Logic as a Formal Method

Antony Galton
Department of Computer Science
University of Exeter
Exeter EX4 4PT

June 25, 1997

Introduction

The aim of this article is to present, in outline, a representative selection of the ways in which formal logic has been of service to computer science. Logic offers so many possibilities of application, and there are so many diverse groups of researchers developing logic-based applications, that it will be impossible in the space available to do justice to the whole field. Indeed it will be impossible even to mention everything that is going on, let alone say anything about it. Therefore I confine myself to a few areas which I believe, taken together, give a fair impression of the promise that formal logic holds as a tool for computer scientists. I assume in this article that the reader has a working knowledge of the classical Propositional and Predicate Calculi: a lightning sketch of these systems can be found in my article 'Classical Logic: a Crash Course for Beginners' which appears earlier in this issue; for more details, the reader is urged to consult a textbook such as my *Logic for Information Technology* (Wiley, 1990).

1 Applications of Classical Logic

First-order logic has a number of virtues which make it a valuable tool in Computer Science. The first of these is that it is an artificial language, totally under our control, with none of the maverick and unpredictable ambiguities that pervade ordinary language. Once one has fixed the domain of the interpretation, and the denotation of the constants, predicates and function symbols, the meaning of every formula is thereby fixed too, in a completely unambiguous way. It is therefore a good medium in which to codify precisely the facts and rules pertaining to the sorts of domains we are interested in when thinking computationally—which can mean either the domains we want our computations to be *about*, or the domain of computation itself, for example if we want to reason about the behaviour of programs. Second, and no less important, is the fact that the language of the Predicate Calculus comes with a ready-made inferential apparatus, enabling us not just to *express* the facts pertaining to our chosen domain but also to *reason* with them in a way that is guaranteed to be logically correct. And finally, a virtue that is often claimed for the Predicate Calculus is that it is *universal* in that it does not prejudge its possible domains of application. It is not entirely clear to me how far this claim is really correct, but there certainly seem to be potential application areas which pose considerable difficulty for the Predicate Calculus, for example reasoning about *mass terms* such as 'water' or 'gold', where it is only with the greatest artificiality that we can conceptualise the domain in terms of a set of discrete individuals. For computer science, however, this limitation is not serious, since in this discipline we do almost always think of things in just the discrete kind of way that is required for a direct application of the Predicate Calculus.

1.1 Program specification

My first example of such an application concerns the range of activities to which we may apply the term *specification*. Writing programs is only a meaningful activity so long as one starts off with some idea, however vague, of how one wants one's program to behave. In small-scale 'recreational' programming, and in some areas of Artificial Intelligence, the idea may only ever be formulated in the vaguest way, programmers relying on their ability to recognize intuitively when they have come up with something interesting that conforms to their original idea. In a commercial or industrial context, however, it has long been recognized that a systematic codification of the desired program behaviour is an absolute prerequisite for responsible programming. How systematic should one be? According to the Formal Methods school, nothing less than a rigorous specification in first-order logic or some comparable formalism will do, for without such a specification, it is not only impossible to be sure that the program behaves as desired, it is even left indeterminate what the desired program behaviour is in the first place.

Suppose, to take a simple example, that one wanted to develop a library information system which users can consult in order to find out about what books, periodicals, etc., the library possesses, and also about their own use of the library—who has a given book on loan, when it is due back, and so on. The domain of the program consists of a number of different types of object: library items such as books and periodicals, catalogue numbers assigned to these, locations where items are physically stored in the library, and individual borrowers, who may have various different statuses (e.g., in a university library, undergraduate, postgraduate, staff), their addresses, and so on. There are innumerable constraints that must be satisfied, for example that a book cannot be on loan to more than one user at a time, that each copy of every book has a catalogue number that identifies it uniquely, that each borrower has a unique status which determines the normal period of loan for items borrowed, that certain items may have restricted loan periods, and so on. Somehow the finished information system must behave in such a way that these constraints are all satisfied. How is the programmer to ensure that this is achieved? Of course the answer is by being systematic, and from what we have already said it should be clear that a very good way of being systematic is to express all the relevant constraints in logical form.

Thus, for example, the first constraint mentioned above might come out looking something like this:

$$\forall x \forall y \forall z (Book(x) \wedge On\_loan(x, y) \wedge On\_loan(x, z) \rightarrow y = z).$$

The complete set of such constraints will constitute a formal specification of the information system, laying down criteria for what is to count as a correct implementation of the system. There are several things that can be done with such a specification. Ideally, we would like to be able to do the following:

- Use the specification to prove that the implementation is correct—i.e., that it does not violate any of the constraints[1];
- Even better, by systematically transforming the specification, to produce a working program from it automatically, in such a way that the correct behaviour of the program is guaranteed.
- By logical inference from the specification to determine what further properties any correct implementation of the specification must possess.

[1]In order to do this the programming *language* will also need to be specified formally.

It is generally found that logic alone is not enough; the logic, which is as we have remarked highly general, has to be embedded in procedures that are more specifically tailored to the computational context. What is important, though, is that we can effect a clean separation between the specification of *what* we are trying to achieve—that is, how the finished program should behave—and the implementational details, the procedures *by which* the desired behaviour is to be realised. The specification should thus be presented as a set of declarations of what is required, rather than in terms of algorithms, so the declarative nature of logic suits it to that purpose. Formal specification methods such as Z and VDM generally contain a substantial core of pure logic, although as already hinted they contain much else besides.

1.2 Program verification

Given a formal specification and an actual program, how can one tell whether the latter is correct in relation to the former, that it actually behaves as specified? Formal methods of *program verification* are designed to allow one to check just this. In general terms, the behaviour of a program, or of a self-contained module within a program, can be specified by

1. the class of inputs it is to accept;
2. the class of outputs it is to deliver;
3. the required relation between the input and the output.

For example, a program to compute the quotient and remainder when one natural number[2] is divided by another can be specified as follows:

1. Input: two natural numbers x, y, where $y > 0$.
2. Output: two natural numbers q, r.
3. Input-Output Relation: $x = qy + r$ and $r < y$.

An example procedure, in Pascal, is

```
procedure quot_rem(x,y: integer; var q,r: integer);

begin
q:=0; r:=x;
while r>=y do
       begin
       r:=r-y; q:=q+1
       end
end.
```

We assume here that the input condition, that $x \geq 0$ and $y > 0$, is satisfied before the procedure is called.

