Chapter 6
Static Analysis

J. C. Huang
Department of Computer Science
University of Houston

Static Analysis

Static analysis is a process in which we attempt to find faults in a program by examining the source code systematically without test-executing it.

What can we do with it?

It can be used to

- find symptom of possible programming faults, and
- explicates the computation performed by the program.

Anomalies

Sometimes part of a program may be abnormally formed. We call that an *anomaly* instead of a fault because it may or may not cause the program to fail. Nevertheless, it is a symptom of possible programming error.

Types of anomalies

Possible anomalies include

- Structural flaws in a program module,
- Flaws in module interface,
- Errors in event sequencing.

Types of structural flaw detectable

- Extraneous entities
- Improper loop constructs.
- Improper loop nesting.
- Unreferenced labels.
- Unreachable statements.
- Transfer of control into a loop.

Note that it is difficult, if not impossible, to create a construct of any of the last four types unless the use of GOTO statement is allowed.

Example

For example, in C++, a beginner may write

```
char* p;
strcpy( p, "Houston" );
```

which is syntactically correct but semantically wrong. It should be written like

```
char* p;
p = buffer;
strcpy( p, "Houston" );
```

Types of interface flaw detectable

- Inconsistencies in the declaration of data structures.
- Improper linkage among modules (e.g., discrepancy in the number and types of parameters).
- Flaws in other inter-program communication mechanism such as common blocks.

Detectable event-sequencing errors

- Priority interrupt handling conflict
- Error in file handling
- Data-flow anomaly
- Anomaly in concurrent programs

Data-flow Anomaly

When a program is being executed, it may act on a variable (datum) in three different ways, namely, *define, reference,* and *undefine*.

Data-flow Anomaly (continued)

The dataflow with respect to a variable is said to be anomalous if the variable is either undefined and referenced, defined and then undefined, or defined and defined again.

Data-flow Anomaly (continued)

The presence of a data-flow anomaly in the program is only a symptom of possible programming error. The program may or may not be in error.

Data-Flow Anomaly Detection in Concurrent Programs

Possible events that may occur:

- define
- reference
- undefine
- schedule
- unschedule (not scheduled)
- wait

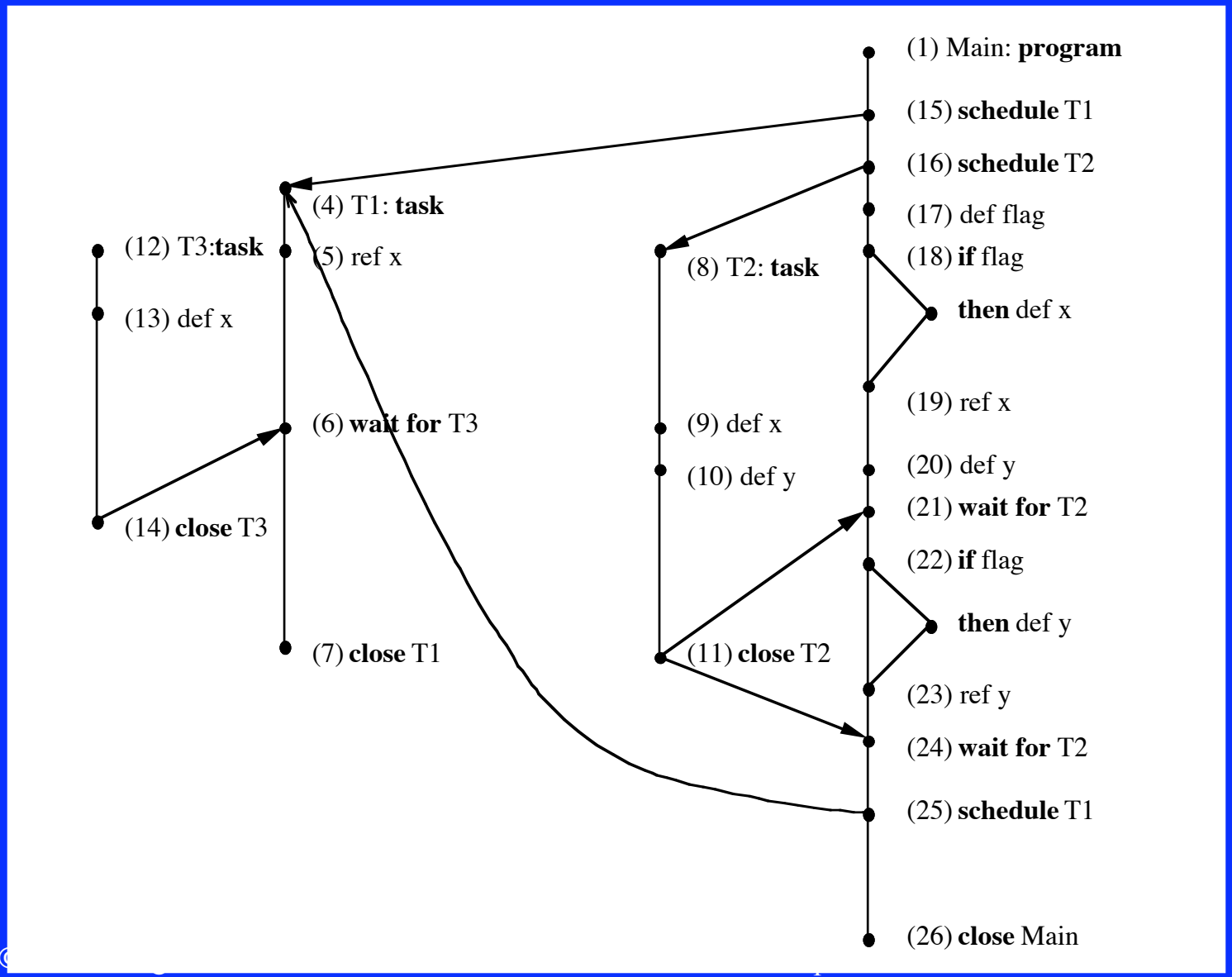
Possible types of anomaly:

- a dead definition of a variable
- waiting for a process not scheduled
- scheduling a process in parallel with itself
- waiting for a process guaranteed to have terminated previously
- referencing an uninitialized variable
- referencing a variable which is being defined by a parallel process
- referencing a variable whose value is indeterminate

Example program

(See the slide in Chapter 6a.)

