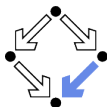


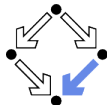
# Specifying and Verifying Programs (Part 1)

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

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



# 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  $Q$ .

$$\{x = a \wedge y = b\} x := x + y \{x = a + y \wedge y = b\}$$

- **Predicate Transformers:**  $wp(c, Q) = P$

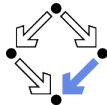
- If the execution of command  $c$  shall yield a post-state with property  $Q$ , it must be executed in a pre-state with property  $P$ .

$$wp(x := x + y, x = a + y \wedge y = b) = (x + y = a + y \wedge y = b)$$

- **State Relations:**  $c : [P \Rightarrow Q]^x, \dots$

- The post-state generated by the execution of command  $c$  is related to the pre-state by  $P \Rightarrow Q$  (where only variables  $x, \dots$  have changed).

$$x = x + y : [\text{var } x = \text{old } x + \text{old } y]^x$$



- 
1. **The Hoare Calculus**
  2. Checking Verification Conditions
  3. Predicate Transformers
  4. Generating Verification Conditions
  5. Termination
  6. Proving Verification Conditions
  7. Abortion
  8. Procedures

# 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.

# The Rules of the Hoare Calculus

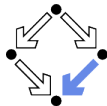


Hoare calculus rules are inference rules with Hoare triples as proof goals.

$$\frac{\{P_1\} c_1 \{Q_1\} \dots \{P_n\} c_n \{Q_n\} \quad VC_1, \dots, VC_m}{\{P\} c \{Q\}}$$

- Application of a rule to a triple  $\{P\} c \{Q\}$  to be verified yields
  - other triples  $\{P_1\} c_1 \{Q_1\} \dots \{P_n\} c_n \{Q_n\}$  to be verified, and
  - formulas  $VC_1, \dots, VC_m$  (the **verification conditions**) to be proved.
- Given a Hoare triple  $\{P\} c \{Q\}$  as the root of the **verification tree**:
  - The rules are repeatedly applied until the leaves of the tree do not contain any more Hoare triples.
  - If all verification conditions in the tree can be proved, the root of the tree represents a valid Hoare triple.

The Hoare calculus generates verification conditions such that the validity of the conditions implies the validity of the original Hoare triple.



# Weakening and Strengthening

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

- Logical derivation:  $\frac{A_1 \ A_2}{B}$ 
  - 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 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.

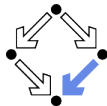
# Special Commands



$\{P\}$  **skip**  $\{P\}$        $\{\text{true}\}$  **abort**  $\{\text{false}\}$

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

Useful commands for reasoning and program transformations.



# 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

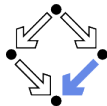
- To make sure that  $Q$  holds for  $x$  after the assignment of  $e$  to  $x$ , it suffices to make sure that  $Q$  holds for  $e$  before the assignment.

## ■ Partial correctness

- Evaluation of  $e$  may abort.

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





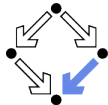
# Array Assignments

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

- An array is modelled as a **function**  $a : I \rightarrow V$ .
  - Index set  $I$ , value set  $V$ .
  - $a[i] = e$  ... 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 “ $a := a[i \mapsto e]$ ”.

$$\{\underline{a[i \mapsto x]}[1] > 0\} \quad a[i] := x \quad \{a[1] > 0\}$$

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



# Array Assignments

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

$$\begin{aligned} & Q[\underline{a[i \mapsto e][j]}] \\ & \quad \rightsquigarrow \\ & (i = j \Rightarrow Q[e]) \wedge (i \neq j \Rightarrow Q[a[j]]) \end{aligned}$$

## ■ Array Axioms

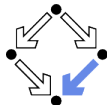
$$i = j \Rightarrow a[i \mapsto e][j] = e$$

$$i \neq j \Rightarrow \underline{a[i \mapsto e][j]} = a[j]$$

$$\begin{array}{lll} \{ \underline{a[i \mapsto x][1]} > 0 \} & a[i] := x & \{ a[1] > 0 \} \\ \{ (i = 1 \Rightarrow x > 0) \wedge (i \neq 1 \Rightarrow \underline{a[1]} > 0) \} & a[i] := x & \{ a[1] > 0 \} \end{array}$$

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

# Command Sequences



$$\frac{\{P\} c_1 \{R\} \quad \{R\} c_2 \{Q\}}{\{P\} c_1; c_2 \{Q\}}$$

## ■ 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  $R$  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).

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

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

# Conditionals



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

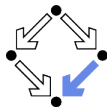
$$\frac{\{P \wedge b\} c \{Q\} \quad (P \wedge \neg b) \Rightarrow Q}{\{P\} \text{ if } b \text{ then } c \{Q\}}$$

## ■ 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.

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

# Loops



$$\{ \text{true} \} \text{ loop } \{ \text{false} \} \quad \frac{\{ I \wedge b \} c \{ I \}}{\{ I \} \text{ while } b \text{ do } c \{ I \wedge \neg b \}}$$

## ■ Interpretation:

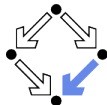
- The **loop** command does not terminate and thus trivially satisfies partial correctness.
  - Axiom implies  $\{P\} \text{ loop } \{Q\}$  for arbitrary  $P, Q$ .
- If it is the case that
  - $I$  holds before the execution of the **while**-loop and
  - $I$  also holds after every iteration of the loop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



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

## ■ 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 $I$ .

- Strongest relationship between all variables modified in loop body.