A logical system specifically designed for program verification was developed by C. A. R. Hoare over twenty years ago (Hoare, 1969). It makes use of first-order logic together with a special form of *program logic* in which to express propositions of the form 'if before the execution of a piece of code

[2]The *natural numbers* are the non-negative integers, i.e., 0, 1, 2, 3, ...

S, the values of the program variables satisfy the formula P, then after execution of S they will satisfy the formula Q'. The notation used to express this is $\{P\}S\{Q\}$. Here P is known as a *precondition* and Q as a *postcondition*. In general one wants the precondition to be as weak as possible, to cover the greatest possible range of initial states (hence the *weakest precondition* rules of Dijkstra, 1976), and the postcondition to be as strong as possible, so that one gets as detailed a picture of the output state as possible. For example, the formula

$$\{y = k\}\texttt{x} := \texttt{y} + \texttt{1}\{x = k + 1 \wedge y = k\}$$

states that if we execute the instruction `x:=y+1` when $y = k$, then in the resulting state we will have $x = k + 1$ and $y = k$ (note that the initial value of x is irrelevant, so it is not mentioned in the precondition). In general, Hoare laid down as an axiom (the *Axiom of Assigment*) the rule

$$\{P[x'/x]\}\texttt{x} := \texttt{x}'\{P\}.$$

Here $P[x'/x]$ is the statement obtained from P by replacing each occurrence of 'x' by 'x''—so, in the example above we have

$$\text{‘}x = k + 1 \wedge y = k\text{’}[y + 1/x]$$

i.e.,'$y + 1 = k + 1 \wedge y = k$', which is equivalent to what we had before since $y + 1 = k + 1$ is equivalent to $y = k$ and for any formula ϕ, $\phi \wedge \phi$ is equivalent to ϕ.

In addition to the axiom of assignment, Hoare used a number of rules of inference, including

- two *Rules of Consequence*:

  (Cons 1) If $\{P\}S\{Q\}$, and Q implies R, then $\{P\}S\{R\}$;
  (Cons 2) If P implies Q, and $\{Q\}S\{R\}$ then $\{P\}S\{R\}$

  which enable one to weaken the postcondition or strengthen the precondition,

- a *Rule of Composition*:

  (Comp) If $\{P\}S_1\{Q\}$ and $\{Q\}S_2\{R\}$ then $\{P\}S_1; S_2\{R\}$

  which enables pre- and post-conditions to be stated for compound instructions built up by concatenating simpler ones, and

- a *Rule of Iteration*:

  (it) If $\{P \wedge B\}S\{P\}$ then $\{P\}\texttt{while B do S}\{\neg B \wedge P\}$

  which enables us to handle `while`-loops.

One can now *prove* that the Pascal procedure above meets the specification for the quotient-remainder computation. Specifically, the relation

$$\{y > 0\}\texttt{quot\_rem(x, y, q, r)}\{x = qy + r \wedge r < y\}$$

can be derived as a theorem of the system defined by Hoare's rules. To illustrate part of the derivation, we note that by the rule of assignment we have

$$\{x = (q + 1)y + (r - y)\}\texttt{r} := \texttt{r} - \texttt{y}; \texttt{q} := \texttt{q} + \texttt{1}\{x = qy + r\};$$

but the precondition here is equivalent to $x = qy + r$, so we have

$$\{x = qy + r\}\texttt{r} := \texttt{r} - \texttt{y}; \texttt{q} := \texttt{q} + \texttt{1}\{x = qy + r\},$$

making this formula an *invariant* for the `while`-loop of the procedure. We now have, from the rule of iteration,

$\{x = qy + r\}$
`while r >= y do begin r := r - y; q := q + 1 end`
$\{\neg r \geq y \wedge x = qy + r\}$;

but since $x = qy + r$ is implied by $q = 0 \wedge r = x$, and $\neg r \geq y$ implies $r < y$, the rules of consequence give us

$\{q = 0 \wedge r = x\}$
`while r >= y do begin r := r - y; q := q + 1 end`
$\{r < y \wedge x = qy + r\}$.

It is not hard to see how to get from here to the full correctness proof. Note that we several times made use of equivalences and implications between formulae; of course in a fully formal proof we should have to *prove* these equivalences, and that is where first-order logic comes in. The whole proof makes use of both ordinary standard logic and Hoare's special logic of programs. This illustrates the point made above that first-order logic achieves its greatest power when used in conjunction with other formalisms. For further details of methods such as those mentioned here, see for example Dijkstra (1976) or Backhouse (1986).

1.3 Program synthesis

In the quotient-remainder example above we gave the specification and the program independently of one another, and then showed how to prove that the program is correct with respect to the specification. In actual programming practice, though, it is obviously desirable that the construction of the program should be *guided* by the specification, so that it is not as it were just a matter of luck that the program should happen to satisfy it; formal specification methods such as Z and VDM are designed to facilitate this (see Jones, 1986, for VDM, and Diller, 1990 for Z). Ideally, we would like a systematic method for *deriving* the program from the specification, and the goal of a number of researchers is to make the transformation of specification to program so systematic that it can be actually automated, so that the activity of programming is in effect reduced to that of composing specifications. This goal defines the area of *Automated Program Synthesis* (APS).

The approach to APS that makes greatest use of logic is the so-called *Deductive Approach*, pioneered by Manna and Waldinger (1980). The central idea behind the deductive approach is that a formal specification of the form

> Given inputs $x_1, x_2, \ldots, x_n$ satisfying the formula
>
> $$P(x_1, x_2, \ldots, x_n)$$
>
> find outputs $z_1, z_2, \ldots, z_m$ such that the formula
>
> $$Q(x_1, \ldots, x_n, z_1, \ldots, z_m)$$
>
> holds

can be written as a *Specification Theorem*

$$\forall x_1 \cdots \forall x_n (P(x_1, \ldots, x_n) \rightarrow \exists z_1 \cdots \exists z_m Q(x_1, \ldots, x_n, z_1, \ldots, z_m))$$

which has to be true in order for there to exist a program meeting the specification (for if the required outputs do not even exist, then obviously no program can deliver them). To synthesize the required program we attempt to prove the specification theorem; our proof must be *constructive*, i.e., we prove that something exists by showing how to construct it. The program itself then emerges as a by-product of the proof.

Here I shall discuss a variant of the deductive approach which uses the *Constructive Matching* methodology of Fraňová (1988). I shall illustrate this method by showing how it can be used to synthesize a quotient-remainder program. The specification theorem in this case is

$$\forall x \forall y (y > 0 \rightarrow \exists q \exists r (x = qy + r \wedge r < y)).$$

Note that we assume the domain is the set of natural numbers.

