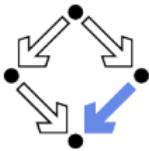
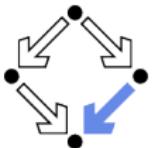


# Specifying and Verifying Programs

Wolfgang Schreiner  
[Wolfgang.Schreiner@risc.jku.at](mailto:Wolfgang.Schreiner@risc.jku.at)

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





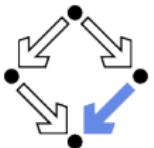
# Specifying and Verifying Programs

---

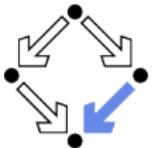
We will discuss two (closely interrelated) calculi.

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

The Hoare calculus can be easily applied in manual verifications; for automation, the predicate transformers calculus is more suitable (both calculi can be also combined).



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



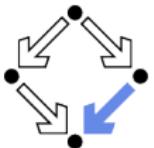
# The Hoare Calculus

---

First/best-known calculus for program reasoning (C. A. R. Hoare, 1969).

- “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

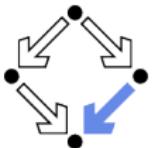
---

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.



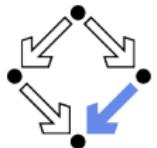
# Weakening and Strengthening

---

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

- 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.



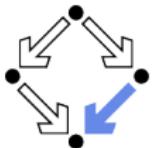
# Special Commands

---

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

- 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  $\{P\} \text{ abort } \{Q\}$  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  $e$ .

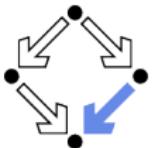
## ■ Interpretation

- 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.

$$\begin{array}{lll} \{x + 3 < 5\} & x := x + 3 & \{x < 5\} \\ \{x < 2\} & x := x + 3 & \{x < 5\} \end{array}$$



# Array Assignments

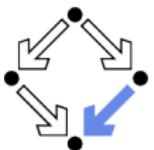
---

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

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

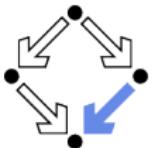
$$\begin{aligned} Q[\underline{a[i \mapsto e]}[j]] \\ \rightsquigarrow \\ (i = j \Rightarrow Q[e]) \wedge (i \neq j \Rightarrow Q[a[j]]) \end{aligned}$$

## ■ Array Axioms

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

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

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



# Command Sequences

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

## ■ Interpretation

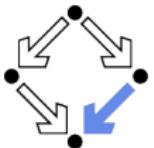
- To show that, if  $P$  holds before the execution of  $c_1; c_2$ , then  $Q$  holds afterwards, it suffices to show for some  $R$  that
  - if  $P$  holds before  $c_1$ , that  $R$  holds afterwards, and that
  - if  $R$  holds before  $c_2$ , then  $Q$  holds afterwards.

## ■ Problem: find suitable $R$ .

- Easy in many cases (see later).

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

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



# Conditionals

---

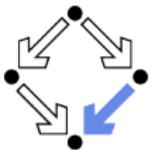
$$\frac{\{P \wedge b\} \ c_1 \ \{Q\} \quad \{P \wedge \neg b\} \ c_2 \ \{Q\}}{\{P\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \ \{Q\}}$$

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

## ■ 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.

$$\frac{\{x \neq 0 \wedge x \geq 0\} \ y := x \ \{y > 0\} \quad \{x \neq 0 \wedge x \not\geq 0\} \ y := -x \ \{y > 0\}}{\{x \neq 0\} \text{ if } x \geq 0 \text{ then } y := x \text{ else } y := -x \ \{y > 0\}}$$



# Loops

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

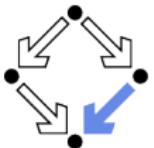
## ■ Interpretation:

- 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)

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

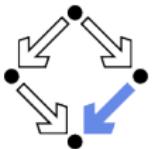
## ■ Interpretation:

- 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

---

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

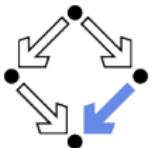
$$(n \geq 0 \wedge s = 0 \wedge i = 1) \Rightarrow I$$

$$\begin{aligned} & \{I \wedge i \leq n\} \quad s := s + i; i := i + 1 \quad \{I\} \\ & (I \wedge i \not\leq n) \Rightarrow s = \sum_{j=1}^n j \end{aligned}$$

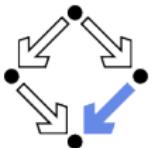
---


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

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



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



# A Program Verification

---

- Verification of the following Hoare triple:

$$\{Input\} \text{ while } i \leq n \text{ do } (s := s + i; i := i + 1) \{Output\}$$

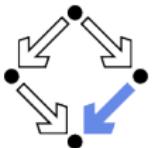
- Auxiliary predicates:

$$Input : \Leftrightarrow n \geq 0 \wedge s = 0 \wedge i = 1$$
$$Output : \Leftrightarrow s = \sum_{j=1}^n j$$
$$Invariant : \Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n + 1$$

- 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.

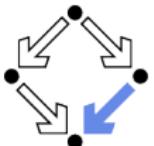


# RISCAL: Checking Program Execution

```
val N:Nat; type number = N[N]; type index = N[N+1]; type result = N[N·(1+N)/2];

proc summation(n:number): result
  requires n ≥ 0;
  ensures result = ∑j:number with 1 ≤ j ∧ j ≤ n. j;
{
  var s:result := 0;
  var i:index := 1;
  while i ≤ n do
    invariant s = ∑j:number with 1 ≤ j ∧ j ≤ i-1. j;
    invariant 1 ≤ i ∧ i ≤ n+1;
  {
    s := s+i;
    i := i+1;
  }
  return s;
}
```

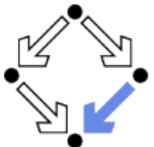
We check for some  $N$  the program execution; this implies that the invariant is not too strong.



# 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 =  $\sum_{j \text{ number}} 1 \leq j \wedge j \leq n. j$ ;  
pred Invariant(n:number, s:result, i:index) ⇔  
  (s =  $\sum_{j \text{ number}} 1 \leq j \wedge j \leq i-1. j$ )  $\wedge$  1 ≤ i  $\wedge$  i ≤ n+1;  
  
theorem A(n:number, s:result, i:index) ⇔  
  Input(n, s, i)  $\Rightarrow$  Invariant(n, s, i);  
theorem B(n:number, s:result, i:index) ⇔  
  Invariant(n, s, i)  $\wedge$  i ≤ n  $\Rightarrow$  Invariant(n, s+i, i+1);  
theorem C(n:number, s:result, i:index) ⇔  
  Invariant(n, s, i)  $\wedge$   $\neg(i \leq n)$   $\Rightarrow$  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:

$$\{olda = a \wedge oldx = x\}$$

$$i := 0; r := -1; n = |a|$$

**while**  $i < n \wedge r = -1$  **do**

**if**  $a[i] = x$

**then**  $r := i$

**else**  $i := i + 1$

$$\{a = olda \wedge x = oldx \wedge$$

$$((r = -1 \wedge \forall i : 0 \leq i < |a| \Rightarrow a[i] \neq x) \vee$$

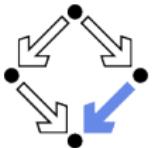
$$(0 \leq r < |a| \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x))\}$$

$$Invariant : \Leftrightarrow olda = a \wedge oldx = x \wedge n = |a| \wedge$$

$$0 \leq i \leq n \wedge \forall j : 0 \leq j < i \Rightarrow a[j] \neq x \wedge$$

$$(r = -1 \vee (r = i \wedge i < n \wedge a[r] = x))$$

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

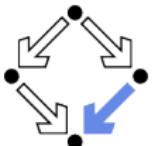


# RISCAL: Checking Program Execution

```
val N:N; val M:N;
type index = Z[-1,N]; type elem = N[M]; type array = Array[N,elem];

proc search(a:array, x:elem): index
    ensures (result = -1  $\wedge$   $\forall i:index. 0 \leq i \wedge i < N \Rightarrow a[i] \neq x$ )  $\vee$ 
        ( $0 \leq result \wedge result < N \wedge$ 
          $a[result] = x \wedge \forall i:index. 0 \leq i \wedge i < result \Rightarrow a[i] \neq x$ );
{
    var i:index = 0;
    var r:index = -1;
    while i < N  $\wedge$  r = -1 do
        invariant  $0 \leq i \wedge i \leq N \wedge \forall j:index. 0 \leq j \wedge j < i \Rightarrow a[j] \neq x$ ;
        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* : $\Leftrightarrow olda = a \wedge oldx = x \wedge n = |a| \wedge$   
 $0 \leq i \leq n \wedge \forall j : 0 \leq j < i \Rightarrow a[j] \neq x \wedge$   
 $(r = -1 \vee (r = i \wedge i < n \wedge a[r] = x))$

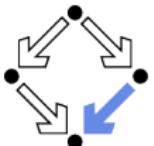
*A* : $\Leftrightarrow Input \Rightarrow Invariant$

*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.

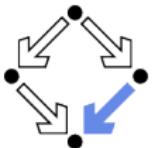


# RISCAL: Checking Verification Conditions

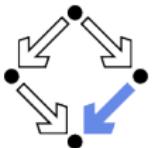
```
pred Input(i:index, r:index) ⇔ i = 0 ∧ r = -1;
pred Output(a:array, x:elem, i:index, r:index) ⇔
  (r = -1 ∧ ∀i:index. 0 ≤ i ∧ i < N ⇒ a[i] ≠ x) ∨
  (0 ≤ r ∧ r < N ∧ a[r] = x ∧ ∀i:index. 0 ≤ i ∧ i < r ⇒ a[i] ≠ x);
pred Invariant(a:array, x:elem, i:index, r:index) ⇔
  0 ≤ i ∧ i ≤ N ∧ (∀j:index. 0 ≤ j ∧ j < i ⇒ a[j] ≠ x) ∧
  (r = -1 ∨ (r = i ∧ i < N ∧ a[r] = x));

theorem A(a:array, x:elem, i:index, r:index) ⇔
  Input(i, r) ⇒ Invariant(a, x, i, r);
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);
theorem B2(a:array, x:elem, i:index, r:index) ⇔
  Invariant(a, x, i, r) ∧ i < N ∧ r = -1 ∧ a[i] ≠ x ⇒
  Invariant(a, x, i+1, r);
theorem C(a:array, x:elem, i:index, r:index) ⇔
  Invariant(a, x, i, r) ∧ ¬(i < N ∧ r = -1) ⇒
  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. Termination
  5. Abortion
  6. Generating Verification Conditions
  7. Proving Verification Conditions
  8. Procedures



# Backward Reasoning

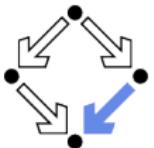
Implication of rule for command sequences and rule for assignments:

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

## ■ 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.

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



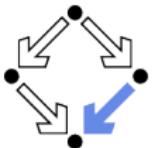
# Weakest Preconditions

---

A calculus for “backward reasoning” (E.W. Dijkstra, 1975).

- **Predicate transformer  $\text{wp}$** 
  - Function “ $\text{wp}$ ” 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  $\{\text{wp}(c, Q)\} \subset \{Q\}$ .
- $\text{wp}(c, Q)$  is the **weakest** such precondition.
  - Take any  $P$  such that  $\{P\} \subset \{Q\}$ .
  - Then  $P \Rightarrow \text{wp}(c, Q)$ .
- Consequence:  $\{P\} \subset \{Q\}$  iff  $(P \Rightarrow \text{wp}(c, Q))$ 
  - We want to prove  $\{P\} \subset \{Q\}$ .
  - 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.

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

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

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

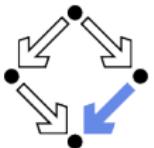
$$\text{wp}(c_1; c_2, Q) = \text{wp}(c_1, \text{wp}(c_2, Q))$$

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

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

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

Loops represent a special problem (see later).



