PREV
A. The Pseudocode Notati

NEXT
Programming Problems

Appendix B. Review of Mathematical Logic

Mathematical logic is used to state correctness properties and to prove that a program or algorithm has these properties. Fortunately, the logic needed in this book is not very complex, and consists of the propositional calculus (which is reviewed in this appendix) and its extension to temporal logic (Chapter 4). Variables in concurrent algorithms typically take on a small number of values, so that the number of *different* states in a computation is finite and relatively small. If a complicated calculation is required, it is abstracted away. For example, a concurrent algorithm may calculate heat flow over a surface by dividing up the surface into many cells, computing the heat flow in each and then composing the results. The concurrent algorithm will *not* concern itself with the calculations using real numbers, but only with the division of the calculation into tasks, and the synchronization and communications among the tasks.

For more on mathematical logic in the context of computer science, see my textbook [9].

The Propositional Calculus

Syntax

We assume the existence of an unbounded set of atomic propositions {*p*, *q*, *r*, ... }, one unary operator ¬ (*not*), and four binary operators, ∨ (*disjunction*, *or*), ∧ (*conjunction*, *and*), → (*implication*) and ↔ (*equivalence*). A *formula* of the propositional calculus is constructed inductively as the smallest set such that an atomic

NOTE

In an implication $(p \rightarrow q)$, p is called the *antecedent* and q is called the *consequent*.

As with expressions in programming languages, parentheses can be omitted by assigning a precedence to each operator: $\neg$ has the highest precedence, followed in descending order by $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. We can further omit parentheses since it follows from the definition of the semantics of formulas that all the binary operators are associative except for $\rightarrow$. Therefore, the meaning of the formula $(p \vee q) \wedge r$ is the same as that of $p \vee (q \vee r)$, and it can be written $p \vee q \vee r$. However, $p \rightarrow q \rightarrow r$ must be written as $(p \rightarrow q) \rightarrow r$ or $p \rightarrow (q \rightarrow r)$, because the meanings of the two formulas are different.

Here are some examples of formulas of the propositional calculus:

p, $p \rightarrow (q \rightarrow p)$, $(\neg p \rightarrow \neg q) \leftrightarrow (q \rightarrow p)$, $(p \vee q) \rightarrow (p \wedge q)$, $p \rightarrow \neg p$.

Semantics

The semantics of a formula is obtained by defining a function v, called an *assignment*, that maps the set of atomic propositions into $\{T, F\}$ (or $\{true, false\}$). Given an assignment, it can be inductively extended to an *interpretation* of a formula (which we will also denote by v) using the following rules:

A	v(A1)	v(A2)	v(A)
$\neg A_1$	T		F
$\neg A_1$	F		T
$A_1 \vee A_2$	F	F	F
$A_1 \vee A_2$	otherwise		T
$A_1 \wedge A_2$	T	T	T
$A_1 \wedge A_2$	otherwise		F
$A_1 \rightarrow A_2$	T	F	F

A	v(A1)	v(A2)	v(A)
$A_1 \rightarrow A_2$	otherwise		T
$A_1 \leftrightarrow A_2$	$v(A_1) = v(A_2)$		T
$A_1 \leftrightarrow A_2$	$v(A_1) \neq v(A_2)$		F

A formula f is said to be *true* in an interpretation v if $v(f) = T$ and *false* if $v(f) = F$. Under the assignment $v(p) = T$, $v(q) = F$, the formula $p \rightarrow (q \rightarrow p)$ is true, as can be seen by inductively checking the interpretation using the above table: $v(q \rightarrow p) = T$ since $v(q) = F$ and $v(p) = T$, from which it follows that $v(p \rightarrow (q \rightarrow p)) = T$. On the other hand, the formula $(p \vee q) \rightarrow (p \wedge q)$ is false in the same interpretation, since $v(p \vee q) = T$ and $v(p \wedge q) = F$, which is precisely the one case in which the implication is false.