The proof of the theorem uses mathematical induction, using x as the induction variable[3]. The *base case* is therefore when $x = 0$, and the theorem reduces in this case to

$$\forall y (y > 0 \rightarrow \exists q \exists r (0 = qy + r \wedge r < y)).$$

Constructive matching requires us to match 0 with $qy + r$; given the axioms for addition and multiplication (in §3 above), there are only two ways of doing this: either by putting $q = r = 0$ or by putting $y = r = 0$. The second conflicts with the condition $y > 0$ in the antecedent, so we are left with the first, for which we must now check that $r < y$ holds; since $r = 0$ and $y > 0$, it does. This gives us the construction 'if $x = 0$, then put $q = 0$ and $r = 0$', which in turn gives us the first part of the synthesized program

```
if x=0 then
   begin
   q:=0; r:=0
   end
else ...
```

For the 'else' part, we use the fact that if $x > 0$ then $x = s(a)$ for some a (where s denotes the successor function). In this case the specification theorem becomes

$$\forall y (y > 0 \rightarrow \exists q \exists r (s(a) = qy + r \wedge r < y)),$$

and we may make use in addition of the *induction hypothesis*[4], which is

$$\forall y (y > 0 \rightarrow \exists q \exists r (a = qy + r \wedge r < y)).$$

Constructive matching requires us to match $s(a)$ with $qy + r$. Without even using the induction hypothesis, we can immediately write down two 'trivial' solutions, namely $q = 0 \wedge r = s(a)$ and $y = 0 \wedge r = s(a)$. The second conflicts with the condition $y > 0$, but the first is acceptable so long as $r < y$, i.e., $s(a) < y$. This gives us the next part of the synthesized program:

[3]The best choice of the induction variable is not always a straightforward matter, and is one of the chief obstacles in the way of a full automation of the procedure we outline; here we go for the one which, as it turns out, leads to the simplest proof.

[4]Here we are assuming as a working hypothesis that the required result holds for the case $x = a$, and we want to show that on this basis it follows that the result must hold for $x = s(a)$; that way, given that it holds for $x = 0$ (the base case), we will have proved that it holds also for $x = s(0), s(s(0)), s(s(s(0))), \ldots$—in short, for every natural number.

```
... if x=s(a) and s(a)<y then
       begin
       q:= 0; r:= s(a)
       end
    else ...
```

which may be simplified to

```
... if x<y then
       begin
       q:= 0; r:= x
       end
    else ...
```

We must now consider the case where $s(a) \geq y$, the 'non-trivial' solutions. For these we must take into account the induction hypothesis. For given y, this guarantees us the existence of values q_1 and r_1 such that $a = q_1 y + r_1$ and $r_1 < y$. To match $s(a)$ with $qy + r$, then, we must try to match $s(q_1 y + r_1)$ with $qy + r$. The axioms for addition and multiplication give us the following possibilities:

1. $qy = 0 \wedge r = s(q_1 y + r_1)$
2. $qy = s(q_1 y + r_1) \wedge r = 0$
3. $qy = q_1 y \wedge r = s(r_1)$
4. $qy = s(q_1 y) \wedge r = r_1$

Since $y > 0$ and $a = q_1 y + r_1$, the first possibility implies $q = 0 \wedge r = s(a)$, which duplicates the trivial solution already found. Of the remaining cases, numbers 2 and 4 lead nowhere since we can't match qy with either $s(q_1 y + r_1)$ or $s(q_1 r)$. This leaves number 3, which can be simplified to $q = q_1 \wedge r = s(r_1)$. This solution is acceptable so long as the condition $r < y$ is satisfied, i.e., $s(r_1) < y$. The axioms for '$<$' (not given here) break this down into $r_1 < y \wedge s(r_1) \neq y$. The induction hypothesis gives us $r_1 < y$, so we are left with $s(r_1) \neq y$ as a condition for this solution to be acceptable. To convert this into the next part of the algorithm, all we need is to note that q_1 and r_1 are the values returned by the quotient-remainder procedure when called with inputs a and y, so we have

```
... begin
    quot_rem(x-1,y,q1,r1);
    if r1+1<>y then
       begin
       q:=q1; r:=r1+1
       end
    else ...
```

(Here we represent a by $x - 1$ since $x = s(a)$.)

Finally we are left with the 'failure case', in which $s(r_1) = y$. This will occur when x is a multiple of y, though we don't need to know this in order to execute the proof. For this case we have to prove a 'missing lemma', namely that the induction hypothesis implies

$$\exists q \exists r (s(a) = s(r_1) q + r \wedge r < s(r_1))$$

(this is got by substituting $s(r_1)$ for y in the earlier formula). We must match $s(a)$ with $s(r_1)q+r$. But substituting $s(r_1)$ for y in the induction hypothesis gives us $a = s(r_1)q_1 + r_1$, which implies (via the addition and multiplication axioms) $s(a) = s(r_1)s(q_1)$. So we are really trying to match $s(r_1)s(q_1)$ with $s(r_1)q + r$, which we can do by putting $q = s(q_1) \wedge r = 0$. This enables us to complete the algorithm as

```
        begin
        q:=q1+1; r:=0
        end
```

Putting it all together, then, we have succeeded in synthesizing the following recursive procedure for computing quotient and remainder:

```
procedure quot_rem(x,y: integer; var q,r: integer);

begin
if x=0 then
   begin
   q:=0; r:=0
   end
else if x<y then
        begin
        q:= 0; r:= x
        end
     else
         begin
         quot_rem(x-1,y,q1,r1);
         if r1+1<>y then
            begin
            q:=q1; r:=r1+1
            end
         else begin
              q:=q1+1; r:=0
              end
         end
end.
```

This program may not be maximally efficient; it may be necessary to transform it in some way to improve it in this respect. Nonetheless it is impressive that by an almost purely mechanical procedure we have been able to synthesize a program from the specification at all! It is guaranteed to be correct so long as the condition $y > 0$ is satisfied.

The goal of APS is to take techniques like this and refine them so that they can be applied practically to much more complicated cases, the sort of cases that one is likely to encounter in 'real life'. It has to be said that we are as yet nowhere near to achieving this goal; nonetheless I think that the enterprise is a good illustration of the power and insight that purely mechanical operations on logical formalisms can provide.

