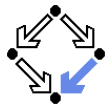


# Hoare Calculus and Predicate Transformers

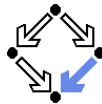
Wolfgang Schreiner  
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)  
Johannes Kepler University, Linz, Austria  
<http://www.risc.uni-linz.ac.at>



1. The Hoare Calculus for Non-Loop Programs
2. Predicate Transformers
3. Partial Correctness of Loop Programs
4. Total Correctness of Loop Programs
5. Abortion
6. Procedures

## The Hoare Calculus

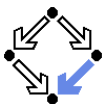


Calculus for reasoning about imperative programs.

- **“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 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.

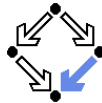
## General Rules



$$\frac{P \Rightarrow Q}{\{P\} \{Q\}} \quad \frac{P \Rightarrow P' \quad \{P'\} c \{Q'\} \quad Q' \Rightarrow Q}{\{P\} c \{Q\}}$$

- **Logical derivation**:  $\frac{A_1 \quad A_2}{B}$ 
  - 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 sentences**:
  - To show that, if  $P$  holds in a state, then  $Q$  holds in the same state (no command is executed), it suffices to show  $P$  implies  $Q$ .
    - Hoare triples are ultimately reduced to classical logic.
  - 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



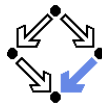
Commands modeling “emptiness” and abortion.

$$\{P\} \mathbf{skip} \{P\} \quad \{\mathbf{true}\} \mathbf{abort} \{\mathbf{false}\}$$

- 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  $\{P\} \mathbf{abort} \{Q\}$  for arbitrary  $P, Q$ .

Useful commands for reasoning and program transformations.

## Array Assignments



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

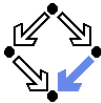
- An array is modelled as a function  $a : I \rightarrow V$ 
  - Index set  $I$ , value set  $V$ .
  - $a[i] = e \dots a$  holds at index  $i$  the value  $e$ .
- **Updated array**  $a[i \mapsto e]$ 
  - Array that is constructed from  $a$  by mapping index  $i$  to value  $e$ .
  - Axioms (for all  $a : I \rightarrow V, i \in I, j \in I, e \in V$ ):

$$\begin{aligned} i = j &\Rightarrow a[i \mapsto e][j] = e \\ i \neq j &\Rightarrow a[i \mapsto e][j] = a[j] \end{aligned}$$

$$\begin{aligned} \{a[i \mapsto x][1] > 0\} \quad a[i] := x \quad \{a[1] > 0\} \\ \{(i = 1 \Rightarrow x > 0) \wedge (i \neq 1 \Rightarrow a[1] > 0)\} \quad a[i] := x \quad \{a[1] > 0\} \end{aligned}$$

Index violations and pointer semantics of arrays not yet considered.

## Scalar Assignments

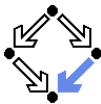


$$\{Q[e/x]\} x := e \{Q\}$$

- **Syntax**
  - Variable  $x$ , expression  $e$ .
  - $Q[e/x] \dots Q$  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.

$$\begin{aligned} \{x + 3 < 5\} \quad x := x + 3 \quad \{x < 5\} \\ \{x < 2\} \quad x := x + 3 \quad \{x < 5\} \end{aligned}$$

## Command Sequences

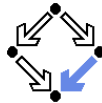


$$\frac{\{P\} c_1 \{R_1\} \quad R_1 \Rightarrow R_2 \quad \{R_2\} c_2 \{Q\}}{\{P\} c_1; c_2 \{Q\}}$$

- **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  $R_1$  and  $R_2$  with  $R_1 \Rightarrow R_2$  that
    - if  $P$  holds before  $c_1$ , that  $R_1$  holds afterwards, and that
    - if  $R_2$  holds before  $c_2$ , then  $Q$  holds afterwards.
- **Problem:** find suitable  $R_1$  and  $R_2$ 
  - Easy in many cases (see later).

$$\frac{\{x + y - 1 > 0\} y := y - 1 \{x + y > 0\} \quad \{x + y > 0\} x := x + y \{x > 0\}}{\{x + y - 1 > 0\} y := y - 1; x := x + y \{x > 0\}}$$

