Propositional Dynamic Logic

CS 112: Lecture Notes

November 23, 2010

My class notes are largely based on the presentation in the book, *Dynamic Logic* (2000), by David Harel, Dexter Kozen, and Jerzy Tiuryn. This is a wonderful book, and very clearly lays out the motivation for the expressiveness present in dynamic logic. For the notes, I've reprinted parts of their earlier article from 1984, much of which is reworked in the 2000 book. I am also referencing the forthcoming article by Jess and myself, which deals with a first-order fragment of DL, focusing on motion predicates in language. See the webpage for that article.

PREFACE

Dynamic Logic (DL) is a formal system for reasoning about programs. Traditionally, this has meant formalizing correctness specifications and proving rigorously that those specifications are met by a particular program. Other activities fall into this category as well: determining the equivalence of programs, comparing the expressive power of various programming constructs, synthesizing programs from specifications, *etc*. Formal systems too numerous to mention have been proposed for these purposes, each with its own peculiarities.

DL can be described as a blend of three complementary classical ingredients: first-order predicate logic, modal logic, and the algebra of regular events. These components merge to form a system of remarkable unity that is theoretically rich as well as practical.

The name *Dynamic Logic* emphasizes the principal feature distinguishing it from classical predicate logic. In the latter, truth is *static*: the truth value of a formula φ is determined by a valuation of its free variables over some structure. The valuation and the truth value of φ it induces are regarded as immutable; there is no formalism relating them to any other valuations or truth values. In Dynamic Logic, there are explicit syntactic constructs called *programs* whose main role is to change the values of variables, thereby changing the truth values of formulas. For example, the program $x := x + 1$ over the natural numbers changes the truth value of the formula "x is even".

Such changes occur on a metalogical level in classical predicate logic. For example, in Tarski's definition of truth of a formula, if $u : \{x, y, \ldots\} \to \mathbb{N}$ is a valuation of variables over the natural numbers $\mathbb{N}$, then the formula $\exists x\ x^2 = y$ is defined to be true under the valuation u iff there exists an $a \in \mathbb{N}$ such that the formula $x^2 = y$ is true under the valuation $u[x/a]$, where $u[x/a]$ agrees with u everywhere except x, on which it takes the value a. This definition involves a metalogical operation that produces $u[x/a]$ from u for all possible values $a \in \mathbb{N}$. This operation becomes explicit in DL in the form of the program $x := ?$, called a *nondeterministic* or *wildcard assignment*. This is a rather unconventional program, since it is not effective; however, it is quite useful as a descriptive tool. A more conventional way to obtain a square root of y, if it exists, would be the program

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

In DL, such programs are first-class objects on a par with formulas, complete with a collection of operators for forming compound programs inductively from a basis of primitive programs. To discuss the effect of the execution of a program α on the truth of a formula φ, DL uses a modal construct $<\alpha>\varphi$, which intuitively states, "It is possible to execute α starting from the current state and halt in a state satisfying φ." There is also the dual construct $[\alpha]\varphi$, which intuitively states, "If α halts when started in the current state, then it does so in a state satisfying φ." For example, the first-order formula $\exists x\ x^2 = y$ is equivalent to the DL formula $<x := ?>\ x^2 = y$. In order to instantiate the quantifier effectively, we might replace the nondeterministic assignment inside the < > with the **while** program (1); over $\mathbb{N}$, the two formulas would be equivalent.

Apart from the obvious heavy reliance on classical logic, computability theory and programming, the subject has its roots in the work of [Thiele, 1966] and [Engeler, 1967] in the late 1960's, who were the first to advance the idea of formulating and investigating formal systems dealing with properties of programs in an abstract setting. Research in program verification flourished thereafter with the work of many researchers, notably [Floyd, 1967], [Hoare, 1969], [Manna, 1974], and [Salwicki, 1970]. The first precise development of a DL-like system was carried out by [Salwicki, 1970], following [Engeler, 1967]. This system was called Algorithmic Logic. A similar system, called Monadic Programming Logic, was developed by [Constable, 1977]. Dynamic Logic, which emphasizes the modal nature of the program/assertion interaction, was introduced by [Pratt, 1976].