1.4 Logic Programming

The drive towards automated synthesis opens up the prospect of programming by writing specifications: that is, instead of writing a program, the programmer writes a specification which is then automatically converted into a program. In a sense this is already something we are doing all the time, for what else is a program written in a high-level programming language but a specification for the low-level code into which it is compiled? However, there is another sense in which it is quite misleading to regard a Pascal program, say, as a specification, and that is that it already consists of a sequence of *instructions* which the computer is to follow in order to derive the output from the input rather than a bare statement of the relations that are to hold between the input and the output. Thus Pascal, and the majority of other widely-used programming languages which resemble it in this respect, is an *imperative* language, whereas ideally a specification language should be purely *declarative*, i.e., it should enable one to describe conditions on the input-output relation independently of any particular procedure for realizing them.

If, therefore, we are ever to program by writing specifications, we require our programming languages to be declarative, just like our specification languages. A program written in such a language could then be regarded as an *executable specification*. There are two main classes of declarative programming languages in existence, the *functional* languages, of which Lisp is a rather impure example, and Miranda a purer one, and the *logic programming* languages, of which so far only Prolog has come to be widely used, although a recently developed language called Gödel seem to be gaining in popularity.

In a functional language the input-output relation is specified as a function whose values for a given set of inputs is the corresponding set of outputs. The language thus consists of a set of expressions denoting primitive functions and operations for constructing new functions from old, and the task of implementing it is that of specifying algorithms for evaluating the complex functional expressions built up in this way. Several different logical systems may be used to provide the foundation for functional languages, for example the λ-calculus or Martin-Löf's intuitionistic type theory. On the latter, see section 2.1.

Turning now to logic programming, the central idea here is that a statement of the form 'if P then Q' can be regarded either declaratively, as asserting something which may or may not be true, or *procedurally*, as telling you that if you want to know whether Q is true, you should try finding out whether P is true. Suppose, for example, we have the following set of rules and facts:

1. If it is sunny then it is warm.
2. If it is daytime and there are no clouds then it is sunny.
3. It is daytime.
4. There are no clouds.

and suppose we are asked 'Is it warm?'. From 1 we know that it is warm if it is sunny, so that in effect we have replaced the original question by a new question, 'Is it sunny?'. From 2 we know that it is sunny if it is daytime and there are no clouds, so now we can replace our question by the two questions 'Is it daytime?' and 'Are there no clouds?'. From 3 and 4 we know that the answer to each of these questions is 'yes'; this gets transmitted back up the chain of questions to give us an answer 'yes' to the original question 'Is it warm?'.

This example could be translated directly into Prolog, and the resulting code might look like:

```
warm :- sunny.
sunny :- daytime, no_clouds.
```

```
daytime.
no_clouds.
```

With this program loaded, we can then pose the query

```
?- warm.
```

to the Prolog interpreter, which will then set in train a sequence of moves which amount to a proof that the statements in the program logically imply the statement `warm`.

This example makes use of no resources beyond the Propositional Calculus, and in fact only involves a restricted subset of formulae known as *Horn clauses*. A Horn clause in the Propositional Calculus is a formula of the form

$$B_1 \wedge B_2 \wedge \cdots \wedge B_n \rightarrow H,$$

where the B_i and H are atomic formulae (i.e., single schematic letters). This corresponds to the Prolog clause

```
H :- B1, B2, ..., Bn.
```

The logic of Horn clauses has a sound and complete proof theory which uses a single inference rule, called *resolution*, which says that given the clauses

$$B'_1 \wedge B'_2 \wedge \cdots \wedge B'_n \rightarrow H'$$

and

$$B_1 \wedge B_2 \wedge \cdots \wedge B_m \rightarrow H,$$

if $H' \equiv B_i$ for some i then we may infer the clause

$$B_1 \wedge \cdots \wedge B_{i-1} \wedge B'_1 \wedge B'_2 \wedge \cdots \wedge B'_n \wedge B_{i+1} \wedge \cdots \wedge B_m \rightarrow H.$$

Logic programming would be of little use if it did not go beyond the Propositional Calculus, but in fact the ideas presented above can be extended in a rather natural way to the Predicate Calculus. In this setting, a Horn clause is now defined as a formula having the form

$$\forall x_1 \forall x_2 \cdots \forall x_k (B_1 \wedge B_2 \wedge \cdots \wedge B_n \rightarrow H)$$

where each B_i and H is an atom, that is a formula obtained by filling in the argument places of a simple predicate by constants, variables, or complex terms derived from these using function symbols, and $x_1, x_2, \ldots, x_n$ are all the variables that occur in any of the B_i or H. To extend the resolution procedure to Horn clauses of this kind, we make use of a pattern-matching algorithm called the *unification algorithm* which enables us to determine whether two atoms have a common instance. For example, given the two clauses $\forall x \forall y (P(x) \wedge Q(g(x), y) \rightarrow R(y))$ and $\forall z (S(z) \rightarrow Q(z, f(z)))$, we can observe that the atoms involving Q can be unified by substituting $g(x)$ for z and $f(g(x))$ for y. We can therefore resolve the two clauses to give $\forall x (P(g(x)) \wedge S(g(x)) \rightarrow R(f(g(x))))$.

Prolog notation dispenses with quantifiers since Horn clauses do not contain existential quantifiers and all the universal quantifiers are placed in a string at the head of the clause. Thus the two examples above would be written as

```
r(Y) :- p(X), q(g(X),Y).
q(Z,f(Z)) :- s(Z).
```

Note Prolog's odd convention of writing predicates, function symbols and constants beginning with lower case letters and variables with upper case.

To illustrate the use of Prolog as an executable specification language, we shall consider some simple list-processing tasks. Prolog notates a list in the form `[X|Xs]`, where `X` is the first element of the list and `Xs` is tail of the list, i.e., the list consisting of the remaining elements. An individual list with known members may be written out in full as, e.g.,

```
[monday, tuesday, wednesday, thursday, friday, saturday, sunday].
```

The empty list is written as `[]`. Suppose we want to specify the relation $Reverse$ in which a list stands to that list which contains the same elements but in the reverse order. If we know that the reverse of a list $L = (l_1, l_2, \ldots, l_n)$ is $R = (l_n, l_{n-1}, \ldots, l_1)$, then we know that the reverse of the list $[h|L] = (h, l_1, l_2, \ldots, l_n)$, obtained by adding an element h to the beginning of L, will be the list S obtained by adding h to the *end* of R. We also know that the reverse of the empty list is itself. This gives us two Prolog clauses

```
reverse([],[]).
reverse([H|L],S) :- reverse(L,R), add_to_end(H,R,S).
```