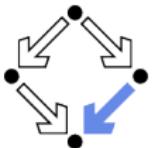
# Example

---

$$\begin{aligned}
 WP &= \text{wp}(\mathbf{if } a[i] < x \mathbf{then } \{a[i] := a[i-1]; i := i-1\}, a[i+1] = b) \\
 &= (a[i] < x \Rightarrow WP_1) \wedge (\neg(a[i] < x) \Rightarrow a[i+1] = b) \\
 &\equiv (a[i] < x \Rightarrow WP_1) \wedge (a[i] \geq x \Rightarrow a[i+1] = b)
 \end{aligned}$$

$$\begin{aligned}
 WP_1 &= \text{wp}(\{a[i] := a[i-1]; i := i-1\}, a[i+1] = b) \\
 &= \text{wp}(a[i] := a[i-1], a[(i-1)+1] = b) \\
 &\equiv \text{wp}(a[i] := a[i-1], a[i] = b) \\
 &= \text{wp}(a := a[i \mapsto a[i-1]], a[i] = b) \\
 &= a[i \mapsto a[i-1]][i] = b \\
 &\equiv (i = i \Rightarrow a[i-1] = b) \wedge (i \neq i \Rightarrow a[i] = b) \\
 &\equiv a[i-1] = b
 \end{aligned}$$

$$WP \equiv \underline{(a[i] < x \Rightarrow a[i-1] = b)} \wedge \underline{(a[i] \geq x \Rightarrow a[i+1] = b)}$$



# Forward Reasoning

---

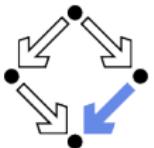
Sometimes, we want to derive a postcondition from a given precondition.

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

## ■ 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  $e[x_0/x]$ .

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



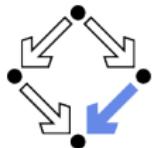
# Strongest Postcondition

---

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.

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

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

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

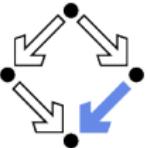
$$\text{sp}(c_1; c_2, P) = \text{sp}(c_2, \text{sp}(c_1, P))$$

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

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

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

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



# Example

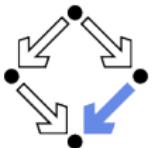
$$\begin{aligned}
 SP &= \text{sp}(\text{if } a[i] < x \text{ then } \{a[i] := a[i-1]; i := i-1\}, a[i] = b) \\
 &= SP_1 \vee (a[i] = b \wedge \neg(a[i] < x)) \equiv SP_1 \vee (a[i] = b \wedge a[i] \geq x) \\
 &\equiv SP_1 \vee (b \geq x \wedge a[i] = b)
 \end{aligned}$$

$$\begin{aligned}
 SP_1 &= \text{sp}(\{a[i] := a[i-1]; i := i-1\}, a[i] = b \wedge a[i] < x) \\
 &\equiv \text{sp}(\{a[i] := a[i-1]; i := i-1\}, a[i] = b \wedge b < x) \\
 &= \text{sp}(i:=i-1, SP_2)
 \end{aligned}$$

$$\begin{aligned}
 SP_2 &= \text{sp}(a[i]:=a[i-1], a[i] = b \wedge b < x) \\
 &= \text{sp}(a:=a[i \mapsto a[i-1]], a[i] = b \wedge b < x) \\
 &= \exists a_0 : a_0[i] = b \wedge b < x \wedge a = a_0[i \mapsto a_0[i - 1]] \\
 &\equiv b < x \wedge \exists a_0 : a_0[i] = b \wedge a = a_0[i \mapsto a_0[i - 1]] \\
 &\equiv b < x \wedge a[i] = a[i - 1]
 \end{aligned}$$

$$\begin{aligned}
 SP_1 &\equiv \text{sp}(i:=i-1, b < x \wedge a[i] = a[i - 1]) \\
 &= \exists i_0 : b < x \wedge a[i_0] = a[i_0 - 1] \wedge i = i_0 - 1 \\
 &\equiv b < x \wedge \exists i_0 : a[i_0] = a[i_0 - 1] \wedge i_0 = i + 1 \\
 &\equiv b < x \wedge a[i + 1] = a[(i + 1) - 1] \equiv b < x \wedge a[i + 1] = a[i]
 \end{aligned}$$

$$SP \equiv (b < x \wedge a[i + 1] = a[i]) \vee (b \geq x \wedge a[i] = b)$$



# Hoare Calc. and Predicate Transformers

---

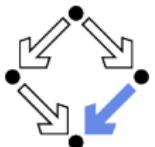
In practice, often a combination of the calculi is applied.

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

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

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

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



# Weakest Liberal Preconditions for Loops

---

Why not apply predicate transformers to loops?

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

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

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

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

## ■ Interpretation

- Weakest precondition that ensures that loops stops in a state satisfying  $Q$ , unless it aborts or runs forever.

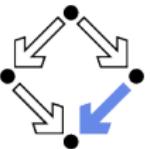
## ■ Infinite sequence of predicates $L_i(Q)$ :

- 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: $L_i(Q) = \text{wp}(\text{if}_i, Q)$

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

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



# Example

---

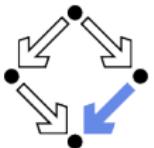
$\text{wp}(\text{while } i < n \text{ do } i := i + 1, Q)$

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

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

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

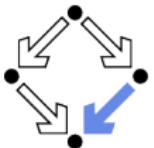
$$\begin{aligned} L_3(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, \\ &\quad (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow (i + 1 \not< n \Rightarrow Q[i + 1/i])))) \\ &\Leftrightarrow (i \not< n \Rightarrow Q) \wedge \\ &\quad (i < n \Rightarrow ((i + 1 \not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



# 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  $\neg(P \Rightarrow L_i(Q)), \dots$ 
  - $\{P\} \text{ while } b \text{ do } c \{Q\}$  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.

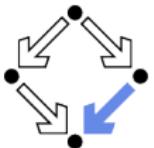


# Preconditions for Loops with Invariants

```
wp(while b do invariant I; cx, ..., Q) =  
  let oldx = x, ... in  
  I ∧ (∀x, ... : I ∧ b ⇒ wp(c, I)) ∧  
  (∀x, ... : I ∧ ¬b ⇒ 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** ( $s := s + i; i := i + 1$ )

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

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

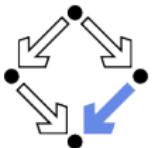
## ■ Weakest precondition:

$\text{wp}(\text{while } i \leq n \text{ do invariant } I; c^{s,i}, Q) =$   
**let**  $olds = s, oldi = i$  **in**  
 $I \wedge (\forall s, i : I \wedge i \leq n \Rightarrow I[i + 1/i][s + i/s]) \wedge$   
 $(\forall s, i : I \wedge \neg(i \leq n) \Rightarrow Q)$

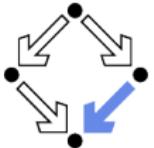
## ■ Verification condition:

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

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



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



# Termination

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

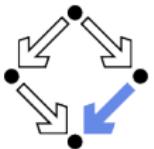
$$\{ \text{false} \} \text{ loop } \{ \text{false} \} \quad \frac{I \Rightarrow t \geq 0 \quad \{ I \wedge b \wedge t = N \} \ c \ \{ I \wedge t < N \}}{\{ I \} \text{ while } b \text{ do } c \ \{ I \wedge \neg b \}}$$

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

- 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

---

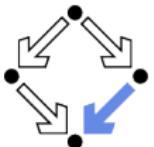
$$I : \Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$$

$$t := n - i + 1$$

$$\frac{(n \geq 0 \wedge i = 1 \wedge s = 0) \Rightarrow I \quad I \Rightarrow n - i + 1 \geq 0}{\{I \wedge i \leq n \wedge n - i + 1 = N\} \quad s := s + i; i := i + 1 \quad \{I \wedge n - i + 1 < N\}}$$

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

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

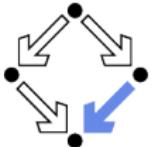


# Termination in RISCAL

---

```
while i ≤ n do
    invariant s =  $\sum_{j:\text{number with } 1 \leq j \wedge j \leq i-1} j$ ;
    invariant  $1 \leq i \wedge i \leq n+1$ ;
    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);
```

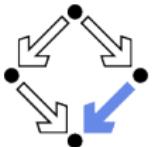


# Termination in RISCAL

```

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) ⇔ ...

```



# 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.

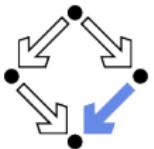
- Preserves property:  $\{P\} c \{Q\}$  iff  $(P \Rightarrow \text{wp}(c, Q))$

- 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

---

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

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

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

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

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

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

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

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

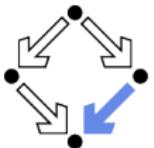
$$L_3(Q) = (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \text{wp}(i := i + 1, L_2(Q)))$$

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

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

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

...

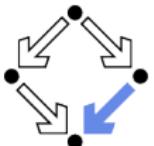


# Weakest Preconditions for Loops

---

- 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”:
  - $\forall i \in \mathbb{N} : L_i(Q) \Rightarrow \text{wp}(\mathbf{while } b \mathbf{ do } c, Q).$
  - $\forall P : (\forall i \in \mathbb{N} : L_i(Q) \Rightarrow P) \Rightarrow (\text{wp}(\mathbf{while } b \mathbf{ do } c, Q) \Rightarrow P).$
- 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  $\{P\} \subset \{Q\}.$ 
  - 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.

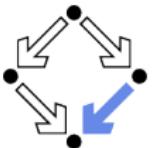


# Weakest Precondition with Measures

```
wp(while b do invariant I; decreases t; cx, ..., Q) =  
  let oldx = x, ... in  
  I ∧ (∀x, ... : I ∧ b ⇒ wp(c, I)) ∧  
  (∀x, ... : I ∧ ¬b ⇒ Q) ∧  
  (∀x, ... : I ⇒ t ≥ 0) ∧  
  (∀x, ... : I ∧ b ⇒ let T = t in wp(c, t < T))
```

- 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** ( $s := s + i; i := i + 1$ )

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

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

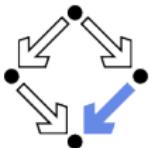
$t := n + 1 - i$

## ■ Weakest precondition:

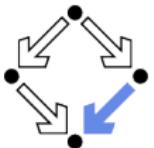
$$\begin{aligned} \text{wp}(\text{while } i \leq n \text{ do invariant } I; c^{s,i}, Q) = \\ \text{let } olds = s, oldi = i \text{ in} \\ I \wedge (\forall s, i : I \wedge i \leq n \Rightarrow I[s + i/s, i + 1/i]) \wedge \\ (\forall s, i : I \wedge \neg(i \leq n) \Rightarrow Q) \wedge \\ (\forall s, i : I \Rightarrow t \geq 0) \wedge \\ (\forall s, i : I \wedge i \leq n \Rightarrow \text{let } T = n + 1 - i \text{ in } n + 1 - (i + 1) < T) \end{aligned}$$

## ■ Verification condition:

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



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



# Abortion

---

New rules to prevent abortion.

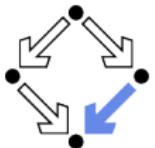
{false} **abort** {true}

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

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

- 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

---

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

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

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

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

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

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

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

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

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

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

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

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

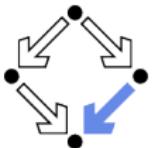
$D(e_1 < e_2) = D(e_1) \wedge D(e_2).$

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

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

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

Assumes that expressions have already been type-checked.



# Abortion

---

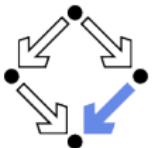
Slight modification of existing rules.

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

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

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

Expressions must be defined in any context.



# Abortion

---

Similar modifications of weakest preconditions.

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

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

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

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

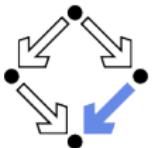
$$\text{wp}(\mathbf{if } b \mathbf{ then } c, Q) = D(b) \wedge (b \Rightarrow \text{wp}(c, Q)) \wedge (\neg b \Rightarrow Q)$$

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

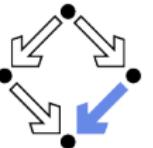
$$L_0(Q) = \text{false}$$

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

$\text{wp}(c, Q)$  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. Termination
  5. Abortion
  6. Generating Verification Conditions
  7. Proving Verification Conditions
  8. Procedures



# RISCAL and Verification Conditions

RISC Algorithm Language (RISCAL)

File Edit SMT TP 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[N];
5 type index = N[N+1];
6 type result = N[N-(1+N)/2];
7
8 proc summation(n:number): result
9   requires n ≥ 0;
10  ensures result = ∑:number with 1 ≤ j ≤ n. j;
11 {
12   var s:result = 0;
13   var i:index = 1;
14   while i ≤ n do
15     invariant s = ∑:number with 1 ≤ j ≤ i. j;
16     invariant 1 ≤ i ≤ n+1;
17     decreases n-i+1;
18   {
19     s = s+i;
20     i = i+1;
21   }
22   return s;
23 }
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
31 pred Output(n:number, s:result) =
32   s = ∑:number with 1 ≤ j ≤ n. j;
33
34 pred Invariant(n:number, s:result, i:index) =
35   (s = ∑:number with 1 ≤ j ≤ i. j) ∧ 1 ≤ i ≤ n+1;
36
37 fun Termination(n:number, s:result, i:index): number
38   = n-1;
39
40 theorem A(n:number, s:result, i:index)
41   requires n ≥ 0;
```

Analysis

Translation:  Nondeterminism Default Value: 0 Other Values:

Execution:  Silent Inputs: Per Mille: Branches: Depth:

Visualization:  Trace  Tree Width: 800 Height: 600

Parallelism:  Multi-Threaded Threads: 4  Distributed Servers:

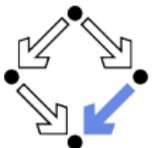
Operation:  summation(Z)

at unknown position:  
theorem is not true  
ERROR encountered in execution (3 ms).  
Executing summation @ PostSat(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation @ PostNotTrivialAll(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation @ PostNotTrivialSome().  
Execution completed (0 ms).  
Executing summation @ PostUnique(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation @ CorrOp(Z) with all 5 inputs.  
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).  
Executing summation @ LongOp(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation @ LongOp1(Z) with all 5 inputs.  
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).  
Executing summation @ LongOp2(Z) with all 5 inputs.  
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).  
Executing summation @ LongOp3(Z) with all 5 inputs.  
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).  
Executing summation @ LongOp4(Z) with all 5 inputs.  
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).  
Executing summation @ LongOp5(Z) with all 5 inputs.  
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).  
Executing summation @ PrepOp0(Z) with all 5 inputs.  
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).  
Executing summation @ PrepOp1(Z) with all 5 inputs.  
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).