Background material on mathematical logic, computability, formal languages and automata, and program verification can be found in [Shoenfield, 1967] (logic), [Rogers, 1967] (recursion theory), [Kozen, 1997a] (formal languages, automata, and computability), [Keisler, 1971] (infinitary logic), [Manna, 1974] (program verification), and [Harel, 1992; Lewis and Papadimitriou, 1981; Davis *et al.*, 1994] (computability and complexity). Much of this introductory material as it pertains to DL can be found in the authors' text [Harel *et al.*, 2000].

There are by now a number of books and survey papers treating logics of programs, program verification, and Dynamic Logic [Apt and Olderog, 1991; Backhouse, 1986; Harel, 1979; Harel, 1984; Parikh, 1981; Goldblatt, 1982; Goldblatt, 1987; Knijnenburg, 1988; Cousot, 1990; Emerson, 1990; Kozen and Tiuryn, 1990]. In particular, much of this chapter is an abbreviated summary of material from the authors' text [Harel *et al.*, 2000], to which we refer the reader for a more complete treatment. Full proofs of many of the theorems cited in this chapter can be found there, as well as extensive introductory material on logic and complexity along with numerous examples and exercises.

1 REASONING ABOUT PROGRAMS

1.1 Programs

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

EXAMPLE 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 in classical first-order logic. 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 section we give a brief introduction to some of these languages and the role they play in program verification.

1.2 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*. 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 "$\ldots$" 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.

1.3 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 propositional 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 vocabulary 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 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.

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 2.2, 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:

$$\textbf{if } \varphi \textbf{ then } \alpha \textbf{ else } \beta \quad \stackrel{\text{def}}{=} \quad \varphi?\,;\,\alpha \cup \neg\varphi?\,;\,\beta \tag{2}$$

$$\textbf{while } \varphi \textbf{ do } \alpha \quad \stackrel{\text{def}}{=} \quad (\varphi?\,;\,\alpha)^*\,;\,\neg\varphi? \tag{3}$$

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.

Recursion

Recursion can appear in programming languages in several forms. Two such manifestations are *recursive calls* and *stacks*. Under certain very general conditions, the two constructs can simulate each other. It can also be shown 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 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

[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.

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.

1.4 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 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

$$\textbf{while } x > 7 \textbf{ do } x := x + 3$$

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

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

Dynamic Logic can reason about the behavior of a program that is manifested in its input/output relation. It is not well suited 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 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 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 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 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.

1.5 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 14.2.

2 PROPOSITIONAL DYNAMIC LOGIC (PDL)

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. Since PDL is a subsystem of first-order DL, we can be sure that all properties of PDL that we discuss in this section will also be valid in first-order DL.

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

$$[\alpha](\varphi \land \psi) \quad \leftrightarrow \quad [\alpha]\varphi \land [\alpha]\psi. \tag{4}$$

The left-hand side asserts that the formula $\varphi \land \psi$ 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 (4) 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 (4) holds universally, regardless of the domain of computation and the nature of the particular α, φ, and ψ.

As another example, consider

$$[\alpha\,;\,\beta]\varphi \quad \leftrightarrow \quad [\alpha][\beta]\varphi. \tag{5}$$

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) asserts the logical equivalence of these two statements. It holds regardless of the nature of α, β, and φ. Like (4), (5) can be used to simplify the verification of complicated programs.

As a final example, consider the assertion

$$[\alpha]p \quad \leftrightarrow \quad [\beta]p \tag{6}$$

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 (6). 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 (6).

2.1 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 section we will introduce the basic version, called *regular* PDL. Variations of this basic version will be considered in later sections.

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

$\varphi \rightarrow \psi$	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 propositional operators $\wedge$, $\vee$, $\neg$, $\leftrightarrow$, and $\mathbf{1}$ can be defined from $\rightarrow$ and $\mathbf{0}$ in the usual way.