We must also specify the relation Add_to_end: we know that the result of adding an element e to the end of the empty list is the one-element list (e); also, if the result of adding e to the end of a list $L = (l_1, l_2, \ldots, l_n)$ is the list $M = (l_1, l_2, \ldots, l_n, e)$, then the result of adding e to the end of the list $[h|L] = (h, l_1, l_2, \ldots, l_n)$ will be $[h|M] = (h, l_1, l_2, \ldots, l_n, e)$. The required Prolog clauses are thus

```
add_to_end(E,[],[E]).
add_to_end(E,[H|L],[H|M]) :- add_to_end(E,L,M).
```

The four clauses we have written down, which express in a purely declarative way the basic facts about the $Reverse$ and Add_to_end relations, can now be used to answer a query such as

```
?- reverse([a,b,c,d,e],X).
```

to which the Prolog interpreter will duly come up with the answer

```
X = [e,d,c,b,a].
```

It would be idle to pretend that Prolog fulfils all the requirements of an executable specification language: manifestly it does not, as even its most enthusiastic devotees will admit. The way the Prolog interpreter works is sensitive to the ordering of atoms within a clause and to the ordering of clauses within a program; programs which are equivalent from a declarative point of view can turn out to have quite different behaviour in practice—it can often happen, for example, that by changing the order of the clauses one can turn a program which always terminates into a program that never does. On the other hand, merely swapping around clauses in a correct program will never generate a program that is incorrect in the sense of delivering wrong answers, the worst that can happen is that instead of delivering correct answers it fails to deliver anything. But Prolog has in addition a number of *non-logical*, i.e., purely procedural features, most notably the infamous 'cut' operator (written '`!`'), which gives the programmer control over which parts of the search space are examined, and as a result *can* result in incorrect programs unless one is very careful to observe the procedural niceties in one's programming. Another source of possible error is the (again infamous) 'negation by failure' operator (`not`) which was introduced in an attempt to circumvent the limitation to Horn clauses by means of a procedural definition of negation. (For more on this, see section 2.4 on non-monotonic reasoning.)

Thus Prolog is not perfect, nor was it ever claimed to be. For all its faults, though, it has proved to be a very congenial medium in which to encode certain kinds of programming tasks, notably those which essentially involve *recursion* (such as our list-processing examples above) and which figure prominently in Artificial Intelligence. It also serves as a pointer to what might be achieved once the problems have been ironed out. For further details on Prolog, see Sterling and Shapiro (1986), and for Logic Programming generally, Hogger (1990).

2 Beyond Classical Logic

Although classical logic, i.e., the first-order Predicate Calculus, can be, when appropriately handled, a formidable tool for representing and reasoning about almost any domain, it is not the last word in formal logic. During the present century logicians, mathematicians, philosophers and computer scientists have studied a wide range of alternative formalisms designed for specific applications which do not appear to be easily handled by classical logic. In this section we shall briefly review a number of these formalisms, with particular emphasis on those that have attracted the attention of computer scientists.

There are broadly speaking two ways of devising a non-classical logic: one can either take the language of classical logic unaltered, but reinterpret the logical constants (i.e., the connectives and the quantifiers) so that the class of formulae that count as logically true or inferences that count as valid is altered; or one can alter the language itself by introducing new logical constants.

2.1 Intuitionistic Logic

Intuitionistic Logic belongs to the first of these two categories, in that it does not extend the syntax of classical logic, but reinterprets the connectives so that they are no longer truth-functional. Intuitionists are very much concerned with the grounds one might have for asserting a proposition: it is no good just saying that a formula is true, one's conviction of its truth must be grounded in some concrete intuition. As a consequence of this, the intuitionist is disinclined to accept as valid such classical theorems as the Law of the Excluded Middle $P \vee \neg P$ or the Law of Double Negation $\neg\neg P \rightarrow P$. In the former case, the intuitionist would say that one is only warranted in asserting a formula of the form $A \vee B$ if either one is warranted in asserting A or one is warranted in asserting B—and clearly there are cases where one is not warranted either in asserting A or $\neg A$. In mathematics, for example, one may not be able to prove either a proposition or its negation; this is currently the case with Goldbach's conjecture that every even number greater than 2 is the sum of two primes. If we represent this conjecture by G, then the classical logician will be quite happy to assert that $G \vee \neg G$, and will be prepared to use this assertion as a premiss in a proof; the intuitionist however will not be prepared to assert this until a proof of either G or $\neg G$ is available. In the case of the Law of Double Negation, the intuitionist interprets $\neg P$ to mean that one has well-grounded reasons for denying P; so $\neg\neg P$ says that the supposition that one has a proof that P is false is untenable, and the intuitionist will not accept that this amounts to a proof of P, since it is quite possible that *neither* P nor $\neg P$ can be proved.

Intuitionists demand a similar grounding for the existence of objects. One is only warranted in saying that something exists if one has a constructive means to exhibit it. This places limitations on the conditions under which an intuitionist is prepared to accept a formula of the form $\exists x P(x)$. For example, whereas classical mathematicians will accept as a proof of the existence of transcendental (i.e., non-algebraic) numbers the fact that the class of real numbers is of higher cardinality than the class of algebraic numbers, the intuitionist will not be satisfied until some actual number is proved to be transcendental. Again, to the question 'Are there irrational numbers x and y such that x^y is rational?', the classical mathematician will accept the following argument: we know that $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$, 2 is rational, $\sqrt{2}$ is irrational, and either $\sqrt{2}^{\sqrt{2}}$ is rational or it is not; in the former case we can put $x = y = \sqrt{2}$ and in the latter case we can put $x = \sqrt{2}^{\sqrt{2}}$, $y = \sqrt{2}$. The intuitionist, on the other hand, will not accept this until it is shown which of the alternative pairs of values for x and y gives the desired result.

For an intuitionist, the only thing that counts as a proof of a formula $A \vee B$ is either a proof of A or a proof of B (together with an indication of which of these has been proved!); the only thing that counts as a proof of $A \rightarrow B$ is a procedure for converting a proof of A into a proof of B; and the only thing that counts as a proof of $\exists x P(x)$ is a method for constructing an object that satisfies the predicate P. In effect, Intuitionistic Logic has a different conception of the meaning of a formula from Classical Logic. In Classical Logic, the meaning of a formula can be characterised in terms of the class of interpretations which satisfy it, and the meanings of the connectives and quantifiers are specified in terms of the contribution they make to the way in which this class is determined. In Intuitionistic Logic, on the other hand, the meaning of a formula, insofar as it is considered at all, is characterised in terms of the class of possible proofs of the formula, and the meanings of the logical constants are accordingly to be specified in terms of the constraints they place on what is to count as proofs of formulae containing them. For a good introduction to Intuitionistic Logic, see Dummett (1977).