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?

RISCAL implements (a variant of) the wp-calculus for VC generation.



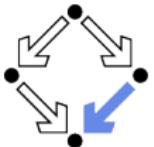
# RISCAL Verification Conditions

---

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

- Implementation preconditions
  - Well-definedness of commands and loop annotations.
  - One condition for every partial function/predicate application.
- 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.
  - One condition for every decreases clause.

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 (see menu “SMT”).

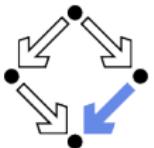
- ➡ Execute Task
- Show Counterexample
- Print Description
- Print Definition
- Apply SMT Solver
- Apply Theorem Prover
- Print Prover Output

**Example:** is loop invariant preserved?

```
s = ( $\sum j:\text{number}$  with  $(1 \leq j) \wedge (j \leq (i-1))$ ). j

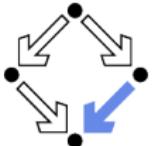
theorem _summation_0_LoopOp3(n:number)
requires n  $\geq$  0;
 $\Leftrightarrow \forall s:\text{result}, i:\text{index}.$  ((( $s = (\sum j:\text{number}$  with  $(1 \leq j) \wedge (j \leq (i-1))$ ). j))
 $\wedge ((1 \leq i) \wedge (i \leq (n+1))) \wedge (i \leq n)) \Rightarrow$ 
 $(\text{let } s = s+i \text{ in } (\text{let } i = i+1 \text{ in}$ 
 $(s = (\sum j:\text{number}$  with  $(1 \leq j) \wedge (j \leq (i-1))$ . j))));
```

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



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

# Proving Verification Conditions



RISCAL also integrates the RISCTP interface to various theorem provers.