The process-augmented flow-graph



Possible anomalies

- An uninitialized variable (x) may be referenced at line 5, as task T1 may execute to completion before T2 begins.

- The definitions of y as found in task T2 (line 10) and the main program (line 20) may be useless since y may be redefined at line 22 before y is ever referenced.

Possible anomalies (continued)

- y is defined by two processes that may be executed concurrently, and thus the reference at line 23 may be to an indeterminate value.

- Variable x is assigned a value by task T2 (line 9) while simultaneously being referenced by the main program at line 19.

Possible anomalies (continued)

- There is a possibility that task T1 will be scheduled in parallel with itself at line 25 since there is no guarantee that T1 terminates after its initial scheduling.
- The wait at line 24 is unnecessary, as T2 was guaranteed to have terminated at line 21, and it has not been scheduled subsequently.
- The wait at line 6 will never be satisfied as T3 was never scheduled.

Symbolic Evaluation (Execution)

The basic idea is to execute the program with symbolic inputs and produce symbolic formulae as output.

Example:

```
read(x, y);
z := x + y;
x := x - y;
z := x * z;
write(z);
```

Ordinary execution with x = 2 and y = 4.

	value trace		
	x	y	z
read(x, y);	2	4	undefined
z := x + y;	2	4	6
x := x - y	-2	4	6
z := x * z;	-2	4	-12
write(z);	-2	4	-12

Symbolic execution with x = a and y = b

	value trace		
	x	y	z
read(x,y);	a	b	undefined
z:=x+y;	a	b	a+b
x:=x-y	a-b	b	a+b
z:=x*z;	a-b	b	a*a-b*b
write(z);	a-b	b	a*a-b*b

Path condition

If the program consists of more than one execution path, it is necessary to choose a path through the program to be followed, and the result of execution should include path condition, or *pc* for short, which is a Boolean expression over the symbolic values.

Comment

Generally speaking, the usefulness of symbolic execution is limited to numerical programs designed to compute a function describable by a closed formula.

Example

For example, the technique is useful to the following Fortran program designed to solve quadratic equations by using the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Program 6.1

(See the text. It is too large to be included in a slide)

A trace subprogram

```
READ (5, 11) A, B, C
/\.NOT. (A .EQ. 0.0 .AND. B .EQ. 0.0 .AND. C .EQ.
   0.0)
/\ (A .NE. 0.0 .OR. B .NE. 0.0)
/\ (A .NE. 0.0)
/\ (C .NE. 0.0)
RREAL = -B/(2.0*A)
DISC = B**2 - 4.0*A*C
RIMAG = SQRT(ABS(DISC))/(2.0*A)
/\.NOT. (DISC .LT. 0.0)
R1 = RREAL + RIMAG
R2 = RREAL - RIMAG
WRITE (6, 31) R1, R2
```

We can rewrite it into the canonical form first,

```
READ (5, 11) A, B, C
/\ (A .NE. 0.0 .OR. B .NE. 0.0 .OR. C .NE. 0.0)
/\ (A .NE. 0.0 .OR. B .NE. 0.0)
/\ (A .NE. 0.0)
/\ (C .NE. 0.0)
/\ (B**2 - 4.0*A*C .GE. 0.0)
RREAL = -B/(2.0*A)
DISC = B**2 - 4.0*A*C
RIMAG = SQRT(ABS(DISC))/(2.0*A)
R1 = RREAL + RIMAG
R2 = RREAL - RIMAG
WRITE (6, 31) R1, R2
```

then the path condition can be simplified to

```
READ (5, 11) A, B, C
/\ (A .NE. 0.0 .OR. B .NE. 0.0)
/\ (A .NE. 0.0)
/\ (C .NE. 0.0)
/\ (B**2 - 4.0*A*C .GE. 0.0)
RREAL = -B/(2.0*A)
DISC = B**2 - 4.0*A*C
RIMAG = SQRT(ABS(DISC))/(2.0*A)
R1 = RREAL + RIMAG
R2 = RREAL - RIMAG
WRITE (6, 31) R1, R2
```

and further simplified to

```
READ (5, 11) A, B, C
/\ (A .NE. 0.0)
/\ (C .NE. 0.0)
/\ (B**2 - 4.0*A*C .GE. 0.0)
RREAL = -B/(2.0*A)
DISC = B**2 - 4.0*A*C
RIMAG = SQRT(ABS(DISC))/(2.0*A)
R1 = RREAL + RIMAG
R2 = RREAL - RIMAG
WRITE (6, 31) R1, R2
```

and then symbolically execute it to yield

```
R1=-B/(2.0*A)
       +SQRT(ABS(B**2-4.0*A*C))/(2.0*A)
R2=-B/(2.0*A)
       -SQRT(ABS(B**2-4.0*A*C))/(2.0*A)
pc:A.NE.0.0.AND.C.NE.0.0
       .AND.B**2-4.0*A*C.GE.0.0
```

This demonstrate the usefulness of a symbolic execution because it clearly indicates what the program will do for the cases where the path condition `pc` is satisfied.

Another possible application

Symbolic execution can also be used to guide simplification of source code. For example, consider the following segment of code:

```
r=a%b;
a=b;
b=r;
r=a%b;
a=b;
b=r;
```

Symbolic execution with a=A and b=B

after execution of of statement	*the symbolic values becomes*
	a=A
	b=B
r=a%b	r=A%B
a=b	a=B
b=r	b=A%B
r=a%B	**r=B%(A%B)**
a=b	**a=A%B**
b=r	**b=B%(A%B)**

Suggested simplification

The result of symbolic execution strongly suggests that the code can be simplified to:

r=B%(A%B)	⇔	a=a%b;
a=A%B		r=b%a;
b=B%(A%B)		b=r;

Comment

In general, the result of a symbolic execution is a set of strings (symbols) representing the values of the program variables. These strings often grow uncontrollably during the execution. Thus the results may not be of much use unless the symbolic execution system is capable of simplifying these strings automatically.

Such a simplifier basically requires the power of a mechanical theorem prover. Therefore, a symbolic execution system is a computationally intensive software system, and is relatively difficult to build.

Program slicing

Program slicing is a method for abstracting from a program. Given a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior.