Intuitionistic logic is held by many to be the natural medium for reasoning about computation because of its emphasis on constructive methods. In the Intuitionistic Type Theory of Martin-Löf (Martin-Löf, 1985), a proposition is now *identified* with a class of acceptable proofs, and is regarded as a *type* of which the individual proofs are exemplars ('proof objects'). The theory provides rules which enable one to derive an expression for an exemplar of a type from an expression for the type itself; in effect this amounts to having a constructive proof theory for the language of types, since an exemplar of a type is a constructive proof of the proposition which the type is taken to represent. Taken together, the rules define a highly abstract general theory which admits of various different interpretations. In particular, the relation between a specification and a successful implementation of it is regarded as analogous to that between a proposition and a constructive proof of it, and hence the types can be interpreted as programming tasks or specifications, and their instances as implementations. In this way the formal apparatus of the type theory can be brought to bear on the tasks of program specification, verification, and synthesis, since expressions representing types can be treated as specifications, expressions representing their exemplars can be treated as implementations, and the rules for constructing exemplars from types can therefore be regarded as procedures (albeit not fully mechanical) for synthesizing programs from specifications.

2.2 Temporal Logic

The objects studied in mathematics are by nature changeless, and therefore *time* does not play an important part in mathematical reasoning. In computation, however, time is an important element because the execution of a program takes time, and the values of the program variables change in the course of execution. In order to reason about the behaviour of programs, therefore, the time factor has to be considered. This is especially important in the case of *concurrent* programs, in which more than one processor is involved, each making use of results produced by others, making it important to coordinate the separate execution sequences to enable them to interact in the way required for successful completion of the computation.

Temporal Logic is a special form of logic designed to handle the time dimension, and it has been found useful for specifying the behaviour of concurrent processes. It augments the syntax of the Predicate Calculus by the addition of *temporal operators*. A typical selection of such operators are $\Box$, $\Diamond$, and $\bigcirc$, whose meanings are given as follows:

- $\Box A$ is true at time t so long as A is true at t and at every later time;
- $\Diamond A$ is true at time t so long as A is true at t or at some later time;
- $\bigcirc A$ is true at time t so long as A is true at the time immediately following t.

Notice that the reference to the time immediately following t in the definition of $\bigcirc$ presupposes that time is treated as a discrete sequence, with an ordering relation like that of the integers; this is a reasonable supposition if the 'times' one is talking about are successive states of a computation, but in a more general context it is unwarranted, and time may alternatively be assumed to be dense like the rational numbers, continuous like the real numbers, or even partially ordered (for example in reasoning about distributed systems, in which the separate computation sequences are to some extent independent of one another).

Temporal formulae can be used to specify the meaning of constructs in a programming language. Let us posit a function $\mathcal{F}$ which assigns to a piece of program code a temporal formula which will be true at a time if and only if that code is executed at that time[5]. The function $\mathcal{F}$ may be specified *compositionally*: both programs and temporal formulae are built up recursively from a basic stock of primitives, and the recursive definition of $\mathcal{F}$ will detail how the temporal formula to be associated with a given composite program is constructed from the temporal formulae associated with the parts out of which the program is built. Thus we need only define $\mathcal{F}$ explicitly for the primitive program elements, and let the compositional rules do the rest. For simplicity we can assume that a primitive program element a can be read either imperatively as an instruction or declaratively as an assertion that the instruction is carried out; thus the primitive elements belong to both the programming language and the temporal logic, and we can put $\mathcal{F}(a) = a$ for each such element a.

We shall illustrate the compositional rules with some examples. The meaning of the *sequential composition* construct, denoted '$\circ$', might be given by the rule

$$\mathcal{F}(a \circ P) = a \wedge \bigcirc\mathcal{F}(P),$$

which says that an execution of $a \circ P$ at a time consists of an execution of a at that time followed by an execution of P at the immediately following time. As another example, the meaning of a conditional instruction might be given by the rule

$$\mathcal{F}(\text{if } c \text{ then } P_1 \text{ else } P_2) = (c \wedge \mathcal{F}(P_1)) \vee (\neg c \wedge \mathcal{F}(P_2)).$$

Finally, for the concurrent programming construct $P_1 \parallel P_2$, which is an instruction to perform P_1 and P_2 in parallel, we might put

$$\mathcal{F}(P_1 \parallel P_2) = \mathcal{F}(P_1) \wedge \mathcal{F}(P_2).$$

This is, it must be stressed, a greatly simplified account; in practice one works with more complicated semantic rules, the details of which will depend on what model of parallel computation

[5]More exactly, it is necessary to specify the environment within which the code fragment is executed; the brief account given here is of necessity vastly over-simplified.

one has in mind. For example, we may be thinking of simulating parallel processing on a sequential machine by means of *interleaving* the parallel execution sequences. In that case, we can introduce an *idle* primitive i, arbitrarily many copies of which can be inserted into the execution sequences for a given program fragment; then two fragments P_1 and P_2 may be interleaved by allowing the primitive actions belonging to each process to occur at times when the other process is idle (for details, see Barringer, 1987).

What does this temporal semantics buy us? Primarily, it enables programs to be proved correct with respect to specifications composed in temporal logic. The specification may require that some actions a and b both occur infinitely often; this is represented by the temporal formula

$$\Box(\Diamond a \wedge \Diamond b).$$

To show that the program P meets the requirement, we must prove that this formula is a logical consequence of the formula $\mathcal{F}(P)$. For this purpose a proof system is supplied for the temporal language in which these formulae is expressed.

Temporal Logics have been used for other purposes than program specification, notably in formal approaches to knowledge representation in Artificial Intelligence. In these applications one makes use of additional temporal operators for looking into the past as well as the future, and a proof system must include rules describing their interaction (e.g., if it is now the case that A then it will always be the case that it *has* been the case that A—in symbols, $A \rightarrow GPA$). For further details, see Galton (1987) and Goldblatt (1987).

