Specifying and Verifying Programs

We will discuss three (closely interrelated) calculi.

- **Hoare Calculus**: \( \{P\} \ c \ \{Q\} \)
  
  - If command \( c \) is executed in a pre-state with property \( P \) and terminates, it yields a post-state with property \( Q \).
  
  \[ x = a \land y = b \] \( x := x + y \) \( x = a + y \land y = b \)

- **Predicate Transformers**: \( \text{wp}(c, Q) = P \)
  
  - If the execution of command \( c \) shall yield a post-state with property \( Q \), it must be executed in a pre-state with property \( P \).
  
  \[ \text{wp}(c, x = a + y \land y = b) = (x + y = a + y \land y = b) \]

- **State Relations**: \( c : [P \Rightarrow Q]^{++} \)
  
  - The post-state generated by the execution of command \( c \) is related to the pre-state by \( P \Rightarrow Q \) (where only variables \( x, \ldots \) have changed).
  
  \[ x = x + y : [\text{var } x = \text{old } x + \text{old } y]^* \]

**The Hoare Calculus**

First and best-known calculus for program reasoning (C.A.R. Hoare).

- **“Hoare triple”**: \( \{P\} \ c \ \{Q\} \)
  
  - Logical propositions \( P \) and \( Q \), program command \( c \).
  
  - The Hoare triple is itself a logical proposition.
  
  - The Hoare calculus gives rules for constructing true Hoare triples.

- **Partial correctness** interpretation of \( \{P\} \ c \ \{Q\} \):
  
  "If \( c \) is executed in a state in which \( P \) holds, then it terminates in a state in which \( Q \) holds unless it aborts or runs forever."

  - Program does not produce wrong result.
  
  - But program also need not produce any result.
  
  - Abortion and non-termination are not (yet) ruled out.

- **Total correctness** interpretation of \( \{P\} \ c \ \{Q\} \):
  
  "If \( c \) is executed in a state in which \( P \) holds, then it terminates in a state in which \( Q \) holds."

  - Program produces the correct result.

We will use the partial correctness interpretation for the moment.
Weakening and Strengthening

\[ P \Rightarrow P' \quad \{ P' \} \quad c \quad \{ Q' \} \quad Q' \Rightarrow Q \quad \{ P \} \quad c \quad \{ Q \} \]

- **Logical derivation:** \( A_1 A_2 \)
  - **Forward:** If we have shown \( A_1 \) and \( A_2 \), then we have also shown \( B \).
  - **Backward:** To show \( B \), it suffices to show \( A_1 \) and \( A_2 \).

**Interpretation of above sentence:**
To show that, if \( P \) holds, then \( Q \) holds after executing \( c \), it suffices to show this for a \( P' \) weaker than \( P \) and a \( Q' \) stronger than \( Q \).

Precondition may be weakened, postcondition may be strengthened.

Special Commands

\{ P \} \quad \textbf{skip} \quad \{ P \} \quad \{ \text{true} \} \quad \textbf{abort} \quad \{ \text{false} \}

- The **skip** command does not change the state; if \( P \) holds before its execution, then \( P \) thus holds afterwards as well.
- The **abort** command aborts execution and thus trivially satisfies partial correctness.
  - Axiom implies \( \{ P \} \quad \textbf{abort} \quad \{ Q \} \) for arbitrary \( P, Q \).

Useful commands for reasoning and program transformations.

Scalar Assignments

\{ Q[e/x] \} \quad x := e \quad \{ Q \}

- **Syntax**
  - Variable \( x \), expression \( e \).
  - \( Q[e/x] \ldots Q \) where every free occurrence of \( x \) is replaced by \( e \).
- **Interpretation**
  - To make sure that \( Q \) holds for \( x \) after the assignment of \( e \) to \( x \), it suffices to make sure that \( Q \) holds for \( e \) before the assignment.
- **Partial correctness**
  - Evaluation of \( e \) may abort.

\[
\begin{align*}
{x + 3 < 5} & \quad x := x + 3 \quad \{ x < 5 \} \\
{x < 2} & \quad x := x + 3 \quad \{ x < 5 \}
\end{align*}
\]