The reduced program, called a *slice*, is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior

Example program P

```
1   begin
2      read(x, y);
3      total := 0.0;
4      sum := 0.0;
5      if x <= 1
6        then sum := y
7        else begin
8                read(z);
9                total := x*y
10             end;
11     write(total, sum)
12  end.
```

Example slice S_1

Slice on the value of z at statement 12:

```
 1  begin
 2     read(x, y);
 5     if x <= 1
 6        then
 7        else begin
 8                read(z);
10             end;
12  end.
```

Example slice S_2

Slice on the value of total at statement 12:

```
 1  begin
 2     read(x, y);
 3     total := 0.0;
 5     if x <= 1
 6       then
 7       else begin
 9              total := x*y
10            end;
12  end.
```

Example slice S_3

Slice on the value of x at statement 9:

```
 1   begin
 2      read(x, y);
12   end.
```

DEF and REF sets

Definition 6.2: Let P be a program, and suppose that the statements are numbered consecutively. Then for each statement n in P we can define two sets: REF(n) is the set of all variables referenced at n, and DEF(n) is the set of all variables defined at n.

Slicing criterion

Definition 6.3: A slicing criterion of program P is an ordered pair (i, V), where i is a statement number in P and V is a subset of the variable in P.

Example slicing criteria

C_1: (12, {z}),

C_2: (12, {total}), and

C_3: (9, {x}).

Value trace

Definition 6.4: A value trace of a program P is a finite list of ordered pairs

$$(n_1, s_1)(n_2, s_2) \ldots (n_k, s_k)$$

where each n_i denotes a statement in P, and each s_i is a vector of values of all variables in P immediately before the execution of n_i.

Example

Consider the program listed in the next slide in which the vector of variables used is

<x, y, z, sum, total>

Example program

```
1  begin
2    read(x, y);
3    total := 0.0;
4    sum := 0.0;
5    if x <= 1
6      then sum := y
7      else begin
8             read(z);
9             total := x*y
10            end;
11   write(total, sum)
12 end.
```

A value trace

T_1: (1, <?, ?, ?, ?, ?>)

(2, <?, ?, ?, ?, ?>)

(3, <X, Y, ?, ?, ?>)

(4, <X, Y, ?, ?, 0.0>)

(5, <X, Y, ?, 0.0, 0.0>)

(6, <X, Y, ?, 0.0, 0.0>)

(11, <X, Y, ?, Y, 0.0>)

(12, <X, Y, ?, Y, 0.0>)

Another possible value trace

T_2: (1, <?, ?, ?, ?, ?>)

(2, <?, ?, ?, ?, ?>)

(3, <X, Y, ?, ?, ?>)

(4, <X, Y, ?, ?, 0.0>)

(7, <X, Y, ?, 0.0, 0.0>)

(8, <X, Y, ?, 0.0, 0.0>)

(9, <X, Y, Z, 0.0, 0.0>)

(10, <X, Y, Z, 0.0, X*Y>)

(11, <X, Y, Z, 0.0, X*Y>)

(12, <X, Y, Z, 0.0, X*Y>)

Remark

In the above we use a question mark (?) to denote an undefined value, and a variable name in upper case to denote the value of that variable obtained through an input statement in the program.

Projection

Definition 6.5: Given a slicing criterion C = (i, V) and a value trace T, we can define a projection function Proj(C, T) that deletes from a value trace all ordered pairs except those with i as the left component, and from the right components of the remaining pairs all values except those of variables in V.

Example projection

$Proj(C_1, T_1)$ = Proj((12, {z}), T_1)

= Proj((12, {z}), (1, <?, ?, ?, ?, ?>)
(2, <?, ?, ?, ?, ?>)
(3, <X, Y, ?, ?, ?>)
(4, <X, Y, ?, ?, 0.0>)
(5, <X, Y, ?, 0.0, 0.0>)
(6, <X, Y, ?, 0.0, 0.0>)
(11, <X, Y, ?, Y, 0.0>)
(12, <X, Y, ?, Y, 0.0>)

= (12, <?>)

Another example projection

$Proj(C_2, T_1)$ = $Proj((12, \{total\}), T_1)$

= $Proj((12, \{total\}), (1, <?, ?, ?, ?, ?>)$
$(2, <?, ?, ?, ?, ?>)$
$(3, <X, Y, ?, ?, ?>)$
$(4, <X, Y, ?, ?, 0.0>)$
$(5, <X, Y, ?, 0.0, 0.0>)$
$(6, <X, Y, ?, 0.0, 0.0>)$
$(11, <X, Y, ?, Y, 0.0>)$
$(12, <X, Y, ?, Y, 0.0>)$

= $(12, <0.0>)$

Yet another example projection

$Proj(C_3, T_2)$ = $Proj((9, \{x\}), T_2)$

= $Proj((9, \{x\}), (1, <?, ?, ?, ?, ?>)$
$(2, <?, ?, ?, ?, ?>)$
$(3, <X, Y, ?, ?, ?>)$
$(4, <X, Y, ?, ?, 0.0>)$
$(7, <X, Y, ?, 0.0, 0.0>)$
$(8, <X, Y, ?, 0.0, 0.0>)$
$(9, <X, Y, Z, 0.0, 0.0>)$
$(10, <X, Y, Z, 0.0, X*Y>)$
$(11, <X, Y, Z, 0.0, X*Y>)$
$(12, <X, Y, Z, 0.0, X*Y>)$

= $(9, <X>)$

Formal definition of a slice

Definition 6.6: A slice S of a program P on a slicing criterion C = (i, V) is any executable program satisfying the following two properties:

(a) S can be obtained from P by deleting zero or more statement from P.

(b) Whenever P halts on an input I with value trace T, S also halts on input I with value trace T', and Proj(C, T) = Proj(C', T'), where C' = (i', V), and i' = i if statement i is in the slice, or i' is the nearest successor to i otherwise.

Example

Again, consider P, the example program listed in the next slide, and the slicing criterion C1 = (12, {z}). According to the above definition, S1 is a slice because if we execute P with any input x = X such that X £ 1, it will produce the value trace T1, and as given previously, Proj(C1, T1) = (12, <?>).

