Verifiable Programming

- Reason about imperative sequential programs such as Java programs
- Imperative program
  - defines state space
    - defined by collection of typed program variables
    - are coordinate axis of state space
  - pattern of actions operating in state space

View of imperative programming

- State space
  - program variables x1,x2, … xn
  - Cartesian product of the variable types T1, T2, … ,Tn (each type stands for the associated value set)
  - one state: n-tuple of values

View of imperative programming

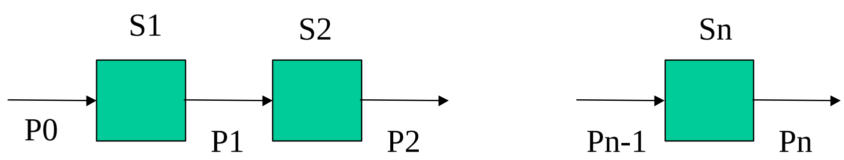
- Actions performed
  - change current state by assigning new values to program variables
  - assignment statement x:=e
    - evaluate expression e; old value of x is lost
    - e may have free variables; value of x depends on current state

View of imperative programming

- Program execution: sequence of actions A1, A2, A3, … causing transitions through a sequence of states s1,s2,… possibly terminating in a final state sn.
- Initial state s1 represents input data and final state sn represents the results

View of imperative programming

- Conditions satisfied after each action
- Special case: assignment statements Si only



Pi express properties of the states possible at the ith stage

Sloppy use: Interlude

- Terms like condition, requirement, assertion and relation are often sloppy in a technical sense. Formulas or predicates (functions with range BOOL)
- Formula $x<2y$: which predicate?

Sloppy use: Interlude

- Formula $x<2y$: which predicate?

$$p_2 \equiv l \;\; x \bullet x < 2y$$
$$p_3 \equiv l \;\; y \bullet x < 2y$$
$$p_4 \equiv l \;\; xyz \bullet x < 2y$$
$$p_1 \equiv l \;\; xy \bullet x < 2y$$

Sloppy use: Interlude

- Formula $x<2y$: which predicate?

$$p_2 \equiv l \;\; x \bullet x < 2y \qquad p_2(v) = (v < 2y)$$
$$p_3 \equiv l \;\; y \bullet x < 2y \qquad p_3(2) = (x < 4)$$
$$p_4 \equiv l \;\; xyz \bullet x < 2y \qquad p_4(5,2,v) = p_1(5,2) = (5 < 4) = f$$
$$p_1 \equiv l \;\; xy \bullet x < 2y$$

Sloppy use: Interlude

$$p \equiv l \;\; x \bullet P \quad gives \quad p(e) = P_e^x$$

x is a list of distinct variables and e is a corresponding list of expressions.

Sloppy use: Interlude

When we use the term predicate for a formula P, we mean

$$l \;\; x \bullet P$$

where x stands for $x_1, x_2, \ldots, x_n$ the list of all program variables.

We assume that functions have never hidden arguments as, for instance

$$p_2 \equiv l \;\; x \bullet x < 2y \quad p_2(v) = (v < 2y)$$

Sloppy use: Interlude

We assume that functions have never hidden arguments as, for instance

$$p_2 \equiv l \quad x \bullet x < 2y \qquad p_2(v) = (v < 2y)$$

The value of an expression e can depend on, at most, the free variables in e.

Definitions

$$s = (s_1, s_2, \cdots, r_n) \quad \textit{satisfies} \quad P$$

$$\textit{iff} \;\; P^{x}_{r}, \textit{ i.e. } P^{x_1, x_2, \ldots, x_n}_{s_1, s_2, \cdots, r_n} \;\; \textit{holds}.$$

Definitions

$$P \approx \{s \bullet P_r^x\}$$

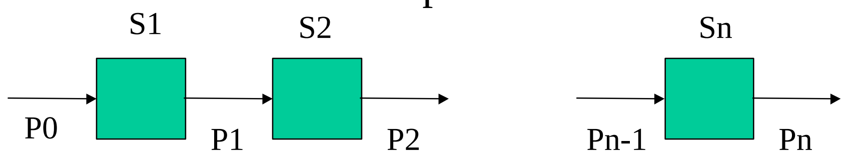
Predicate P as a set of states.