Array Assignments

\{ Q[a[i \mapsto e]/a] \} \quad a[i] := e \quad \{ Q \}

- An array is modelled as a function \( a : I \rightarrow V \).
  - Index set \( I \), value set \( V \).
  - \( a[i] = e \ldots \) array \( a \) contains at index \( i \) the value \( e \).
- **Term** \( a[i \mapsto e] \) ("array \( a \) updated by assigning value \( e \) to index \( i \)"")
  - A new array that contains at index \( i \) the value \( e \).
  - All other elements of the array are the same as in \( a \).
- Thus array assignment becomes a special case of scalar assignment.
  - Think of "\( a[i] := e \)" as "\( a := a[i \mapsto e] \)".

\[
\begin{align*}
\{ a[i \mapsto x][1] > 0 \} & \quad a[i] := x \quad \{ a[1] > 0 \}
\end{align*}
\]

Arrays are here considered as basic values (no pointer semantics).
Array Assignments

How to reason about $a[i \mapsto e]$?

$$
Q[a[i \mapsto e][j]] \iff (i = j \Rightarrow Q[e]) \land (i \neq j \Rightarrow Q[a[j]])
$$

- Array Axioms
  - $i = j \Rightarrow a[i \mapsto e][j] = e$
  - $i \neq j \Rightarrow a[i \mapsto e][j] = a[j]$

Get rid of “array update terms” when applied to indices.

Command Sequences

$\{P\} \ c_1 \ \{R\} \ c_2 \ \{Q\}$

- Interpretation
  - To show that, if $P$ holds before the execution of $c_1; c_2$, then $Q$ holds afterwards, it suffices to show for some $R$ that
    - if $P$ holds before $c_1$, that $R$ holds afterwards, and that
    - if $R$ holds before $c_2$, then $Q$ holds afterwards.
  - Problem: find suitable $R$.
  - Easy in many cases (see later).

$$
\begin{align*}
\{x + y - 1 > 0\} & \ y := y - 1 \ \{x + y > 0\} \\
\{x + y > 0\} & \ x := x + y \ \{x > 0\} \\
\{x + y - 1 > 0\} & \ y := y - 1; x := x + y \ \{x > 0\}
\end{align*}
$$

The calculus itself does not indicate how to find intermediate property.

Conditionals

$\{P \land b\} \ c_1 \ \{Q\} \ \{P \land \neg b\} \ c_2 \ \{Q\}$

- Interpretation
  - To show that, if $P$ holds before the execution of the conditional, then $Q$ holds afterwards,
  - it suffices to show that the same is true for each conditional branch, under the additional assumption that this branch is executed.

$$
\begin{align*}
\{x \neq 0 \land x \geq 0\} & \ y := x \ \{y > 0\} \\
\{x \neq 0 \land x < 0\} & \ y := -x \ \{y > 0\} \\
\{x \neq 0\} & \ \text{if } x \geq 0 \ \text{then } y := x \ \text{else } y := -x \ \{y > 0\}
\end{align*}
$$

Loops

$\{\text{true}\} \ \text{loop} \ \{\text{false}\} \ \{I \land b\} \ c \ \{I \land \neg b\}$

- Interpretation:
  - The loop command does not terminate and thus trivially satisfies partial correctness.
  - Axiom implies $\{P\} \ \text{loop} \ \{Q\}$ for arbitrary $P, Q$.
  - If it is the case that
    - $I$ holds before the execution of the while-loop and
    - $I$ also holds after every iteration of the loop body, then $I$ holds after the execution of the loop (together with the negation of the loop condition $b$).
  - $I$ is a loop invariant.

- Problem:
  - Rule for while-loop does not have arbitrary pre/post-conditions $P, Q$.
  - In practice, we combine this rule with the strengthening/weakening-rule.
Loops (Generalized)

\[ P \Rightarrow I \quad \{ I \land b \} \quad c \quad \{ I \} \quad (I \land \neg b) \Rightarrow Q \]

\[
\{ P \} \quad \text{while } b \quad \text{do} \quad c \quad \{ Q \}
\]

**Interpretation:**

To show that, if before the execution of a while-loop the property \( P \) holds, after its termination the property \( Q \) holds, it suffices to show for some property \( I \) (the loop invariant) that

- \( I \) holds before the loop is executed (i.e. that \( P \) implies \( I \)),
- if \( I \) holds when the loop body is entered (i.e. if also \( b \) holds), that after the execution of the loop body \( I \) still holds,
- when the loop terminates (i.e. if \( b \) does not hold), \( I \) implies \( Q \).