Example program P

```
1  begin
2    read(x, y);
3    total := 0.0;
4    sum := 0.0;
5    if x <= 1
6      then sum := y
7      else begin
8             read(z);
9             total := x*y
10            end;
11   write(total, sum)
12 end.
```

Example (continued)

Now if we execute S_1 with the same input, it should yield the following value trace:

T'_1: (1, <?, ?, ?, ?, ?>)

(2, <?, ?, ?, ?, ?>)

(5, <X, Y, ?, ?, ?>)

(6, <X, Y, ?, ?, ?>)

(12, <X, Y, ? , ?>)

Example (continued)

Since statement 12 exists in P as well as S_1, $C_1 = C'_1$, and

$$
\begin{aligned}
Proj(C'_1, T'_1) &= ((12, \{z\}),\ T'_1) \\
&= (1, \langle ?, ?, ?, ?, ? \rangle) \\
&\quad\ (2, \langle ?, ?, ?, ?, ? \rangle) \\
&\quad\ (5, \langle X, Y, ?, ?, ? \rangle) \\
&\quad\ (6, \langle X, Y, ?, ?, ? \rangle) \\
&\quad\ (12, \langle X, Y, ?, ?, ? \rangle) \\
&= (12, \langle ? \rangle) \\
&= Proj(C_1, T_1)
\end{aligned}
$$

Example (continued)

Hence S_1 is a slice of P.

As yet another example in which C C', consider C = (11, {z}). Since statement 11 is not in S_1, C' will have to be set to (12, {z}) instead because statement 12 is the nearest successor of 11.

Comment

There can be many different slices for a given program and slicing criterion. There is always at least one slice for a given slicing criterion -- the program itself.

Comment

The above definition of a slice is not constructive in that it does not say how to find one. The smaller the slice the better. However, finding minimal slices is equivalent to solving the halting problem -- it is impossible.

Code Inspection

Code inspection (walk-through) is a process designed to assure high quality of the software produced. It should be carried out after the first clean compilation of the code to be inspected, and before any formal testing is done on that code.

Objectives

(a) to find logic errors,

(b) to verify the technical accuracy and completeness of the code,

(c) to verify that the programming language definition used conforms to that of the compiler to be used by the customer,

Objectives (continued)

(d) to ensure that no conflicting assumptions or design decisions have been made in different parts of the code, and

(e) to ensure that good coding practices and standards are used, and the code is easily understandable.

The team should include

(a) the designer who will answer any question,

(b) the moderator who ensures that any discussion is topical and productive,

(c) the paraphraser who steps through the code and paraphrase it in English, and

(d) the librarian or recorder.

Material needed

(a) program listings and design documents,

(b) a list of assumptions and decisions made in coding, and

(c) a participant-prepared list of problems and minor errors.

Comment

The purpose of a code inspection should not be to evaluate the competence of the author of the code, or to unnecessarily criticize coding style. The style of the code should not be discussed unless it prevents the code from meeting the objectives of the code inspection.

Products

(a) a summary report which briefly describes the problems found during the inspection,

(b) a form for listing each problem found so that its disposition or resolution can be recorded, and

(c) a list of updates made to the specifications and changes made to the code.

Reinspect when

(a) a nontrivial change to the code is required, or

(b) the number of problems found exceeds one for every 25 non-commentary lines of the code.

Reschedule when

(a) any mandatory participant can not be in attendance,

(b) the material needed for inspection is not made available to the participants in time for preparation,

(c) there is a strong evidence to indicate that the participants are not properly prepared,

(d) the moderator can not function effectively for some reason, or

(e) material given to the participants is found to be not up-to-date.

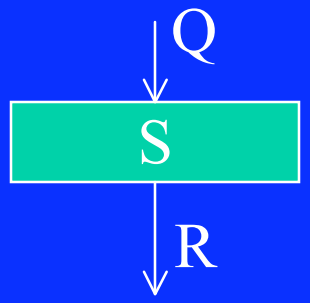
Comment

The process described above is to be carried out manually. Some part of which, however, can be done more readily if proper tools are available.

For example, in preparation for a code inspection, if the programmer find it difficult to understand certain parts of the source code, software tools can be used to facilitate understanding. Such tools can be built based on the program analysis method described in Sec. 1.6, and the technique of program slicing outlined in the next section.

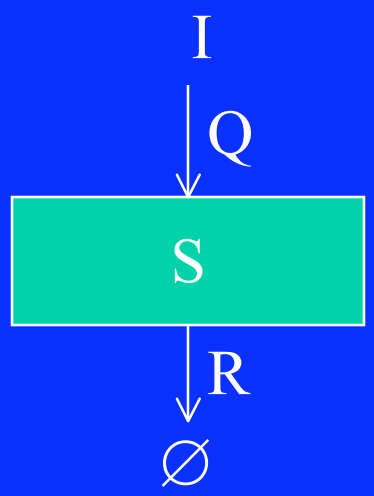
Proving Programs Correct

A common task in program verification is to show that, for a given program S, if a certain *precondition* Q is true before the execution of S then a certain *postcondition* R is true after the execution, provided that S terminates. This proposition is commonly denoted by Q{S}R for short.



Proving Programs Correct (continued)

If we succeeded in showing that Q{S}R is a theorem (i.e., always true), then to show that S is *partially correct*, with respect to some input predicate I and output predicate Ø, is to show that $I \supset Q$ and $R \supset Ø$.



Two alternative approaches

Verification of correctness can be carried out in two ways:

Given S, I, and Ø we may first let R ≡ Ø and show that Q{S}Ø for some predicate Q, and then show that I ⊃ Q.

Alternatively, we may let Q ≡ I and show that I{S}R for some predicate R, and then show that R ⊃ Ø.

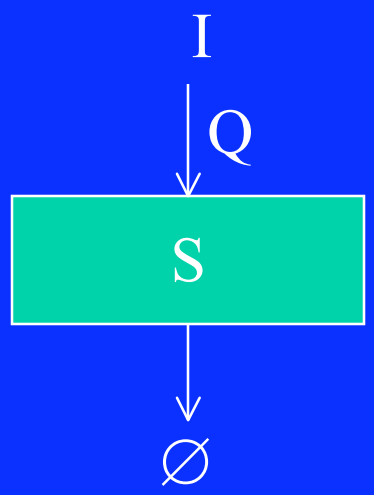
Bottom-up approach

In the first approach the basic problem is to find as weak as possible a condition Q such that $Q\{S\}\emptyset$ and $I \supset Q$.

A possible solution is to use the method of predicate transformation to find the weakest precondition.

Top-down approach

In the second approach the problem is to find as strong as possible a condition R so that I{S}R and $R \supset \emptyset$. This problem is fundamental to the method of inductive assertions.