2.3 Modal Logic

Temporal Logic assigns different truth-values to the same formula at different times, thus in effect allowing us to talk about a temporal sequence of states of the world. Modal Logic generalizes this idea to a collection of possible world-states related to each other in an arbitrary way, not necessarily in the form of a temporal sequence[6]. A model for Modal Logic thus consists of a set W of *possible worlds* with an *accessibility relation* R defined on them so that for each world we can say which worlds are accessible from it (or count as possible relative to it), one of the worlds, say w_0, being singled out as the *actual world*. Each atomic formula of the language is now assigned a truth-value at each possible world; a convenient way of doing this is to define a function V which assigns to each atomic formula the set of possible worlds at which it is true—thus for a formula A we have $V(A) \subseteq W$. The language of Modal Logic includes all the usual connectives together with two *modal operators* $\Box$ and $\Diamond$ which may be read 'necessarily' and 'possibly'. The formal semantics of the modal operators makes use of the idea, due to Leibniz, that necessity may be defined as truth in all possible worlds, and possibility as truth in *some* possible world. Thus for any formula A and world w we put

- $w \in V(\Box A)$ iff for every world w' accessible from w, $w' \in V(A)$;
- $w \in V(\Diamond A)$ iff for some world w' accessible from w, $w' \in V(A)$.

Notice that if we think of worlds as times, and read 'accessible from' as 'not earlier than' then the meanings of the modal operators become precisely what we gave for the temporal operators in the discussion of Temporal Logic. But the accessibility relation does not have to be a temporal ordering;

[6]Historically, it was the other way round. Modal Logic—as the logic of necessity and possibility—dates back to work published by C. I. Lewis in the period 1912–18, whereas Temporal Logic had to await the seminal work of Arthur Prior in the 1950s.

it may be transitive or intransitive, symmetric or asymmetric, reflexive or irreflexive, and so on. Each combination of choices gives rise to a separate modal logic with its own rules. Good general introductions to Modal Logic are Hughes and Cresswell (1968, 1984) and Chellas (1980).

Capable as it is of assuming so many forms, the potential applications of Modal Logic are quite diverse, ranging from reasoning about programs, to reasoning about mathematical proofs, to reasoning about causality, knowledge, or belief. *Dynamic Logic* is a form of modal logic used for describing the behaviour of non-deterministic programs (see Harel, 1984). With non-determinism, when an instruction is executed in a particular state, several resulting states may be possible; for example, the instruction to assign to the variable x an integer randomly chosen from the range $0, \ldots, 3$ can result in a state in which any one of the equations $x = 0, x = 1, x = 2, x = 3$ holds[7]. One often needs to know that some proposition will hold in *whichever* state results from executing a particular instruction, or again that some proposition *may* hold. For this purpose a special *multi-modal* logic is used in which there are infinitely many pairs of modal operators, parametrized by pieces of program code. For a given piece of code p, the modal operators $[p]$ and $\langle p \rangle$ characterize the accessibility relation on program states defined by p, so that the formula $[p]A$ says that A holds in every state accessible from the current state by execution of p, while $\langle p \rangle A$ says that A holds in *some* such state. In our example of the instruction to pick a random number in the range $0, \ldots, 3$ we can put, for example, $[x \leftarrow rand(3)]0 \leq x \leq 3$ and $\langle x \leftarrow rand(3) \rangle x = 0$.

Dynamic Logic provides rules for handling the modal operators associated with complex instructions derived by the application of programming constructs to simpler ones. For example, the *non-deterministic choice* operator, which we may symbolize $\sqcup$, is used in an instruction to perform one of a set of instructions non-deterministically. Then we have

$$[p_1 \sqcup p_2]A \text{ iff } [p_1]A \wedge [p_2]A.$$

If we represent our instruction $x \leftarrow rand(3)$ as

$$p \equiv x \leftarrow 0 \sqcup x \leftarrow 1 \sqcup x \leftarrow 2 \sqcup x \leftarrow 3$$

then since for $i = 0, \ldots, 3$ we have

$$[x \leftarrow i]0 \leq x \leq 3,$$

it follows from the rule for $\sqcup$ that

$$[p]0 \leq x \leq 3.$$

The goal of Dynamic Logic is to apply this kind of reasoning recursively to prove theorems about the behaviour of a complex program, for example a theorem of the form

$$A \rightarrow [p]B,$$

which can be read as saying that the program p transforms an initial state in which A holds into a final state in which B holds—in other words, that p is correct with respect to the specification 'derive B from A'.

It is instructive to compare Dynamic Logic with Temporal Logic. Both of them use modal operators to reason about a structured set of states; in the case of Temporal Logic, these are the sequence of states passed through in the execution of a program, whereas in the case of Dynamic Logic they are the different possibilities that can result from the execution of a program. There is no

[7]Non-determinism may also arise in parallel processing as a result of the unpredictable synchronization of the processors, which may depend on aspects of system usage, etc., which lie outside the control of the program itself.

reason why these two pictures should not be combined, to give a logic of *possible execution sequences* in which there are two kinds of modality, one for looking forward along individual execution paths and one for looking as it were orthogonally to this across the range of different execution paths. For this purpose, Temporal Logic has been extended to a logic of so-called 'branching time' (Ben-Ari, *et al.*, 1981), in which there are two sets of modal operators, enabling one to express distinctions such as that between

> A is true at some time along every execution path

and

> A is true at every time along some execution path

which obviously correspond to very different types of program behaviour.

2.4 Logics for Non-monotonic Reasoning

All the logics we have considered so far are *monotonic*, that is, their inference rules are such that if we can infer a conclusion C from a set of premisses Π then we can still infer C if we add extra premisses to Π to form an enlarged set Π', where $\Pi \subset \Pi'$. The monotonicity derives from the way that validity is characterized in terms of possible interpretations: to say that the premiss set Π implies the conclusion C is to say that C is satisfied in every interpretation which satisfies each member of Π. Now any interpretation which satisfies each member of Π' obviously also satisfies each member of Π, since each member of Π *is* a member of Π'. Hence C is true in every interpretation which satisfies each member of Π', i.e., Π' implies C, so we have monotonicity.

In some applications, however, it appears that the reasoning we are trying to formalize is *non-monotonic*; sometimes the addition of a new piece of information can force us to retract a conclusion which we had drawn on the basis of the information available to us earlier. For example, suppose I look at the railway timetable to see if I can get to Leeds in time for a seminar at 4 p.m., given that I am lecturing in Exeter from 11 a.m. to noon. I find that there is no train that will get me there in time, and conclude that if I am to attend the seminar I shall have to cancel the lecture. Then I find that a supplement to the timetable has been issued which gives details of extra trains running at certain times, and as luck would have it one of these leaves Exeter at 12.30 p.m. and arrives in Leeds at 3.25 p.m. I now retract my earlier conclusion and decide that I can both give my lecture in Exeter and attend the seminar in Leeds.