**Problem:** find appropriate loop invariant \( I \).

The strongest relationship between all variables modified in loop body.

The calculus itself does not indicate how to find suitable loop invariant.

Example

\[ I : \iff s = \sum_{j=1}^{i-1} j \land 1 \leq i \leq n + 1 \]
\[ (n \geq 0 \land i = 1 \land s = 0) \Rightarrow I \]
\[ \{ I \land i \leq n \} \quad s := s + i; \quad i := i + 1 \quad \{ I \} \]
\[ (l \land i \neq n) \Rightarrow s = \sum_{j=1}^{n} j \]
\[ \{ n \geq 0 \land i = 1 \land s = 0 \} \quad \text{while } i \leq n \quad \text{do} \quad (s := s + i; \quad i := i + 1) \quad \{ s = \sum_{j=1}^{n} j \} \]

The invariant captures the “essence” of a loop; only by giving its invariant, a true understanding of a loop is demonstrated.

Backward Reasoning

Implication of rule for command sequences and rule for assignments:

\[
\{ P \} \quad c \quad \{ Q[e/x] \} \\
\{ P \} \quad c; \quad x := e \quad \{ Q \}
\]

**Interpretation**

- If the last command of a sequence is an assignment, we can remove the assignment from the proof obligation.
- By multiple application, assignment sequences can be removed from the back to the front.

\[
\begin{align*}
\{ P \} & \quad x := x + 1; \quad \{ x + 1 = 5 \} & \quad P \Rightarrow x = 4 \\
\{ P \} & \quad y := 2 \ast x; \quad \{ x + 2x = 15 \} & \quad (\Leftrightarrow x = 4) \\
\{ P \} & \quad z := x + y \quad \{ x + y = 15 \} & \quad (\Leftrightarrow x = 5)
\end{align*}
\]
Weakest Preconditions

A calculus for "backward reasoning" (E.W. Dijkstra).

- Predicate transformer \( \text{wp} \)
  - Function "wp" that takes a command \( c \) and a postcondition \( Q \) and returns a precondition.
  - Read \( \text{wp}(c, Q) \) as "the weakest precondition of \( c \) w.r.t. \( Q \)."
  - \( \text{wp}(c, Q) \) is a precondition for \( c \) that ensures \( Q \) as a postcondition.
  - Must satisfy \( \{ \text{wp}(c, Q) \} \ c \{ Q \} \).
  - \( \text{wp}(c, Q) \) is the weakest such precondition.
  - Take any \( P \) such that \( \{ P \} c \{ Q \} \).
  - Then \( P \Rightarrow \text{wp}(c, Q) \).

- Consequence: \( \{ P \} c \{ Q \} \) iff \( (P \Rightarrow \text{wp}(c, Q)) \)
  - We want to prove \( \{ P \} c \{ Q \} \).
  - We may prove \( P \Rightarrow \text{wp}(c, Q) \) instead.

Verification is reduced to the calculation of weakest preconditions.

Weakest Preconditions

The weakest precondition of each program construct.

\[
\begin{align*}
\text{wp}(\text{skip}, Q) &= Q \\
\text{wp}(\text{abort}, Q) &= \text{true} \\
\text{wp}(x := e, Q) &= Q[e/x] \\
\text{wp}(c_1 ; c_2 , Q) &= \text{wp}(c_1 , \text{wp}(c_2, Q)) \\
\text{wp}(\text{if } b \text{ then } c_1 \text{ else } c_2 , Q) &= (b \Rightarrow \text{wp}(c_1, Q)) \land (\neg b \Rightarrow \text{wp}(c_2, Q)) \\
\text{wp}(\text{while } b \text{ do } c , Q) &= (b \Rightarrow \text{wp}(c, Q)) \land (\neg b \Rightarrow Q) \\
\end{align*}
\]

Loops represent a special problem (see later).

Strongest Postcondition

A calculus for forward reasoning.

- Predicate transformer \( \text{sp} \)
  - Function "sp" that takes a precondition \( P \) and a command \( c \) and returns a postcondition.
  - Read \( \text{sp}(c, P) \) as "the strongest postcondition of \( c \) w.r.t. \( P \)."
  - \( \text{sp}(c, P) \) is a postcondition for \( c \) that is ensured by precondition \( P \).
  - Must satisfy \( \{ P \} c \{ \text{sp}(c, P) \} \).
  - \( \text{sp}(c, P) \) is the strongest such postcondition.
  - Take any \( P, Q \) such that \( \{ P \} c \{ Q \} \).
  - Then \( \text{sp}(c, P) \Rightarrow Q \).