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

# Example



$$I :\Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$$

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

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

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

---

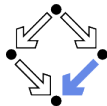
$$\{n \geq 0 \wedge s = 0 \wedge i = 1\} \text{ while } i \leq n \text{ do } (s := s + i; i := i + 1) \{s = \sum_{j=1}^n j\}$$

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



- 
1. The Hoare Calculus
  - 2. Checking Verification Conditions**
  3. Predicate Transformers
  4. Generating Verification Conditions
  5. Termination
  6. Proving Verification Conditions
  7. Abortion
  8. Procedures





# A Program Verification

- Verification of the following Hoare triple:  
 $\{Input\} \text{ while } i \leq n \text{ do } (s := s + i; i := i + 1) \{Output\}$
- Auxiliary predicates:  
 $Input : \Leftrightarrow n \geq 0 \wedge s = 0 \wedge i = 1$   
 $Output : \Leftrightarrow s = \sum_{j=1}^n j$   
 $Invariant : \Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n + 1$
- Verification conditions:  
 $A : \Leftrightarrow Input \Rightarrow Invariant$   
 $B : \Leftrightarrow Invariant \wedge i \leq n \Rightarrow Invariant[i + 1/i][s + i/s]$   
 $C : \Leftrightarrow Invariant \wedge i \not\leq n \Rightarrow Output$

If the verification conditions are valid, the Hoare triple is true.

# RISCAL: Checking Program Execution



```
val N:Nat; type number =  $\mathbb{N}[N]$ ; type index =  $\mathbb{N}[N+1]$ ; type result =  $\mathbb{N}[N \cdot (1+N)/2]$ ;
```

```
proc summation(n:number): result
  requires  $n \geq 0$ ;
  ensures result =  $\sum j:\text{number with } 1 \leq j \wedge j \leq n. j$ ;
{
  var s:result := 0;
  var i:index := 1;
  while  $i \leq n$  do
    invariant s =  $\sum j:\text{number with } 1 \leq j \wedge j \leq i-1. j$ ;
    invariant  $1 \leq i \wedge i \leq n+1$ ;
    {
      s := s+i;
      i := i+1;
    }
  return s;
}
```

We check for some  $N$  the program execution; this implies that the invariant is not too strong.

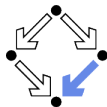
# RISCAL: Checking Verification Conditions



```
pred Input(n:number, s:result, i:index) ⇔
  n ≥ 0 ∧ s = 0 ∧ i = 1;
pred Output(n:number, s:result) ⇔
  s = ∑j:number with 1 ≤ j ∧ j ≤ n. j;
pred Invariant(n:number, s:result, i:index) ⇔
  (s = ∑j:number with 1 ≤ j ∧ j ≤ i-1. j) ∧ 1 ≤ i ∧ i ≤ n+1;

theorem A(n:number, s:result, i:index) ⇔
  Input(n, s, i) ⇒ Invariant(n, s, i);
theorem B(n:number, s:result, i:index) ⇔
  Invariant(n, s, i) ∧ i ≤ n ⇒ Invariant(n, s+i, i+1);
theorem C(n:number, s:result, i:index) ⇔
  Invariant(n, s, i) ∧ ¬(i ≤ n) ⇒ Output(n, s);
```

We check for some  $N$  that the verification conditions are valid; this also implies that the invariant is not too weak.



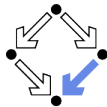
# Another Program Verification

Verification of the following Hoare triple:

$$\{olda = a \wedge oldx = x\}$$
$$i := 0; r := -1; n = |a|$$
$$\mathbf{while} \ i < n \wedge r = -1 \ \mathbf{do}$$
$$\quad \mathbf{if} \ a[i] = x$$
$$\quad \quad \mathbf{then} \ r := i$$
$$\quad \quad \mathbf{else} \ i := i + 1$$
$$\{a = olda \wedge x = oldx \wedge$$
$$\quad ((r = -1 \wedge \forall i : 0 \leq i < |a| \Rightarrow a[i] \neq x) \vee$$
$$\quad (0 \leq r < |a| \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x))\}$$
$$\mathbf{Invariant} : \Leftrightarrow olda = a \wedge oldx = x \wedge n = |a| \wedge$$
$$0 \leq i \leq n \wedge \forall j : 0 \leq j < i \Rightarrow a[j] \neq x \wedge$$
$$(r = -1 \vee (r = i \wedge i < n \wedge a[r] = x))$$

Find the smallest index  $r$  of an occurrence of value  $x$  in array  $a$  ( $r = -1$ , if  $x$  does not occur in  $a$ ).

# RISCAL: Checking Program Execution

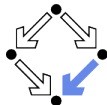


```
val N:N; val M:N;
type index =  $\mathbb{Z}[-1,N]$ ; type elem =  $\mathbb{N}[M]$ ; type array = Array[N,elem];

proc search(a:array, x:elem): index
  ensures (result = -1  $\wedge$   $\forall i:\text{index}. 0 \leq i \wedge i < N \Rightarrow a[i] \neq x$ )  $\vee$ 
    (0  $\leq$  result  $\wedge$  result  $<$  N  $\wedge$ 
      a[result] = x  $\wedge$   $\forall i:\text{index}. 0 \leq i \wedge i <$  result  $\Rightarrow a[i] \neq x$ );
{
  var i:index = 0;
  var r:index = -1;
  while i < N  $\wedge$  r = -1 do
    invariant 0  $\leq$  i  $\wedge$  i  $\leq$  N  $\wedge$   $\forall j:\text{index}. 0 \leq j \wedge j <$  i  $\Rightarrow a[j] \neq x$ ;
    invariant r = -1  $\vee$  (r = i  $\wedge$  i < N  $\wedge$  a[r] = x);
  {
    if a[i] = x
      then r := i;
      else i := i+1;
    }
  }
  return r;
}
```

We check for some  $N, M$  the program execution.