Assumption about the language used

We assume that programs are written in a language consisting of the following statements:

(1) *assignment statements*: x := e;

(2) *conditional statements*: **if** B **then** S **else** S';

(3) *repetitive statements*: **while** B **do** S;

and a program is constructed by concatenating such statements.

INTDIV: an example program

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

Example

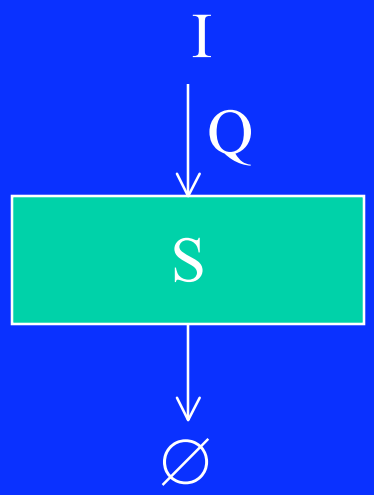
Suppose we wish to verify that program INTDIV is partially correct with respect to input predicate I: $x \geq 0 \wedge y > 0$ and output predicate ∅: $x = r + q \times y \wedge r < y \wedge r \geq 0$, i.e., to prove that

$$(x \geq 0 \wedge y > 0)\{\text{INTDIV}\}(x = r + q \times y \wedge r < y \wedge r \geq 0)$$

is a theorem.

The Predicate Transformation Method: Bottom-Up Approach

Recall that in the first approach, given S, I, and Ø, the basic problem is to find as weak as possible a condition Q such that Q{S}Ø, and then determine if $I \supset Q$.



Weakest precondition

Let S be a programming construct and R be a predicate or condition (henceforth we shall use the terms predicate, condition, and logical expression interchangeably). Then wp(S, R) denotes the *weakest precondition* for the initial state such that an execution of S will properly terminate, leaving it in a final state satisfying the condition R.

wp(S, R)

is called a *predicate transformer* and has the following properties:

1. For any S, wp(S, F) ≡ F
2. For any program S and any predicates S, Q, and R, if Q ⊃ R then wp(S, Q) ⊃ wp(S, R).
3. For any programming construct S and any predicates Q and R, (wp(S, Q) ∧ wp(S, R)) ≡ wp(S, Q ∧ R).
4. For any deterministic programming construct S and any predicates Q and R,
   (wp(S, Q) ∨ wp(S, R)) ≡ wp(S, Q ∨ R).

`skip` and `abort`

We shall define two special statements: `skip` and `abort`.

The statement `skip` is the same as the null statement in a high-level language, or the "no-op" instruction in an assembly language. Its meaning can be given as wp(`skip`, R) ≡ R for any predicate R.

The statement `abort`, when executed, will not lead to a final state. Its meaning is defined as wp(`abort`, R) ≡ F for any predicate R.

$$wp(x:=E, R) \equiv R_{E \rightarrow x}$$

R	*x := E*	$R_{E \rightarrow x}$	*simplified to*
x = 0	x := 0	0 = 0	T
a > 1	x := 10	a > 1	a > 1
x < 10	x := x + 1	x + 1 < 10	x < 9
x ≠ y	x := x - y	x - y ≠ y	x ≠ 2y

$$wp(S_1;S_2, R)$$

For a sequence of two programming constructs S_1 and S_2,

$$wp(S_1;S_2, R) \equiv wp(S_1, wp(S_2, R)).$$

wp(**if** B **then** S_1 **else** S_2, R)

wp(**if** B **then** S_1 **else** S_2, R) ≡

$B \wedge wp(S_1, R) \vee \neg B \wedge wp(S_2, R)$.

wp(**while** B **do** S, R)

wp(**while** B **do** S, R) $\equiv (\exists j)_{j \geq 0}(A_j(R))$,

where

$A_0(R) \equiv \neg B \wedge R$ and

$A_{j+1}(R) \equiv B \wedge wp(S, A_j(R))$ for all $j \geq 0$.

Example: proving INTDIV correct

We first compute

wp(**while** r ≥ y **do begin** r := r - y; q := q + 1 **end**, x = r + q × y ∧ r < y ∧ r ≥ 0)

where B ≡ r ≥ y

R ≡ x = r + q × y ∧ r < y ∧ r ≥ 0

S: r := r - y; q := q + 1;

Example (continued)

$$A_0(R) \equiv \neg B \wedge R$$
$$\equiv r < y \wedge x = r + q \times y \wedge r < y \wedge r \geq 0$$
$$\equiv x = r + q \times y \wedge r < y \wedge r \geq 0$$
$$A_1(R) \equiv B \wedge wp(S, A_0(R))$$
$$\equiv r \geq y \wedge wp(r := r - y; q := q + 1, x = r + q \times y \wedge r < y \wedge r \geq 0)$$
$$\equiv r \geq y \wedge x = r - y + (q + 1) \times y \wedge r - y < y \wedge r - y \geq 0$$
$$\equiv x = r + q \times y \wedge r < 2 \times y \wedge r \geq y$$

Example (continued)

$$A_2(R) \equiv B \wedge wp(S, A_1(R))$$
$$\equiv x = r + q \times y \wedge r < 3 \times y \wedge r \geq 2 \times y$$
$$A_3(R) \equiv B \wedge wp(S, A_2(R))$$
$$\equiv x = r + q \times y \wedge r < 4 \times y \wedge r \geq 3 \times y$$

Example (continued)

From these we may guess that

$A_j(R) \equiv B \wedge wp(S, A_{j-1}(R))$

$\equiv x = r + q \times y \wedge r < (j+1) \times y \wedge r \geq j \times y$

and we have to prove that our guess is correct by mathematical induction.

Example (continued)

Assume that $A_j(R)$ is as given above, then

$A_0(R) \equiv x = r + q \times y \wedge r < (0+1) \times y \wedge r \geq 0 \times y$

$\equiv x = r + q \times y \wedge r < y \wedge r \geq 0$

$A_{j+1}(R) \equiv B \wedge wp(S, A_j(R))$

$\equiv r \geq y \wedge wp(r := r - y; q := q + 1, x = r + q \times y \wedge r < (j+1) \times y \wedge r \geq j \times y)$

$\equiv r \geq y \wedge x = r - y + (q + 1) \times y \wedge r - y < (j+1) \times y \wedge r - y \geq j \times y$

$\equiv x = r+q \times y \wedge r < ((j+1)+1) \times y \wedge r \geq (j+1) \times y$

Example (continued)

These two instances of $A_j(R)$ show that if $A_j(R)$ is correct then $A_{j+1}(R)$ is also correct as given above.