- Consequence: \( \{ P \} c \{ Q \} \) iff \( (\text{sp}(c, P) \Rightarrow Q) \).
  - We want to prove \( \{ P \} c \{ Q \} \).
  - We may prove \( \text{sp}(c, P) \Rightarrow Q \) instead.

Verification is reduced to the calculation of strongest postconditions.

Forward Reasoning

Sometimes, we want to derive a postcondition from a given precondition.

\[
\{ P \} x := e \{ \exists x_0 : P[x_0/x] \land x = e[x_0/x] \}
\]

- Forward Reasoning
  - What is the maximum we know about the post-state of an assignment \( x := e \), if the pre-state satisfies \( P \)?
  - We know that \( P \) holds for some value \( x_0 \) (the value of \( x \) in the pre-state) and that \( x \) equals \( e[x_0/x] \).

\[
\begin{align*}
&\{ x \geq 0 \land y = a \} \\
&x := x + 1 \\
&\{ \exists x_0 : x_0 \geq 0 \land y = a \land x = x_0 + 1 \} \\
&(\Leftrightarrow \{ x_0 : x_0 \geq 0 \land x = x_0 + 1 \} \land y = a) \\
&(\Leftrightarrow x > 0 \land y = a)
\end{align*}
\]

Strongest Postcondition

A calculus for forward reasoning.

- Predicate transformer \( \text{sp} \)
  - Function "sp" that takes a precondition \( P \) and a command \( c \) and returns a postcondition.
  - Read \( \text{sp}(c, P) \) as "the strongest postcondition of \( c \) w.r.t. \( P \)."
  - \( \text{sp}(c, P) \) is a postcondition for \( c \) that is ensured by precondition \( P \).
  - Must satisfy \( \{ P \} c \{ \text{sp}(c, P) \} \).
  - \( \text{sp}(c, P) \) is the strongest such postcondition.
  - Take any \( P, Q \) such that \( \{ P \} c \{ Q \} \).
  - Then \( \text{sp}(c, P) \Rightarrow Q \).

- Consequence: \( \{ P \} c \{ Q \} \) iff \( (\text{sp}(c, P) \Rightarrow Q) \).
  - We want to prove \( \{ P \} c \{ Q \} \).
  - We may prove \( \text{sp}(c, P) \Rightarrow Q \) instead.

Verification is reduced to the calculation of strongest postconditions.
Strongest Postconditions

The strongest postcondition of each program construct.

\[
\begin{align*}
\text{sp}(\text{skip}, P) &= P \\
\text{sp}(\text{abort}, P) &= \text{false} \\
\text{sp}(x := e, P) &= \exists x_0 : P[x_0/x] \land x = e[x_0/x] \\
\text{sp}(c_1 ; c_2, P) &= \text{sp}(c_2, \text{sp}(c_1, P)) \\
\text{sp}(\text{if } b \text{ then } c_1 \text{ else } c_2, P) &= \text{sp}(c_1, P \land b) \lor \text{sp}(c_2, P \land \neg b) \\
\text{sp}(\text{while } b \text{ do } c, P) &= \text{sp}(c, P \land b) \lor (P \land \neg b)
\end{align*}
\]

Forward reasoning as a (less-known) alternative to backward-reasoning.

Hoare Calculus and Predicate Transformers

In practice, often a combination of the calculi is applied.

\[
\{P\} \ c_1 ; \text{while } b \text{ do } c ; c_2 \ {Q}\]

- Assume \(c_1\) and \(c_2\) do not contain loop commands.
- It suffices to prove

\[
\{\text{sp}(P, c_1)\} \text{ while } b \text{ do } c \ {\text{wp}(c_2, Q)}
\]

Predicate transformers are applied to reduce the verification of a program to the Hoare-style verification of loops.