# Conditionals



$$\frac{\{P \wedge b\} c_1 \{Q\} \quad \{P \wedge \neg b\} c_2 \{Q\}}{\{P\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{Q\}}$$

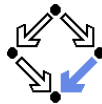
$$\frac{\{P \wedge b\} c \{Q\} \quad (P \wedge \neg b) \Rightarrow Q}{\{P\} \text{ if } b \text{ then } c \{Q\}}$$

## 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.

$$\frac{\{x \neq 0 \wedge x \geq 0\} y := x \{y > 0\} \quad \{x \neq 0 \wedge x < 0\} y := -x \{y > 0\}}{\{x \neq 0\} \text{ if } x \geq 0 \text{ then } y := x \text{ else } y := -x \{y > 0\}}$$

# Backward Reasoning



Implication of rule for command sequences and rule for assignments:

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

## 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.

$\{P\}$	$\{P\}$	$\{P\}$	$\{P\}$	$P \Rightarrow x = 4$
$x := x+1;$	$x := x+1;$	$x := x+1;$	$\{x+1 = 5\}$	
$y := 2*x;$	$y := 2*x;$	$\{x+2x = 15\}$	$(\Leftrightarrow x = 4)$	
$z := x+y$	$\{x+y = 15\}$	$(\Leftrightarrow 3x = 15)$		
$\{z = 15\}$		$(\Leftrightarrow x = 5)$		

# 1. The Hoare Calculus for Non-Loop Programs

## 2. Predicate Transformers

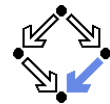
## 3. Partial Correctness of Loop Programs

## 4. Total Correctness of Loop Programs

## 5. Abortion

## 6. Procedures

# Weakest Preconditions



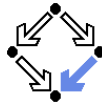
A calculus for “backward reasoning”.

## Predicate transformer 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

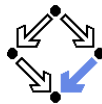


The weakest precondition of each program construct.

$\text{wp}(\text{skip}, Q) \Leftrightarrow Q$   
 $\text{wp}(\text{abort}, Q) \Leftrightarrow \text{true}$   
 $\text{wp}(x := e, Q) \Leftrightarrow Q[e/x]$   
 $\text{wp}(c_1; c_2, Q) \Leftrightarrow \text{wp}(c_1, \text{wp}(c_2, Q))$   
 $\text{wp}(\text{if } b \text{ then } c_1 \text{ else } c_2, Q) \Leftrightarrow (b \Rightarrow \text{wp}(c_1, Q)) \wedge (\neg b \Rightarrow \text{wp}(c_2, Q))$   
 $\text{wp}(\text{if } b \text{ then } c, Q) \Leftrightarrow (b \Rightarrow \text{wp}(c, Q)) \wedge (\neg b \Rightarrow Q)$

Alternative formulation of a program calculus.

## Strongest Postcondition

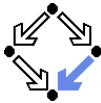


A calculus for forward reasoning.

- **Predicate transformer sp**
  - Function “sp” that takes a precondition  $P$  and a command  $c$  and returns a postcondition.
  - Read  $\text{sp}(P, c)$  as “the strongest postcondition of  $c$  w.r.t.  $P$ ”.
- $\text{sp}(P, c)$  is a **postcondition** for  $c$  that is ensured by precondition  $P$ .
  - Must satisfy  $\{P\} c \{ \text{sp}(P, c) \}$ .
- $\text{sp}(P, c)$  is the **strongest** such postcondition.
  - Take any  $P, Q$  such that  $\{P\} c \{Q\}$ .
  - Then  $\text{sp}(P, c) \Rightarrow Q$ .
- Consequence:  $\{P\} c \{Q\}$  iff  $(\text{sp}(P, c) \Rightarrow Q)$ .
  - We want to prove  $\{P\} c \{Q\}$ .
  - We may prove  $\text{sp}(P, c) \Rightarrow Q$  instead.

Verification is reduced to the calculation of strongest postconditions.