# The Verification Conditions



*Input* : $\Leftrightarrow$   $olda = a \wedge oldx = x \wedge n = length(a) \wedge i = 0 \wedge r = -1$

*Output* : $\Leftrightarrow$   $a = olda \wedge x = oldx \wedge$   
 $((r = -1 \wedge \forall i : 0 \leq i < length(a) \Rightarrow a[i] \neq x) \vee$   
 $(0 \leq r < length(a) \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x))$

*Invariant* : $\Leftrightarrow$   $olda = a \wedge oldx = x \wedge n = |a| \wedge$   
 $0 \leq i \leq n \wedge \forall j : 0 \leq j < i \Rightarrow a[j] \neq x \wedge$   
 $(r = -1 \vee (r = i \wedge i < n \wedge a[r] = x))$

*A* : $\Leftrightarrow$  *Input*  $\Rightarrow$  *Invariant*

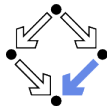
*B*<sub>1</sub> : $\Leftrightarrow$  *Invariant*  $\wedge i < n \wedge r = -1 \wedge a[i] = x \Rightarrow$  *Invariant*[*i*/*r*]

*B*<sub>2</sub> : $\Leftrightarrow$  *Invariant*  $\wedge i < n \wedge r = -1 \wedge a[i] \neq x \Rightarrow$  *Invariant*[*i* + 1/*i*]

*C* : $\Leftrightarrow$  *Invariant*  $\wedge \neg(i < n \wedge r = -1) \Rightarrow$  *Output*

The verification conditions *A*, *B*<sub>1</sub>, *B*<sub>2</sub>, *C* must be valid.

# RISCAL: Checking Verification Conditions



```
pred Input(i:index, r:index)  $\Leftrightarrow$  i = 0  $\wedge$  r = -1;
pred Output(a:array, x:elem, i:index, r:index)  $\Leftrightarrow$ 
  (r = -1  $\wedge$   $\forall$ i:index. 0  $\leq$  i  $\wedge$  i < N  $\Rightarrow$  a[i]  $\neq$  x)  $\vee$ 
  (0  $\leq$  r  $\wedge$  r < N  $\wedge$  a[r] = x  $\wedge$   $\forall$ i:index. 0  $\leq$  i  $\wedge$  i < r  $\Rightarrow$  a[i]  $\neq$  x);
pred Invariant(a:array, x:elem, i:index, r:index)  $\Leftrightarrow$ 
  0  $\leq$  i  $\wedge$  i  $\leq$  N  $\wedge$  ( $\forall$ j:index. 0  $\leq$  j  $\wedge$  j < i  $\Rightarrow$  a[j]  $\neq$  x)  $\wedge$ 
  (r = -1  $\vee$  (r = i  $\wedge$  i < N  $\wedge$  a[r] = x));

theorem A(a:array, x:elem, i:index, r:index)  $\Leftrightarrow$ 
  Input(i, r)  $\Rightarrow$  Invariant(a, x, i, r);
theorem B1(a:array, x:elem, i:index, r:index)  $\Leftrightarrow$ 
  Invariant(a, x, i, r)  $\wedge$  i < N  $\wedge$  r = -1  $\wedge$  a[i] = x  $\Rightarrow$ 
  Invariant(a, x, i, i);
theorem B2(a:array, x:elem, i:index, r:index)  $\Leftrightarrow$ 
  Invariant(a, x, i, r)  $\wedge$  i < N  $\wedge$  r = -1  $\wedge$  a[i]  $\neq$  x  $\Rightarrow$ 
  Invariant(a, x, i+1, r);
theorem C(a:array, x:elem, i:index, r:index)  $\Leftrightarrow$ 
  Invariant(a, x, i, r)  $\wedge$   $\neg$ (i < N  $\wedge$  r = -1)  $\Rightarrow$ 
  Output(a, x, i, r);
```

We check for some  $N, M$  that the verification conditions are valid.



- 
1. The Hoare Calculus
  2. Checking Verification Conditions
  - 3. Predicate Transformers**
  4. Generating Verification Conditions
  5. Termination
  6. Proving Verification Conditions
  7. Abortion
  8. Procedures



# 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.

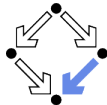
$$\begin{array}{l} \{P\} \\ x := x+1; \\ y := 2*x; \\ z := x+y \\ \{z = 15\} \end{array}$$

$$\begin{array}{l} \{P\} \\ x := x+1; \\ y := 2*x; \\ \{x + y = 15\} \end{array}$$

$$\begin{array}{l} \{P\} \\ x := x+1; \\ \{x + 2x = 15\} \\ (\Leftrightarrow 3x = 15) \\ (\Leftrightarrow x = 5) \end{array}$$

$$\begin{array}{l} \{P\} \\ \{x + 1 = 5\} \\ (\Leftrightarrow x = 4) \end{array}$$

$$P \Rightarrow x = 4$$



# Weakest Preconditions

A calculus for “backward reasoning” (E.W. Dijkstra).

- **Predicate transformer  $wp$**

- Function “ $wp$ ” that takes a command  $c$  and a postcondition  $Q$  and returns a precondition.

- Read  $wp(c, Q)$  as “the weakest precondition of  $c$  w.r.t.  $Q$ ”.

- $wp(c, Q)$  is a **precondition** for  $c$  that ensures  $Q$  as a postcondition.

- Must satisfy  $\{wp(c, Q)\} c \{Q\}$ .

- $wp(c, Q)$  is the **weakest** such precondition.

- Take any  $P$  such that  $\{P\} c \{Q\}$ .

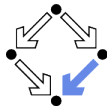
- Then  $P \Rightarrow wp(c, Q)$ .

- Consequence:  $\{P\} c \{Q\}$  iff  $(P \Rightarrow wp(c, Q))$

- We want to prove  $\{P\} c \{Q\}$ .

- We may prove  $P \Rightarrow wp(c, Q)$  instead.

**Verification is reduced to the calculation of weakest preconditions.**



# Weakest Preconditions

---

The weakest precondition of each program construct.

$$\text{wp}(\mathbf{skip}, Q) = Q$$

$$\text{wp}(\mathbf{abort}, Q) = \text{true}$$

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

$$\text{wp}(c_1; c_2, Q) = \text{wp}(c_1, \text{wp}(c_2, Q))$$

$$\text{wp}(\mathbf{if } b \mathbf{ then } c_1 \mathbf{ else } c_2, Q) = (b \Rightarrow \text{wp}(c_1, Q)) \wedge (\neg b \Rightarrow \text{wp}(c_2, Q))$$

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

$$\text{wp}(\mathbf{while } b \mathbf{ do } c, Q) = \dots$$

Loops represent a special problem (see later).