Weakest Liberal Preconditions for Loops

Why not apply predicate transformers to loops?

\[
\begin{align*}
\text{wp}(\text{loop}, Q) &= \text{true} \\
\text{wp}(\text{while } b \text{ do } c, Q) &= L_0(Q) \land L_1(Q) \land L_2(Q) \land \ldots \\
L_0(Q) &= \text{true} \\
L_{i+1}(Q) &= (\neg b \Rightarrow Q) \land (b \Rightarrow \text{wp}(c, L_i(Q)))
\end{align*}
\]

- Interpretation
  - Weakest precondition that ensures that loops stops in a state satisfying \(Q\), unless it aborts or runs forever.
  - Infinite sequence of predicates \(L_i(Q)\):
    - Weakest precondition that ensures that after less than \(i\) iterations the state satisfies \(Q\), unless the loop aborts or does not yet terminate.
  - Alternative view: \(L_i(Q) = \text{wp}(i, Q)\)

Example

\[
\begin{align*}
\text{wp}(\text{while } i < n \text{ do } i := i + 1, Q) \\
&= \text{L}_0(Q) = \text{true} \\
&= \text{L}_1(Q) = (i \not< n \Rightarrow Q) \land (i < n \Rightarrow \text{wp}(i := i + 1, Q)) \\
&= \text{L}_2(Q) = (i \not< n \Rightarrow Q) \land (i < n \Rightarrow \text{wp}(i := i + 1, Q)) \\
&= \text{L}_3(Q) = (i \not< n \Rightarrow Q) \land (i < n \Rightarrow \text{wp}(i := i + 1, Q)) \\
&= (i \not< n \Rightarrow Q) \land (i < n \Rightarrow (i + 1 \not< n \Rightarrow Q[i + 1/i])) \\
&= (i \not< n \Rightarrow Q[i + 1/i]) \land (i + 1 < n \Rightarrow (i + 2 \not< n \Rightarrow Q[i + 2/i]))
\end{align*}
\]
Weakest Liberal Preconditions for Loops

Sequence \( L_i(Q) \) is monotonically increasing in strength:

\[
\forall i \in \mathbb{N} : L_{i+1}(Q) \Rightarrow L_i(Q).
\]

The weakest precondition is the “lowest upper bound”:

\[
\forall i \in \mathbb{N} : \text{wp}(\text{while } b \text{ do } c, Q) \Rightarrow L_i(Q).
\]

We can only compute weaker approximation \( L_i(Q) \).

\[
\text{wp}(\text{while } b \text{ do } c, Q) \Rightarrow L_i(Q).
\]

We want to prove \( \{P\} \text{ while } b \text{ do } c \{Q\} \).

This is equivalent to proving \( P \Rightarrow \text{wp}(\text{while } b \text{ do } c, Q) \).

Thus \( P \Rightarrow L_i(Q) \) must hold as well.

If we can prove \( \neg(P \Rightarrow L_i(Q)) \), . . .


Falsification is possible by use of approximation \( L_i \), but verification is not.

A Constructive Definition of Arrays

\[
\text{newcontext "arrays2";}
\]

\[
\text{INDEX: TYPE = NAT;}
\]

\[
\text{ELEM: TYPE;}
\]

\[
\text{ARR: TYPE = [INDEX, ARRAY INDEX OF ELEM];}
\]

\[
\text{any: ARRAY INDEX OF ELEM;}
\]

\[
\text{anylem: ELEM;}
\]

\[
\text{anyarray: ARR;}
\]

\[
\text{content: ARR -> (ARRAY INDEX OF ELEM) = LAMBDA(a:ARR): a.1;}
\]

% the array operations

\[
\text{length: ARR -> INDEX = LAMBDA(a:ARR): a.0;}
\]

\[
\text{new: INDEX -> ARR = LAMBDA(n:INDEX): (n, any);}\]

\[
\text{put: (ARR, INDEX, ELEM) -> ARR = LAMBDA(a:ARR, i:INDEX, e:ELEM): IF i < length(a) THEN (length(a), content(a) WITH [i]:=e) ELSE anyarray ENDIF;}
\]

\[
\text{get: (ARR, INDEX) -> ELEM = LAMBDA(a:ARR, i:INDEX): IF i < length(a) THEN content(a)[i] ELSE anyelem ENDIF;}
\]

