Techniques for algorithm verification

Stefan D. Bruda

CS 310, Winter 2025

Correctness Statements

- A block of code with its assertions is a logical formula called correctness statement (aka Hoare triple)

```
ASSERT(P) C ASSERT(Q)
    P {C} Q
```

- We prove the validity of correctness statements using a proof system of axioms and inference rules
  - One of many variants of Hoare logic
- Assignment statement needs an axiom:
  - Motivating example: `n == n0 { n = n - 1; } n == n0 - 1`
  - First axiom attempt: $V == I \quad \{V = E;\} \quad V == [E](V \mapsto I)$
    - Supplementary condition: V is distinct from I
    - Only works for pre-conditions of form $V == I$; no good for example for `n > 0 { n = n - 1; } n >= 0`
  - It turns out that it is better to reason backward
  - Assignment axiom (C. A. R. Hoare):

$$[Q](V \mapsto E) \quad \{V = E;\} \quad Q$$

    - Example: `n-1 >= 0 { n = n - 1; } n >= 0`
    - Example: `x - y >= 0 { x = x - y; } x >= 0`

Substitutions

- $[Q](I \mapsto E)$ = like Q except that all the occurrences of I are replaced by E
- Use parentheses as necessary
  - `[2*i == j](i↦i-1)` is `2*(i-1) == j` rather than `2*i-1 == j`
- Only free occurrences of I are replaced
  - A variable introduced by a quantifier is bound, all other variables are free
    - `present <=> Exists (k=0; k<n) A[k] == x`: `k` is bound; `present`, `A`, and `x` are free

```
[ForAll (i=0; i<n) A[i]>0](i↦i+1)
                    →  ForAll (i=0; i<n) A[i]>0
[ForAll (i=0; i<n) A[i]>0](n↦n+1)
                    →  ForAll (i=0; i<n+1) A[i]>0
[i>0 => Exists(i=0; i<n) A[i]>0](i↦i+1)
                    →  (i+1)>0 => Exists(i=0; i<n) A[i]>0
```

- Rename bound identifiers if substitution $I \mapsto E$ causes free identifiers in E to become bound

```
[ForAll (i=0; i<n) A[i] > j](j↦i-1)
                   ↛  [ForAll (i=0; i<n) A[i] > i-1]
                   →  [ForAll (k=0; k<n) A[k] > i-1]
```


Proofs and Proof Tableaux

- A formal proof is a sequence of logical statements
  - A statement is either an axiom or the conclusion of an inference rule
- Difficult to read when it comes to code, so a proof tableau is often used instead
  - Consist of program code, pre- and post-conditions, and all the intermediate assertions
  - Stating non-obvious mathematical facts: comments or the following macro:

    ```
    #define FACT (P)
    ```
  - Must be able to reconstitute the formal proof out of a tableau

Formal proof

1.	`n-1>0 { n=n-1; } n>0`	(assignment)
2.	`n>1 => n-1>0`	(math)
3.	`n>1 { n=n-1; } n>0`	(strengthen)
4.	`n>0 => n>=1`	(math)
5.	`n>1 { n=n-1; } n>=1`	(weakening)

Tableau

```
ASSERT(n>1)          /* 3 */
FACT(n>1 => n-1>0)   /* 2 */
ASSERT(n-1>0)        /* 1 */
n = n-1;
ASSERT(n>0)
FACT(n>0 => n>=1)    /* 4 */
ASSERT(n>=1)         /* 5 */
```

(Note: in practice trivial facts are omitted)

Inference Rules and Proof Tableaux

- Pre-condition strengthening and post-condition weakening

$$\frac{P' \{C\} Q \qquad P \Rightarrow P'}{P \{C\} Q} \qquad \frac{P \{C\} Q \qquad Q \Rightarrow Q'}{P \{C\} Q'}$$

- Sequencing

$$\frac{P \{C_0\} Q \qquad Q \{C_1\} R}{P \{C_0\ C_1\} R}$$

$$\frac{P \{C_0\} Q \qquad Q' \{C_1\} R \qquad Q \Rightarrow Q'}{P \{C_0\ C_1\} R}$$

- If statements

$$\frac{P \texttt{ \&\& } B \{C_0\} Q \qquad P \texttt{ \&\& !}B \{C_1\} Q}{P \{\texttt{if (}B\texttt{) }C_0\texttt{ else }C_1\} Q}$$

  - Else-less if statements are best represented in tableaux using an empty else branch (empty C_1)

- Sequencing

```
ASSERT(P)
C0
ASSERT(Q)
C1
ASSERT(R)
```

- If statement

```
ASSERT(P)
if (B)
   ASSERT(P && B)
   C0
   ASSERT(Q)
else
   ASSERT(P && !B)
   C1
   ASSERT(Q)
ASSERT(Q)
```