# Forward Reasoning



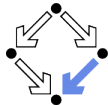
Sometimes, we want to derive a postcondition from a given precondition.

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

## ■ 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  $e[x_0/x]$ .

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

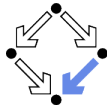


# Strongest Postcondition

A calculus for forward reasoning.

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

**Verification is reduced to the calculation of strongest postconditions.**



# Strongest Postconditions

---

The strongest postcondition of each program construct.

$$\text{sp}(\text{skip}, P) = P$$

$$\text{sp}(\text{abort}, P) = \text{false}$$

$$\text{sp}(x := e, P) = \exists x_0 : P[x_0/x] \wedge x = e[x_0/x]$$

$$\text{sp}(c_1; c_2, P) = \text{sp}(c_2, \text{sp}(c_1, P))$$

$$\text{sp}(\text{if } b \text{ then } c_1 \text{ else } c_2, P) \Leftrightarrow \text{sp}(c_1, P \wedge b) \vee \text{sp}(c_2, P \wedge \neg b)$$

$$\text{sp}(\text{if } b \text{ then } c, P) = \text{sp}(c, P \wedge b) \vee (P \wedge \neg b)$$

$$\text{sp}(\text{while } b \text{ do } c, P) = \dots$$

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

# Hoare Calc. and Predicate Transformers



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

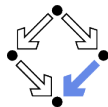
$$\{P\} c_1; \mathbf{while} \ b \ \mathbf{do} \ c; c_2 \ \{Q\}$$

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

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

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?

$$\text{wp}(\mathbf{loop}, Q) = \text{true}$$

$$\text{wp}(\mathbf{while } b \mathbf{ do } c, Q) = L_0(Q) \wedge L_1(Q) \wedge L_2(Q) \wedge \dots$$

$$L_0(Q) = \text{true}$$

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

## ■ 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.

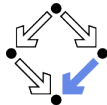
## ■ Alternative view: $L_i(Q) = \text{wp}(\text{if}_i, Q)$

$$\text{if}_0 = \mathbf{loop}$$

$$\text{if}_{i+1} = \mathbf{if } b \mathbf{ then } (c; \text{if}_i)$$



# Example



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

$$L_0(Q) = \text{true}$$

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

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

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

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

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

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

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

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

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

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

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

# 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”:
  - $\forall i \in \mathbb{N} : \text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q) \Rightarrow L_i(Q)$ .
  - $\forall P : (\forall i \in \mathbb{N} : P \Rightarrow L_i(Q)) \Rightarrow (P \Rightarrow \text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q))$ .
- We can only compute weaker **approximation**  $L_i(Q)$ .
  - $\text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q) \Rightarrow L_i(Q)$ .
- We want to prove  $\{P\} \mathbf{while\ } b \ \mathbf{do\ } c \{Q\}$ .
  - This is equivalent to proving  $P \Rightarrow \text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q)$ .
  - Thus  $P \Rightarrow L_i(Q)$  must hold as well.
- If we can prove  $\neg(P \Rightarrow L_i(Q))$ , ...
  - $\{P\} \mathbf{while\ } b \ \mathbf{do\ } c \{Q\}$  does **not** hold.
  - If we fail, we may try the easier proof  $\neg(P \Rightarrow L_{i+1}(Q))$ .

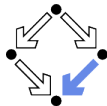
Falsification is possible by use of approximation  $L_i$ , but verification is not.

# Preconditions for Loops with Invariants


$$\text{wp}(\text{while } b \text{ do invariant } I; c^{x, \dots}, Q) =$$
$$\text{let } oldx = x, \dots \text{ in}$$
$$I \wedge (\forall x, \dots : I \wedge b \Rightarrow \text{wp}(c, I)) \wedge$$
$$(\forall x, \dots : I \wedge \neg b \Rightarrow Q)$$

- Loop body  $c$  only modifies variables  $x, \dots$ .
- Loop is annotated with invariant  $I$ .
  - May refer to new values  $x, \dots$  of variables after every iteration.
  - May refer to original values  $oldx, \dots$  when loop started execution.
- Generated verification condition ensures:
  1.  $I$  holds in the initial state of the loop.
  2.  $I$  is preserved by the execution of the loop body  $c$ .
  3. When the loop terminates,  $I$  ensures postcondition  $Q$ .

This precondition is only “weakest” relative to the invariant.



# Example

**while**  $i \leq n$  **do** ( $s := s + i; i := i + 1$ )

$c^{s,i} := (s := s + i; i := i + 1)$

$I := s = olds + \left( \sum_{j=oldi}^{i-1} j \right) \wedge oldi \leq i \leq n + 1$

■ **Weakest precondition:**

$wp(\mathbf{while} \ i \leq n \ \mathbf{do} \ \mathbf{invariant} \ I; \ c^{s,i}, Q) =$

**let**  $olds = s, oldi = i$  **in**

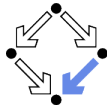
$I \wedge (\forall s, i : I \wedge i \leq n \Rightarrow I[i + 1/i][s + i/s]) \wedge$

$(\forall s, i : I \wedge \neg(i \leq n) \Rightarrow Q)$

■ **Verification condition:**

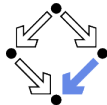
$n \geq 0 \wedge i = 1 \wedge s = 0 \Rightarrow wp(\dots, s = \sum_{j=1}^n j)$

Many verification systems implement (a variant of) this calculus.