A formula f is *satisfiable* if it is true in *some* interpretation, and it is *valid* if it is true in *all* interpretations. Similar definitions can be given in terms of falsehood: A formula f is *unsatisfiable* if it is false in *all* interpretations, and it is *falsifiable* if it is false in *some* interpretation. The two sets of definitions are *dual*: f is satisfiable if and only if $\neg f$ is falsifiable, and f is valid if and only if $\neg f$ is unsatisfiable. It is easy to show that a formula is satisfiable or falsifiable since you only have to come up with one interpretation in which the formula is true or false, respectively; it is difficult to show that a formula is valid or unsatisfiable since you have to check *all* interpretations.

We have already shown that $(p \vee q) \rightarrow (p \wedge q)$ is not valid (falsifiable) by finding an assignment in which it is false. We leave it to the reader to check that $p \rightarrow (q \rightarrow p)$ is valid, while $(p \vee q) \rightarrow (p \wedge q)$, though not valid, is satisfiable.

Logical Equivalence

Two formulas are *logically equivalent* to each other if their values are the same under all interpretations. Logical equivalence is important, because logically equivalent formulas can be substituted for one another. This is often done implicitly, just as we replace the arithmetic expression 1 + 3 by the expression 2 + 2 without explicit justification.

The two formulas $p \rightarrow q$ and $\neg p \vee q$ are logically equivalent because they are both false in any interpretation in which $v(p) = T$, $v(q) = F$, and are both true under all other interpretations. Other important equivalences are *deMorgan's laws*: $\neg (p \wedge q)$ is logically equivalent to $\neg p \vee \neg q$ and $\neg(p \vee q)$ is logically equivalent to $\neg p \wedge \neg q$. For a more complete list, see [9, Section 2.4].

Induction

The concept of proof by numerical induction should be familiar from mathematics. Suppose that we want to prove that a formula $F(n)$ containing a variable n is true for all possible values of n in the non-negative integers. There are two steps to the proof:

1. Prove $F(0)$. This is called the *base case*.

2. Assume that $F(m)$ is true for all $m \le n$ and prove $F(n + 1)$. This is called the *inductive step* and the assumption $F(m)$ for $m \le n$ is called the *inductive hypothesis*.

If these are proved, we can conclude that $F(n)$ is true for all values of n. This form of induction is called *generalized induction*, because we assume $F(m)$ for *all* numbers m less than $n + 1$ and not just for n.

Let us use generalized induction to prove the formula $F(n)$: if n is a positive even number, then it is the sum of two odd numbers:

1. We need to prove two base cases: $F(1)$ is trivially true because 1 is not even, and $F(2)$ is true because $2 = 1 + 1$.

2. Let us assume the inductive hypothesis $F(m)$ for $m \le n$ and prove $F(n + 1)$. The proof is divided into two cases:

    1. $n + 1$ is odd. In this case, the proof is trivial because the antecedent is false.

    2. $n + 1$ is even and $n + 1 > 2$. But $n - 1$ is also even and $n - 1 \ge 2$, so by the inductive hypothesis, there are two odd numbers $2i + 1$ and $2j + 1$ such that $n - 1 = (2i + 1) + (2j + 1)$. A bit of arithmetic gives us:

       $n + 1 = (n - 1) + 2 = (2i + 1) + (2j + 1) + 2 = (2i + 1) + (2(j + 1) + 1)$,

       showing that $n + 1$ is also the sum of two odd numbers.

Computational induction is used to prove programs; the induction is on the states of a computation rather than on natural numbers, but the concept is similar because the states are also arranged in an increasing sequence.

Frequently, we will want to prove invariants, which are formulas that are valid in every state of a computation. These will be proved using induction on the computation: the base case is to show that the formula is true in the initial step; the inductive step is to show that if the formula is assumed to be true in a state, it remains true in any successor state.

Proof Methods

Model Checking

f is valid if and only if $\neg f$ is unsatisfiable, so in order to show that f is valid, search for an interpretation that satisfies $\neg f$. If you find one, f is falsifiable, that is, it is not valid; if not, $\neg f$ is unsatisfiable, so f is valid. In mathematical logic, this dual approach is used in important techniques for proving validity: semantic tableaux, analytic tableaux and resolution.