Definitions

More later.

Flow charts

- May be generalized to arbitrary flow charts with branchings and cycles whose arcs are annotated with state predicates.



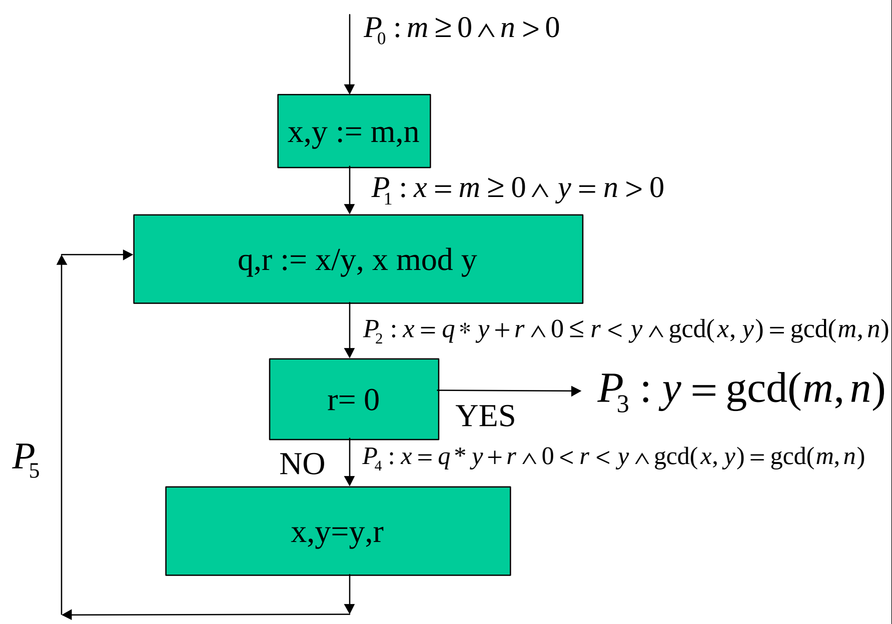
Flow charts

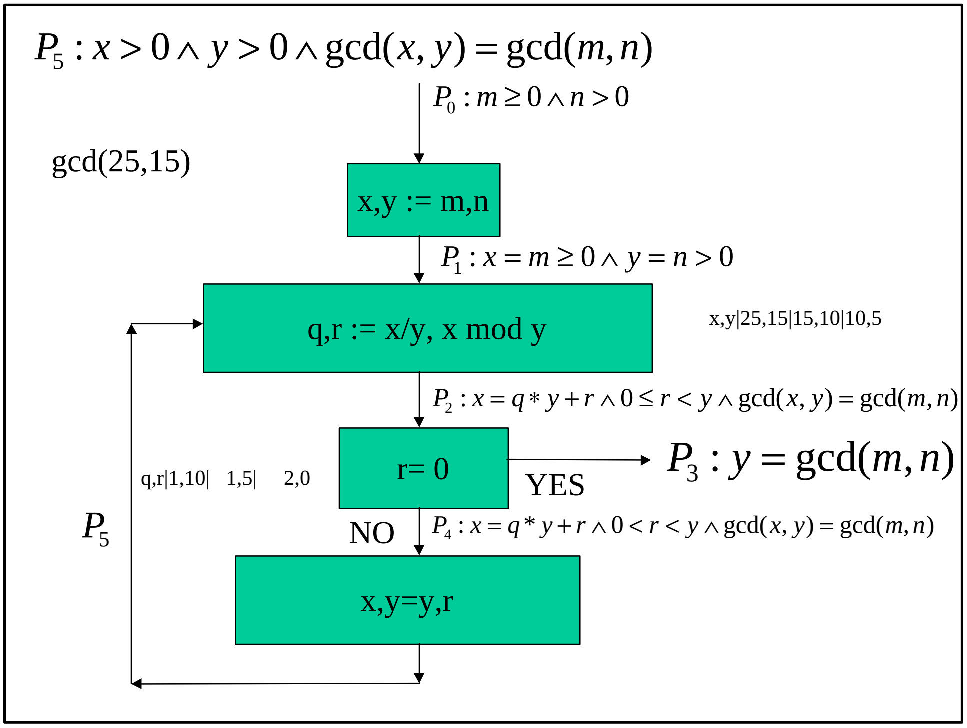
- Arcs: represent sets of states
- Nodes: represent state transitions
- Only nodes are important for execution but state predicates are of fundamental importance to understand programs and to reason about them