## Forward Reasoning



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

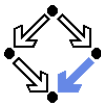
$$\{P\} x := e \{ \exists x_0 : P[x_0/x] \wedge x = e[x_0/x] \}$$

### 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  $e[x_0/x]$ .

$$\begin{aligned} & \{x \geq 0 \wedge y = a\} \\ & \quad x := x + 1 \\ & \{ \exists x_0 : x_0 \geq 0 \wedge y = a \wedge x = x_0 + 1 \} \\ & (\Leftrightarrow (\exists x_0 : x_0 \geq 0 \wedge x = x_0 + 1) \wedge y = a) \\ & (\Leftrightarrow x > 0 \wedge y = a) \end{aligned}$$

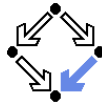
## Strongest Postconditions



The strongest postcondition of each program construct.

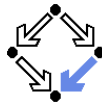
$\text{sp}(P, \text{skip}) \Leftrightarrow P$   
 $\text{sp}(P, \text{abort}) \Leftrightarrow \text{false}$   
 $\text{sp}(P, x := e) \Leftrightarrow \exists x_0 : P[x_0/x] \wedge x = e[x_0/x]$   
 $\text{sp}(P, c_1; c_2) \Leftrightarrow \text{sp}(\text{sp}(P, c_1), c_2)$   
 $\text{sp}(P, \text{if } b \text{ then } c_1 \text{ else } c_2) \Leftrightarrow \text{sp}(P \wedge b, c_1) \vee \text{sp}(P \wedge \neg b, c_2)$   
 $\text{sp}(P, \text{if } b \text{ then } c) \Leftrightarrow \text{sp}(P \wedge b, c) \vee (P \wedge \neg b)$

The use of predicate transformers is an alternative/supplement to the Hoare calculus; this view is due to Dijkstra.



1. The Hoare Calculus for Non-Loop Programs
2. Predicate Transformers
3. Partial Correctness of Loop Programs
4. Total Correctness of Loop Programs
5. Abortion
6. Procedures

## Example



$$I := s = \sum_{j=1}^{i-1} j \wedge (n \geq 0 \Rightarrow 1 \leq i \leq n+1) \wedge (n < 0 \Rightarrow i = 1)$$

$$(i = 1 \wedge s = 0) \Rightarrow I$$

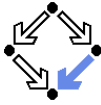
$$\{I \wedge i \leq n\} s := s + i; i := i + 1 \{I\}$$

$$(I \wedge i \not\leq n) \Rightarrow s = \sum_{j=1}^n j$$

$$\frac{}{\{i = 1 \wedge s = 0\} \text{ while } i \leq n \text{ do } (s := s + i; i := i + 1) \{s = \sum_{j=1}^n j\}}$$

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

## The Hoare Calculus and Loops

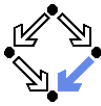


$$\{\text{true}\} \text{ loop } \{\text{false}\} \quad \frac{P \Rightarrow I \quad \{I \wedge b\} c \{I\} \quad (I \wedge \neg b) \Rightarrow Q}{\{P\} \text{ while } b \text{ do } c \{Q\}}$$

### Interpretation:

- The **loop** command does not terminate and thus trivially satisfies partial correctness.
  - Axiom implies  $\{P\} \text{ loop } \{Q\}$  for arbitrary  $P, Q$ .
- 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  $I$ .
  - Strongest relationship between all variables modified in loop body.

## Practical Aspects



We want to verify the following program:

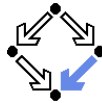
$$\{P\} c_1; \text{ while } b \text{ do } c; c_2 \{Q\}$$

- Assume  $c_1$  and  $c_2$  do not contain loop commands.
- It suffices to prove

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

Verification of loops is the core of most program verifications.

## Weakest Liberal Preconditions for Loops



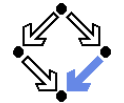
$\text{wp}(\text{loop}, Q) \Leftrightarrow \text{true}$   
 $\text{wp}(\text{while } b \text{ do } c, Q) \Leftrightarrow \forall i \in \mathbb{N} : L_i(Q)$

$L_0(Q) \Leftrightarrow \text{true}$   
 $L_{i+1}(Q) \Leftrightarrow (\neg b \Rightarrow Q) \wedge (b \Rightarrow \text{wp}(c, L_i(Q)))$

### 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:  $L_i(Q) \Leftrightarrow \text{wp}(\text{if}_i, Q)$ 
  - $\text{if}_0 := \text{loop}$
  - $\text{if}_{i+1} := \text{if } b \text{ then } (c; \text{if}_i)$