Example (continued)

Hence

wp(**while** r ≥ y **do begin** r := r - y; q := q + 1 **end**,

$$x = r + q \times y \wedge r < y \wedge r \geq 0)$$

$$\equiv (\exists j)_{j \geq 0}(A_j(R))$$

$$\equiv (\exists j)_{j \geq 0}(x = r + q \times y \wedge r < (j+1) \times y \wedge r \geq j \times y)$$

Example (continued)

$$wp(q:=0;\ r:=x,\ (\exists j)_{j\geq 0}(x=r+q\times y \wedge r<(j+1)\times y \wedge r\geq j\times y))$$
$$\equiv (\exists j)_{j\geq 0}(x < (j+1) \times y \wedge x \geq j \times y)$$

which is implied by $x \geq 0 \wedge y > 0$, and hence the proof that the following is a theorem:

$$(x\geq 0 \wedge y>0)\{\text{INTDIV}\}(x=r+q\times y \wedge r<y \wedge r\geq 0).$$

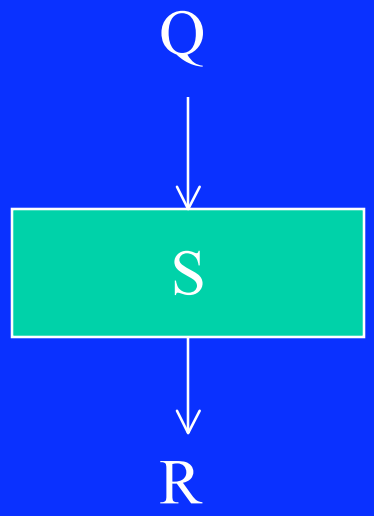
Partial correctness and strong verification

Recall that Q{S}R is a shorthand notation for the proposition: "if Q is true before the execution of S then R is true after the execution, provided that S terminates". Termination of the program has to be proved separately.

If Q ≡ wp(S, R), however, termination of the program is guaranteed. In that case, we can write Q[S]R instead, which is a shorthand notation for the proposition: "if Q is true before the execution of S then R is true after the execution of S, and the execution will terminate".

The Inductive Assertion Method: Top-Down Approach

In the top-down approach, given a program S and a predicates Q, the basic problem is to find as strong as possible a condition R such that Q{S}R.



Assignment statement

If S is an assignment statement of the form x := E, where x is a variable and E is an expression, we have

$$Q\{x := E\}(Q' \wedge x = E')_{x' \rightarrow E^{-1}}$$

where Q' and E' are obtained from Q and E, respectively, by replacing every occurrence of x with x', and then replace every occurrence of x' with E^{-1}, such that $x = E' \equiv x' = E^{-1}$.

Given Q and x := E, construct $(Q' \wedge x = E')_{x' \rightarrow E}$-1 as follows.

1. Write $Q \wedge x = E$.
2. Replace every occurrence of x in Q and E with x' to yield $Q' \wedge x = E'$.
3. If x' occurs in E' then construct $x' = E^{-1}$ from $x = E'$ such that $x = E' \equiv x' = E^{-1}$, else E^{-1} does not exist.
4. If E^{-1} exists then replace every occurrence of x' in $Q' \wedge x = E'$ with E^{-1}. Otherwise, replace every atomic predicate in $Q' \wedge x = E'$ having at least one occurrence of x' with T (the constant predicate TRUE).

Example

Q	x:=E	$(Q' \wedge x=E')_{x' \rightarrow E}$-1	simplified to
$x = 0$	$x := 10$	$T \wedge x = 10$	$x = 10$
$a > 1$	$x := 1$	$a > 1 \wedge x = 1$	$a > 1 \wedge x = 1$
$x < 10$	$x := x + 1$	$x - 1 < 10$	$x < 11$
$x \neq y$	$x := x - y$	$x + y \neq y$	$x \neq 0$

A notational convention

As explained earlier, it is convenient to use |-P to denote the fact that P is a theorem (i.e., always true).

A verification rule may be stated in the form "if |-X then |-Y," which says that if proposition X has been proved as a theorem then Y also is thereby proved as a theorem.

An important fact

Note that $Q[S]R \supset Q\{S\}R$, but not the other way around.

Can you prove that $Q[S]R \not\subset Q\{S\}R$?

Rule 1:

For an assignment statement of the form x := E

$$\vdash Q\{x := E\}(Q' \wedge x = E')_{x' \rightarrow E}^{-1}$$

Rule 2:

For a conditional statement of the form

if B **then** S_1 **else** S_2

If $\vdash Q \wedge B\{S_1\}R_1$ and $\vdash Q \wedge \neg B\{S_2\}R_2$

then $\vdash Q\{$**if** B **then** S_1 **else** $S_2\}R_1 \vee R_2$.

Rule 3

For a loop construct of the form **while** B **do** S

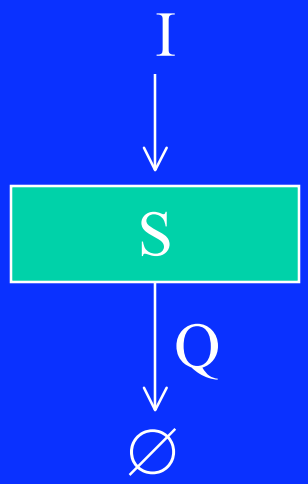
If $\vdash Q \supset R$ and $\vdash (R \wedge B)\{S\}R$

then $\vdash Q\{$**while** B **do** S$\}(\neg B \wedge R)$.

This rule is commonly known as the *invariant-relation theorem*, and any predicate R satisfying the premise is called a *loop invariant* of the loop construct **while** B **do** S.

The top-down strategy

Thus the partial correctness of program S with respect to input condition I and output condition Ø can be proved by showing that $I\{S\}Q$ and $Q \supset Ø$.



The proof can be constructed in smaller steps

if S is a long sequence of statements. Specifically, if S is $S_1;S_2; \ldots ;S_n$ then $I\{S_1;S_2; \ldots ;S_n\}\emptyset$ can be proved by showing that $I\{S_1\}P_1$, $P_1\{S_2\}P_2$, ... , and $P_{n-1}\{S_n\}\emptyset$ for some predicates P_1, P_2, ... , and P_{n-1}. P_is are called inductive assertions, and this method of proving program correctness is called the inductive assertion method.