The advantage of this method is that the search can be systematic and efficient, so that it can be mechanized by a computer program.

Software systems called *model checkers* combine a (concurrent) program together with the negation of a logical formula expressing a correctness property, and then search for a satisfying scenario. If one is found, the correctness property does not hold, and the scenario produced by the model checker is a falsifying interpretation. If an exhaustive search turns up no such scenario, the program is correct.

Deductive Proof

The more classical approach to showing that a formula is valid is to prove it. We assume that certain formulas, called *axioms*, are valid, and that certain *rules* can be used to infer new true formulas from existing ones. A *proof* is a sequence of formulas, each one of which is either an axiom, or a formula that can be inferred from previous formulas using a rule. A formula has been proved if it is the last formula in a proof sequence. For a proof system to be useful it must be *sound* and preferably *complete*: a proof system is sound if every formula it proves is valid and it is complete if every valid formula can be proved. There is an axiom system for propositional LTL that is sound and complete (see [9, Section 12.2]).

Material Implication

Many correctness properties can be expressed as implications $p \rightarrow q$. An examination of the table on page 322 shows that the only way that $p \rightarrow q$ can be false is if $v(p) = T$ and $v(q) = F$. This is somewhat counterintuitive, because it means, for example, that the formula

$(1 + 1 = 3) \rightarrow (1 + 1 = 2)$

is true. This definition, called *material implication*, is the one that is used in mathematics.

Suppose that $p \rightarrow q$ is true in a state and that we want to prove that it remains true in any successor state. We need only concern ourselves with: (a) states in which $v(p) = v(q) = T$, but “suddenly” in the next state $v(q)$ becomes false while $v(p)$ remains true, and (b) states in which $v(p) = v(q) = F$, but “suddenly” in the next state $v(p)$ becomes true while $v(q)$ remains false. If we can show that neither of these situations can occur, the inductive step has been proved.[1]

If the correctness claim is an implication of the form “the computation is at statement `p3` implies that *flag* = 1,” the entire verification is reduced to two claims:

(a) any statement of process `p` leading to the control pointer pointing to `p3` cannot be taken if *flag* = 1 is false or if the statement leads to a state in which *flag* = 1 becomes or remains true, and (b) if the control pointer of process `p` is at statement `p3`, then no other process can falsify *flag* = 1.

Psychologists have investigated the difficulties of reasoning with material implication using *Wason selection tasks*. Suppose that the following four cards are placed before you:

$\boxed{p3}$ $\boxed{p5}$ $\boxed{flag = 1}$ $\boxed{flag = 0}$

On one side of each card is written the current location of the computation (denoted *p*3 if the control pointer is at statement `p3` and similarly for *p*5), and on the other side the value of *flag*. What is the smallest set of cards that you need to turn over to check that they all describe true instances of the claim "if the computation is at statement `p3`, then *flag* = 1"? The answer is that you need to turn over the first card (to ensure that the true antecedent is associated with a true consequent on the other side of the card) and the fourth card (to ensure that the false consequent is associated with a false antecedent on the other side of the card). If the antecedent is false (the second card), it doesn't matter what the consequent is, and if the consequent is true (the third card), it doesn't matter what the antecedent is.

Correctness of Sequential Programs

Let us assume that a program `P` contains only one read statement as its first statement, and one write statement as its last statement:

```
read(x1,x2,. . . ,xM), . . . , write(y1,y2,. . . ,yN).
```

There are two definitions of correctness of `P`. Informally, they are:

Partial correctness. *If* `P` halts, the answer is "correct."

Total correctness. `P` *does* halt and with the answer is "correct."

These concepts can be formalized mathematically. We use the notation $\bar{x}$ and $\bar{y}$ to denote sequences of (mathematical) variables that contain the values of the (program) variables `(x1,x2,. . . ,xM)` and `(y1,y2,. . . ,yN)`, and similarly for sequences of values ā and $\bar{b}$. Let $A(\bar{x})$ be a logical formula with free variables $\bar{x}$ called the *precondition*, and $B(\bar{x}, \bar{y})$ be a logical formula with free variables $\bar{x}$, $\bar{y}$ called the *postcondition*.