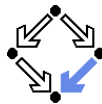
## Example



$\text{wp}(\text{while } i < n \text{ do } i := i + 1, Q)$

$L_0(Q) \Leftrightarrow \text{true}$   
 $L_1(Q) \Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, \text{true}))$   
 $\Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{true})$   
 $\Leftrightarrow (i \not< n \Rightarrow Q)$   
 $L_2(Q) \Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, i \not< n \Rightarrow Q))$   
 $\Leftrightarrow (i \not< n \Rightarrow Q) \wedge$   
 $(i < n \Rightarrow (i + 1 \not< n \Rightarrow Q[i + 1/i]))$   
 $L_3(Q) \Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1,$   
 $(i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow (i + 1 \not< n \Rightarrow Q[i + 1/i])))$   
 $\Leftrightarrow (i \not< n \Rightarrow Q) \wedge$   
 $(i < n \Rightarrow ((i + 1 \not< n \Rightarrow Q[i + 1/i]) \wedge$   
 $(i + 1 < n \Rightarrow (i + 2 \not< n \Rightarrow Q[i + 2/i])))$

## Weakest Liberal Preconditions for Loops

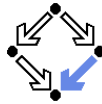


- Sequence  $L_i(Q)$  is monotonically increasing in strength:
  - $\forall i \in \mathbb{N} : L_{i+1}(Q) \Rightarrow L_i(Q)$ .
- The weakest precondition is the “lowest upper bound”:
  - $\forall i \in \mathbb{N} : \text{wp}(\text{while } b \text{ do } c, Q) \Rightarrow L_i(Q)$ .
  - $\forall P : (\forall i \in \mathbb{N} : P \Rightarrow L_i(Q)) \Rightarrow (P \Rightarrow \text{wp}(\text{while } b \text{ do } c, Q))$ .
- 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  $\neg(P \Rightarrow L_i(Q))$ , ...
  - $\{P\} \text{ while } b \text{ do } c \{Q\}$  does **not** hold.
  - If we fail, we may try the easier proof  $\neg(P \Rightarrow L_{i+1}(Q))$ .

Falsification is possible by use of approximation  $L_i$ , but verification is not.

1. The Hoare Calculus for Non-Loop Programs
2. Predicate Transformers
3. Partial Correctness of Loop Programs
4. Total Correctness of Loop Programs
5. Abortion
6. Procedures

## Total Correctness of Loops



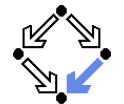
Hoare rules for **loop** and **while** are replaced as follows:

$$\{\text{false}\} \text{loop} \{\text{false}\} \quad \frac{P \Rightarrow I \quad I \wedge b \Rightarrow t > 0 \quad \{I \wedge b \wedge t = N\} c \quad \{I \wedge t < N\} (I \wedge \neg b) \Rightarrow Q}{\{P\} \text{while } b \text{ do } c \quad \{Q\}}$$

- New interpretation of  $\{P\} c \{Q\}$ .
  - If execution of  $c$  starts in a state where  $P$  holds, then execution **terminates** in a state where  $Q$  holds, unless it aborts.
  - Non-termination is ruled out, abortion not (yet).
  - The **loop** command thus does not satisfy total correctness.
- **Termination term  $t$** .
  - Denotes a natural number before and after every loop iteration.
  - If  $t = N$  before an iteration, then  $t < N$  after the iteration.
  - Consequently, if term denotes zero, loop must terminate.

Instead of the natural numbers, any *well-founded ordering* may be used for the domain of  $t$ .