We reason in ways similar to this all the time; probably more of our everyday reasoning is non-monotonic than not. For this reason, it might appear that the kinds of logical formalism we have looked at above are ill-adapted to handling everyday modes of reasoning, and many researchers, particularly in Artificial Intelligence, have sought to remedy this by devising logical systems containing non-monotonic inference rules.

The simplest example of this is the *Negation by Failure* rule in Prolog. This is an attempt to model exactly the kind of reasoning illustrated in our timetable example. Such reasoning makes use of what is known as the *Closed World Assumption*, which amounts to assuming that the information you have is complete. As applied to a timetable, the Closed World Assumption is that if the timetable doesn't say there is a train running between two places at a given time, then there isn't such a train. Normally this is a reasonable assumption to make, since the compilers of timetables make every effort to ensure that no information is missed out. More generally, the Closed World Assumption applied to a set of propositions (or database) Δ is that if Δ does not imply A, then we may infer that $\neg A$ is true. The Negation by Failure rule implements this as follows: to prove $\neg A$, try all possible ways of proving A, and consider $\neg A$ proved just so long as you fail. In practice, Negation by Failure is only an imperfect implementation of the Closed World Assumption, as it is quite possible that the systematic attempt to prove A leads into an infinite regress, or a loop, so that the attempt never terminates in failure. However, when handled carefully, Negation by Failure (which is the only kind of negation available to Prolog) is an indispensible tool which greatly enhances the power of Prolog as a programming language.

To see why inference under the Closed World Assumption is bound to be non-monotonic, we need only reflect that if Δ does not imply A, then using the Closed World Assumption we have

From Δ we may infer $\neg A$;
from $\Delta \cup \{A\}$ we may not infer $\neg A$;
and $\Delta \subset \Delta \cup \{A\}$

which amounts to saying that our inference rule is non-monotonic. An unkind way of putting it would be to say that the inference rule is invalid—which of course, strictly speaking, it is, but the point is that often it is not practicable to reason validly all the time, and what is needed is a systematic and responsible codification of the kinds of plausible but invalid reasoning that are useful to us in everyday life.

A large number of different non-monotonic formalisms have been proposed; a good survey is Etherington (1988). Many of them can be described semantically in terms of restrictions on what is to count as a model for a set of formulae. In classical logic, *any* interpretation that satisfies the formulae is counted as a model, and validity can be defined as truth of the conclusion in all possible models of the premisses. Validity so defined is monotonic since any model for an extended set of premisses is automatically a model for the original set. If now we select only certain interpretations as models of a given formula then validity can be non-monotonic because the *selected* models of the extended set of premisses need not be selected models of the original set. (For example, the Closed World Assumption in effect selects models which are minimal in the sense that no smaller model will satisfy the formula; but a minimal model for $\Delta \cup \{A\}$ will not be a minimal model for Δ unless Δ actually implies A.) The task of formalizing non-monotonic inference amounts, in effect, to defining a set of purely formal criteria for selecting the preferred models—if, as seems to me quite likely, the criteria we use in practice are dependent on the specific content of the premisses, then it would seem that no formal criteria can exist, and we must look elsewhere than formal logic for a proper treatment of non-monotonic reasoning.

3 Concluding Remarks

In the space available I have only been able to scratch the surface of the wide variety of ways in which logic has been applied as a formal method in computer science. There are many types of logic, of interest to computer scientists, which I have not mentioned at all, for example Many-Sorted Logic, Second-Order Logic, Many-Valued Logic, Conditional Logic, Relevant Logic, Deontic Logic, Linear Logic, and so on. The aim of my survey has been to convince you that Formal Logic has an important role to play in Computer Science, and to give some idea of the variety of services it can deliver. For further information, the reader is encouraged to follow up the references cited below.

Acknowledgments

I should like to thank John Gooday, Brian Lings, and two anonymous referees for their helpful remarks on the first draft of this article.

References

Roland C. Backhouse (1986), *Program Construction and Verification*. Englewood Cliffs, NJ: Prentice-Hall International.

Howard Barringer (1987), 'The Use of Temporal Logic in the Compositional Specification of Concurrent Systems', in Galton (1987).

M. Ben-Ari, A. Pnueli, and Z. Manna (1981), 'The Temporal Logic of Branching Time', *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*.

Brian F. Chellas (1980), *Modal Logic: An Introduction*. Cambridge: Cambridge University Press.

Edsger W. Dijkstra (1976), *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall.

Antoni Diller (1990), *Z: An Introduction to Formal Methods*. Chichester: John Wiley and Sons.

Michael Dummett (1977), *Elements of Intuitionism*. Oxford: Oxford University Press.

David Etherington (1988), *Reasoning with Incomplete Information*. Los Altos, CA: Morgan Kaufman.

Marta Fraňová (1988), *Fundamentals of a New Methodology for Program Synthesis from Formal Specifications: CM-construction of Atomic Formulae*, Ph.D. Thesis, Université de Paris-Sud, Orsay.

Antony Galton (1987), *Temporal Logics and their Applications*, London: Academic Press.

Robert Goldblatt (1987), *Logics of Time and Computation*, Stanford CA: Center for the Study of Language and Information.

David Harel (1984), 'Dynamic Logic', in D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic*, Volume II. Dordrecht: D. Reidel.

C. A. R. Hoare (1969), 'An Axiomatic Basis for Computer Programming', *Communications of the ACM*, Vol. 12, pp. 576–583.

Christopher Hogger (1990), *Essentials of Logic Programming*. Oxford: Clarendon Press.

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

G. E. Hughes and M. J. Cresswell, *A Companion to Modal Logic*, Methuen.

Cliff B. Jones (1986), *Systematic Software Development Using VDM*. Englewood Cliffs, NJ: Prentice-Hall International.

Zohar Manna and Richard Waldinger (1980), 'A Deductive Approach to Program Synthesis', *ACM Transactions on Programming Languages and Systems*, Vol. 2, pp. 90–121.

P. Martin-Löf (1985), ‘Constructive Mathematics and Computer Programming’, in C. A. R. Hoare and J. C. Shepherdson (eds.), *Mathematical Logic and Programming Languages*, Prentice-Hall.

Leon Sterling and Ehud Shapiro (1986), *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass.: MIT Press.