Partial correctness

Program `P` is *partially correct with respect to* $A(\bar{x})$ *and* $B(\bar{x}, \bar{y})$ if and only if for all values of ā:

if A (ā) is true,

and ā are the input values read into `(x1,x2,. . . ,xM)`,

and `P` terminates writing out the values $\bar{b}$ of `(y1,y2,. . . ,yN)`,

then $B(\bar{a}, \bar{b})$ is true.

Total correctness

Program `P` is *totally correct with respect to* $A(\bar{x})$ *and* $B(\bar{x}, \bar{y})$ if and only if for all values of *ā*:

if $A(\bar{x})$ is true,

and ā are the input values read into `(x1,x2,. . . ,xM)`,

then `P` terminates writing out the values $\bar{b}$ of `(y1,y2,. . . ,yN)`,

and $B(\bar{a}, \bar{b})$ is true.

Consider the following program, and take a few minutes to try to understand what the program does before reading on:

Table B.1. Verification example

```
integer x1, integer x2

integer y1 ← 0,
integer y2 ← 0,
integer y3
```

```
p1: read(x1,x2)
p2: y3 ← x1
p3: while y3 ≠ 0
p4:   if y2+1 = x2
p5:      y1 ← y1 + 1
p6:      y2 ← 0
p7:   else
p8:      y2 ← y2 + 1
p9:   y3 ← y3 - 1
p10: write(y1,y2)
```

The program is partially correct with respect to $A\,(x1\,,\,x2)$ defined as *true* and $B(x1\,,\,x2\,,\,y1\,,\,y2)$ defined as

$(x1 = x2 \cdot y1 + y2) \wedge (0 \leq y2 < x2)$.

The program computes the result and the remainder of dividing the integer $x1$ by the integer $x2$. It does this by counting down from `x1` in the variable `y3`, adding one to the remainder `y2` on each execution of the loop body. When the remainder would become equal to the divisor `x2`, it is zeroed out and the result `y1` is incremented.

It is true that if zero or a negative number is read into $x2$, then the program will not terminate, but termination is not required for partial correctness; the only requirement is that *if* the program terminates, *then* the postcondition holds. On the other hand, the program *is* totally correct with respect to the precondition $A\,(x1\,,\,x2)$ defined as $(x1 \geq 0) \wedge (x2 > 0)$ and the same postcondition.

This emphasizes that it is always wrong to speak of a *program* as being correct. A program can only be correct with respect to its specification, here its pre- and postconditions.

The problem with the concepts of partial and total correctness is that they are appropriate only for programs that terminate. While there are concurrent programs that terminate (such as scientific simulations that use parallelism to compute the behavior of complex systems), most concurrent programs (such as operating systems and real-time controllers) are designed to be non-terminating. The term *reactive* is often used to emphasize that the program is designed to repeatedly react to external stimuli (such clicking a button on a mouse) or to repeatedly sample an external sensor. If one of these systems ever does terminate, we say that the computer has “crashed” and consider it a bug! A new definition of correctness is needed for concurrent programs.

Verification in SPARK

Most of the proof of the algorithm for integer division can be mechanically carried out by SPARK [5]. Listing B.1 shows the algorithm written in the SPARK language, which is a subset of Ada augmented with annotations. (Some of the variable names have been changed for clarity.) The annotation **derives**, together with the mode declarations **in** and **out** of the parameters, is used for checking information flow. **derives** states that the values of `Q` and `R` that are returned from the procedure are obtained from the values of `X1` and `X2` that enter the procedure.

The proof annotations are the precondition **pre** that states that the dividend `X1` is non-negative and that the divisor `X2` is positive, and the postcondition **post** that states the relation among the variables required for the correctness of integer division. The **assert** annotation specifies what must be true upon entering the **while** loop, namely that the division relation hold as long as the value of `N` is added to `Q*X2+R`. Clearly, when leaving the loop, the value of `N` will be zero and the relation will hold among the variables that are the parameters of the procedure.