## Example



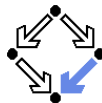
$$I := s = \sum_{j=1}^{i-1} j \wedge (n \geq 0 \Rightarrow 1 \leq i \leq n+1) \wedge (n < 0 \Rightarrow i = 1)$$

$$(i = 1 \wedge s = 0) \Rightarrow I \quad I \wedge i \leq n \Rightarrow n - i + 1 > 0$$

$$\frac{\{I \wedge i \leq 0 \wedge n - i + 1 = N\} s := s + i; i := i + 1 \quad \{I \wedge n - i + 1 < N\} (I \wedge i \not\leq n) \Rightarrow s = \sum_{j=1}^n j}{\{i = 1 \wedge s = 0\} \text{while } i \leq n \text{ do } (s := s + i; i := i + 1) \quad \{s = \sum_{j=1}^n j\}}$$

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

## Weakest Preconditions for Loops



$$\text{wp}(\text{loop}, Q) \Leftrightarrow \text{false}$$

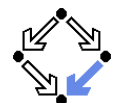
$$\text{wp}(\text{while } b \text{ do } c, Q) \Leftrightarrow \exists i \in \mathbb{N} : L_i(Q)$$

$$L_0(Q) := \text{false}$$

$$L_{i+1}(Q) := (\neg b \Rightarrow Q) \wedge (b \Rightarrow \text{wp}(c, L_i(Q)))$$

- **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  $(P \Rightarrow \text{wp}(c, Q))$ 
  - Now for **total correctness** interpretation of Hoare calculus.
- Preserves alternative view:  $L_i(Q) \Leftrightarrow \text{wp}(\text{if}_i, Q)$ 
  - $\text{if}_0 := \text{loop}$
  - $\text{if}_{i+1} := \text{if } b \text{ then } (c; \text{if}_i)$

## Example



$$\text{wp}(\text{while } i < n \text{ do } i := i + 1, Q)$$

$$L_0(Q) := \text{false}$$

$$L_1(Q) := (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_0(Q)))$$

$$\Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{false})$$

$$\Leftrightarrow i \not< n \wedge Q$$

$$L_2(Q) := (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_1(Q)))$$

$$\Leftrightarrow (i \not< n \Rightarrow Q) \wedge$$

$$i < n \Rightarrow (i + 1 \not< n \wedge Q[i + 1/i])$$

$$L_3(Q) := (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_2(Q)))$$

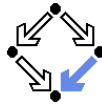
$$\Leftrightarrow (i \not< n \Rightarrow Q) \wedge$$

$$(i < n \Rightarrow ((i + 1 \not< n \Rightarrow Q[i + 1/i]) \wedge$$

$$(i + 1 < n \Rightarrow (i + 2 \not< n \wedge Q[i + 2/i])))$$

...

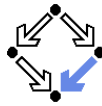
## Weakest Preconditions for Loops



- Sequence  $L_i(Q)$  is now monotonically **decreasing** in strength:
  - $\forall i \in \mathbb{N} : L_i(Q) \Rightarrow L_{i+1}(Q)$ .
- The weakest precondition is the “greatest lower bound”:
  - $\forall i \in \mathbb{N} : L_i(Q) \Rightarrow \text{wp}(\text{while } b \text{ do } c, Q)$ .
  - $\forall P : (\forall i \in \mathbb{N} : L_i(Q) \Rightarrow P) \Rightarrow (\text{wp}(\text{while } b \text{ do } c, Q) \Rightarrow P)$ .
- We can only compute a stronger approximation  $L_i(Q)$ .
  - $L_i(Q) \Rightarrow \text{wp}(\text{while } b \text{ do } c, Q)$ .
- We want to prove  $\{P\} c \{Q\}$ .
  - It suffices to prove  $P \Rightarrow \text{wp}(\text{while } b \text{ do } c, Q)$ .
  - It thus also suffices to prove  $P \Rightarrow L_i(Q)$ .
  - If proof fails, we may try the easier proof  $P \Rightarrow L_{i+1}(Q)$

Verifications are typically not successful with finite approximation of weakest precondition.

## Abortion



New rules to prevent abortion.

$$\frac{\{\text{false}\} \text{ abort } \{\text{true}\}}{\{Q[e/x] \wedge D(e)\} x := e \{Q\}}$$

$$\{Q[a[i \mapsto e]/a] \wedge D(e) \wedge 0 \leq i < \text{length}(a)\} a[i] := e \{Q\}$$

- 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.