Flow chart-based verification

- 50 year anniversary
- J. von Neumann and H.H. Goldstine 1947
  - assertion boxes
  - operation boxes
- Robert Floyd (1967): 20 years later
- A predicate P associated with an arc A is intended to signify that the state satisfies P whenever control passes along A.

$P_5 : x > 0 \wedge y > 0 \wedge \gcd(x, y) = \gcd(m, n)$





Proof approach: Induction

- Show that assertion on each arc is satisfied whenever control reaches that arc, provided P_0 holds initially.
- For each node in the graph and every pair of incoming and outgoing arcs a1,a2 we prove that if the assertion on a1 holds before the operation, then the assertion on a2 holds after the operation.

Proof approach: Induction

- As a result of these proofs the theorem follows by induction on the number of operations performed.

Proof approach: Induction

- P1 and P5 imply $x \geq 0 \wedge y > 0 \wedge \gcd(x, y) = \gcd(m, n)$

$P_1 : x = m \geq 0 \wedge y = n > 0$

$P_5 : x > 0 \wedge y > 0 \wedge \gcd(x, y) = \gcd(m, n)$

$P_2 : x = q * y + r \wedge 0 \leq r < y \wedge \gcd(x, y) = \gcd(m, n)$

- P1 implies P2. P5 implies P2.

Proof approach: Induction

- On the YES branch, we know x is a multiple of y and therefore P3 follows.

$$P_2 : x = q * y + r \wedge 0 \le r < y \wedge \gcd(x, y) = \gcd(m, n)$$

$$P_3 : y = \gcd(m, n)$$

Proof approach: Induction

- On the NO branch, r is non-zero, which by P2 gives 0<r in P4.

$$P_2 : x = q * y + r \wedge 0 \le r < y \wedge \gcd(x, y) = \gcd(m, n)$$

$$P_4 : x = q * y + r \wedge 0 < r < y \wedge \gcd(x, y) = \gcd(m, n)$$

Proof approach: Induction

- The simultaneous assignment x,y:=y,r shows that what is true for y,r in P4 must hold for x,y in P5. x>0 and y>0 is clear.

$$P_4 : x = q * y + r \wedge 0 < r < y \wedge \gcd(x, y) = \gcd(m, n)$$

$$P_5 : x > 0 \wedge y > 0 \wedge \gcd(x, y) = \gcd(m, n)$$

- To justify gcd(x,y) = gcd(m,n): Crux

Proof approach: Induction

- Must show that P4 implies gcd(y,r)=gcd(m,n). P4 gives x=q*y+r. If y and r are evenly divisible by k, then so is x.
- F(x) = {k s.t. k divides x evenly}

$$F(y) \cap F(r) \subseteq F(x)$$

$$P_4 : x = q * y + r \wedge 0 < r < y \wedge \gcd(x, y) = \gcd(m, n)$$

Proof approach: Induction

- P4 also gives r=x -q*y.