Proof requires guesswork

Required inductive assertions for constructing a proof often have to be found by guesswork, based on one's understanding of the program in question, especially if a loop construct is involved. No algorithm for this purpose exists, although some heuristics have been developed to aid the search.

Proving the correctness of INTDIV

I: $x \geq 0 \wedge y > 0$

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

∅: $x = r + q \times y \wedge r \geq 0 \wedge r < y$

Proving INTDIV (continued)

I: $x \geq 0 \wedge y > 0$

begin

q := 0;

$x \geq 0 \wedge y > 0 \wedge q = 0$ (*by Rule 1*)

r := x;

while r ≥ y **do**

begin r := r - y; q := q + 1 **end**

end.

∅: $x = r + q \times y \wedge r \geq 0 \wedge r < y$

Proving INTDIV (continued)

I: $x \geq 0 \wedge y > 0$

begin

q := 0;

$x \geq 0 \wedge y > 0 \wedge q = 0$

r := x;

$x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x$ (*by Rule 1*)

while r ≥ y **do**

begin r := r - y; q := q + 1 **end**

end.

∅: $x = r + q \times y \wedge r \geq 0 \wedge r < y$

Proving INTDIV (continued)

I: $x \geq 0 \wedge y > 0$

begin

q := 0;

r := x;

$x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x$

while r ≥ y **do**

begin r := r - y; q := q + 1 **end**

$x = r + q \times y \wedge r \geq 0 \wedge r < y$

end.

∅: $x = r + q \times y \wedge r \geq 0 \wedge r < y$

Proving INTDIV (continued)

Obviously

$$x = r + q \times y \wedge r \geq 0 \wedge r < y$$

implies (in fact it is identical to)

$$\varnothing$$

and hence the proof.

Comment on the above method

There are many variations to the inductive-assertion method. The above version is designed, as an integral part of this section, to show that a correctness proof can be constructed in a top-down manner. As such, we assume that a program is composed of a concatenation of statements, and an inductive assertion is to be inserted between such statements only.

Comment (continued)

The problem is that most programs contain nested loops and compound statements, which may render applications of Rules 2 and 3 hopelessly complicated.

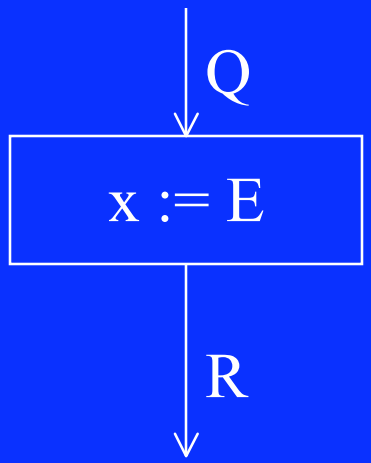
The complication induced by nested loops and compound statements can be eliminated by representing the program as a flowchart.

A variation of the inductive assertion method

In this method, the program is represented as a flowchart, and appropriate assertions are placed on various points in the control flow. These assertions "cut" the flowchart into a set of paths.

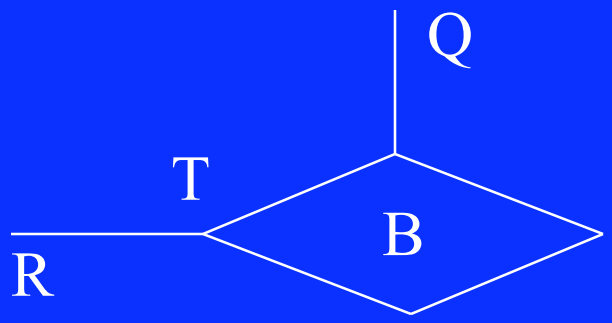
A *path* between assertions Q and R is formed by a single sequence of statements that will be executed if the control flow traverses from Q to R in an execution, and contains no other assertions. It is possible that Q and R are the same.

Basic path 1



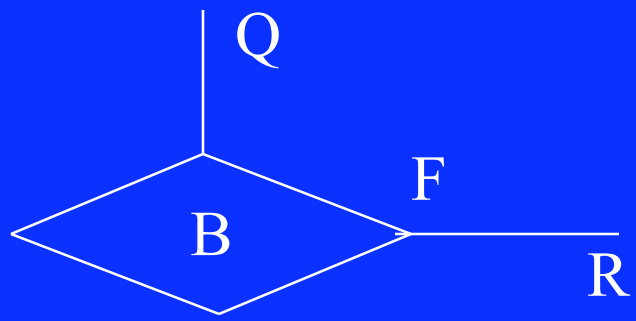
Associated lemma: $(Q' \wedge x = E')_{x' \rightarrow E}\text{-}1 \supset R$

Basic path 2



Associated lemma: $Q \wedge B \supset R$

Basic path 3



Associated lemma: $Q \wedge \neg B \supset R$

The proof

In this method, we shall let the input predicate be the starting assertion at the program entry, and let the output predicate be the ending assertion at the program exit. To prove the correctness of the program is to show that every lemma associated with a basic path is a theorem.

The proof (continued)

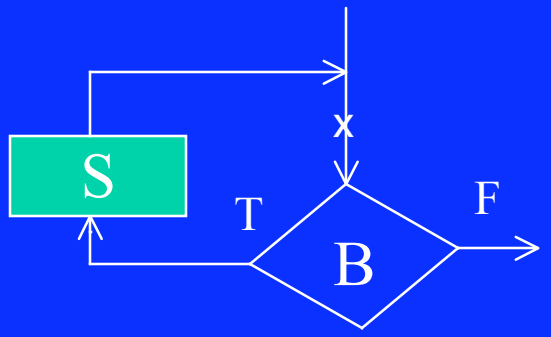
If we succeeded in doing that, then due to transitivity of the implication relation, it implies that, if the input predicate is true at the program entry, the output predicate will be true also if and when the control reaches the exit (i.e., if the execution terminates). Therefore it constitutes a proof of the partial correctness of the program.