## 1. The Hoare Calculus for Non-Loop Programs

## 2. Predicate Transformers

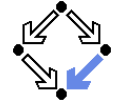
## 3. Partial Correctness of Loop Programs

## 4. Total Correctness of Loop Programs

## 5. Abortion

## 6. Procedures

## Definedness of Expressions



$$D(0) \Leftrightarrow \text{true}.$$

$$D(1) \Leftrightarrow \text{true}.$$

$$D(x) \Leftrightarrow \text{true}.$$

$$D(a[i]) \Leftrightarrow D(i) \wedge 0 \leq i < \text{length}(a).$$

$$D(e_1 + e_2) \Leftrightarrow D(e_1) \wedge D(e_2).$$

$$D(e_1 * e_2) \Leftrightarrow D(e_1) \wedge D(e_2).$$

$$D(e_1 / e_2) \Leftrightarrow D(e_1) \wedge D(e_2) \wedge e_2 \neq 0.$$

$$D(\text{true}) \Leftrightarrow \text{true}.$$

$$D(\text{false}) \Leftrightarrow \text{true}.$$

$$D(\neg b) \Leftrightarrow D(b).$$

$$D(b_1 \wedge b_2) \Leftrightarrow D(b_1) \wedge D(b_2).$$

$$D(b_1 \vee b_2) \Leftrightarrow D(b_1) \wedge D(b_2).$$

$$D(e_1 < e_2) \Leftrightarrow D(e_1) \wedge D(e_2).$$

$$D(e_1 \leq e_2) \Leftrightarrow D(e_1) \wedge D(e_2).$$

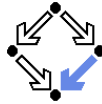
$$D(e_1 > e_2) \Leftrightarrow D(e_1) \wedge D(e_2).$$

$$D(e_1 \geq e_2) \Leftrightarrow D(e_1) \wedge D(e_2).$$

Assumes that expressions have already been type-checked.



# Abortion



Slight modification of existing rules.

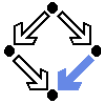
$$\frac{\{P \wedge b \wedge D(b)\} c_1 \{Q\} \quad \{P \wedge \neg b \wedge D(b)\} c_2 \{Q\}}{\{P\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{Q\}}$$

$$\frac{\{P \wedge b \wedge D(b)\} c \{Q\} \quad (P \wedge \neg b \wedge D(b)) \Rightarrow Q}{\{P\} \text{ if } b \text{ then } c \{Q\}}$$

$$\frac{P \Rightarrow I \quad I \Rightarrow (T \in \mathbb{N} \wedge D(b)) \quad \{I \wedge b \wedge T = t\} c \{I \wedge T < t\} \quad (I \wedge \neg b) \Rightarrow Q}{\{P\} \text{ while } b \text{ do } c \{Q\}}$$

Expressions must be defined in any context.

# Abortion



Similar modifications of weakest preconditions.

$$\text{wp}(\text{abort}, Q) \Leftrightarrow \text{false}$$

$$\text{wp}(x := e, Q) \Leftrightarrow Q[e/x] \wedge D(e)$$

$$\text{wp}(\text{if } b \text{ then } c_1 \text{ else } c_2, Q) \Leftrightarrow$$

$$D(b) \wedge (b \Rightarrow \text{wp}(c_1, Q)) \wedge (\neg b \Rightarrow \text{wp}(c_2, Q))$$

$$\text{wp}(\text{if } b \text{ then } c, Q) \Leftrightarrow D(b) \wedge (b \Rightarrow \text{wp}(c, Q)) \wedge (\neg b \Rightarrow Q)$$

$$\text{wp}(\text{while } b \text{ do } c, Q) \Leftrightarrow \exists i \in \mathbb{N} : L_i(Q)$$

$$L_0(Q) \Leftrightarrow \text{false}$$

$$L_{i+1}(Q) \Leftrightarrow D(b) \wedge (\neg b \Rightarrow Q) \wedge (b \Rightarrow \text{wp}(c, L_i(Q)))$$

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

## 1. The Hoare Calculus for Non-Loop Programs

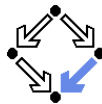
## 2. Predicate Transformers

## 3. Partial Correctness of Loop Programs

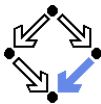
## 4. Total Correctness of Loop Programs

## 5. Abortion

## 6. Procedures



# Procedure Specifications



global  $F$ ;  
 requires  $Pre$ ;  
 ensures  $Post$ ;  
 $o = p(i) \{ c \}$

### ■ Specification of procedure $o = p(i)$ .

■ Input parameter  $i$ , output parameter  $o$ .