$$F(x) \cap F(y) \subseteq F(r)$$

$$P_4 : x = q * y + r \wedge 0 < r < y \wedge \gcd(x, y) = \gcd(m, n)$$

Proof approach: Induction

- Intersection on both sides with F(y)

$$F(y) \cap F(r) \subseteq F(x)$$

$$F(x) \cap F(y) \subseteq F(r)$$

$$F(y) \cap F(r) \subseteq F(x) \cap F(y)$$

$$F(x) \cap F(y) \subseteq F(r) \cap F(y)$$

$$P_4 : x = q * y + r \wedge 0 < r < y \wedge \gcd(x, y) = \gcd(m, n)$$

Proof approach: Induction

$$F(y) \cap F(r) \subseteq F(x) \cap F(y)$$
$$F(x) \cap F(y) \subseteq F(r) \cap F(y)$$
$$F(y) \cap F(r) \supseteq F(x) \cap F(y)$$
$$F(y) \cap F(r) = F(x) \cap F(y)$$

gcd(y,r)=gcd(x,y)   QED

$$P_4 : x = q * y + r \wedge 0 < r < y \wedge \gcd(x, y) = \gcd(m, n)$$

Conditional correctness

- y=gcd(m,n) whenever the algorithm terminates (control eventually reaches the YES branch).
- Next prove termination: show that progress toward a goal is being made. Show that y decreases with each execution of the loop: y:=r is the only assignment to y in loop.

Prove termination

- P4 asserts that $r<y$. y remains positive. Thus the execution must stop.

External/internal documentation

- P0 and P3 together comprise the external specification: What a user must know.
- The other assertions: internal documentation: explain to a reader how and why the algorithm works.
- Not all assertions are equally important: if P2 is given, can easily find the others.

Loop invariant

- P2 is called a loop invariant: remains true every time around the loop.
- Loop invariant provides essential and non-obvious information.
- Note that most proof steps are trivial mathematically.

Use program notation

```
const m,n : Int; {m >=0 and n>0}
var x,y : Int = m,n;
loop {const q : Int = x/y;}
  const r : Int = x mod y;
  {x=q*y+r and 0<=r<y and gcd(x,y)=gcd(m,n)}
while r != 0;
  x,y := y,r;
repeat
{y = gcd(m,n)}
```

Two important remarks about Floyd/Hoare style verification

- Not applicable to life-size programs unless one is very careful about program structure. Consider only small part of total state space.
- State assertions should not be afterthoughts; they belong to the program design phase. Non-trivial invariants are difficult to come up with.

Help during program design phase

- Compute y for given x with relative accuracy eps : Real > 0.
- Use iteration: y, z (current term), k
- plausible initializations:
  - y:=0 or 1 or 1+x
  - z:=1 or x
  - k:=0 or 1

$$y = e^{x} = \sum_{x=0}^{\infty} \frac{x^{k}}{k!}$$

Help during program design phase

- Plausible loop assignment statements
  - y:=y+z or y+z*x/k or y+z*x/(k+1)
  - z:=z*x/k or z*x/(k+1)
  - k:=k+1

$$y = e^x = \sum_{x=0}^{\infty} \frac{x^k}{k!}$$

- 3*2*2*3*2 = 72 possible statement sets

Help during program design phase

- Choose a good loop invariant first
- k=1
- y=1+x; z=x

$$I : y = e^x = \sum_{x=0}^{\infty} \frac{x^k}{k!} \wedge z = \frac{x^k}{k!}$$

- while abs(z) > = eps*y;
  - k=k+1; z:=z*x/k; y:=y+z;
- repeat

Help during program design phase/ignore rounding errors

- Const x, eps : Real; {eps > 0}
- var k : Int =1;
- var y,z : Real =1+x; x;
- loop $\{I : y = e^x = \sum_{x=0}^{\infty} \frac{x^k}{k!} \wedge z = \frac{x^k}{k!}\}$
  - while abs(z) > = eps*y;
  - k:=k+1; z:=z*x/k; y:=y+z;
- repeat $\{y = e^x = \sum_{x=0}^{\infty} \frac{x^k}{k!} \wedge abs\left(\frac{x^k}{k!}\right) < eps * y\}$