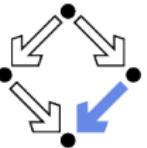
- Menu “TP” and menu entry “Apply Theorem Prover”
    - Tries to prove verification condition for *arbitrary* type sizes.
    - “Apply Prover to All Theorems”: multiple proofs (in parallel).
    - “Print Prover Output”: shows details of proof attempt.
    - “Open Theorem Prover GUI”: open the RISTP web interface.

```

File Edit SMT TP Help
File /src/tutorials/courses/wis2024/formal/videos/02-verify/0/search2.tpt
Analysis
Translation: QF_ABSTRACTION Default Value: 0 Other Values: ...
Execution: Silent Inputs: Per Mile: Branches: Depth: ...
Visualization: Trace Tree Width: 1500 Height: 800
Parallelization: Multi-Threaded Threads: 4 Distributed Servers: ...
Operations: □ searchAtIndex(1, I)
RISCA Algorithm Language 4.3.8 (Alpha 15, 2024)
https://www.risc.at/research/formal/software/RISCA/
(C) 2016-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCA -h" to see the available command line options.
-----
Reading file /src/tutorials/courses/wis2024/formal/slides/02-verify/I
Linuxarch2.tpt
Using N3.
Using N3.
Computing the value of searchAtIndex.
Type checking and translation completed.
Parallel execution with 4 threads (no output is shown)...
Execution completed (548 ms, see "Print Prover Output").

1 // linear search: given an array a and key x,
2 // return the smallest position i where x occurs in a
3 // if x does not occur in a
4 //   then return -1
5
6 val MIN = 0
7 val MAX = 1000
8
9 type index = Z[1..N];
10 type elem = N[0..MAX];
11
12 proc searchAtIndex(a: elem[], x: elem): index
13   ensures result >= 1 & result <= N;
14   ensures result = 0 & x = a[0] => x = a[0];
15   ensures result = result + 1 & x = a[result] => x = a[result];
16   ensures result = result + 1 & x = a[result] => x = a[result];
17
18 var i: index := 0;
19 var z: index := -3;
20 while i <= z do
21   if a[i] <= x & a[i] >= N then
22     invariant "[i] index: 0 <= i < N";
23     invariant "[i] index: 0 <= i <= a[i]" & x <= a[i];
24     invariant "[i+1] index: 1 <= i + 1 < N & a[i] <= a[i+1]";
25     invariant "[i+1] index: 1 <= i + 1 < N & a[i] <= a[i+1]";
26     invariant "[i+1] index: 1 <= i + 1 < N & a[i] <= a[i+1]";
27   else i := i + 1;
28 end
29
30 return z;
31
32
33 // the verification conditions to be proved
34 // for the total correctness of the program
35 // are as follows:
36
37 pred Input(index: index; x: index) =
38   1 = 0 & x = -1;
39
40
41 print "Input: index = " + index + ", x = " + x;
42 print "Result: " + z;
43
44 print "Verification conditions to be proved:";
45 print "1. searchAtIndex([1..N], 0) = 0";
46 print "2. searchAtIndex([1..N], 1) = 1";
47 print "3. searchAtIndex([1..N], 2) = 2";
48 print "4. searchAtIndex([1..N], 3) = 3";
49 print "5. searchAtIndex([1..N], 4) = 4";
50 print "6. searchAtIndex([1..N], 5) = 5";
51 print "7. searchAtIndex([1..N], 6) = 6";
52 print "8. searchAtIndex([1..N], 7) = 7";
53 print "9. searchAtIndex([1..N], 8) = 8";
54 print "10. searchAtIndex([1..N], 9) = 9";
55 print "11. searchAtIndex([1..N], 10) = 10";
56
57 print "12. searchAtIndex([1..N], 11) = -1";
58
59 print "13. searchAtIndex([1..N], 12) = -1";
60
61 print "14. searchAtIndex([1..N], 13) = -1";
62
63 print "15. searchAtIndex([1..N], 14) = -1";
64
65 print "16. searchAtIndex([1..N], 15) = -1";
66
67 print "17. searchAtIndex([1..N], 16) = -1";
68
69 print "18. searchAtIndex([1..N], 17) = -1";
70
71 print "19. searchAtIndex([1..N], 18) = -1";
72
73 print "20. searchAtIndex([1..N], 19) = -1";
74
75 print "21. searchAtIndex([1..N], 20) = -1";
76
77 print "22. searchAtIndex([1..N], 21) = -1";
78
79 print "23. searchAtIndex([1..N], 22) = -1";
80
81 print "24. searchAtIndex([1..N], 23) = -1";
82
83 print "25. searchAtIndex([1..N], 24) = -1";
84
85 print "26. searchAtIndex([1..N], 25) = -1";
86
87 print "27. searchAtIndex([1..N], 26) = -1";
88
89 print "28. searchAtIndex([1..N], 27) = -1";
90
91 print "29. searchAtIndex([1..N], 28) = -1";
92
93 print "30. searchAtIndex([1..N], 29) = -1";
94
95 print "31. searchAtIndex([1..N], 30) = -1";
96
97 print "32. searchAtIndex([1..N], 31) = -1";
98
99 print "33. searchAtIndex([1..N], 32) = -1";
100
101 print "34. searchAtIndex([1..N], 33) = -1";
102
103 print "35. searchAtIndex([1..N], 34) = -1";
104
105 print "36. searchAtIndex([1..N], 35) = -1";
106
107 print "37. searchAtIndex([1..N], 36) = -1";
108
109 print "38. searchAtIndex([1..N], 37) = -1";
110
111 print "39. searchAtIndex([1..N], 38) = -1";
112
113 print "40. searchAtIndex([1..N], 39) = -1";
114
115 print "41. searchAtIndex([1..N], 40) = -1";
116
117 print "42. searchAtIndex([1..N], 41) = -1";
118
119 print "43. searchAtIndex([1..N], 42) = -1";
120
121 print "44. searchAtIndex([1..N], 43) = -1";
122
123 print "45. searchAtIndex([1..N], 44) = -1";
124
125 print "46. searchAtIndex([1..N], 45) = -1";
126
127 print "47. searchAtIndex([1..N], 46) = -1";
128
129 print "48. searchAtIndex([1..N], 47) = -1";
130
131 print "49. searchAtIndex([1..N], 48) = -1";
132
133 print "50. searchAtIndex([1..N], 49) = -1";
134
135 print "51. searchAtIndex([1..N], 50) = -1";
136
137 print "52. searchAtIndex([1..N], 51) = -1";
138
139 print "53. searchAtIndex([1..N], 52) = -1";
140
141 print "54. searchAtIndex([1..N], 53) = -1";
142
143 print "55. searchAtIndex([1..N], 54) = -1";
144
145 print "56. searchAtIndex([1..N], 55) = -1";
146
147 print "57. searchAtIndex([1..N], 56) = -1";
148
149 print "58. searchAtIndex([1..N], 57) = -1";
150
151 print "59. searchAtIndex([1..N], 58) = -1";
152
153 print "60. searchAtIndex([1..N], 59) = -1";
154
155 print "61. searchAtIndex([1..N], 60) = -1";
156
157 print "62. searchAtIndex([1..N], 61) = -1";
158
159 print "63. searchAtIndex([1..N], 62) = -1";
160
161 print "64. searchAtIndex([1..N], 63) = -1";
162
163 print "65. searchAtIndex([1..N], 64) = -1";
164
165 print "66. searchAtIndex([1..N], 65) = -1";
166
167 print "67. searchAtIndex([1..N], 66) = -1";
168
169 print "68. searchAtIndex([1..N], 67) = -1";
170
171 print "69. searchAtIndex([1..N], 68) = -1";
172
173 print "70. searchAtIndex([1..N], 69) = -1";
174
175 print "71. searchAtIndex([1..N], 70) = -1";
176
177 print "72. searchAtIndex([1..N], 71) = -1";
178
179 print "73. searchAtIndex([1..N], 72) = -1";
180
181 print "74. searchAtIndex([1..N], 73) = -1";
182
183 print "75. searchAtIndex([1..N], 74) = -1";
184
185 print "76. searchAtIndex([1..N], 75) = -1";
186
187 print "77. searchAtIndex([1..N], 76) = -1";
188
189 print "78. searchAtIndex([1..N], 77) = -1";
190
191 print "79. searchAtIndex([1..N], 78) = -1";
192
193 print "80. searchAtIndex([1..N], 79) = -1";
194
195 print "81. searchAtIndex([1..N], 80) = -1";
196
197 print "82. searchAtIndex([1..N], 81) = -1";
198
199 print "83. searchAtIndex([1..N], 82) = -1";
199
200 print "84. searchAtIndex([1..N], 83) = -1";
200
201 print "85. searchAtIndex([1..N], 84) = -1";
201
202 print "86. searchAtIndex([1..N], 85) = -1";
202
203 print "87. searchAtIndex([1..N], 86) = -1";
203
204 print "88. searchAtIndex([1..N], 87) = -1";
204
205 print "89. searchAtIndex([1..N], 88) = -1";
205
206 print "90. searchAtIndex([1..N], 89) = -1";
206
207 print "91. searchAtIndex([1..N], 90) = -1";
207
208 print "92. searchAtIndex([1..N], 91) = -1";
208
209 print "93. searchAtIndex([1..N], 92) = -1";
209
210 print "94. searchAtIndex([1..N], 93) = -1";
210
211 print "95. searchAtIndex([1..N], 94) = -1";
211
212 print "96. searchAtIndex([1..N], 95) = -1";
212
213 print "97. searchAtIndex([1..N], 96) = -1";
213
214 print "98. searchAtIndex([1..N], 97) = -1";
214
215 print "99. searchAtIndex([1..N], 98) = -1";
215
216 print "100. searchAtIndex([1..N], 99) = -1";
216
217 print "101. searchAtIndex([1..N], 100) = -1";
217
218 print "102. searchAtIndex([1..N], 101) = -1";
218
219 print "103. searchAtIndex([1..N], 102) = -1";
219
220 print "104. searchAtIndex([1..N], 103) = -1";
220
221 print "105. searchAtIndex([1..N], 104) = -1";
221
222 print "106. searchAtIndex([1..N], 105) = -1";
222
223 print "107. searchAtIndex([1..N], 106) = -1";
223
224 print "108. searchAtIndex([1..N], 107) = -1";
224
225 print "109. searchAtIndex([1..N], 108) = -1";
225
226 print "110. searchAtIndex([1..N], 109) = -1";
226
227 print "111. searchAtIndex([1..N], 110) = -1";
227
228 print "112. searchAtIndex([1..N], 111) = -1";
228
229 print "113. searchAtIndex([1..N], 112) = -1";
229
230 print "114. searchAtIndex([1..N], 113) = -1";
230
231 print "115. searchAtIndex([1..N], 114) = -1";
231
232 print "116. searchAtIndex([1..N], 115) = -1";
232
233 print "117. searchAtIndex([1..N], 116) = -1";
233
234 print "118. searchAtIndex([1..N], 117) = -1";
234
235 print "119. searchAtIndex([1..N], 118) = -1";
235
236 print "120. searchAtIndex([1..N], 119) = -1";
236
237 print "121. searchAtIndex([1..N], 120) = -1";
237
238 print "122. searchAtIndex([1..N], 121) = -1";
238
239 print "123. searchAtIndex([1..N], 122) = -1";
239
240 print "124. searchAtIndex([1..N], 123) = -1";
240
241 print "125. searchAtIndex([1..N], 124) = -1";
241
242 print "126. searchAtIndex([1..N], 125) = -1";
242
243 print "127. searchAtIndex([1..N], 126) = -1";
243
244 print "128. searchAtIndex([1..N], 127) = -1";
244
245 print "129. searchAtIndex([1..N], 128) = -1";
245
246 print "130. searchAtIndex([1..N], 129) = -1";
246
247 print "131. searchAtIndex([1..N], 130) = -1";
247
248 print "132. searchAtIndex([1..N], 131) = -1";
248
249 print "133. searchAtIndex([1..N], 132) = -1";
249
250 print "134. searchAtIndex([1..N], 133) = -1";
250
251 print "135. searchAtIndex([1..N], 134) = -1";
251
252 print "136. searchAtIndex([1..N], 135) = -1";
252
253 print "137. searchAtIndex([1..N], 136) = -1";
253
254 print "138. searchAtIndex([1..N], 137) = -1";
254
255 print "139. searchAtIndex([1..N], 138) = -1";
255
256 print "140. searchAtIndex([1..N], 139) = -1";
256
257 print "141. searchAtIndex([1..N], 140) = -1";
257
258 print "142. searchAtIndex([1..N], 141) = -1";
258
259 print "143. searchAtIndex([1..N], 142) = -1";
259
260 print "144. searchAtIndex([1..N], 143) = -1";
260
261 print "145. searchAtIndex([1..N], 144) = -1";
261
262 print "146. searchAtIndex([1..N], 145) = -1";
262
263 print "147. searchAtIndex([1..N], 146) = -1";
263
264 print "148. searchAtIndex([1..N], 147) = -1";
264
265 print "149. searchAtIndex([1..N], 148) = -1";
265
266 print "150. searchAtIndex([1..N], 149) = -1";
266
267 print "151. searchAtIndex([1..N], 150) = -1";
267
268 print "152. searchAtIndex([1..N], 151) = -1";
268
269 print "153. searchAtIndex([1..N], 152) = -1";
269
270 print "154. searchAtIndex([1..N], 153) = -1";
270
271 print "155. searchAtIndex([1..N], 154) = -1";
271
272 print "156. searchAtIndex([1..N], 155) = -1";
272
273 print "157. searchAtIndex([1..N], 156) = -1";
273
274 print "158. searchAtIndex([1..N], 157) = -1";
274
275 print "159. searchAtIndex([1..N], 158) = -1";
275
276 print "160. searchAtIndex([1..N], 159) = -1";
276
277 print "161. searchAtIndex([1..N], 160) = -1";
277
278 print "162. searchAtIndex([1..N], 161) = -1";
278
279 print "163. searchAtIndex([1..N], 162) = -1";
279
280 print "164. searchAtIndex([1..N], 163) = -1";
280
281 print "165. searchAtIndex([1..N], 164) = -1";
281
282 print "166. searchAtIndex([1..N], 165) = -1";
282
283 print "167. searchAtIndex([1..N], 166) = -1";
283
284 print "168. searchAtIndex([1..N], 167) = -1";
284
285 print "169. searchAtIndex([1..N], 168) = -1";
285
286 print "170. searchAtIndex([1..N], 169) = -1";
286
287 print "171. searchAtIndex([1..N], 170) = -1";
287
288 print "172. searchAtIndex([1..N], 171) = -1";
288
289 print "173. searchAtIndex([1..N], 172) = -1";
289
290 print "174. searchAtIndex([1..N], 173) = -1";
290
291 print "175. searchAtIndex([1..N], 174) = -1";
291
292 print "176. searchAtIndex([1..N], 175) = -1";
292
293 print "177. searchAtIndex([1..N], 176) = -1";
293
294 print "178. searchAtIndex([1..N], 177) = -1";
294
295 print "179. searchAtIndex([1..N], 178) = -1";
295
296 print "180. searchAtIndex([1..N], 179) = -1";
296
297 print "181. searchAtIndex([1..N], 180) = -1";
297
298 print "182. searchAtIndex([1..N], 181) = -1";
298
299 print "183. searchAtIndex([1..N], 182) = -1";
299
300 print "184. searchAtIndex([1..N], 183) = -1";
300
301 print "185. searchAtIndex([1..N], 184) = -1";
301
302 print "186. searchAtIndex([1..N], 185) = -1";
302
303 print "187. searchAtIndex([1..N], 186) = -1";
303
304 print "188. searchAtIndex([1..N], 187) = -1";
304
305 print "189. searchAtIndex([1..N], 188) = -1";
305
306 print "190. searchAtIndex([1..N], 189) = -1";
306
307 print "191. searchAtIndex([1..N], 190) = -1";
307
308 print "192. searchAtIndex([1..N], 191) = -1";
308
309 print "193. searchAtIndex([1..N], 192) = -1";
309
310 print "194. searchAtIndex([1..N], 193) = -1";
310
311 print "195. searchAtIndex([1..N], 194) = -1";
311
312 print "196. searchAtIndex([1..N], 195) = -1";
312
313 print "197. searchAtIndex([1..N], 196) = -1";
313
314 print "198. searchAtIndex([1..N], 197) = -1";
314
315 print "199. searchAtIndex([1..N], 198) = -1";
315
316 print "200. searchAtIndex([1..N], 199) = -1";
316
317 print "201. searchAtIndex([1..N], 200) = -1";
317
318 print "202. searchAtIndex([1..N], 201) = -1";
318
319 print "203. searchAtIndex([1..N], 202) = -1";
319
320 print "204. searchAtIndex([1..N], 203) = -1";
320
321 print "205. searchAtIndex([1..N], 204) = -1";
321
322 print "206. searchAtIndex([1..N], 205) = -1";
322
323 print "207. searchAtIndex([1..N], 206) = -1";
323
324 print "208. searchAtIndex([1..N], 207) = -1";
324
325 print "209. searchAtIndex([1..N], 208) = -1";
325
326 print "210. searchAtIndex([1..N], 209) = -1";
326
327 print "211. searchAtIndex([1..N], 210) = -1";
327
328 print "212. searchAtIndex([1..N], 211) = -1";
328
329 print "213. searchAtIndex([1..N], 212) = -1";
329
330 print "214. searchAtIndex([1..N], 213) = -1";
330
331 print "215. searchAtIndex([1..N], 214) = -1";
331
332 print "216. searchAtIndex([1..N], 215) = -1";
332
333 print "217. searchAtIndex([1..N], 216) = -1";
333
334 print "218. searchAtIndex([1..N], 217) = -1";
334
335 print "219. searchAtIndex([1..N], 218) = -1";
335
336 print "220. searchAtIndex([1..N], 219) = -1";
336
337 print "221. searchAtIndex([1..N], 220) = -1";
337
338 print "222. searchAtIndex([1..N], 221) = -1";
338
339 print "223. searchAtIndex([1..N], 222) = -1";
339
340 print "224. searchAtIndex([1..N], 223) = -1";
340
341 print "225. searchAtIndex([1..N], 224) = -1";
341
342 print "226. searchAtIndex([1..N], 225) = -1";
342
343 print "227. searchAtIndex([1..N], 226) = -1";
343
344 print "228. searchAtIndex([1..N], 227) = -1";
344
345 print "229. searchAtIndex([1..N], 228) = -1";
345
346 print "230. searchAtIndex([1..N], 229) = -1";
346
347 print "231. searchAtIndex([1..N], 230) = -1";
347
348 print "232. searchAtIndex([1..N], 231) = -1";
348
349 print "233. searchAtIndex([1..N], 232) = -1";
349
350 print "234. searchAtIndex([1..N], 233) = -1";
350
351 print "235. searchAtIndex([1..N], 234) = -1";
351
352 print "236. searchAtIndex([1..N], 235) = -1";
352
353 print "237. searchAtIndex([1..N], 236) = -1";
353
354 print "238. searchAtIndex([1..N], 237) = -1";
354
355 print "239. searchAtIndex([1..N], 238) = -1";
355
356 print "240. searchAtIndex([1..N], 239) = -1";
356
357 print "241. searchAtIndex([1..N], 240) = -1";
357
358 print "242. searchAtIndex([1..N], 241) = -1";
358
359 print "243. searchAtIndex([1..N], 242) = -1";
359
360 print "244. searchAtIndex([1..N], 243) = -1";
360
361 print "245. searchAtIndex([1..N], 244) = -1";
361
362 print "246. searchAtIndex([1..N], 245) = -1";
362
363 print "247. searchAtIndex([1..N], 246) = -1";
363
364 print "248. searchAtIndex([1..N], 247) = -1";
364
365 print "249. searchAtIndex([1..N], 248) = -1";
365
366 print "250. searchAtIndex([1..N], 249) = -1";
366
367 print "251. searchAtIndex([1..N], 250) = -1";
367
368 print "252. searchAtIndex([1..N], 251) = -1";
368
369 print "253. searchAtIndex([1..N], 252) = -1";
369
370 print "254. searchAtIndex([1..N], 253) = -1";
370
371 print "255. searchAtIndex([1..N], 254) = -1";
371
372 print "256. searchAtIndex([1..N], 255) = -1";
372
373 print "257. searchAtIndex([1..N], 256) = -1";
373
374 print "258. searchAtIndex([1..N], 257) = -1";
374
375 print "259. searchAtIndex([1..N], 258) = -1";
375
376 print "260. searchAtIndex([1..N], 259) = -1";
376
377 print "261. searchAtIndex([1..N], 260) = -1";
377
378 print "262. searchAtIndex([1..N], 261) = -1";
378
379 print "263. searchAtIndex([1..N], 262) = -1";
379
380 print "264. searchAtIndex([1..N], 263) = -1";
380
381 print "265. searchAtIndex([1..N], 264) = -1";
381
382 print "266. searchAtIndex([1..N], 265) = -1";
382
383 print "267. searchAtIndex([1..N], 266) = -1";
383
384 print "268. searchAtIndex([1..N], 267) = -1";
384
385 print "269. searchAtIndex([1..N], 268) = -1";
385
386 print "270. searchAtIndex([1..N], 269) = -1";
386
387 print "271. searchAtIndex([1..N], 270) = -1";
387
388 print "272. searchAtIndex([1..N], 271) = -1";
388
389 print "273. searchAtIndex([1..N], 272) = -1";
389
390 print "274. searchAtIndex([1..N], 273) = -1";
390
391 print "275. searchAtIndex([1..N], 274) = -1";
391
392 print "276. searchAtIndex([1..N], 275) = -1";
392
393 print "277. searchAtIndex([1..N], 276) = -1";
393
394 print "278. searchAtIndex([1..N], 277) = -1";
394
395 print "279. searchAtIndex([1..N], 278) = -1";
395
396 print "280. searchAtIndex([1..N], 279) = -1";
396
397 print "281. searchAtIndex([1..N], 280) = -1";
397
398 print "282. searchAtIndex([1..N], 281) = -1";
398
399 print "283. searchAtIndex([1..N], 282) = -1";
399
400 print "284. searchAtIndex([1..N], 283) = -1";
400
401 print "285. searchAtIndex([1..N], 284) = -1";
401
402 print "286. searchAtIndex([1..N], 285) = -1";
402
403 print "287. searchAtIndex([1..N], 286) = -1";
403
404 print "288. searchAtIndex([1..N], 287) = -1";
404
405 print "289. searchAtIndex([1..N], 288) = -1";
405
406 print "290. searchAtIndex([1..N], 289) = -1";
406
407 print "291. searchAtIndex([1..N], 290) = -1";
407
408 print "292. searchAtIndex([1..N], 291) = -1";
408
409 print "293. searchAtIndex([1..N], 292) = -1";
409
410 print "294. searchAtIndex([1..N], 293) = -1";
410
411 print "295. searchAtIndex([1..N], 294) = -1";
411
412 print "296. searchAtIndex([1..N], 295) = -1";
412
413 print "297. searchAtIndex([1..N], 296) = -1";
413
414 print "298. searchAtIndex([1..N], 297) = -1";
414
415 print "299. searchAtIndex([1..N], 298) = -1";
415
416 print "300. searchAtIndex([1..N], 299) = -1";
416
417 print "301. searchAtIndex([1..N], 300) = -1";
417
418 print "302. searchAtIndex([1..N], 301) = -1";
418
419 print "303. searchAtIndex([1..N], 302) = -1";
419
420 print "304. searchAtIndex([1..N], 303) = -1";
420
421 print "305. searchAtIndex([1..N], 304) = -1";
421
422 print "306. searchAtIndex([1..N], 305) = -1";
422
423 print "307. searchAtIndex([1..N], 306) = -1";
423
424 print "308. searchAtIndex([1..N], 307) = -1";
424
425 print "309. searchAtIndex([1..N], 308) = -1";
425
426 print "310. searchAtIndex([1..N], 309) = -1";
426
427 print "311. searchAtIndex([1..N], 310) = -1";
427
428 print "312. searchAtIndex([1..N], 311) = -1";
428
429 print "313. searchAtIndex([1..N], 312) = -1";
429
430 print "314. searchAtIndex([1..N], 313) = -1";
430
431 print "315. searchAtIndex([1..N], 314) = -1";
431
432 print "316. searchAtIndex([1..N], 315) = -1";
432
433 print "317. searchAtIndex([1..N], 316) = -1";
433
434 print "318. searchAtIndex([1..N], 317) = -1";
434
435 print "319. searchAtIndex([1..N], 318) = -1";
435
436 print "320. searchAtIndex([1..N], 319) = -1";
436
437 print "321. searchAtIndex([1..N], 320) = -1";
437
438 print "322. searchAtIndex([1..N], 321) = -1";
438
439 print "323. searchAtIndex([1..N], 322) = -1";
439
440 print "324. searchAtIndex([1..N], 323) = -1";
440
441 print "325. searchAtIndex([1..N], 324) = -1";
441
442 print "326. searchAtIndex([1..N], 325) = -1";
442
443 print "327. searchAtIndex([1..N], 326) = -1";
443
444 print "328. searchAtIndex([1..N], 327) = -1";
444
445 print "329. searchAtIndex([1..N], 328) = -1";
445
446 print "330. searchAtIndex([1..N], 329) = -1";
446
447 print "331. searchAtIndex([1..N], 330) = -1";
447
448 print "332. searchAtIndex([1..N], 331) = -1";
448
449 print "333. searchAtIndex([1..N], 332) = -1";
449
450 print "334. searchAtIndex([1..N], 333) = -1";
450
451 print "335. searchAtIndex([1..N], 334) = -1";
451
452 print "336. searchAtIndex([1..N], 335) = -1";
452
453 print "337. searchAtIndex([1..N], 336) = -1";
453
454 print "338. searchAtIndex([1..N], 337) = -1";
454
455 print "339. searchAtIndex([1..N], 338) = -1";
455
456 print "340. searchAtIndex([1..N], 339) = -1";
456
457 print "341. searchAtIndex([1..N], 340) = -1";
457
458 print "342. searchAtIndex([1..N], 341) = -1";
458
459 print "343. searchAtIndex([1..N], 342) = -1";
459
460 print "344. searchAtIndex([1..N], 343) = -1";
460
461 print "345. searchAtIndex([1..N], 344) = -1";
461
462 print "346. searchAtIndex([1..N], 345) = -1";
462
463 print "347. searchAtIndex([1..N], 346) = -1";
463
464 print "348. searchAtIndex([1..N], 347) = -1";
464
465 print "349. searchAtIndex([1..N], 348) = -1";
465
466 print "350. searchAtIndex([1..N], 349) = -1";
466
467 print "351. searchAtIndex([1..N], 350) = -1";
467
468 print "352. searchAtIndex([1..N], 351) = -1";
468
469 print "353. searchAtIndex([1..N], 352) = -1";
469
470 print "354. searchAtIndex([1..N], 353) = -1";
470
471 print "355. searchAtIndex([1..N], 354) = -1";
471
472 print "356. searchAtIndex([1..N], 355) = -1";
472
473 print "357. searchAtIndex([1..N], 356) = -1";
473
474 print "358. searchAtIndex([1..N], 357) = -1";
474
475 print "359. searchAtIndex([1..N], 358) = -1";
475
476 print "360. searchAtIndex([1..N], 359) = -1";
476
477 print "361. searchAtIndex([1..N], 360) = -1";
477
478 print "362. searchAtIndex([1..N], 361) = -1";
478
479 print "363. searchAtIndex([1..N], 362) = -1";
479
480 print "364. searchAtIndex([1..N], 363) = -1";
480
481 print "365. searchAtIndex([1..N], 364) = -1";
481
482 print "366. searchAtIndex([1..N], 365) = -1";
482
483 print "367. searchAtIndex([1..N], 366) = -1";
483
484 print "368. searchAtIndex([1..N], 367) = -1";
484
485 print "369. searchAtIndex([1..N], 368) = -1";
485
486 print "370. searchAtIndex([1..N], 369) = -1";
486
487 print "371. searchAtIndex([1..N], 370) = -1";
487
488 print "372. searchAtIndex([1..N], 371) = -1";
488
489 print "373. searchAtIndex([1..N], 372) = -1";
489
490 print "374. searchAtIndex([1..N], 373) = -1";
490
491 print "375. searchAtIndex([1..N], 374) = -1";
491
492 print "376. searchAtIndex([1..N], 375) = -1";
492
493 print "377. searchAtIndex([1..N], 376) = -1";
493
494 print "378. searchAtIndex([1..N], 377) = -1";
494
495 print "379. searchAtIndex([1..N], 378) = -1";
495
496 print "380. searchAtIndex([1..N], 379) = -1";
496
497 print "381. searchAtIndex([1..N], 380) = -1";
497
498 print "382. searchAtIndex([1..N], 381) = -1";
498
499 print "383. searchAtIndex([1..N], 382) = -1";
499
500 print "384. searchAtIndex([1..N], 383) = -1";
500
501 print "385. searchAtIndex([1..N], 384) = -1";
501
502 print "386. searchAtIndex([1..N], 385) = -1";
502
503 print "387. searchAtIndex([1..N], 386) = -1";
503
504 print "388. searchAtIndex([1..N], 387) = -1";
504
505 print "389. searchAtIndex([1..N], 388) = -1";
505
506 print "390. searchAtIndex([1..N], 389) = -1";
506
507 print "391. searchAtIndex([1..N], 390) = -1";
507
508 print "392. searchAtIndex([1..N], 391) = -1";
508
509 print "393. searchAtIndex([1..N], 392) = -1";
509
510 print "394. searchAtIndex([1..N],
```