- 
1. The Hoare Calculus
  2. Checking Verification Conditions
  3. Predicate Transformers
  - 4. Generating Verification Conditions**
  5. Termination
  6. Proving Verification Conditions
  7. Abortion
  8. Procedures

# RISCAL and Verification Conditions



RISC Algorithm Language (RISCAL)

File Edit Help

File: summation.txt

```
1 // summation: return the sum of all values from 1 to n
2
3 val N: Nat;
4 type number = %[N];
5 type index = %[N+1];
6 type result = %[N*(1+N)/2];
7
8 proc summation(n: number): result
9   requires n >= 0;
10  ensures result =  $\sum_{j: \text{number with } 1 \leq j \wedge j \leq n. j}$ ;
11 {
12   var s: result = 0;
13   var i: index = 1;
14   while i <= n do
15     invariant s =  $\sum_{j: \text{number with } 1 \leq j \wedge j \leq i-1. j}$ ;
16     invariant 1 <= i & i <= n+1;
17     decreases n+1-i;
18     {
19       s = s+i;
20       i = i+1;
21     }
22   }
23   return s;
24 }
25 // the verification conditions to be proved
26 // for the total correctness of the program
27
28 pred Input(n: number, s: result, i: index) =
29   n >= 0 & s = 0 & i = 1;
30 pred Output(n: number, s: result) =
31   s =  $\sum_{j: \text{number with } 1 \leq j \wedge j \leq n. j}$ ;
32
33 pred Invariant(n: number, s: result, i: index) =
34   ( $s = \sum_{j: \text{number with } 1 \leq j \wedge j \leq i-1. j}$ ) & 1 <= i & i <= n+1;
35
36 fun Termination(n: number, s: result, i: index): number =
37   n+1-i;
38
39 theorem A(n: number, s: result, i: index) =
40   Input(n, s, i) = Invariant(n, s, i);
41
42 theorem T(n: number, s: result, i: index) =
43   Invariant(n, s, i) = Termination(n, s, i) >= 0;
44
```

Analysis

Translation:  Nondeterminism Default Value: 0 Other Values: [ ]

Execution:  Silent Inputs: [ ] Per Mile: [ ] Branches: [ ]

Visualization:  Trace  Tree Width: 800 Height: 600

Parallelism:  Multi-Threaded Threads: 4  Distributed Servers: [ ]

Operation: [ summation(Z) ]

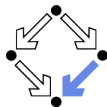
Tasks

- summation(Z)
  - Execute operation
  - Validate specification
    - Execute specification
    - Is precondition satisfiable?
    - Is precondition not trivial?
    - Is postcondition always satisfiable?
    - Is postcondition always not trivial?
    - Is postcondition sometimes not trivial?
    - Is result uniquely determined?
  - Verify specification preconditions
  - Verify correctness of result
    - Is result correct?
  - Verify iteration and recursion
    - Does loop invariant initially hold?
    - Does loop invariant initially hold?
    - Is loop measure non-negative?
    - Is loop invariant preserved?
    - Is loop invariant preserved?
    - Is loop measure decreased?
  - Verify implementation preconditions
    - Is assigned value legal?
    - Is assigned value legal?

Executing summation\_0\_CorrOp0(Z) with all 5 inputs.  
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_LoopOp0(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_LoopOp1(Z) with all 5 inputs.  
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_LoopOp2(Z) with all 5 inputs.  
Execution completed for ALL inputs (4 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_LoopOp3(Z) with all 5 inputs.  
Execution completed for ALL inputs (6 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_LoopOp4(Z) with all 5 inputs.  
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_LoopOp5(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_PredOp0(Z) with all 5 inputs.  
Execution completed for ALL inputs (6 ms, 5 checked, 0 inadmissible).  
Executing summation\_0\_PredOp1(Z) with all 5 inputs.  
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).

RISCAL implements Dijkstra's calculus for VC generation.

# RISCAL Verification Conditions

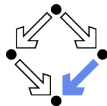


RISCAL splits Dijkstra's single condition  $Input \Rightarrow wp(C, Output)$  into many "fine-grained" verification conditions:

- **Is result correct?**
  - One condition for every ensures clause.
- **Does loop invariant initially hold? Is loop invariant preserved?**
  - Partial correctness.
  - One condition for every invariant clause.
- **Is loop measure non-negative? Is loop measure decreased?**
  - Termination (later).
  - One condition for every decreases clause.
- **Specification and implementation preconditions**
  - Well-definedness of formulas and commands (later).
  - One condition for every partial function/predicate application.

Click on a condition to see the affected commands; if the procedure contains conditionals, a condition is generated for each execution branch.

# Checking Verification Conditions



- **Double-click** a condition to have it checked.
  - Checked conditions turn from red to blue.
- **Right-click** a condition to see a pop-up menu.
  - Check verification condition (same as double-click)
  - Show variable values that invalidate condition.
  - Print relevant program information (e.g. invariant).
  - Print verification condition itself.
  - Apply SMT solver for faster checking.

- ➡ Execute Task
- Show Counterexample
- Print Description
- Print Definition
- Apply SMT Solver

**Example:** is loop invariant preserved?

```
s = ( $\sum$ j:number with (1 ≤ j) ∧ (j ≤ (i-1)). j)
```

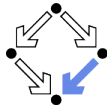
```
theorem _summation_0_LoopOp3(n:number)
```

```
requires n ≥ 0;
```

```
⇔  $\forall$ s:result,i:index. (((s = ( $\sum$ j:number with (1 ≤ j) ∧ (j ≤ (i-1)). j))  
  ∧ ((1 ≤ i) ∧ (i ≤ (n+1)))) ∧ (i ≤ n)) ⇒  
  (let s = s+i in (let i = i+1 in  
    (s = ( $\sum$ j:number with (1 ≤ j) ∧ (j ≤ (i-1)). j)))));
```

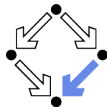
**Important:** check models with small type sizes.