Proof of Fundamental Array Properties

% the classical array axioms as formulas to be proved

length1: FORMULA

\[
\text{FORALL(n:INDEX): length(new(n)) = n;}
\]

length2: FORMULA

\[
\text{FORALL(a:ARR, i:INDEX, e:ELEM): i < length(a) => length(put(a, i, e)) = length(a);}\]

get1: FORMULA

\[
\text{FORALL(a:ARR, i:INDEX, e:ELEM): i < length(a) => get(put(a, i, e), i) = e;}
\]

get2: FORMULA

\[
\text{FORALL(a:ARR, i, j:INDEX, e:ELEM): i < length(a) AND j < length(a) AND i /= j => get(put(a, i, e), j) = get(a, j);}\]
Proof of a Higher-Level Array Property

% extensionality on low-level arrays
extensionality: AXIOM
FORALL(a, b: ARRAY INDEX OF ELEM):
a = b <-> (FORALL(i: INDEX): a[i] = b[i]);

% unassigned parts hold identical values
unassigned: AXIOM
FORALL(a: ARRAY, i: NAT):
(i >= length(a)) => content(a)[i] = content(b)[i];

% extensionality on arrays to be proved
extensionality: FORMULA
FORALL(a, b: ARRAY INDEX OF ELEM):
FORALL(a: ARRAY, i: NAT):
(i < length(a) => get(a, i) = get(b, i));

The Verification Conditions

A: Input => Invariant
B_1: Invariant \land i < n \land r = -1 \land a[i] = x \Rightarrow Invariant[i/r]
B_2: Invariant \land i < n \land r = -1 \land a[i] \neq x \Rightarrow Invariant[i+1/i]
C: Invariant \land \neg(i < n \land r = -1) \Rightarrow Output

Input: olda = a \land oldx = x \land n = length(a) \land i = 0 \land r = -1
Output: a = olda \land x = oldx \land
((r = -1 \land \forall i: 0 \leq i < length(a) \Rightarrow a[i] \neq x) \lor
(0 \leq r < length(a) \land a[r] = x \land \forall i: 0 \leq i < r \Rightarrow a[i] \neq x))

Invariant: olda = a \land oldx = x \land n = length(a) \land
0 \leq i \land 0 \leq j < i \Rightarrow a[j] \neq x \land
(r = -1 \lor \{ i \land i < n \land a[r] = x \})

The verification conditions A, B_1, B_2, C have to be proved.

A Program Verification

Verification of the following Hoare triple:
\{ olda = a \land oldx = x \land n = length(a) \land i = 0 \land r = -1 \}
while i < n \land r = -1 do
    if a[i] = x
    then r := i
    else i := i + 1
\{ a = olda \land x = oldx \land
((r = -1 \land \forall i: 0 \leq i < |a| \Rightarrow a[i] \neq x) \lor
(0 \leq r < |a| \land a[r] = x \land \forall i: 0 \leq i < r \Rightarrow a[i] \neq x)) \}
Find the smallest index r of an occurrence of value x in array a (r = -1, if x does not occur in a).
The Verification Conditions (Contd)

...  

A: FORMULA  
Input => Invariant(a, x, i, n, r);

B1: FORMULA  
Invariant(a, x, i, n, r) AND i < n AND r = -1 AND get(a,i) = x  
=> Invariant(a,x,i,n,i);

B2: FORMULA  
Invariant(a, x, i, n, r) AND i < n AND r = -1 AND get(a,i) /= x  
=> Invariant(a,x,i+1,n,r);

C: FORMULA  
Invariant(a, x, i, n, r) AND NOT(i < n AND r = -1)  
=> Output;

The Proofs

A:  
B1: (2 user actions) (1 user action)  
B2:  
C: (3 user actions) (6 user actions)

Termination

1. The Hoare Calculus
2. Predicate Transformers
3. Proving Verification Conditions
4. Termination
5. Abortion
6. Procedures

1. The Hoare Calculus
2. Predicate Transformers
3. Proving Verification Conditions
4. Termination
5. Abortion
6. Procedures
Weakest Preconditions for Loops

\[
wp(\text{while } b \text{ do } c, Q) = L_0(Q) \lor L_1(Q) \lor L_2(Q) \lor \ldots
\]

\[
L_0(Q) = \text{false}
\]

\[
L_i(Q) = (¬b \Rightarrow Q) \land (b \Rightarrow wp(c, L_i(Q)))
\]

- New interpretation
  - Weakest precondition that ensures that the loop terminates in a state in which \(Q\) holds, unless it aborts.

- New interpretation of \(L_i(Q)\)
  - Weakest precondition that ensures that the loop terminates after less than \(i\) iterations in a state in which \(Q\) holds, unless it aborts.