Many (but typically not all) automatic proof attempts may succeed.



# Example: Linear Search

Does the quantified loop invariant initially hold?

RISCTP

Proof - Without Type-Checking Theorems - Method: SMT + MESON Timeout [s]: 5 : Multi-Threaded n Threads: 4 :

Expand:  Axioms:  Int\*  Maps:  Data Equality:  Off  Low  Med  High  Max SMT:  Off  Min  Max Display:  Problems  Proofs  Search Limit:  Depth Size: 4 :  Iterate  Single Goal



**Proof Status: Success**

**Proof Output**

**Input File**

**Proof Problem**

**[?] Problem Simplification:**

```
[1] search_0_LoopOn1(Array[2],2) (rule [V=R | 3-L] on the goal)
{:} search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on the goal)
[1] search_0_LoopOn1(Array[2],2) (rule [V|R | 3-L] on the goal)
{:} search_0_LoopOn1(Array[2],2) (rule [V|R | 3-L] on the goal)
{:} search_0_LoopOn1(Array[2],2) (rule [def] on the pool)
[1] search_0_LoopOn1(Array[2],2) (rule [V|R | 3-L] on the goal)
{:} search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on the goal)
[1] search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on the goal)
{:} search_0_LoopOn1(Array[2],2) (rule [def] on [48])
[1] search_0_LoopOn1(Array[2],2) (rule [def] on [49])
[1] search_0_LoopOn1(Array[2],2) (rule [def] on [53])
[1] search_0_LoopOn1(Array[2],2) (rule [def] on [53])
{:} search_0_LoopOn1(Array[2],2) (rule [def] on [53])
[1] search_0_LoopOn1(Array[2],2) (rule [def] on [55])
[1] search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on [55])
{:} search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on [57])
[1] search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on [57])
{:} search_0_LoopOn1(Array[2],2) (rule [~R | V=R | A-L] on [59])
{:} search_0_LoopOn1(Array[2],2) (rule [~L-L] on [59] and [60])
{:} search_0_LoopOn1(Array[2],2) (rule [T-R | A-L] on [59]) closes the p
```

(Ww hide universally quantified knowledge)

```
1:[S0+1]~>0(0+1,1)
2:[S1+0]~>0(1+0,1)

45:[S-1+0]~>0(1) < 0
48:N0(type) Nat:type(MN)
49:N0(type) Nat:type(M)
53:[`search_0_LoopOn1(Array[2],2),1] array:type(x0)
54:[`search_0_LoopOn1(Array[2],2),1] elem:type(x0)
```

goal[`,search\_0\_LoopOn1(Array[2],2),2] index:0~>0(1) ≤ j 0 ≤ i ≤ N ⇒ (0 ≤ j) × (j < 0) ⇒ (~>0(x0),x0)]

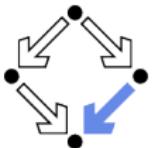
**[?] Subproblems:**

**[?] Clause Forms:**

**[?] Proofs:**

**[?] Proof Search:**

Proof method MESON: proof problem is already closed by simplification.



# Example: Linear Search

Does the quantified loop invariant initially hold?

(We hide universally quantified knowledge)

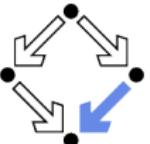
```
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_LoopOp1(Array[§],§).2.1.1] 0 ≤ x§
56:['_search_0_LoopOp1(Array[§],§).2.1.2] x§ ≤ M
57:['_search_0_LoopOp1(Array[§],§).2.2.1.1] 0 ≤ (j§+1)
58:['_search_0_LoopOp1(Array[§],§).2.2.1.2] j§ ≤ N
59:['_search_0_LoopOp1(Array[§],§).2.2.2.1.1] 0 ≤ j§
60:['_search_0_LoopOp1(Array[§],§).2.2.2.1.2] j§ < 0
```

---

```
goal:['_search_0_LoopOp1(Array[§],§).2.2.2.2] ¬'=§0'(a§[j§],x§)
```

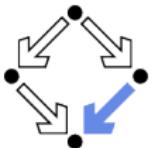
In the next (and final) step, it is recognized that the assumptions  $0 \leq j§$  and  $j§ \leq 0$  are inconsistent.

## Example: Linear Search



Is the quantified loop invariant preserved by the first conditional branch?

Problem is closed by simplification, proof search, and SMT solving.



# Example: Linear Search

Is the quantified loop invariant preserved by the first conditional branch?