The possibility operator $<\,>$ is the modal dual of the necessity operator $[\,]$. It is defined by

$$<\alpha>\varphi \stackrel{\text{def}}{=} \neg[\alpha]\neg\varphi.$$

The propositions $[\alpha]\varphi$ and $<\alpha>\varphi$ 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 $<\,>$ and $[\,]$ is that $<\alpha>\varphi$ 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

$$\begin{array}{rcl}
\mathbf{skip} & \stackrel{\text{def}}{=} & \mathbf{1}? \\
\mathbf{fail} & \stackrel{\text{def}}{=} & \mathbf{0}? \\
\mathbf{if}\ \varphi_1 \rightarrow \alpha_1 \ \mid \cdots \mid \ \varphi_n \rightarrow \alpha_n\ \mathbf{fi} & \stackrel{\text{def}}{=} & \varphi_1?;\alpha_1 \cup \cdots \cup \varphi_n?;\alpha_n \\
\mathbf{do}\ \varphi_1 \rightarrow \alpha_1 \ \mid \cdots \mid \ \varphi_n \rightarrow \alpha_n\ \mathbf{od} & \stackrel{\text{def}}{=} & (\bigcup_{i=1}^{n} \varphi_i?;\alpha_i)^*;(\bigwedge_{i=1}^{n} \neg\varphi_i)? \\
\mathbf{if}\ \varphi\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta & \stackrel{\text{def}}{=} & \mathbf{if}\ \varphi \rightarrow \alpha \ \mid\ \neg\varphi \rightarrow \beta\ \mathbf{fi} \\
 & = & \varphi?;\alpha \cup \neg\varphi?;\beta \\
\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha & \stackrel{\text{def}}{=} & \mathbf{do}\ \varphi \rightarrow \alpha\ \mathbf{od} \\
 & = & (\varphi?;\alpha)^*;\neg\varphi? \\
\mathbf{repeat}\ \alpha\ \mathbf{until}\ \varphi & \stackrel{\text{def}}{=} & \alpha;\mathbf{while}\ \neg\varphi\ \mathbf{do}\ \alpha \\
 & = & \alpha;(\neg\varphi?;\alpha)^*;\varphi? \\
\{\varphi\}\,\alpha\,\{\psi\} & \stackrel{\text{def}}{=} & \varphi \rightarrow [\alpha]\psi.
\end{array}$$

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. We will argue later that the formal definitions of these operators given above correctly model their intuitive behavior.

2.2 Semantics

The semantics of PDL comes from the semantics for modal logic. 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

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

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,

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

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

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

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.

$$\begin{array}{rcll} \mathfrak{m}_{\mathfrak{K}}(\varphi \to \psi) & \stackrel{\text{def}}{=} & (K - \mathfrak{m}_{\mathfrak{K}}(\varphi)) \cup \mathfrak{m}_{\mathfrak{K}}(\psi) & \\ \mathfrak{m}_{\mathfrak{K}}(\mathbf{0}) & \stackrel{\text{def}}{=} & \varnothing & \\ \mathfrak{m}_{\mathfrak{K}}([\alpha]\varphi) & \stackrel{\text{def}}{=} & K - (\mathfrak{m}_{\mathfrak{K}}(\alpha) \circ (K - \mathfrak{m}_{\mathfrak{K}}(\varphi))) & \\ & = & \{u \mid \forall v \in K \text{ if } (u,v) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ then } v \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\} & \\ \mathfrak{m}_{\mathfrak{K}}(\alpha;\beta) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha) \circ \mathfrak{m}_{\mathfrak{K}}(\beta) & (7) \\ & = & \{(u,v) \mid \exists w \in K\ (u,w) \in \mathfrak{m}_{\mathfrak{K}}(\alpha) \text{ and } (w,v) \in \mathfrak{m}_{\mathfrak{K}}(\beta)\} & \\ \mathfrak{m}_{\mathfrak{K}}(\alpha \cup \beta) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha) \cup \mathfrak{m}_{\mathfrak{K}}(\beta) & \\ \mathfrak{m}_{\mathfrak{K}}(\alpha^*) & \stackrel{\text{def}}{=} & \mathfrak{m}_{\mathfrak{K}}(\alpha)^* \;=\; \bigcup_{n \geq 0} \mathfrak{m}_{\mathfrak{K}}(\alpha)^n & (8) \\ \mathfrak{m}_{\mathfrak{K}}(\varphi?) & \stackrel{\text{def}}{=} & \{(u,u) \mid u \in \mathfrak{m}_{\mathfrak{K}}(\varphi)\}. & \end{array}$$

The operator $\circ$ in (7) is relational composition. In (8), the first occurrence of * is the iteration symbol of PDL, and the second is the reflexive transitive closure operator on binary relations. Thus (8) 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:

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

The defined operators inherit their meanings from these definitions:

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

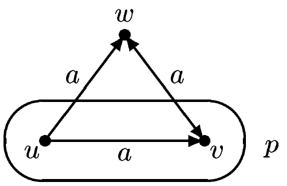
In addition, the **if-then-else**, **while-do**, and guarded commands inherit their semantics from the above definitions, and the input/output relations given by the formal semantics capture their intuitive operational meanings. For example, the relation associated with the program **while** φ **do** 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 $u_n \notin \mathfrak{m}_{\mathfrak{K}}(\varphi)$.

