**Program Correctness**

* **Program correctness**
  + Program specification – describes desired output for admissible inputs
    - Describes what the program should compute
    - A program’s correctness is relative to its specification
  + Testing vs. formal verification
    - “Testing can show the presence of bugs, but never their absence”
    - Impossible to determine all possible bugs – there are infinitely (or finitely but very) many possible inputs
    - Formal verification describes the program specification in logic & shows that a program satisfies its specification using a proof theory
    - Formal verification checks for all inputs
* **Core programming language**
  + Imperative program – consists of commands that modify the values of variables
  + Sequential program – has no concurrency
  + Transformational program – given inputs, the program is expected to compute outputs and terminate
  + State of a program is the values of variables at some point in the execution of the program
    - Expressions and commands evaluate relative to the current state
    - Each command in a program can change the state
  + Variable assignment
    - V := E
    - V (in the next state) is assigned the value of expression E (in the current state)
  + Postcondition
    - A wff about variables in the state after the program is completed
    - E.g. “compute a number stored in Y, whose square is less than the input X”
      * Y2 < X
  + Precondition
    - A wff about variables in the state before the program has started
    - E.g. previous example requires
      * X > 0
  + Assertions can be made in the form: (a triple)
    - Assert(P); ← P = precondition (write assert(true) if no constraints)
    - C; ← C = program
    - Assert(Q); ← Q = postcondition
  + Partial correctness
    - |=par assert(P); C; assert(Q);
    - i.e. for all program executions of C that start in a state satisfying P, if the execution of C terminates ⇒ Q is satisfied at termination
    - If C never terminates, Q doesn’t have to be satisfied
  + Total correctness
    - |= tot assert(P); C; assert(Q);
    - i.e. for all starting states satisfying P, C is guaranteed to terminate & Q is satisfied at termination
  + Logical variables – variables that do not appear the program
    - E.g. consider:

assert(x > 0); ← refers to value of x before program execution

y := 1;

x := 2;

assert(y2 < x); ← value of x is changed during execution

* + - Use logical variable:

assert(x = x0 ∧ x > 0);

y := 1;

x := 2;

assert(y2 < x0); ← x0 refers to value at the beginning of program

* + Ex:

y := 1;

z := 0;

while !(z == x) do {

z := z + 1;

y := y \* z

}

* + - Assert(x ≥ 0); ... assert(y = x!);
  + Ex:

y := 1;

while !(x == 0) do {

y := y \* x;

x := x – 1

}

* + - Assert(x = x0 ∧ x ≥ 0); … assert(y = x0!)
* **Program correctness – proof theory**
  + A.k.a. axiomatic semantics
  + Proof theories for PC are sound but often not complete – since programs work with numbers
  + Assignment rule
    - Axiom (no premises):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

assert(P[E/Var]);

Var := E;

assert(P); Asn

* + - Where Var = a variable; E = an expression; P[E/Var] a substitution of Var for E
    - Ex:

assert(2 = y); % work backwards from postcondition; this must be the

x := 2; % precondition for the postcondition to be true

assert(x = y); % asn

* + - Ex:

assert(z = x + y + 1 – 4);

x := x + y + 1

assert(z = x – 4); % asn

* + Composition rule

assert(P); assert(Q);

C1; C2;

assert(Q); assert(R);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ composition

assert(P);

C1;

C2;

assert(R);

* + - i.e. steps in a proof can be chained together
  + Implied rule

P’ |− P

Assert(P);

C;

Assert(Q);

Q |− Q’

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ implied

Assert(P’);

C;

Assert(Q’);

* + - P’ |− P and Q |− Q’ are verification conditions
    - E.g.

assert(P);

assert(Q); % implied (VC 1)

% VC 1

P |− Q

* + Program correctness proofs consist of:
    - Annotated program
      * Has a precondition, postcondition, and assertions between all commands
      * Every assertion except the precondition requires a justification using proof rules
      * Every assertion is a wff that is true whenever the program reaches that point in its execution (given the initial state satisfies the precondition)
      * Usually work backwards from postcondition to precondition
      * Ex:

assert(P);

C1;

assert(X1); proof rule

C2;

assert(X2); proof rule

…

assert(Q); proof rule

* + - Proofs of verification conditions (VCs) – label with # in annotated program, proved separately after the program
  + In A |− B or A ⇒ B
    - A is stronger than B; B is weaker than A
    - Precondition strengthening – assuming more than needed
      * i.e. P’ |− P; assert(P); C; assert(Q);
      * → assert(P’); C; assert(Q);
    - Postcondition weakening – concluding less than what is actually true
      * i.e. assert(P); C; assert(Q); Q’ |− Q
      * → assert(P); C; assert(Q’);
  + Derived assignment rule

assert(P);

Var := E;

assert(Q); % derived asn (VC 1)

% VC 1

P |− Q[E/Var]

* + One-armed conditional (if-then)

assert(P);

if B then {

assert(P ∧ B); % if-then

C;

assert(Q); % rule

};

assert(Q); % if-then (VC 1)

% VC 1

P ∧ ¬B |− Q

* + - i.e. if B is true then get assert(Q) from inside if block
    - if B is not true then get assert(Q) from P ∧ ¬B |− Q
  + Two-armed conditional (if-then-else)
    - P is unknown when working backwards

assert(P);

if B then {

assert(P ∧ B); % if-then-else

C1;

assert(Q); % rule

} else {

assert(P ∧ ¬B); % if-then-else

C2;

assert(Q); % rule

};

assert(Q); % if-then-else

* + - Modified two-armed conditional – allows working backwards

assert((B ⇒ P1) ∧ (B ⇒ P2));

if B then {

assert(P1); % if-then-else

C1;

assert(Q); % rule

} else {

assert(P2); % if-then-else

C2;

assert(Q); % rule

};

assert(Q); % if-then-else

* + While loop rule
    - Inv = loop invariant
      * True both before & after the execution of the loop body
      * By induction, Inv is preserved for every iteration of the loop
      * i.e. Inv is true before & after the entire while loop
      * Doesn’t have to be true throughout an iteration – only at beginning & end
      * Is usually a formalization of what the loop should accomplish

assert(P);

assert(Inv); % implied (VC 1)

while B do {

assert(Inv ∧ B); % Inv & guard

C;

assert(Inv); % rule

}

assert(Inv ∧ ¬B); % partial-while

assert(Q); % implied (VC 2)

% VC 1

P |− Inv

% VC 2

Inv ∧ ¬B |− Q

* Program correctness counterexamples

assert(true);

a := x + 1;

if (a – 1 = 0) {

y := 1;

} else {

y := a + 1;

};

assert(y = x + 1);

* + Provide initial state (values of program & logical variables)
    - E.g. a = doesn’t matter; x = 1; y = doesn’t matter
  + Final state
    - E.g. a = 2; x = 1; y = 3
  + Demonstrate that the precondition is satisfied by the initial state
    - E.g. [true] = T
  + Demonstrate that the postcondition is not satisfied by the final state
    - E.g. [y = x + 1] = (3 = 1 + 1) = F