Goal:  $\neg'=\$0'([](a\$,\_j\$),[](a\$,i\$)) \quad [ \_search\_0\_LoopOp6(Array[Z],Z)' . 2.2.2.2.1.1.2 ]$  (proof depth: 0, proof size: 1)

Goal:  $\neg'=\$0'([](a\$,\_j\$),[](a\$,i\$))$

To prove the goal, we assume its negation

[1]  $'=\$0'([](a\$,\_j\$),[](a\$,i\$))$

and show a contradiction. For this, consider knowledge  $[ \_search\_0\_LoopOp6(Array[Z],Z)' . 2.2.2.2.1.1.2 ]$  with the following instance:

$\forall j@113:index. \leq(0,+(j@113,1)) \wedge \leq(j@113,N\$) \wedge \leq(0,j@113) \wedge <(j@113,i\$) \wedge '=\$0'([](a\$,\_j@113),[](a\$,i\$)) \rightarrow \perp$

Assumption [1] matches the literal  $'=\$0'([](a\$,\_j@113),[](a\$,i\$))$  on the left side of this clause by the following substitution:

$j@113 \rightarrow j\$$

Therefore, applying this substitution and dropping the literal, we know:

$\leq(0,+(j\$,1)) \wedge \leq(j\$,N\$) \wedge \leq(0,j\$) \wedge <(j\$,i\$) \rightarrow \perp$

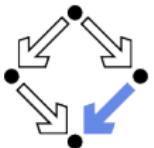
Therefore, to show a contradiction, we prove this subgoal:

$\leq(0,+(j\$,1)) \wedge \leq(j\$,N\$) \wedge \leq(0,j\$) \wedge <(j\$,i\$)$

SUCCESS: goal  $\neg'=\$0'([](a\$,\_j\$),[](a\$,i\$)) \quad [ \_search\_0\_LoopOp6(Array[Z],Z)' . 2.2.2.2.1.1.2 ]$  has been proved with the following substitution:

$j@113 \rightarrow j\$$

Invariant has to be instantiated with constant  $j\$$  for variable  $j$ .



# Example: Linear Search

Is the quantified loop invariant preserved by the first conditional branch?

Goal:  $\leq(0, +(j\$1))$  (proof depth: 1, proof size: 2)

Goal:  $\leq(0, +(j\$1))$

Assumptions:

[1] ' $=\leq 0'$ ( $[](a\$, j\$), [](a\$, i\$)$ )

The goal has been proved by the SMT solver: the solver states by the output

unsat

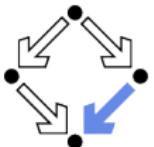
the unsatisfiability of the negated goal in conjunction with this knowledge:

$[_\text{search\_0\_LoopOp6}(\text{Array}[\mathbb{Z}], \mathbb{Z})'.2.2.2.2.2.2.2.1.1] \leq(0, j\$)$

SUCCESS: goal  $\leq(0, +(j\$1))$  has been proved with the following substitution:

$j@113 \rightarrow j\$$

Option “SMT: Med”: subgoals are closed by the SMT solver.



# Example: Linear Search

Is the quantified loop invariant preserved by the first conditional branch?

Proof problem: '\_search\_0\_LoopOp6(Array[Z],Z)'

The problem has been closed by the SMT solver: the solver states by the output

unsat

the unsatisfiability of these clauses that arise from the negation of the theorem to be proved:

```
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.1.1.2]  $\forall j:\text{index}.$   $\leq(0,+(j,1)) \wedge \leq(j,N\$) \wedge \leq(0,j) \wedge <(j,i\$) \wedge '=\$0'([](a\$,j),[](a\$,i\$)) \rightarrow \perp$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.1.2]  $\leq(j\$,N\$)$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.1.1]  $\leq(0,j\$)$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.1.2]  $<(j\$,i\$)$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.2.2]  $'=\$0'([](a\$,j\$),[](a\$,i\$))$ 
```

In more detail, the solver states the unsatisfiability of these clause instances:

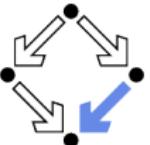
```
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.1.1.2.1]  $\leq(0,+(j\$,1)) \wedge \leq(j\$,N\$) \wedge \leq(0,j\$) \wedge <(j\$,i\$) \wedge '=\$0'([](a\$,j\$),[](a\$,i\$)) \rightarrow \perp$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.1.2]  $\leq(j\$,N\$)$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.2.1.1]  $\leq(0,j\$)$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.2.1.2]  $<(j\$,i\$)$ 
['_search_0_LoopOp6(Array[Z],Z)' .2.2.2.2.2.2.2.2]  $'=\$0'([](a\$,j\$),[](a\$,i\$))$ 
```

Thus the theorem is valid.

-----  
SUCCESS: goal '\_search\_0\_LoopOp6(Array[Z],Z)' has been proved.

Option “SMT: Max”: a proof outline is produced by the SMT solver.

## Example: Linear Search



Is quantified loop invariant preserved by the second conditional branch?

RISCTP

Prove Without Type-Checking Theorems - Method: SMT + MESON Timeout: 6s - Multi-Threaded: 4 Threads: 4

Expand: 6 Axioms: 0 Ints: 0 Maps: 0 Data Equality: Off Low Med High Max SMT: Off Min Med Max Display: - Problems: 0 Proofs: Search Limit: Depth Size: 4 : Iterate Single Goal

**Proof Status: Success**

**Prover Output:**

**Input File:**

**Proof Problem:**

**Problem Simplification:**

[1+] search 0 LoopOp?Array([Z],2) rule [V-R] [-l-] on the goal

**Subproblems:**

- 1. 'search 0 LoopOp?Array([Z],2)'

**Clauses Formulas:**

- 1. 'search 0 LoopOp?Array([Z],2)'

**Proofs:**

1. [1+] search 0 LoopOp?Array([Z],2) 0 (success)
  - (-) search 0 LoopOp?Array([Z],2) 2.2.2.2.2.2.2 (success)
  - (-) 'search 0 LoopOp?Array([Z],2) 2.2.2.2.2.2.2 (iteration 1) (success)
    - ↑⇒#0@11h(0x1): search 0 LoopOp?Array([Z],2.2.2.2.1.1.1) (success)
      - [+] #0@11h(0x1) (success)

**Proof Search:**

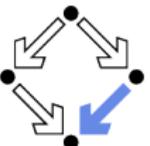
```

26.[!port2] VcInt(yInt, [(y <= x) | (y < x)]) 
27.[!port1] VcInt(yInt, [(y <= x) | (y = y)]) 
28.[!port2] VcInt(yInt, [(y <= x) | (y <= y) | (y >= 0) | (y = 0)]) 
29.[!port1] VcInt(yInt, [(y <= x) | (y >= 0)]) 
30.[!port2] VcInt(yInt, [(y <= x) | (y >= 0) | (y = 0)]) 
31.[!port1] VcInt(yInt, [(y <= x) | (y >= 0) | (y = 0) | (y < 1)]) 
32.[!port1] VcInt(yInt, [(y <= x) | (y >= 0) | (y = 1)]) 
33.[!port1] VcInt(yInt, [(y <= y) | (y < x)]) 
34.[!port1] VcInt(yInt, [(y <= y) | (y <= y)]) 
35.[!port1] VcInt(yInt, [(y <= y) | (y <= 1)]) 
36.[!port1] VcInt(yInt, [(y <= y)]) 
37.[!port1] VcInt(yInt, [(y <= y) | (y <= 0)]) 
38.[!port1] VcInt(yInt, [(y <= y) | (y <= 0) | (y = 1)]) 
39.[!port1] VcInt(yInt, [(y <= y) | (y <= 0) | (y = 1)]) 
40.[!port1] VcInt(yInt, [(y <= y) | (y <= 0) | (y = 0)]) 
41.[!port1] VcInt(yInt, [(y <= y) | (y <= 0) | (y = 0)]) 
42.[!posy] VcInt(yInt, [(y <= y) | (0 < y)]) 
43.[!posy] VcInt(yInt, [(y <= y) | (0 < y)]) 
44.[!posy] 0 < 1 
45.[!posy] 0 < 0(1) < 0 
46.[!posy] 0 < 0(1) < x(4 < x) 
47.[!posy] VcInt(yInt, [(y <= x)]) 
48.[!NTypey] y <= N 
49.[!NTypey] 0 <= M 
50.msp0109[Vm1Map[Int,Int,2]:Map[Int,Int]](VcInt,~>0)(m1[1],m2[0])~>~1(m1,m2) 
51.msp0110[Vm1Map[Int,Int,1]:Map[Int,Int]](VcInt,~>1)(m1[0],m2,vInt,~>1)(m1[0],0)~>~1(m1[0],v) 
52.msp0111[Vm1Map[Int,Int,1]:Map[Int,Int]](VcInt,~>0)(m1[0],m2,vInt,~>0)(m1[0],0)~>~1(m1[0],v) 
53.msp0112[Vm1Map[Int,Int,1]:Map[Int,Int]](VcInt,~>1)(m1[0],m2,vInt,~>1)(m1[0],0)~>~1(m1[0],v) 
54.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
55.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
56.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
57.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
58.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
59.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
60.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
61.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
62.[!search 0 LoopOp?Array([Z],2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
63.[!search 0 LoopOp?Array([Z],2.2.2.2.2.2.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
64.[!search 0 LoopOp?Array([Z],2.2.2.2.2.2.1.1.1)](VcInt,~>0)(m1[0],m2[0])~>~1(m1[0],m2[0]) 
goal?_search 0 LoopOp?Array([Z],2.2.2.2.2.2.2)~>~1(m1[0],m2[0])

```

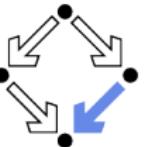
Problem is closed by simplification, proof search, and SMT solving.

## Example: Linear Search



Is quantified loop invariant preserved by the second conditional branch?

Proof with knowledge  $j \leq i$  is split into one case  $j = i$  (which is closed by simplification) and one case  $j < i$  (which is closed by proof search as in the first conditional branch).



# Example: Linear Search

Is result correct?

RISCTP

Proof Without Type-Checking Theorems Method: SMT  MESON Timeout [s]: 5 Multi-Threaded: 4 Threads: 4

**Proof Status:** Success

**Proof Output**

**Input File**

**Proof Problem**

[ $\vdash$ ] Problem Simplification:  
[ $\vdash$ ] search\_0\_CorrOp0(Array[Z]) (rule (R-3) on the goal)

[ $\vdash$ ] Subproblems:  
1. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,1)  
2. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,2)  
3. search\_0\_CorrOp0(Array[Z],1,1,1,2,1,2)  
4. search\_0\_CorrOp0(Array[Z],1,1,2,1,2,1)  
5. search\_0\_CorrOp0(Array[Z],1,1,2,2,2)  
6. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,1)  
7. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,2)  
8. search\_0\_CorrOp0(Array[Z],2,1,1,1,2,1,2)  
9. search\_0\_CorrOp0(Array[Z],2,1,1,2,1,2,1)  
10. search\_0\_CorrOp0(Array[Z],2,1,2,1,2,2,2)

[ $\square$ ] Clause Forms:  
1. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,1)  
2. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,2)  
3. search\_0\_CorrOp0(Array[Z],1,1,1,2,1,2)  
4. search\_0\_CorrOp0(Array[Z],1,1,2,1,2,1)  
5. search\_0\_CorrOp0(Array[Z],1,1,2,2,2)  
6. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,1)  
7. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,2)  
8. search\_0\_CorrOp0(Array[Z],2,1,1,1,2,1,2)  
9. search\_0\_CorrOp0(Array[Z],2,1,1,2,1,2,1)  
10. search\_0\_CorrOp0(Array[Z],2,1,2,1,2,2,2)

[ $\square$ ] Proofs:  
1. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,1,1,2,1) (success)  
2. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,1,1,2,2) (success)  
3. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,1,2,1,2) (success)  
4. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,2,1,2,1) (success)  
5. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,2,2,2) (success)  
6. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,1) (success)  
7. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,2) (success)  
8. search\_0\_CorrOp0(Array[Z],2,1,1,1,2,1,2) (success)  
9. search\_0\_CorrOp0(Array[Z],2,1,1,2,1,2,1) (success)  
10. search\_0\_CorrOp0(Array[Z],2,1,2,1,2,2,2) (success)

RISCTP Algorithm Language (RISCAL); RISCTP Theorem Prover

Proof Status: Success

Proof Output

Input File

Proof Problem

[ $\vdash$ ] Problem Simplification:  
[ $\vdash$ ] search\_0\_CorrOp0(Array[Z]) (rule (R-3) on the goal)

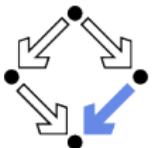
[ $\vdash$ ] Subproblems:  
1. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,1)  
2. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,2)  
3. search\_0\_CorrOp0(Array[Z],1,1,1,2,1,2)  
4. search\_0\_CorrOp0(Array[Z],1,1,2,1,2,1)  
5. search\_0\_CorrOp0(Array[Z],1,1,2,2,2)  
6. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,1)  
7. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,2)  
8. search\_0\_CorrOp0(Array[Z],2,1,1,1,2,1,2)  
9. search\_0\_CorrOp0(Array[Z],2,1,1,2,1,2,1)  
10. search\_0\_CorrOp0(Array[Z],2,1,2,1,2,2,2)

Clause Forms:  
1. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,1)  
2. search\_0\_CorrOp0(Array[Z],1,1,1,1,2,2)  
3. search\_0\_CorrOp0(Array[Z],1,1,1,2,1,2)  
4. search\_0\_CorrOp0(Array[Z],1,1,2,1,2,1)  
5. search\_0\_CorrOp0(Array[Z],1,1,2,2,2)  
6. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,1)  
7. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,2)  
8. search\_0\_CorrOp0(Array[Z],2,1,1,1,2,1,2)  
9. search\_0\_CorrOp0(Array[Z],2,1,1,2,1,2,1)  
10. search\_0\_CorrOp0(Array[Z],2,1,2,1,2,2,2)

Proofs:  
1. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,1,1,2,1) (success)  
2. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,1,1,2,2) (success)  
3. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,1,2,1,2) (success)  
4. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,2,1,2,1) (success)  
5. [ $\vdash$ ] search\_0\_CorrOp0(Array[Z],1,1,2,2,2) (success)  
6. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,1) (success)  
7. search\_0\_CorrOp0(Array[Z],2,1,1,1,1,2,2) (success)  
8. search\_0\_CorrOp0(Array[Z],2,1,1,1,2,1,2) (success)  
9. search\_0\_CorrOp0(Array[Z],2,1,1,2,1,2,1) (success)  
10. search\_0\_CorrOp0(Array[Z],2,1,2,1,2,2,2) (success)

RISCTP Algorithm Language (RISCAL); RISCTP Theorem Prover

Problem is decomposed into five subproblems closed by proof search.



