α.

Let RPDL and LPDL denote the logics obtained by augmenting PDL with the **wf** and **halt** predicates, respectively.[1] It follows from the preceding discussion that

$$\mathsf{PDL} \leq \mathsf{LPDL} \leq \mathsf{RPDL} \leq \text{the propositional } \mu\text{-calculus.}$$

Moreover, all these inclusions are known to be strict.

The logic LPDL is powerful enough to express the total correctness of nondeterministic programs. The total correctness of α with respect to precondition φ and postcondition ψ is expressed

$$TC(\varphi, \alpha, \psi) \iff^{\text{def}} \varphi \rightarrow \mathbf{halt}\,\alpha \wedge [\alpha]\psi.$$

1 The L in LPDL stands for "loop" and the R in RPDL stands for "repeat." We retain these names for historical reasons.

Conversely, **halt** can be expressed in terms of TC:

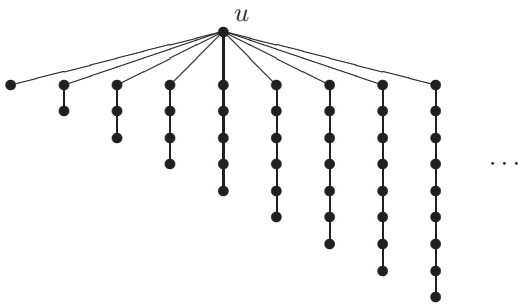
$$\mathbf{halt}\,\alpha \iff TC(\mathbf{1},\alpha,\mathbf{1}).$$

The filtration lemma fails for RPDL, LPDL, and the propositional μ-calculus (except under certain strong syntactic restrictions which render formulas like (10.6.1) ineffable; see Pratt (1981a)). This can be seen by considering the model $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ with

$$K \stackrel{\text{def}}{=} \{(i,j) \in \mathbb{N}^2 \mid 0 \le j \le i\} \cup \{u\}$$

and atomic program a with

$$\mathfrak{m}_{\mathfrak{K}}(a) \stackrel{\text{def}}{=} \{((i,j),(i,j-1)) \mid 1 \le j \le i\} \cup \{(u,(i,i)) \mid i \in \mathbb{N}\}.$$



The state u satisfies $\mathbf{halt}\,a^*$ and $\mathbf{wf}\,a$, but its equivalence class in any finite filtrate does not satisfy either of these formulas. It follows that

THEOREM 10.16: PDL $<$ LPDL.

Proof By the preceding argument and Lemma 6.4, neither $\mathbf{halt}\,a^*$ nor $\mathbf{wf}\,a$ is equivalent to any PDL formula. ■

THEOREM 10.17: LPDL $<$ RPDL.

Proof sketch. For any i, let $\mathfrak{A}_n$ and $\mathfrak{B}_n$ be the structures of Figures 10.1 and 10.2, respectively. The state t_i of $\mathfrak{B}_n$ is identified with the state s_0 in its own copy of $\mathfrak{A}_n$. For any n and $i \le n$, $\mathfrak{A}_n, s_i \vDash \mathbf{wf}\,a^*b$, but $\mathfrak{B}_n, t_i \vDash \neg\mathbf{wf}\,a^*b$. However, for each formula φ of LPDL, it is possible to find a large enough n such that for all $i \le n$, $\mathfrak{B}_n, t_i \vDash \varphi$ iff $\mathfrak{A}_n, s_0 \vDash \varphi$. This is proved by induction on the structure of φ. For the