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 $<\alpha>p \leftrightarrow <\beta>p$ is valid.

EXAMPLE 2. 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

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

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



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

$$<a^*>[(aa)^*]p \wedge <a^*>[(aa)^*]\neg p.$$

2.3 Computation Sequences

Let α be a program. Recall from Section 1.3 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:

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

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

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

has as computation sequences all strings of the form

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

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 3.

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

2.4 Satisfiability and Validity

The definitions of satisfiability and validity of propositions come from modal logic. Let $\mathfrak{K} = (K,\ \mathfrak{m}_{\mathfrak{K}})$ be a Kripke frame and let φ be a proposition. We have defined in Section 2.2 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

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

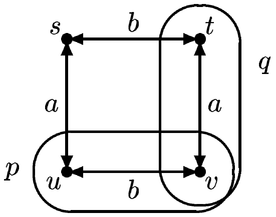
is *sound* if φ is a logical consequence of $\{\varphi_1, \ldots, \varphi_n\}$.

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 4. 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

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

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



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

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

Also, let α be the program

$$\alpha \quad = \quad (aa \cup bb \cup (ab \cup ba)(aa \cup bb)^*(ab \cup ba))^*.$$

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 $\mathfrak{K}$.

EXAMPLE 5. The formula

$$p \wedge [a^*]((p \rightarrow [a]\neg p) \wedge (\neg p \rightarrow [a]p)) \quad \leftrightarrow \quad [(aa)^*]p \wedge [a(aa)^*]\neg p$$

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

2.5 Basic Properties

THEOREM 6. *The following are valid formulas of* PDL*:*

(i) $<\alpha>(\varphi \vee \psi) \quad \leftrightarrow \quad <\alpha>\varphi \vee <\alpha>\psi$

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

(iii) $<\alpha>\varphi \wedge [\alpha]\psi \quad \rightarrow \quad <\alpha>(\varphi \wedge \psi)$

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

(v) $<\alpha>(\varphi \wedge \psi) \quad \rightarrow \quad <\alpha>\varphi \wedge <\alpha>\psi$

(vi) $[\alpha]\varphi \vee [\alpha]\psi \quad \rightarrow \quad [\alpha](\varphi \vee \psi)$

(vii) $<\alpha>\mathbf{0} \quad \leftrightarrow \quad \mathbf{0}$

(viii) $[\alpha]\varphi \quad \leftrightarrow \quad \neg<\alpha>\neg\varphi.$

(ix) $<\alpha \cup \beta>\varphi \quad \leftrightarrow \quad <\alpha>\varphi \vee <\beta>\varphi$

(x) $[\alpha \cup \beta]\varphi \quad \leftrightarrow \quad [\alpha]\varphi \wedge [\beta]\varphi$

(xi) $<\alpha\,;\,\beta>\varphi \quad \leftrightarrow \quad <\alpha><\beta>\varphi$

(xii) $[\alpha\,;\,\beta]\varphi \quad \leftrightarrow \quad [\alpha]\,[\beta]\varphi$

(xiii) $<\varphi?>\psi \quad \leftrightarrow \quad (\varphi \wedge \psi)$

(xiv) $[\varphi?]\psi \quad \leftrightarrow \quad (\varphi \rightarrow \psi)$.

THEOREM 7. *The following are sound rules of inference of* PDL*:*

(i) *Modal generalization (GEN):*

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

(ii) *Monotonicity of* $<\alpha>$*:*

$$\frac{\varphi \rightarrow \psi}{<\alpha>\varphi \rightarrow <\alpha>\psi}$$

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

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

The *converse operator* $^-$ is a program operator with semantics

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

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 8. *For any programs* α *and* β*,*

(i) $\mathfrak{m}_{\mathfrak{K}}((\alpha \cup \beta)^-) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\alpha^- \cup \beta^-)$

(ii) $\mathfrak{m}_{\mathfrak{K}}((\alpha\,;\,\beta)^-) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\beta^-\,;\,\alpha^-)$

(iii) $\mathfrak{m}_{\mathfrak{K}}(\varphi?^-) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\varphi?)$

(iv) $\mathfrak{m}_{\mathfrak{K}}(\alpha^{*-}) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\alpha^{-*})$

(v) $\mathfrak{m}_{\mathfrak{K}}(\alpha^{--}) \quad = \quad \mathfrak{m}_{\mathfrak{K}}(\alpha)$.