- 
1. The Hoare Calculus
  2. Checking Verification Conditions
  3. Predicate Transformers
  4. Generating Verification Conditions
  - 5. Termination**
  6. Proving Verification Conditions
  7. Abortion
  8. Procedures

# Termination



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

$$\frac{\{\text{false}\} \text{loop} \{\text{false}\} \quad I \Rightarrow t \geq 0 \quad \{I \wedge b \wedge t = N\} c \quad \{I \wedge t < N\}}{\{I\} \text{while } b \text{ do } c \quad \{I \wedge \neg b\}}$$

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

- 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 measure  $t$  (term type-checked to denote an integer)**.
  - Becomes smaller by every iteration of the loop.
  - But does not become negative.
  - Consequently, the loop must eventually terminate.

The initial value of  $t$  limits the number of loop iterations.

*Any well-founded ordering may be used as the domain of  $t$ .*

# Example



$$l : \Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$$
$$t := n - i + 1$$

$$(n \geq 0 \wedge i = 1 \wedge s = 0) \Rightarrow l \quad l \Rightarrow n - i + 1 \geq 0$$

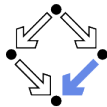
$$\{l \wedge i \leq n \wedge n - i + 1 = N\} s := s + i; i := i + 1 \{l \wedge n - i + 1 < N\}$$

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

---

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

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



# Weakest Preconditions for Loops

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

$\text{wp}(\text{while } b \text{ do } c, Q) = L_0(Q) \vee L_1(Q) \vee L_2(Q) \vee \dots$

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

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

## ■ 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



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

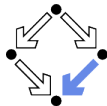
$$L_0(Q) = \text{false}$$

$$\begin{aligned} L_1(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_0(Q))) \\ &\Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{false}) \\ &\Leftrightarrow i \not< n \wedge Q \end{aligned}$$

$$\begin{aligned} L_2(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_1(Q))) \\ &\Leftrightarrow (i \not< n \Rightarrow Q) \wedge \\ &\quad i < n \Rightarrow (i + 1 \not< n \wedge Q[i + 1/i]) \end{aligned}$$

$$\begin{aligned} L_3(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_2(Q))) \\ &\Leftrightarrow (i \not< n \Rightarrow Q) \wedge \\ &\quad (i < n \Rightarrow ((i + 1 \not< n \Rightarrow Q[i + 1/i]) \wedge \\ &\quad (i + 1 < n \Rightarrow (i + 2 \not< n \wedge Q[i + 2/i]))) \end{aligned}$$

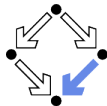
...



# 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”:
  - $\forall i \in \mathbb{N} : L_i(Q) \Rightarrow \text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q)$ .
  - $\forall P : (\forall i \in \mathbb{N} : L_i(Q) \Rightarrow P) \Rightarrow (\text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q) \Rightarrow P)$ .
- We can only compute a stronger approximation  $L_i(Q)$ .
  - $L_i(Q) \Rightarrow \text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q)$ .
- We want to prove  $\{P\} c \{Q\}$ .
  - It suffices to prove  $P \Rightarrow \text{wp}(\mathbf{while\ } b \ \mathbf{do\ } c, Q)$ .
  - It thus also suffices to prove  $P \Rightarrow L_i(Q)$ .
  - If proof fails, we may try the easier proof  $P \Rightarrow L_{i+1}(Q)$

However, verifications are typically not successful with any finite approximation of the weakest precondition.



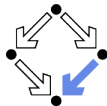
# Weakest Precondition with Measures

$$\begin{aligned} \text{wp}(\text{while } b \text{ do invariant } I; \text{decreases } t; c^{x, \dots}, Q) = \\ \text{let } \text{old}x = x, \dots \text{ in} \\ I \wedge (\forall x, \dots : I \wedge b \Rightarrow \text{wp}(C, I)) \wedge \\ (\forall x, \dots : I \wedge \neg b \Rightarrow Q) \wedge \\ (\forall x, \dots : I \Rightarrow t \geq 0) \wedge \\ (\forall x, \dots : I \wedge b \Rightarrow \text{let } T = t \text{ in } \text{wp}(c, t < T)) \end{aligned}$$

- Loop body  $c$  only modifies variables  $x, \dots$ .
- Loop is annotated with termination measure (term)  $t$ .
  - May refer to new values  $x, \dots$  of variables after every iteration.
- Generated verification condition ensures:
  1.  $t$  is non-negative before/after every loop iteration.
  2.  $t$  is decremented by the execution of the loop body  $c$ .

Also here any well-founded ordering may be used as the domain of  $t$ .

# Example



**while**  $i \leq n$  **do** ( $s := s + i; i := i + 1$ )

$c^{s,i} := (s := s + i; i := i + 1)$

$I := s = olds + \left( \sum_{j=oldi}^{i-1} j \right) \wedge oldi \leq i \leq n + 1$

$t := n + 1 - i$

■ **Weakest precondition:**

$wp(\mathbf{while} \ i \leq n \ \mathbf{do} \ \mathbf{invariant} \ I; \ c^{s,i}, Q) =$

**let**  $olds = s, oldi = i$  **in**

$I \wedge (\forall s, i : I \wedge i \leq n \Rightarrow I[s + i/s, i + 1/i]) \wedge$

$(\forall s, i : I \wedge \neg(i \leq n) \Rightarrow Q) \wedge$

$(\forall s, i : I \Rightarrow t \geq 0) \wedge$

$(\forall s, i : I \wedge i \leq n \Rightarrow \mathbf{let} \ T = n + 1 - i \ \mathbf{in} \ n + 1 - (i + 1) < T)$

■ **Verification condition:**

$n \geq 0 \wedge i = 1 \wedge s = 0 \Rightarrow wp(\dots, s = \sum_{j=1}^n j)$



# Termination in RISCAL

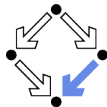


```
while i ≤ n do
  invariant s = ∑j:number with 1 ≤ j ∧ j ≤ i-1. j;
  invariant 1 ≤ i ∧ i ≤ n+1;
  decreases n+1-i;
{
  s := s+i;
  i := i+1;
}

fun Termination(n:number, s:result, i:index): number =
  n+1-i;
theorem T(n:number, s:result, i:index) ⇔
  Invariant(n, s, i) ⇒ Termination(n, s, i) ≥ 0;
theorem B(n:number, s:result, i:index) ⇔
  Invariant(n, s, i) ∧ i ≤ n ⇒
  Invariant(n, s+i, i+1) ∧
  Termination(n, s+i, i+1) < Termination(n, s, i);
```

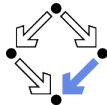
Termination conditions manually constructed or automatically derived.