# Example: Linear Search

Is result correct?

```
proc search(a:array, x:elem): index
ensures
(result = -1  $\wedge$   $\forall i:index. 0 \leq i \wedge i < N \Rightarrow a[i] \neq x$ )  $\vee$ 
( $0 \leq result \wedge result < N \wedge a[result] = x \wedge \forall i:index. 0 \leq i \wedge i < result \Rightarrow a[i] \neq x$ );
```

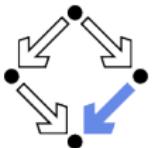
(We hide universally quantified knowledge)

```
1:[\$0+1] '=\$0'(0+1,1)
2:[\$1+0] '=\$0'(1+0,1)
44:[\$0<1] 0 < 1
45:[\$-1<0] '-\$0'(1) < 0
48:[N\$type] 0 ≤ N
49:[M\$type] 0 ≤ M
55:['_search_0_CorrOp0(Array[Z],Z)'2.1.1] 0 ≤ x\$̄
56:['_search_0_CorrOp0(Array[Z],Z)'2.1.2] x\$̄ ≤ M
57:['_search_0_CorrOp0(Array[Z],Z)'2.2.1.1] 0 ≤ (i\$̄+1)
58:['_search_0_CorrOp0(Array[Z],Z)'2.2.1.2] i\$̄ ≤ N
59:['_search_0_CorrOp0(Array[Z],Z)'2.2.2.1.1] 0 ≤ (r\$̄+1)
60:['_search_0_CorrOp0(Array[Z],Z)'2.2.2.1.2] r\$̄ ≤ N
61:['_search_0_CorrOp0(Array[Z],Z)'2.2.2.2.1.1] 0 ≤ i\$̄
63:['_search_0_CorrOp0(Array[Z],Z)'2.2.2.2.1.1.2] '=\$0'(r\$̄,-\$0'(1))  $\vee$  (('=\$0'(r\$̄,i\$̄)  $\wedge$  (i\$̄ < N))  $\wedge$  '=\$0'(a\$̄[r\$̄],x\$̄))
64:['_search_0_CorrOp0(Array[Z],Z)'2.2.2.2.1.2] -(i\$̄ < N)  $\wedge$  '=\$0'(r\$̄,-\$0'(1))
65:['_search_0_CorrOp0(Array[Z],Z)'2.2.2.2.2.1] -('=\$0'(r\$̄,-\$0'(1))  $\wedge$  ( $\forall i:index. (('-\$0'(1) \leq i) \wedge (i \leq N)) \Rightarrow (((0 \leq i) \wedge (i < N)) \Rightarrow (\neg'=\$0'(a\$̄[i],x\$̄))))$ )
```

---

goal:['\_search\_0\_CorrOp0(Array[Z],Z)'2.2.2.2.2.2] (((0 ≤ r\\$̄)  $\wedge$  (r\\$̄ < N))  $\wedge$  '=\\$0'(a\\$̄[r\\$̄],x\\$̄))  $\wedge$  ( $\forall i:index. (('-\$0'(1) \leq i) \wedge (i \leq N)) \Rightarrow (((0 \leq i) \wedge (i < N)) \Rightarrow (\neg'=\$0'(a\$̄[i],x\$̄))))$ )

At first, the decomposition yields the second part of the disjunction as the goal  
(with the negation of the first part as knowledge).



# Example: Linear Search

---

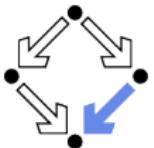
Is result correct?

$$(((0 \leq r\$) \wedge (r\$ < N)) \wedge '=\$0'(a\$[r\$], x\$)) \wedge (\forall i:\text{index}. (((\$0'(1) \leq i) \wedge (i \leq N)) \Rightarrow (((0 \leq i) \wedge (i < r\$)) \Rightarrow (\neg '=\$0'(a\$[i], x\$)))))$$

The further decomposition yields four subproblems with the following goals which are then decomposed into five open subproblems as follows:

- $(0 \leq r)$   $\rightsquigarrow$  2 subproblems, 1 closed, 1 open: subproblem 1.
- $(r < N)$   $\rightsquigarrow$  3 subproblems, 2 closed, 1 open: subproblem 2.
- $(a[r] = x)$   $\rightsquigarrow$  2 subproblems, 1 closed, 1 open: subproblem 3.
- $(\forall i: \dots a[i] \neq x)$   $\rightsquigarrow$  4 subproblems, 2 closed, 2 open: subproblems 4, 5.

We show the derivation and solution of subproblem 5.



# Example: Linear Search

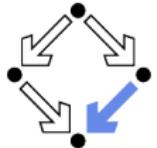
Is result correct?

(We hide universally quantified knowledge)

```

1:[\$0+1] '=\$0'(0+1,1)
2:[\$1+0] '=\$0'(1+0,1)
44:[\$0<1] 0 < 1
45:[\$-1<0] '-\$0'(1) < 0
48:[N\$type] 0 ≤ N
49:[M\$type] 0 ≤ M
55:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.1] 0 ≤ x\$̄
56:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.2] x\$̄ ≤ M
57:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.1.1] 0 ≤ (i\$̄+1)
58:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.1.2] i\$̄ ≤ N
59:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.1] 0 ≤ (r\$̄+1)
60:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.2] r\$̄ ≤ N
61:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.1] 0 ≤ i\$̄
63:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.2] '=\$0'(r\$̄,-\$0'(1)) ∨ (('=\$0'(r\$̄,i\$̄) ∧ (i\$̄ < N)) ∧ '=\$0'(a\$̄[r\$̄],x\$̄))
64:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.2] ¬((i\$̄ < N) ∧ '=\$0'(r\$̄,-\$0'(1)))
65:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1] ¬('=\$0'(r\$̄,-\$0'(1)) ∧ (∀i:index. ((('-\$0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < N)) ⇒ (¬'=\$0'(a\$̄[i],x\$̄))))))
goal:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2] ∀i:index. ((('-\$0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < N)) ⇒ (¬'=\$0'(a\$̄[i],x\$̄))))
```

The last of the four initial subproblems (the goal is to show that value  $x$  does not occur in array  $a$  at any index less than result  $r$ ).



# Example: Linear Search

Is result correct?

(We hide universally quantified knowledge)

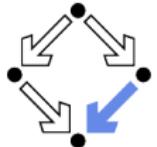
```

1:[\$0+1] '=\$0'(0+1,1)
2:[\$1+0] '=\$0'(1+0,1)
44:[\$0<1] 0 < 1
45:[\$-1<0] '-\$0'(1) < 0
48:[N\$type] 0 ≤ N
49:[M\$type] 0 ≤ M
55:[`_search_0_CorrOp0(Array[\$],\$).2.1.1] 0 ≤ x\$ 
56:[`_search_0_CorrOp0(Array[\$],\$).2.1.2] x\$ ≤ M
57:[`_search_0_CorrOp0(Array[\$],\$).2.2.1.1] 0 ≤ (i\$+1)
58:[`_search_0_CorrOp0(Array[\$],\$).2.2.1.2] i\$ ≤ N
59:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.1.1] 0 ≤ (r\$+1)
60:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.1.2] r\$ ≤ N
61:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.1.1.1] 0 ≤ i\$ 
63:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.1.1.2] '=\$0'(r\$, -\$0'(1)) ∨ (('=\$0'(r\$, i\$) ∧ (i\$ < N)) ∧ '=\$0'(a\$[r\$], x\$))
64:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.1.2] ¬((i\$ < N) ∧ '=\$0'(r\$, -\$0'(1)))
65:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.2.1] ¬(=\$0'(r\$, -\$0'(1)) ∧ (∀i:index. ((('-\$0'(1) ≤ i) ∧ (i ≤ N)) ⇒ ((0 ≤ i) ∧ (i < N)) ⇒ (¬'=\$0'(a\$[i], x\$)))))
66:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.2.2.1.1] 0 ≤ (i\$0+1)
67:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.2.2.2.1.2] i\$0 ≤ N
68:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.2.2.2.2.1.1] 0 ≤ i\$0
69:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.2.2.2.2.1.2] i\$0 < r\$

goal:[`_search_0_CorrOp0(Array[\$],\$).2.2.2.2.2.2.2.2] ¬'=\$0'(a\$[i\$0], x\$)

```

The subproblem after further decomposition; now a case split is going to be performed on disjunction formula 63.



# Example: Linear Search

Is result correct?

(We hide universally quantified knowledge)

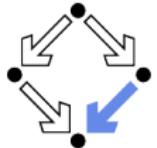
```

1:[\$0+1] '=\$0'(0+1,1)
2:[\$1+0] '=\$0'(1+0,1)
44:[\$0<1] 0 < 1
45:[\$-1<0] '-\$0'(1) < 0
48:[N\$type] 0 ≤ N
49:[M\$type] 0 ≤ M
55:[`_search_0_CorrOp0(Array[\$],\$)`.2.1.1] 0 ≤ x\$ 
56:[`_search_0_CorrOp0(Array[\$],\$)`.2.1.2] x\$ ≤ M
57:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.1.1] 0 ≤ (i\$+1)
58:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.1.2] i\$ ≤ N
59:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.1.1] 0 ≤ (r\$+1)
60:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.1.2] r\$ ≤ N
61:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.1.1.1] 0 ≤ i\$ 
63:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.1.1.2.2] ("=\$0'(r\$ ,i\$) ∧ (i\$ < N)) ∨ '=\$0'(a\$[r\$],x\$)
64:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.1.2] ¬((i\$ < N) ∧ '=\$0'(r\$ ,'-\$0'(1)))
65:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.2.1] ¬("=\$0'(r\$ ,'-\$0'(1)) ∧ (∀i:index. ((('-\$0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < N)) ⇒ ('=\$0'(a\$[i],x\$))))))
66:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.2.2.1.1] 0 ≤ (i\$0+1)
67:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.2.2.2.1.2] i\$0 ≤ N
68:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.2.2.2.2.1.1] 0 ≤ i\$0
69:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.2.2.2.2.1.2] i\$0 < r\$

goal:[`_search_0_CorrOp0(Array[\$],\$)`.2.2.2.2.2.2.2.2.2] ¬'=\$0'(a\$[i\$0],x\$)

```

The second case: result  $r$  equals loop variable  $i$  which is less than array length  $N$  and  $x$  occurs at index  $r$  in  $a$ .



# Example: Linear Search

Is result correct?