The proof (continued)

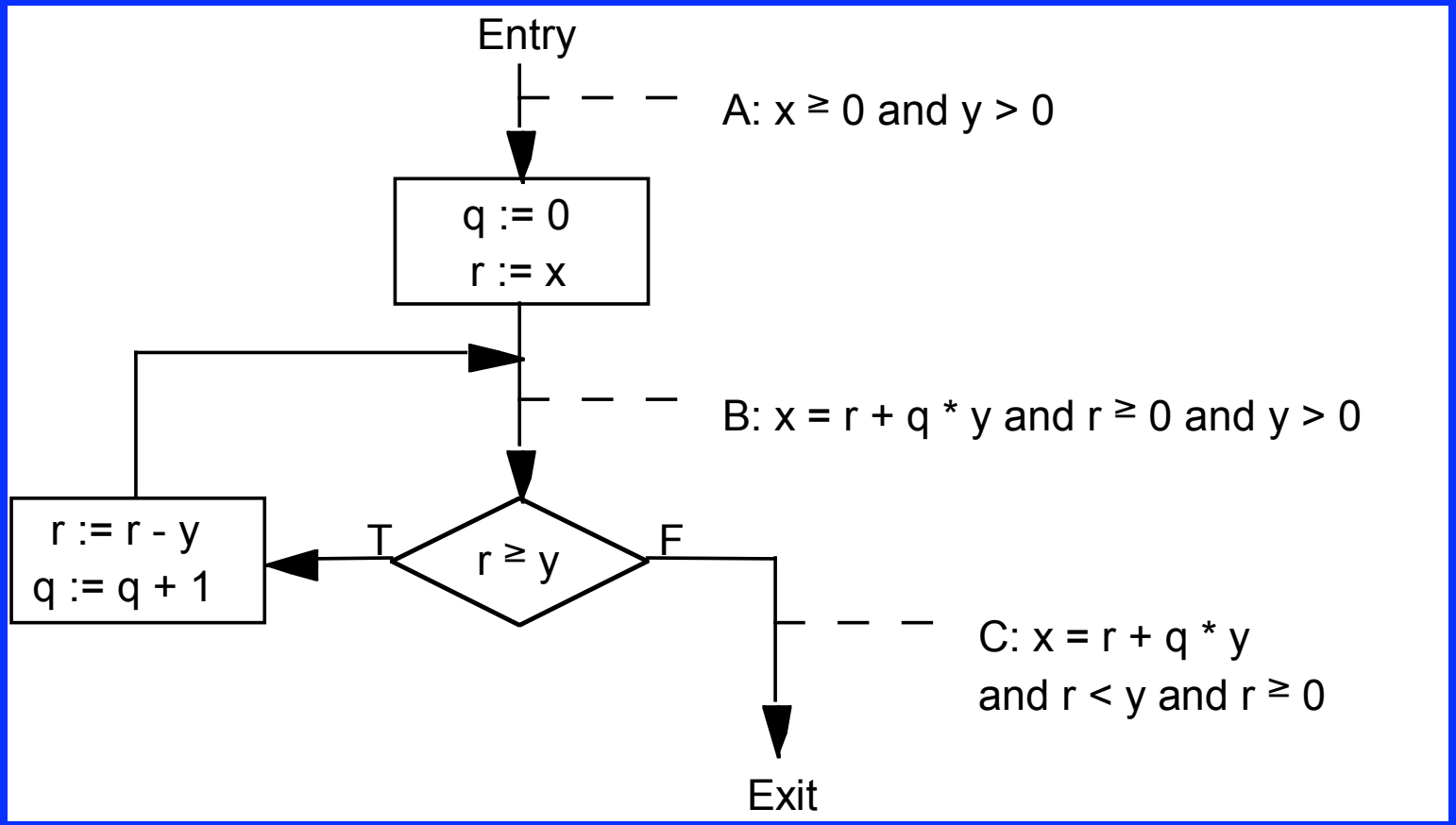
In practice, we work with composite paths instead of simple paths to reduce the number of lemma needs to be proved. A composite path is a path formed by a concatenation of more than one simple path. The lemma associated with a composite path can be constructed by observing that the effect produced by a composite path is the conjunction of that produced by its constituent simple paths.

The proof (continued)

At least one assertion should be inserted into each loop so that any path is of finite length.



Flowchart of program INTDIV



Example (continued)

Three assertions are used: A is the input predicate, C is the output predicate, and B is the assertion used to cut the loop. Assertion B cannot be simply q = 0 *and* r = x because B is not merely the ending point of path AB, it is also the beginning and ending points of path BB. Therefore, we have to guess the assertion at that point that will lead us to a successful proof. In this case, it is not difficult to guess because the output predicate provides a strong hint as to what we need at that point.

Example (continued)

There are three paths: AB, BB, and BC.

Path AB: $x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x \supset x = r + q * y \wedge r \geq 0 \wedge y > 0$

Path BB: $x = r + q*y \wedge r \geq 0 \wedge y > 0 \wedge r \geq y \wedge r' = r - y \wedge q' = q + 1 \supset x = r' + q' * y \wedge r' \geq 0 \wedge y > 0$

Path BC: $x = r + q * y \wedge r \geq 0 \wedge y > 0 \wedge \neg(r \geq y) \supset x = r + q * y \wedge r < y \wedge r \geq 0$

Example (continued)

These three lemmas can be readily proved as follows.

Lemma for Path AB: Substitute 0 for q and r for x in the consequence.

Lemma for Path BB: Eliminate q' and r' and simplify.

Lemma for Path BC: Use the fact that $\neg(r \geq y)$ is $r < y$, and simplify.

Common error

A common error made in constructing a correctness proof is that the guessed assertion is either stronger or weaker than what is needed. Let P be the correct inductive assertion to use in proving $I\{S_1;S_2\}O$, that is, $I\{S_1\}P$ and $P\{S_2\}O$ are both a theorem. If the guessed assertion is too weak, say, $P \vee \Delta$, where Δ is some extraneous predicate, $I\{S_1\}(P\vee\Delta)$ is still a theorem, but $(P\vee\Delta)\{S_2\}O$ may not be. On the other hand, if the guessed assertion is too strong, say, $P \wedge \Delta$, $(P\wedge\Delta)\{S_2\}O$ is still a theorem but $I\{S_1\}(P\wedge\Delta)$ may not be.

Common error (continued)

Consequently, if one failed to construct a proof by using the inductive assertion method, it does not necessarily mean that the program is incorrect. Failure of a proof could result either from an incorrect program or incorrect choices of inductive assertions. In comparison, the bottom-up (predicate transformation) method does not have this disadvantage.