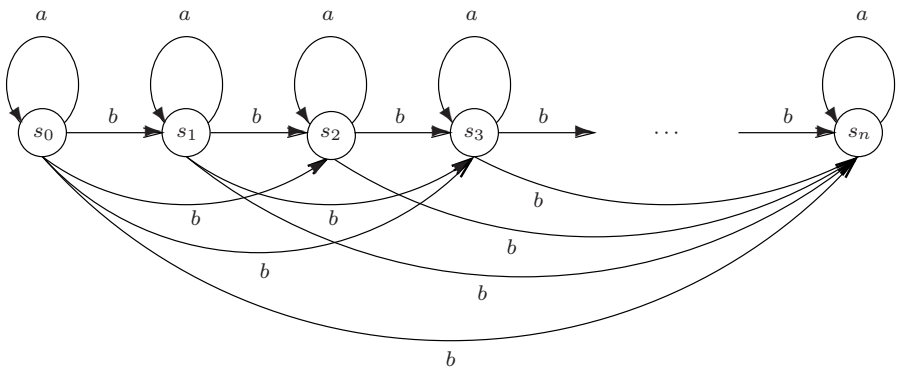


Figure 10.1
The structure $\mathfrak{A}_n$

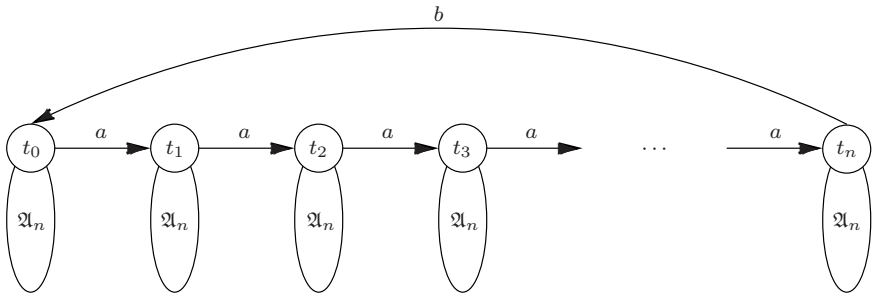
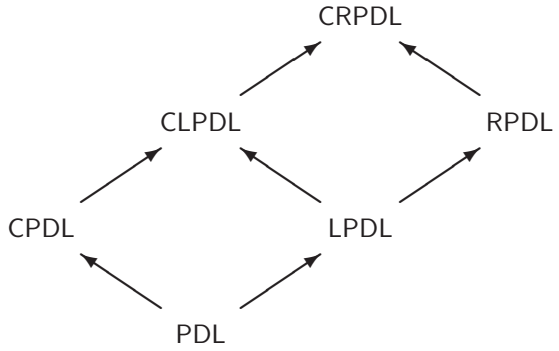


Figure 10.2
The structure $\mathfrak{B}_n$

case of **halt** α, one uses the fact that in order to capture the infinite path of a's and b's in $\mathfrak{B}_n$ with a $\neg$**halt** α clause, say $\neg$**halt** $(a^*b)^*$ for example, there must exist an infinite computation of α that after some finite bounded length consists solely of a's. Hence, this particular $\neg$**halt** α clause is already satisfied in $\mathfrak{A}_n$ for sufficient large n. The argument is similar to the proof of the pumping lemma for regular languages; see Hopcroft and Ullman (1979) or Kozen (1997a). ■

It is possible to extend Theorem 10.17 to versions CRPDL and CLPDL in which converse is allowed in addition to **wf** or **halt**. Also, the proof of Theorem 10.15 goes through for LPDL and RPDL, so that $\langle a^- \rangle \mathbf{1}$ is not expressible in either. Theorem 10.16 goes through for the converse versions too. We obtain the situation illustrated in the following figure, in which the arrows indicate $<$ and the absence of a path between two logics means that each can express properties that the other cannot.



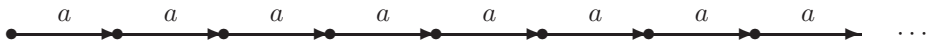
The filtration lemma fails for all **halt** and **wf** versions as in the proof of Theorem 10.16. However, satisfiable formulas of the μ-calculus (hence of RPDL and LPDL) do have finite models. This finite model property is not shared by CLPDL or CRPDL.

THEOREM 10.18: The CLPDL formula

$$\neg\mathbf{halt}\, a^* \wedge [a^*]\mathbf{halt}\, a^{*-}$$

is satisfiable but has no finite model.