- Preserves property: \(\{P\} c \{Q\}\) iff \((P \Rightarrow wp(c, Q))\)

- Now for total correctness interpretation of Hoare calculus.

- Preserves alternative view: \(L_i(Q) \iff wp(i, i+1)\)

\[
\begin{align*}
I : &\iff s = \sum_{j=1}^{i-1} j \land 1 \leq i \leq n + 1 \\
(n \geq 0 \land i = 1 \land s = 0) &\Rightarrow I \land I \Rightarrow n - i + 1 \geq 0 \\
\{I \land i \leq n \land n - i + 1 = N\} &\Rightarrow s = \sum_{j=1}^{i-1} j \\
(l_i &\leq n) \Rightarrow s = \sum_{j=1}^{l_i} j
\end{align*}
\]

In practice, termination is easy to show (compared to partial correctness).

---

Example

\[
L_i(Q) = \text{false}
\]

\[
L_{i+1}(Q) = (¬b \Rightarrow Q) \land (b \Rightarrow wp(c, L_i(Q)))
\]

- New interpretation
  - Weakest precondition that ensures that the loop terminates in a state in which \(Q\) holds, unless it aborts.

- New interpretation of \(L_i(Q)\)
  - Weakest precondition that ensures that the loop terminates after less than \(i\) iterations in a state in which \(Q\) holds, unless it aborts.

- Preserves property: \(\{P\} c \{Q\}\) iff \((P \Rightarrow wp(c, Q))\)

- Now for total correctness interpretation of Hoare calculus.

- Preserves alternative view: \(L_i(Q) \iff wp(i, i+1)\)

\[
\begin{align*}
I : &\iff s = \sum_{j=1}^{i-1} j \land 1 \leq i \leq n + 1 \\
(n \geq 0 \land i = 1 \land s = 0) &\Rightarrow I \land I \Rightarrow n - i + 1 \geq 0 \\
\{I \land i \leq n \land n - i + 1 = N\} &\Rightarrow s = \sum_{j=1}^{i-1} j \\
(l_i &\leq n) \Rightarrow s = \sum_{j=1}^{l_i} j
\end{align*}
\]

In practice, termination is easy to show (compared to partial correctness).
1. The Hoare Calculus
2. Predicate Transformers
3. Proving Verification Conditions
4. Termination
5. Abortion
6. Procedures

Abortion

New rules to prevent abortion.

\[
\{\text{false}\} \text{ abort } \{\text{true}\}
\]
\[
\{Q[e/x] \land D(e)\} x := e \{Q\}
\]
\[
\{Q[a[i \mapsto e]/a] \land D(e) \land 0 \leq i < \text{length}(a)\} \ a[i] := e \{Q\}
\]

- New interpretation of \{P\} \ c \ {Q}\.
- If execution of \ c \ starts in a state, in which property \ P \ holds, then it does not abort and eventually terminates in a state in which \ Q \ holds.

Sources of abortion.
- Division by zero.
- Index out of bounds exception.

\(D(e)\) makes sure that every subexpression of \ e \ is well defined.

Definedness of Expressions

\(D(0) = \text{true}\).
\(D(1) = \text{true}\).
\(D(x) = \text{true}\).
\(D(a[i]) = D(i) \land 0 \leq i < \text{length}(a)\).
\(D(e_1 + e_2) = D(e_1) \land D(e_2)\).
\(D(e_1 \times e_2) = D(e_1) \land D(e_2)\).
\(D(e_1/e_2) = D(e_1) \land D(e_2) \land e_2 \neq 0\).
\(D(\text{true}) = \text{true}\).
\(D(\text{false}) = \text{true}\).
\(D(\neg b) = D(b)\).
\(D(b_1 \land b_2) = D(b_1) \land D(b_2)\).
\(D(b_1 \lor b_2) = D(b_1) \land D(b_2)\).
\(D(e_1 < e_2) = D(e_1) \land D(e_2)\).
\(D(e_1 \leq e_2) = D(e_1) \land D(e_2)\).
\(D(e_1 > e_2) = D(e_1) \land D(e_2)\).
\(D(e_1 \geq e_2) = D(e_1) \land D(e_2)\).