# Termination in RISCAL



```
while i < N ∧ r = -1 do
  invariant 0 ≤ i ∧ i ≤ N;
  invariant ∀j:index. 0 ≤ j ∧ j < i ⇒ a[j] ≠ x;
  invariant r = -1 ∨ (r = i ∧ i < N ∧ a[r] = x);
  decreases if r = -1 then N-i else 0;
{
  if a[i] = x
    then r := i;
    else i := i+1;
}
```

```
fun Termination(a:array, x:elem, i:index, r:index): index =
  if r = -1 then N-i else 0;
theorem T(a:array, x:elem, i:index, r:index) ⇔
  Invariant(a, x, i, r) ⇒ Termination(a, x, i, r) ≥ 0;
theorem B1(a:array, x:elem, i:index, r:index) ⇔
  Invariant(a, x, i, r) ∧ i < N ∧ r = -1 ∧ a[i] = x ⇒
    Invariant(a, x, i, i) ∧
    Termination(a, x, i, i) < Termination(a, x, i, r);
theorem B2(a:array, x:elem, i:index, r:index) ⇔ ...
```



- 
1. The Hoare Calculus
  2. Checking Verification Conditions
  3. Predicate Transformers
  4. Generating Verification Conditions
  5. Termination
  - 6. Proving Verification Conditions**
  7. Abortion
  8. Procedures

# RISC ProofNavigator: A Theory of Arrays



```
% constructive array definition
newcontext "arrays2";

% the types
INDEX: TYPE = NAT;
ELEM:  TYPE;
ARR:   TYPE =
  [INDEX, ARRAY INDEX OF ELEM];

% error constants
any:    ARRAY INDEX OF ELEM;
anyelem: ELEM;
anyarray: ARR;

% a selector operation
content:
  ARR -> (ARRAY INDEX OF ELEM) =
    LAMBDA(a:ARR): a.1;

% the array operations
length: ARR -> INDEX =
  LAMBDA(a:ARR): a.0;
new: INDEX -> ARR =
  LAMBDA(n:INDEX): (n, any);
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;
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

FORALL(n:INDEX): length(new(n)) = n;

length2: FORMULA

FORALL(a:ARR, i:INDEX, e:ELEM):

i < length(a) => length(put(a, i, e)) = length(a);

get1: FORMULA

FORALL(a:ARR, i:INDEX, e:ELEM):

i < length(a) => get(put(a, i, e), i) = e;

get2: FORMULA

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);

▽ [adu]: expand length, get, put, content

▽ [c3b]: scatter

[qid]: proved (CVCL)

# The Verification Conditions



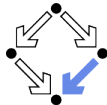
```
newcontext      Input: BOOLEAN = olda = a AND oldx = x AND
  "linsearch";      n = length(a) AND i = 0 AND r = -1;

% declaration      Output: BOOLEAN = a = olda AND
% of arrays        ((r = -1 AND
...                (FORALL(j:NAT): j < length(a) =>
                    get(a,j) /= x)) OR
a: ARR;           (0 <= r AND r < length(a) AND get(a,r) = x AND
olda: ARR;        (FORALL(j:NAT):
x: ELEM;          j < r => get(a,j) /= x)));
oldx: ELEM;

i: NAT;           Invariant: (ARR, ELEM, NAT, NAT, INT) -> BOOLEAN =
n: NAT;           LAMBDA(a: ARR, x: ELEM, i: NAT, n: NAT, r: INT):
r: INT;           olda = a AND oldx = x AND
                  n = length(a) AND i <= n AND
                  (FORALL(j:NAT): j < i => get(a,j) /= x) AND
                  (r = -1 OR (r = i AND i < n AND get(a,r) = x));
...

```

# 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  $\text{get}(a,i) = x$   
=> Invariant(a,x,i,n,i);

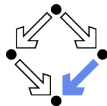
B2: FORMULA

Invariant(a, x, i, n, r) AND  $i < n$  AND  $r = -1$  AND  $\text{get}(a,i) \neq x$   
=> 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: [bca]: expand Input, Invariant  
[fuo]: scatter  
[bxg]: proved (CVCL)

(2 user actions)

B1: [p1b]: expand Invariant  
[lf6]: proved (CVCL)

(1 user action)

B2: [q1b]: expand Invariant in 6kv  
[slx]: scatter  
[a1y]: auto  
[cch]: proved (CVCL)  
[b1y]: proved (CVCL)  
[c1y]: proved (CVCL)  
[d1y]: proved (CVCL)  
[e1y]: proved (CVCL)

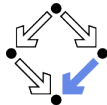
(3 user actions)

C: [dca]: expand Invariant, Output in zfg  
[tvy]: scatter  
[dcu]: auto  
[t4c]: proved (CVCL)  
[ecu]: split pkg  
[kel]: proved (CVCL)  
[lel]: scatter  
[lvn]: auto  
[lap]: proved (CVCL)  
[fcu]: auto  
[blt]: proved (CVCL)  
[gcu]: proved (CVCL)

(6 user actions)



# Termination



```
Termination: (ARR, ELEM, NAT, NAT, INT) -> INT =  
  LAMBDA(a: ARR, x: ELEM, i: NAT, n: NAT, r: INT):  
    IF r=-1 THEN n-i ELSE 0 ENDIF;
```

T: FORMULA

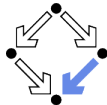
```
Invariant(a, x, i, n, r) => Termination(a, x, i, n, r) >= 0;
```

B1: FORMULA