Proof Let φ be the formula in the statement of the theorem. This formula is satisfied in the infinite model



To show it is satisfied in no finite model, suppose $\mathfrak{K}, s \vDash \varphi$. By (10.6.2) and (10.6.5),

$$\begin{aligned}\mathbf{halt}\, a^* &\iff \mathbf{wf}\, a \wedge [a^*]\mathbf{halt}\, a \\ &\iff \mathbf{wf}\, a \wedge [a^*]\mathbf{1} \\ &\iff \mathbf{wf}\, a,\end{aligned}$$

thus $\mathfrak{K}, s \vDash \neg\mathbf{wf}\, a$. This says that there must be an infinite a-path starting at s. However, no two states along that path can be identical without violating the clause $[a^*]\mathbf{halt}\, a^{*-}$ of φ, thus $\mathfrak{K}$ is infinite. ■

As it turns out, Theorem 10.18 does not prevent CRPDL from being decidable.

THEOREM 10.19: The validity problems for CRPDL, CLPDL, RPDL, LPDL, and the propositional μ-calculus are all decidable in deterministic exponential time.

Obviously, the simpler the logic, the simpler the arguments needed to show exponential time decidability. Over the years all these logics have been gradually shown to be decidable in exponential time by various authors using various techniques. Here we point to the exponential time decidability of the propositional μ-calculus with forward and backward modalities, proved in Vardi (1998b), from which all these can be seen easily to follow. The proof in Vardi (1998b) is carried out by exhibiting an exponential time decision procedure for two-way alternating automata on infinite trees.

As mentioned above, RPDL possesses the finite (but not necessarily the small and not the collapsed) model property.

THEOREM 10.20: Every satisfiable formula of RPDL, LPDL, and the propositional μ-calculus has a finite model.

Proof sketch. The proof uses the fact that every automaton on infinite trees that accepts some tree accepts a tree obtained by unwinding a finite graph. For a satisfiable formula φ in these logics, it is possible to transform the finite graph obtained in this way from the automaton for φ into a finite model of φ. ■

CRPDL and CLPDL are extensions of PDL that, like PDL$+a^{\Delta}b^{\Delta}$ (Theorems 9.5 and 9.9), are decidable despite lacking a finite model property.

Complete axiomatizations for RPDL and LPDL can be obtained by embedding them into the μ-calculus (see Section 17.4).

10.7 Concurrency and Communication

Another interesting extension of PDL concerns concurrent programs. Recall the intersection operator $\cap$ introduced in Section 10.4. The binary relation on states corresponding to the program $\alpha \cap \beta$ is the intersection of the binary relations corresponding to α and β. This can be viewed as a kind of concurrency operator that admits transitions to those states that both α and β would have admitted.

In this section, we consider a different and perhaps more natural notion of concurrency. The interpretation of a program will not be a binary relation on states, which relates initial states to possible final states, but rather a relation between a states and *sets* of states. Thus $\mathfrak{m}_{\mathfrak{K}}(\alpha)$ will relate a start state u to a collection of sets

of states U. The intuition is that starting in state u, the (concurrent) program α can be run with its concurrent execution threads ending in the set of final states U. The basic concurrency operator will be denoted here by $\wedge$, although in the original work on concurrent Dynamic Logic (Peleg (1987b,c,a)) the notation $\cap$ is used.

The syntax of *concurrent* PDL is the same as PDL, with the addition of the clause:

- if $\alpha, \beta \in \Pi$, then $\alpha \wedge \beta \in \Pi$.

The program $\alpha \wedge \beta$ means intuitively, "Execute α and β in parallel."

The semantics of concurrent PDL is defined on Kripke frames $\mathfrak{K} = (K, \mathfrak{m}_{\mathfrak{K}})$ as with PDL, except that for programs α,

$$\mathfrak{m}_{\mathfrak{K}}(\alpha) \quad \subseteq \quad K \times 2^K.$$

Thus the meaning of α is a collection of *reachability pairs* of the form (u, U), where $u \in K$ and $U \subseteq K$. In this brief description of concurrent PDL, we require that structures assign to atomic programs *sequential*, non-parallel, meaning; that is, for each $a \in \Pi_0$, we require that if $(u, U) \in \mathfrak{m}_{\mathfrak{K}}(a)$, then $\#U = 1$. The true parallelism will stem from applying the concurrency operator to build larger sets U in the reachability pairs of compound programs. We shall not provide the details here; the reader is referred to Peleg (1987b,c).