While Loops

- A single assertion called loop invariant will usually do

$$\frac{I \texttt{ \&\& } B \{C\} I}{I \{\texttt{while (}B\texttt{) }C\} I \texttt{ \&\& !}B}$$

  - The invariant I must be preserved by the loop body
  - The invariant is also a pre-condition
  - The invariant I as well as $!B$ must be both true after the execution of the loop no matter how many times the loop executes

- Important consideration: loop termination
  - The loop invariant says nothing about termination
  - Sometimes a suitable pre-condition ensures termination
  - More generally, a variant can be shown to exist (tricky, no algorithmic method)
    - Must be positive immediately before the loop
    - Must decrease monotonically and continuously at each iteration (cannot skip values!)
    - Must ensure loop termination when it reaches 0

Tableau:

```
ASSERT(I)
while (B)
   ASSERT(I && B)
   C
   ASSERT(I)
ASSERT(I && !B)
```

Shortcut:

```
while (B) INVAR(I)
   C
```

where:

```
#define INVAR(I)
```

Local Variables

- A local variable should not be mentioned in the interface, including the pre- and post-conditions that surround the block that defines it:

$$\frac{P\ \{C\}\ Q}{P\ \{T\ I;\ C\}\ Q}$$

  - T is a type
  - The identifier I cannot be free in P or Q
  - Essentially, a local variable does not affect reasoning about the program, provided that the relevant pre- and post-conditions can be expressed without referring to that variable

For Loops

- `for` $(A_0;\ B;\ A_1)$ `C` is equivalent with $A_0;$ `while`(B) $\{$`C` $A_1;\}$.
- The following proof tableau shows this:

```
ASSERT(P)
A0
ASSERT(I)
while(B) {
    ASSERT(I && B)
    C
    A1
    ASSERT(I)
}
ASSERT(I && !B)
ASSERT(Q)
```

- The pre-condition `P` must ensure that `I` holds immediately after the execution of A_0
- `I` must be an invariant of `C` A_1;
- `I` and `!B` must together imply the desired post-condition `Q`

Array Component Assignment

- The usual assignment correctness statement will not work for assigning to array components
- We treat the result of the assignment `A[I]=E` as the new array $(\mathtt{A}|\mathtt{I} \mapsto \mathtt{E})$ such that
  - $(\mathtt{A}|\mathtt{I} \mapsto \mathtt{E})[\mathtt{I}'] = E$ whenever $\mathtt{I}' = \mathtt{I}$
  - $(\mathtt{A}|\mathtt{I} \mapsto \mathtt{E})[\mathtt{I}'] = \mathtt{A}[\mathtt{I}']$ whenever $\mathtt{I}' \neq \mathtt{I}$
- The assignment rule becomes:

$$[\mathtt{Q}](\mathtt{A} \mapsto \mathtt{A}')\ \{\mathtt{A}[\mathtt{I}] = \mathtt{E};\}\ \mathtt{Q}$$

  where $\mathtt{A}'$ is $(\mathtt{A}|\mathtt{I} \mapsto \mathtt{E})$
  - That the subscript `I` is in the range allowed for `A` must be verified separately

Do-While Loops

- `do C while (B);` is equivalent to: `{C while (B) C}`. Therefore:

$$\frac{P\ \{C\}\ I \qquad I\ \&\&\ B\ \{C\}\ I}{P\ \{\mathtt{do}\ C\ \mathtt{while}(B);\}\ I\ \&\&\ !B}$$

```
ASSERT(P)
C
ASSERT(I)
while (B)
    ASSERT(I && B)
    C
    ASSERT(I)
ASSERT(I && !B)
```

Combining Correctness Statements

- Potentially useful when we prove two post-conditions:

$$\frac{P\ \{C\}\ Q_1 \qquad P\ \{C\}\ Q_2}{P\ \{C\}\ Q_1\ \&\&\ Q_2}$$

- Potentially useful when we prove a post-condition in two different circumstances:

$$\frac{P_1\ \{C\}\ Q \qquad P_2\ \{C\}\ Q}{P_1\ ||\ P_2\ \{C\}\ Q}$$

- These are also useful in determining the weakest precondition and strongest postcondition
  - Fundamental concepts for the automation of Hoare logic proofs
- Combinations (but only using the same operator) are also sound:

$$\frac{P_1\ \{C\}\ Q_1 \qquad P_2\ \{C\}\ Q_2}{P_1\ ||\ P_2\ \{C\}\ Q_1\ ||\ Q_2} \qquad \frac{P_1\ \{C\}\ Q_1 \qquad P_2\ \{C\}\ Q_2}{P_1\ \&\&\ P_2\ \{C\}\ Q_1\ \&\&\ Q_2}$$