■ A call has form  $y = p(e)$  for expression  $e$  and variable  $y$ .

■ Set of global variables ("frame")  $F$ .

■ Those global variables that  $p$  may read/write (in addition to  $i, o$ ).

■ Let  $f$  denote all variables in  $F$ .

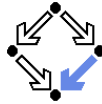
■ Precondition  $Pre$  (may refer to  $i, f$ ).

■ Postcondition  $Post$  (may refer to  $i, f, f_0, o$ ).

■ Proof obligation

$$\{Pre \wedge i_0 = i \wedge f_0 = f\} c \{Post[i_0/i]\}$$

## Procedure Calls



First let us give an alternative (equivalent) version of the assignment rule.

- Original:

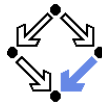
$$\frac{\{D(e) \wedge Q[e/x]\}}{x := e} \{Q\}$$

- Alternative:

$$\frac{\{D(e) \wedge \forall x' : x' = e \Rightarrow Q[x'/x]\}}{x := e} \{Q\}$$

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

## Corresponding Predicate Transformers

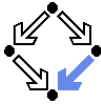


$$\begin{aligned} wp(y = p(e), Q) &\Leftrightarrow \\ &D(e) \wedge Pre[e/i] \wedge \\ &\forall y', f' : \\ &Post[e/i, y'/o, f/f_0, f'/f] \Rightarrow Q[y'/y, f'/f] \end{aligned}$$

$$\begin{aligned} sp(P, y = p(e)) &\Leftrightarrow \\ &\exists y_0, f_0 : \\ &P[y_0/y, f_0/f] \wedge Post[e/y_0/y, f_0/f]/i, y/o \end{aligned}$$

Explicit naming of old/new values required.

## Procedure Calls



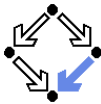
From this, we can derive a rule for the correctness of procedure calls.

$$\frac{\{D(e) \wedge Pre[e/i] \wedge \forall y', f' : Post[e/i, y'/o, f/f_0, f'/f] \Rightarrow Q[y'/y, f'/f]\}}{p(e, y)} \{Q\}$$

- $Pre[e/i]$  refers to the values of the actual argument  $e$  (rather than to the formal parameter  $i$ ).
- $y'$  and  $f'$  denote the values of the vars  $y$ , and  $f$  after the call.
- $Post[. . .]$  refers to the argument values before and after the call.
- $Q[y'/y, f'/f]$  refers to the argument values after the call.

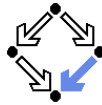
Modular reasoning: rule only relies on the *specification* of  $p$ , not on its implementation.

## Procedure Calls Example



- Procedure specification:
  - global  $f$
  - requires  $f \geq 0 \wedge i > 0$
  - ensures  $f_0 = f \cdot i + o \wedge 0 \leq o < i$
  - $o = \text{divides}F(i)$
- Procedure call:
  - $\{f \geq 0 \wedge f = N \wedge b \geq 0\}$
  - $y = \text{divides}F(b+1)$
  - $\{f \cdot (b+1) \leq N < (f+1) \cdot (b+1)\}$
- To be ultimately proved:
  - $f \geq 0 \wedge f = N \wedge b \geq 0 \Rightarrow$
  - $D(b+1) \wedge f \geq 0 \wedge b+1 > 0 \wedge$
  - $\forall y', f' :$
  - $f = f' \cdot (b+1) + y' \wedge 0 \leq y' < b+1 \Rightarrow$
  - $f' \cdot (b+1) \leq N < (f'+1) \cdot (b+1)$

## Not Yet Covered



- Primitive data types.
  - `int` values are actually finite precision integers.
- More data and control structures.
  - `switch`, `do-while` (easy); `continue`, `break`, `return` (more complicated).
  - Records can be handled similar to arrays.
- Recursion.
  - Procedures may not terminate due to recursive calls.
- Exceptions and Exception Handling.
  - Short discussion in the context of ESC/Java2 later.
- Pointers and Objects.
  - Here reasoning gets complicated.
- ...

The more features are covered, the more complicated reasoning becomes.