The relevant results for this logic are the following:

THEOREM 10.21: PDL $<$ concurrent PDL.

THEOREM 10.22: The validity problem for concurrent PDL is decidable in deterministic exponential time.

Axiom System 5.5, augmented with the following axiom, can be be shown to be complete for concurrent PDL:

$$\langle \alpha \wedge \beta \rangle \varphi \quad \leftrightarrow \quad \langle \alpha \rangle \varphi \wedge \langle \beta \rangle \varphi.$$

10.8 Bibliographical Notes

Completeness and exponential time decidability for DPDL, Theorem 10.1 and the upper bound of Theorem 10.2, are proved in Ben-Ari et al. (1982) and Valiev (1980).

The lower bound of Theorem 10.2 is from Parikh (1981). Theorems 10.4 and 10.5 on SDPDL are from Halpern and Reif (1981, 1983).

That tests add to the power of PDL (Theorem 10.6) is proved in Berman and Paterson (1981), and Theorem 10.7 appears in Berman (1978) and Peterson (1978). It can be shown (Peterson (1978); Berman (1978); Berman and Paterson (1981)) that rich-test PDL is strictly more expressive than poor-test PDL. These results also hold for SDPDL (see Section 10.1).

The results on programs as automata (Theorems 10.10 and 10.11) appear in Pratt (1981b) but the proofs sketched are from Harel and Sherman (1985). The material of Section 10.4 on the intersection of programs is from Harel et al. (1982). That the axioms in Section 10.5 yield completeness for CPDL is proved in Parikh (1978a).

The complexity of PDL with converse and various forms of well-foundedness constructs is studied in Vardi (1985b). Many authors have studied logics with a least-fixpoint operator, both on the propositional and first-order levels (Scott and de Bakker (1969); Hitchcock and Park (1972); Park (1976); Pratt (1981a); Kozen (1982, 1983, 1988); Kozen and Parikh (1983); Niwinski (1984); Streett (1985b); Vardi and Stockmeyer (1985)). The version of the propositional μ-calculus presented here was introduced in Kozen (1982, 1983).

That the propositional μ-calculus is strictly more expressive than PDL with **wf** was show in Niwinski (1984) and Streett (1985b). That this logic is strictly more expressive than PDL with **halt** was shown in Harel and Sherman (1982). That this logic is strictly more expressive than PDL was shown in Streett (1981).

The **wf** construct (actually its complement, **repeat**) is investigated in Streett (1981, 1982), in which Theorems 10.16 (which is actually due to Pratt) and 10.18–10.20 are proved. The **halt** construct (actually its complement, **loop**) was introduced in Harel and Pratt (1978) and Theorem 10.17 is from Harel and Sherman (1982). Finite model properties for the logics LPDL, RPDL, CLPDL, CRPDL, and the propositional μ-calculus were established in Streett (1981, 1982) and Kozen (1988). Decidability results were obtained in Streett (1981, 1982); Kozen and Parikh (1983); Vardi and Stockmeyer (1985); and Vardi (1985b). Deterministic exponential-time completeness was established in Emerson and Jutla (1988) and Safra (1988). For the strongest variant, CRPDL, exponential-time decidability follows from Vardi (1998b).

Concurrent PDL is defined in Peleg (1987b), in which the results of Section 10.7 are proved. Additional versions of this logic, which employ various mechanisms for communication among the concurrent parts of a program, are considered in Peleg (1987c,a). These papers contain many results concerning expressive power, decidability and undecidability for concurrent PDL with communication.

Other work on PDL not described here includes work on nonstandard models, studied in Berman (1979, 1982) and Parikh (1981); PDL with Boolean assignments, studied in Abrahamson (1980); and restricted forms of the consequence problem, studied in Parikh (1981).