DAVID HAREL, DEXTER KOZEN, AND JERZY TIURYN

DYNAMIC LOGIC

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 $u : \{x, y, \dots\} \to \mathbb{N}$ 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

(1) $\quad x := 0\,;$ **while** $x^2 < y$ **do** $x := x + 1.$

In DL, such programs are first-class objects on a par with formulas, complete with a collection of operators for forming compound programs inductively

D.M. Gabbay and F. Guenthner (eds.),
Handbook of Philosophical Logic, 2nd Edition, Volume 4, 99–217.

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 $\langle\alpha\rangle\varphi$, 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 $\langle x := ? \rangle\ x^2 = y$. In order to instantiate the quantifier effectively, we might replace the nondeterministic assignment inside the $\langle\ \rangle$ 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 "..." 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.

http://www.springer.com/978-1-4020-0139-0

Handbook of Philosophical Logic
Gabbay, D.; Guenthner, F. (Eds.)
2002, XIII, 431 p. 1 illus., Hardcover
ISBN: 978-1-4020-0139-0