THEOREM 9. *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) $\langle\alpha^-\rangle[\alpha]\varphi \rightarrow \varphi$.

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

$$\{\langle\alpha^*\rangle\varphi\} \quad \cup \quad \{\neg\varphi, \neg\langle\alpha\rangle\varphi, \neg\langle\alpha^2\rangle\varphi, \ldots\} \tag{9}$$

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. In a nutshell, for any binary relation R, R^* is the $\subseteq$-least reflexive and transitive relation containing R.

THEOREM 10. *The following are valid formulas of* PDL*:*

(i) $[\alpha^*]\varphi \rightarrow \varphi$

(ii) $\varphi \rightarrow \langle\alpha^*\rangle\varphi$

(iii) $[\alpha^*]\varphi \rightarrow [\alpha]\varphi$

(iv) $\langle\alpha\rangle\varphi \rightarrow \langle\alpha^*\rangle\varphi$

(v) $[\alpha^*]\varphi \leftrightarrow [\alpha^*\alpha^*]\varphi$

(vi) $\langle\alpha^*\rangle\varphi \leftrightarrow \langle\alpha^*\alpha^*\rangle\varphi$

(vii) $[\alpha^*]\varphi \leftrightarrow [\alpha^{**}]\varphi$

(viii) $\langle\alpha^*\rangle\varphi \leftrightarrow \langle\alpha^{**}\rangle\varphi$

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

(x) $\texttt{<}\alpha^*\texttt{>}\varphi \;\leftrightarrow\; \varphi \vee \texttt{<}\alpha\texttt{>}\texttt{<}\alpha^*\texttt{>}\varphi.$

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

(xii) $\texttt{<}\alpha^*\texttt{>}\varphi \;\leftrightarrow\; \varphi \vee \texttt{<}\alpha^*\texttt{>}(\neg\varphi \wedge \texttt{<}\alpha\texttt{>}\varphi).$

Semantically, α^* is a reflexive and transitive relation containing α, and Theorem 10 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*:

$$\frac{(\varphi \vee \texttt{<}\alpha\texttt{>}\psi) \to \psi}{\texttt{<}\alpha^*\texttt{>}\varphi \to \psi}$$

(LI) The *loop invariance rule*:

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

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

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

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

$$\texttt{<}\alpha^*\texttt{>}\varphi \quad\to\quad \varphi \vee \texttt{<}\alpha^*\texttt{>}(\neg\varphi \wedge \texttt{<}\alpha\texttt{>}\varphi)$$

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 10(x). Observe that the right-to-left implication of this formula is obtained by substituting $\texttt{<}\alpha^*\texttt{>}\varphi$ for R in the expression

$$\varphi \vee \texttt{<}\alpha\texttt{>}R \quad\to\quad R. \qquad (10)$$

Theorem 10(x) implies that $<\alpha^*>\varphi$ is a solution of (10); that is, (10) is valid when $<\alpha^*>\varphi$ is substituted for R. The rule (RTC) says that $<\alpha^*>\varphi$ 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 (10) 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."

Note that the box form of (IND) bears a strong resemblance to the induction axiom of Peano arithmetic:

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

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

2.6 Encoding Hoare Logic

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

THEOREM 11. *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:*

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

3 FILTRATION AND DECIDABILITY

The *small model property* for PDL 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 result and the technique used to prove it, called *filtration*, come directly from modal logic. 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 described in Section 5.

3.1 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

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

by simultaneous induction. The set $FL(\varphi)$ is called the *Fischer–Ladner closure* of φ. The filtration construction for PDL 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) $FL(\varphi \rightarrow \psi) \stackrel{\text{def}}{=} \{\varphi \rightarrow \psi\} \cup FL(\varphi) \cup FL(\psi)$

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

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

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

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