Example B.1. A SPARK program for integer division

```
--# main_program;
procedure Divide(X1, X2: in Integer; Q, R: out Integer)
--# derives Q, R from X1, X2;
--# pre (X1 >=0) and (X2 >0);
--# post (X1=Q*X2+R) and (X2 >R) and (R>=0);
is
   N: Integer;
begin
  Q := 0; R := 0; N := X1;
 while N /= 0
 --# assert (X1=Q*X2+R+N) and (X2 >R) and (R>=0);
 loop
    if R+1 = X2 then
       Q := Q + 1; R := 0;
    else
       R := R + 1;
    end if;
    N := N - 1;
  end loop;
end Divide;
```

When SPARK is run, it verifies the information flow and then generates four *verification conditions* that must be proved in order to prove the partial correctness of the procedure. A verification condition is a formula that describes the execution of a program along a path in the program. The four conditions are:

1. From the precondition at the beginning of the procedure until the assertion:

   $(X1 \geq 0) \wedge (X2 > 0) \rightarrow (X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0).$

2. From the assertion until the postcondition at the end of the procedure (using the knowledge that the **while** loop terminates, so $N = 0$):

   $(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (N = 0) \rightarrow (X1 = Q \cdot X2 + R) \wedge (X2 > R) \wedge (R \geq 0).$

3. From the assertion, around the loop via the **then** branch, and back to the assertion (using the knowledge that the condition of the **if** statement is true):

$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (R + 1 = X2) \rightarrow (X1 = Q' \cdot X2 + R' + N') \wedge (X2 > R') \wedge (R' \geq 0)$.

The primed variables indicate denote the new values of the variables after executing the assignments in the loop body.

4. From the assertion, around the loop via the **else** branch, and back to the assertion (using the knowledge that the condition of the **if** statement is false):

   $(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (R + 1 \neq X2) \rightarrow (X1 = Q' \cdot X2 + R' + N') \wedge (X2 > R') \wedge (R' \geq 0)$.

The *simplifier* tool of SPARK can reduce the first three formulas to true. It does this by substituting the expressions for the variables in the assignment statements, and then simplifying the formulas using elementary knowledge of arithmetic. For example, substituting the initial values into the first formula gives:

$(X1 \geq 0) \wedge (X2 > 0) \rightarrow (X1 = 0 \cdot X2 + X1) \wedge (X2 > 0) \wedge (0 \geq 0)$.

It does not take much knowledge of arithmetic to conclude that the formula is true.

The condition that goes through the **else** branch cannot be proved by the simplifier. Substituting Q for Q', R +1 for R' and N −1 for N, the formula that must be proved is:

$(X1 = Q \cdot X2 + R + N) \wedge (X2 > R) \wedge (R \geq 0) \wedge (R + 1 \neq X2) \rightarrow (X1 = Q \cdot X2 + R + 1 + N - 1) \wedge (X2 > R + 1) \wedge (R + 1 \geq 0)$.

The first subformulas on either side of the implication are equivalent, so this reduces to:

$(X2 > R) \wedge (R \geq 0) \wedge (R + 1 \neq X2) \rightarrow (X2 > R + 1) \wedge (R + 1 \geq 0)$.

But this is easy for us to prove. By the properties of arithmetic, if $X2 > R$ then $X2 = R + k$ for $k \geq 1$. From the antecedent we know that $R + 1 \neq X2$, so $k > 1$, proving that $X2 > R + 1$. Finally, from $R \geq 0$, it is trivial that $R + 1 \geq 0$.

Another component of SPARK, the *proof checker*, can be used to partially automate the proof of verification conditions such as this one that the simplifier cannot perform.

[1] It is also possible that $v(p) = F$ and $v(q) = T$, but "suddenly" $v(p)$ becomes true *and* v (q) becomes false.