Assumes that expressions have already been type-checked.
Abortion

Similar modifications of weakest preconditions.

\[
\begin{align*}
\wp(\text{abort}, Q) &= \text{false} \\
\wp(x := e, Q) &= Q[e/x] \land D(e) \\
\wp(\text{if } b \text{ then } c_1 \text{ else } c_2, Q) &= \\
&= D(b) \land (b \Rightarrow \wp(c_1, Q)) \land (\neg b \Rightarrow \wp(c_2, Q)) \\
\wp(\text{while } b \text{ do } c, Q) &= (L_0(Q) \lor L_1(Q) \lor L_2(Q) \lor \ldots) \\
\end{align*}
\]

\[
L_0(Q) = \text{false} \\
L_{i+1}(Q) = D(b) \land (\neg b \Rightarrow Q) \land (b \Rightarrow \wp(c, L_i(Q)))
\]

\[
\wp(c, Q) \text{ now makes sure that the execution of } c \text{ does not abort but eventually terminates in a state in which } Q \text{ holds.}
\]

Procedure Specifications

- global \( g \).
- requires \( Pre \).
- ensures \( Post \).
- \( o := p(i) \{ c \} \)

- Specification of a procedure \( p \) implemented by a command \( c \).
  - Input parameter \( i \), output parameter \( o \), global variable \( g \).
  - Command \( c \) may read/write \( i \), \( o \), and \( g \).
  - Precondition \( Pre \) (may refer to \( i, g \)).
  - Postcondition \( Post \) (may refer to \( i, o, g, g_0 \)).
    - \( g_0 \) denotes the value of \( g \) before the execution of \( p \).
- Proof obligation
  \[
  \{ Pre \land i_0 = i \land g_0 = g \} \ c \ \{ Post[i_0/i] \}
  \]

Proof of the correctness of the implementation of a procedure with respect to its specification.

Example

- Procedure specification:
  - global \( g \).
  - requires \( g \geq 0 \land i > 0 \).
  - ensures \( g_0 = g \cdot i + o \land 0 \leq o < i \).
  - \( o := p(i) \{ o := g \% i; \ g := g / i \} \)

- Proof obligation:
  \[
  \{ g \geq 0 \land i > 0 \land i_0 = i \land g_0 = g \} \\
  o := g \% i; \ g := g / i \\
  \{ g_0 = g \cdot i_0 + o \land 0 \leq o < i_0 \}
  \]

A procedure that divides \( g \) by \( i \) and returns the remainder.
Procedure Calls

A call of $p$ provides actual input argument $e$ and output variable $x$.

$$x := p(e)$$

Similar to assignment statement; we thus first give an alternative (equivalent) version of the assignment rule.

- Original:
  $$\{D(e) \land Q[e/x]\}
  x := e
  \{Q\}$$

- Alternative:
  $$\{D(e) \land \forall x': x' = e \Rightarrow Q[x'/x]\}
  x := e
  \{Q\}$$

The new value of $x$ is given name $x'$ in the precondition.

Corresponding Predicate Transformers

$$wp(x = p(e), Q) =
D(e) \land Pre[e/i] \land
\forall x', g':
\text{Post}[e/i, x'/o, g/g_0, g'/g] \Rightarrow Q[x'/x, g'/g]$$

$$sp(P, x = p(e)) =
\exists x_0, g_0:
\text{P}[x_0/y, g_0/g] \land
(\text{Pre}[e/x_0, x_0, g_0/g]/i, g_0/g] \Rightarrow \text{Post}[e/x_0, x_0, g_0/g]/i, x/o])$$

Explicit naming of old/new values required.

Example

- Procedure specification:
  global $g$
  requires $g \geq 0 \land i > 0$
  ensures $g_0 = g \cdot i + o \land 0 \leq o < i$
  $o = p(i) \{o := g \cdot i; g := g/i\}$

- Procedure call:
  $$\{g \geq 0 \land g = N \land b \geq 0\}
  x = p(b + 1)
  \{g \cdot (b + 1) \leq N < (g + 1) \cdot (b + 1)\}$$

To be proved:

$$g \geq 0 \land g = N \land b \geq 0 \Rightarrow
D(b + 1) \land g \geq 0 \land b + 1 > 0 \land
\forall x', g':
\text{g} = g' \cdot (b + 1) + x' \land 0 \leq x' < b + 1 \Rightarrow
\text{g} \cdot (b + 1) \leq N < (g' + 1) \cdot (b + 1)$$