```
Invariant(a, x, i, n, r) AND i < n AND r = -1 AND get(a,i) = x AND  
Termination(a, x, i, n, r) = N  
=> Invariant(a,x,i,n,i) AND Termination(a,x,i,n,i) < N;
```

B2: FORMULA

```
Invariant(a, x, i, n, r) AND i < n AND r = -1 AND get(a,i) /= x AND  
Termination(a, x, i, n, r) = N  
=> Invariant(a,x,i+1,n,r) AND Termination(a,x,i+1,n,r) < N;
```



- 
1. The Hoare Calculus
  2. Checking Verification Conditions
  3. Predicate Transformers
  4. Generating Verification Conditions
  5. Termination
  6. Proving Verification Conditions
  - 7. Abortion**
  8. Procedures

# Abortion



New rules to prevent abortion.

$$\begin{aligned} & \{\text{false}\} \mathbf{abort} \{\text{true}\} \\ & \{Q[e/x] \wedge D(e)\} x := e \{Q\} \\ & \{Q[a[i \mapsto e]/a] \wedge D(e) \wedge D(i) \wedge 0 \leq i < \text{length}(a)\} a[i] := e \{Q\} \end{aligned}$$

- 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



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

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

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

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

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

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

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

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

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

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

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

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

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

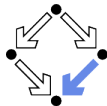
$$D(e_1 \leq e_2) = D(e_1) \wedge D(e_2).$$

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

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

Assumes that expressions have already been type-checked.

# Abortion



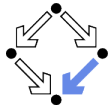
Slight modification of existing rules.

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

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

$$\frac{I \Rightarrow (t \geq 0 \wedge D(b)) \quad \{I \wedge b \wedge t = N\} c \{I \wedge t < N\}}{\{I\} \text{ while } b \text{ do } c \{I \wedge \neg b\}}$$

Expressions must be defined in any context.



# Abortion

Similar modifications of weakest preconditions.

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

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

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

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

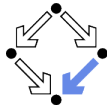
$$\text{wp}(\mathbf{if } b \mathbf{ then } c, Q) = D(b) \wedge (b \Rightarrow \text{wp}(c, Q)) \wedge (\neg b \Rightarrow Q)$$

$$\text{wp}(\mathbf{while } b \mathbf{ do } c, Q) = (L_0(Q) \vee L_1(Q) \vee L_2(Q) \vee \dots)$$

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

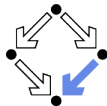
$$L_{i+1}(Q) = D(b) \wedge (\neg b \Rightarrow Q) \wedge (b \Rightarrow \text{wp}(c, L_i(Q)))$$

$\text{wp}(c, Q)$  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
  2. Checking Verification Conditions
  3. Predicate Transformers
  4. Generating Verification Conditions
  5. Termination
  6. Proving Verification Conditions
  7. Abortion
  - 8. Procedures**

# Procedure Specifications



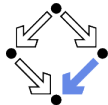
global  $g$ ;  
requires  $Pre$ ;  
ensures  $Post$ ;  
 $o := p(i) \{ c \}$

- 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

$$\{ Pre \wedge i_0 = i \wedge g_0 = g \} c \{ Post[i_0/i] \}$$

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





# Example

## ■ Procedure specification:

global  $g$

requires  $g \geq 0 \wedge i > 0$

ensures  $g_0 = g \cdot i + o \wedge 0 \leq o < i$

$o := p(i) \{ o := g \% i; g := g / i \}$

## ■ Proof obligation:

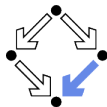
$\{g \geq 0 \wedge i > 0 \wedge i_0 = i \wedge g_0 = g\}$

$o := g \% i; g := g / i$

$\{g_0 = g \cdot i_0 + o \wedge 0 \leq o < i_0\}$

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:

$$\begin{array}{c} \{D(e) \wedge Q[e/x]\} \\ x := e \\ \{Q\} \end{array}$$

- Alternative:

$$\begin{array}{c} \{D(e) \wedge \forall x' : x' = e \Rightarrow Q[x'/x]\} \\ x := e \\ \{Q\} \end{array}$$

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

# Procedure Calls



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

$$\begin{array}{c} \{D(e) \wedge Pre[e/i] \wedge \\ \forall x', g' : Post[e/i, x'/o, g/g_0, g'/g] \Rightarrow Q[x'/x, g'/g]\} \\ x := p(e) \\ \{Q\} \end{array}$$

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

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

# Corresponding Predicate Transformers

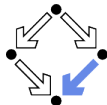


$$\begin{aligned} \text{wp}(x = p(e), Q) = & \\ & D(e) \wedge \text{Pre}[e/i] \wedge \\ & \forall x', g' : \\ & \text{Post}[e/i, x'/o, g/g_0, g'/g] \Rightarrow Q[x'/x, g'/g] \end{aligned}$$

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

Explicit naming of old/new values required.

# Example



## ■ Procedure specification:

global  $g$

requires  $g \geq 0 \wedge i > 0$

ensures  $g_0 = g \cdot i + o \wedge 0 \leq o < i$

$o = p(i) \{ o := g \% i; g := g / i \}$

## ■ Procedure call:

$\{g \geq 0 \wedge g = N \wedge b \geq 0\}$

$x = p(b + 1)$

$\{g \cdot (b + 1) \leq N < (g + 1) \cdot (b + 1)\}$

## ■ To be proved:

$g \geq 0 \wedge g = N \wedge b \geq 0 \Rightarrow$

$D(b + 1) \wedge g \geq 0 \wedge b + 1 > 0 \wedge$

$\forall x', g' :$

$g = g' \cdot (b + 1) + x' \wedge 0 \leq x' < b + 1 \Rightarrow$

$g' \cdot (b + 1) \leq N < (g' + 1) \cdot (b + 1)$