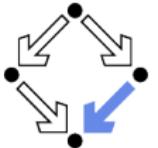
(We hide universally quantified knowledge)

```

1:[\$0+1] '=\$0'(0+1,1)
2:[\$1+0] '=\$0'(1+0,1)
44:[\$0<1] 0 < 1
45:[\$-1<0] '-\$0'(1) < 0
48:[N$type] 0 ≤ N
49:[M$type] 0 ≤ M
55:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.1] 0 ≤ a\$[i\$]
56:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.2] a\$[i\$] ≤ M
57:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.1] 0 ≤ (i\$+1)
58:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.2] i\$ ≤ N
59:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.1] 0 ≤ i\$ 
61:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.2.1.2] i\$ < N
62:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.2] ¬'=\$0'(i\$,-1)
63:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1] ¬'('=\$0'(i\$,'-\$0'(1)) ∧ (Vi:index. ((('-\$0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < N)) ⇒ (¬'=\$0'(a\$[i],a\$[i\$])))))
64:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.1.1] 0 ≤ (i\$0+1)
65:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.1.2] i\$0 ≤ N
66:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2.1.1] 0 ≤ i\$0
67:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2.1.2] i\$0 < i\$
```

goal:`\_search\_0\_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2] ¬'=\\$0'(a\\$[i\\$0],a\\$[i\\$])

After further simplification, another case split is performed on the negated conjunction formula 63 (equivalent to a disjunction of negated formulas).



# Example: Linear Search

Is result correct?

(We hide universally quantified knowledge)

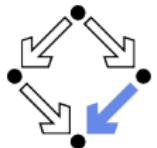
```

1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] -§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.1] 0 ≤ a§[i§]
56:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.2] a§[i§] ≤ M
57:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.1] 0 ≤ (i§+1)
58:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.2] i§ ≤ N
59:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.1] 0 ≤ i§
61:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.2.2.1.2] i§ < N
62:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.2] ¬'=§0'(i§,0-1)
63:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1.2] ¬(∀i:index. ((¬§0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < N)) ⇒ (¬'=§0'(a§[i],a§[i§]))))
64:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.1.1] 0 ≤ (i§0+1)
65:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.1.2] i§0 ≤ N
66:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2.1.1] 0 ≤ i§0
67:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2.1.2] i§0 < i§

goal:`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2.2] ¬'=§0'(a§[i§0],a§[i§])

```

The second case: given constant  $i§$ , array  $a$  holds at some index  $i$  greater equal 0 and less than  $N$  value  $a[i§]$ .



# Example: Linear Search

Is result correct?

(We hide universally quantified knowledge)

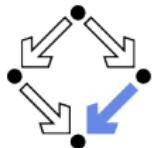
```

1:[\$0+1] '=\$0'(0+1,1)
2:[\$1+0] '=\$0'(1+0,1)
44:[\$0<1] 0 < 1
45:[\$-1<0] '-\$0'(1) < 0
48:[N\$type] 0 ≤ N
49:[M\$type] 0 ≤ M
55:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.1] 0 ≤ a\$[i\$]
56:[`_search_0_CorrOp0(Array[Z],Z)`.2.1.2] a\$[i\$] ≤ M
57:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.1.2] i\$ ≤ N
58:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.1] 0 ≤ i\$ 
60:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.1.2.1.2] i\$ < N
61:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.1.2] '-\$0'(i\$,-1)
62:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1.2.1.2] i\$5 ≤ N
63:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1.2.2.1.1] 0 ≤ i\$5
64:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1.2.2.1.2] i\$5 < N
65:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.1.2.2.2] '=\$0'(a\$[i\$5],a\$[i\$])
66:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.1.2] i\$0 ≤ N
67:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.1.1] 0 ≤ i\$0
68:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.1.2] i\$0 < i\$

goal:[`_search_0_CorrOp0(Array[Z],Z)`.2.2.2.2.2.2.2.2] '-\$0'(a\$[i\$0],a\$[i\$])

```

After further simplification, we have subproblem 5.



# Example: Linear Search

---

Is result correct?

```

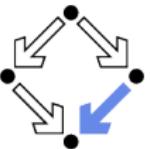
53:[_search_0_CorrOp0(Array[Z],Z)'.1.1] ∀x:Int. (((0 ≤ x) ∧ ((x+1) ≤ N)) ⇒ ((0 ≤ a§[x]) ∧ (a§[x] ≤ M)))
54:[_search_0_CorrOp0(Array[Z],Z)'.1.2] ∀x:Int. ((¬((0 ≤ x) ∧ ((x+1) ≤ N))) ⇒ '=§0'(a§[x],Int§undef))
55:[_search_0_CorrOp0(Array[Z],Z)'.2.1.1] 0 ≤ a§[i§]
56:[_search_0_CorrOp0(Array[Z],Z)'.2.1.2] a§[i§] ≤ M
57:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.1.2] i§ ≤ N
58:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.1.1.1] 0 ≤ i§
59:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.1.1.1.2] ∀j:index. (((0 ≤ (j+1)) ∧ (j ≤ N)) ⇒ (((0 ≤ j) ∧ (j < i§)) ⇒ (¬'=§0'(a§[j],a§[i§]))))
60:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.1.1.2.2.1.2] i§ < N
61:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.1.2] ¬'=§0'(i§,0-1)
62:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.1.2.1.2] i§5 ≤ N
63:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.1.2.2.1.1] 0 ≤ i§5
64:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.1.2.2.1.2] i§5 < N
65:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.1.2.2.2] '=§0'(a§[i§5],a§[i§])
66:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.1.2] i§0 ≤ N
67:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.1.1] 0 ≤ i§0
68:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.1.2] i§0 < i§

goal:[_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.2] ¬'=§0'(a§[i§0],a§[i§])

```

Subproblem 5 with the quantified formulas (except for the theory axioms).

## Example: Linear Search



## Is result correct?

RDC Algorithm Language (RIS-CAL): RISCTP Theorem Prover

**RISCTP**

Prove Without Type-Checking Theorems Method: SMT + MESON Timouts [h]: 5 Multi-Threaded: 4

Expand: 0 Axioms: 0 Ints: 0 Int# Maps: 0 Data Equality: Off Low Med High Max SMT: Off Min Med Max Display: Problems Proofs Search Limit: 0 Depth: Size: 4 Iterate Single Goal

**Proof Status: Success**

Prover Output

Input File

Proof Problem

[4] Problem Simplification:

[\*]: search 0 CorrQpDifray(Z) rule [N-R | N-L] on the goal

[4] Subproblems:

1. search 0 CorrQpDifray(Z) 2|1,1,1,1,1,1
2. search 0 CorrQpDifray(Z) 2|1,1,1,1,1,1,2
3. search 0 CorrQpDifray(Z) 2|1,1,1,2
4. search 0 CorrQpDifray(Z) 2|1,2,2,1
5. search 0 CorrQpDifray(Z) 2|2,2,2

[4] Clause Forms:

1. search 0 CorrQpDifray(Z) 2|1,1,1,1,1,1
2. search 0 CorrQpDifray(Z) 2|1,1,1,1,1,1,2
3. search 0 CorrQpDifray(Z) 2|1,1,1,2
4. search 0 CorrQpDifray(Z) 2|1,2,2,1
5. search 0 CorrQpDifray(Z) 2|2,2,2

[4] Proofs:

- 1. [\*]: search 0 CorrQpDifray(Z) 2|1,1,1,1,1,1 (Success)
- 2. [\*]: search 0 CorrQpDifray(Z) 2|1,1,1,1,1,1,2 (Success)
- 3. [\*]: search 0 CorrQpDifray(Z) 2|1,1,1,2 (Success)
- 4. [\*]: search 0 CorrQpDifray(Z) 2|1,2,2,1 (Success)
- 5. [\*]: search 0 CorrQpDifray(Z) 2|2,2,2 (Success)

[\*]: search 0 CorrQpDifray(Z) 2|2,2,2,2,2,2,2,2 (Success)

[\*]: search 0 CorrQpDifray(Z) 2|2,2,2,2,2,2,2,2,2,2 (Iteration) (Success)

[\*]: search 0 CorrQpDifray(Z) 2|2,2,2,2,2,2,2,2,2,2,2,2 (Success)

[\*]: (1|0,1|0,0,0) (Success)

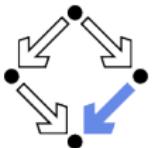
[\*]: (1|0,0|0,0) (Success)

[\*]: (1|0,0|0) (Success)

[\*]: (1|0,0) (Success)

SUCCESS: The proof has been completed.

The problem is closed by proof search and SMT solving.



# Example: Linear Search

Is result correct?

---

Goal:  $\neg'=\$0'([](a\$,\text{i\$0}),[](a\$,i\$)) \text{ ['}\_search\_0\_CorrOp0(Array[\mathbb{Z}],\mathbb{Z})\text{'.2.2.2.2.1.1.1.2] (proof depth: 0, proof size: 1)}$

---

Goal:  $\neg'=\$0'([](a\$,\text{i\$0}),[](a\$,i\$))$

To prove the goal, we assume its negation

[1]  $'=\$0'([](a\$,\text{i\$0}),[](a\$,i\$))$

and show a contradiction. For this, consider knowledge  $['\_search\_0\_CorrOp0(Array[\mathbb{Z}],\mathbb{Z})\text{'.2.2.2.2.1.1.1.2}]$  with the following instance:

$\forall j@113:\text{index}. \le(0,+(j@113,1)) \wedge \le(j@113,\text{N\$}) \wedge \le(0,j@113) \wedge <(j@113,i\$) \wedge '=\$0'([](a\$,\text{j@113}),[](a\$,i\$)) \rightarrow \perp$

Assumption [1] matches the literal  $'=\$0'([](a\$,\text{j@113}),[](a\$,i\$))$  on the left side of this clause by the following substitution:

$j@113 \rightarrow \text{i\$0}$

Therefore, applying this substitution and dropping the literal, we know:

$\le(0,+(i\$0,1)) \wedge \le(i\$0,\text{N\$}) \wedge \le(0,i\$0) \wedge <(i\$0,i\$) \rightarrow \perp$

Therefore, to show a contradiction, we prove this subgoal:

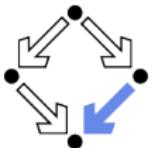
$\le(0,+(i\$0,1)) \wedge \le(i\$0,\text{N\$}) \wedge \le(0,i\$0) \wedge <(i\$0,i\$)$

---

SUCCESS: goal  $\neg'=\$0'([](a\$,\text{i\$0}),[](a\$,i\$)) \text{ ['}\_search\_0\_CorrOp0(Array[\mathbb{Z}],\mathbb{Z})\text{'.2.2.2.2.1.1.1.2]}$  has been proved with the following substitution:

$j@113 \rightarrow \text{i\$0}$

Invariant has to be instantiated with constant  $i\$0$  for variable  $j$ .



# Example: Linear Search

Is result correct?

Goal:  $\leq(0, +(\text{i\$0}, 1))$  (proof depth: 1, proof size: 2)

Goal:  $\leq(0, +(\text{i\$0}, 1))$

Assumptions:

[1] ' $=\$0'([](\text{a\$}, \text{i\$0}), [](\text{a\$}, \text{i\$}))$

The goal has been proved by the SMT solver: the solver states by the output

unsat

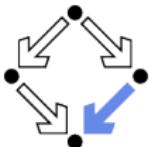
the unsatisfiability of the negated goal in conjunction with this knowledge:

$[_{\text{search\_0\_CorrOp0}}(\text{Array}[\mathbb{Z}], \mathbb{Z})'.2.2.2.2.2.2.2.1.1] \leq(0, \text{i\$0})$

-----  
SUCCESS: goal  $\leq(0, +(\text{i\$0}, 1))$  has been proved with the following substitution:

$j@113 \rightarrow \text{i\$0}$

Option “SMT: Med”: the subproblems are closed by the SMT solver.



# Example: Linear Search

Is result correct?

Proof problem: '\_search\_0\_CorrOp0(Array[Z],Z)'.2.2.2

The problem has been closed by the SMT solver: the solver states by the output

unsat

the unsatisfiability of these clauses that arise from the negation of the theorem to be proved:

```
'_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.1.1.2]  $\forall j:\text{index}.$   $\leq(0,+(j,1)) \wedge \leq(j,\text{N}\$) \wedge \leq(\emptyset,j) \wedge <(j,\text{i\$}) \wedge =\text{S0}'([](a\$),[],(a\$,i\$)) \Rightarrow \perp$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.1.2]  $\leq(i\$,N\$)$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.1.1]  $\leq(\emptyset,i\$0)$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.1.2]  $<(i\$,i\$)$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.2]  $=\text{S0}'([](a\$),i\$0),[],(a\$,i\$))$ 
```

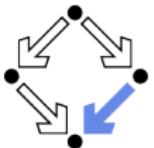
In more detail, the solver states the unsatisfiability of these clause instances:

```
'_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.1.1.2.2]  $\leq(0,+(i\$,1)) \wedge \leq(i\$,N\$) \wedge \leq(\emptyset,i\$0) \wedge <(i\$,i\$) \wedge =\text{S0}'([](a\$),i\$0),[],(a\$,i\$)) \Rightarrow \perp$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.1.2]  $\leq(i\$,N\$)$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.1.1]  $\leq(\emptyset,i\$0)$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.1.2]  $<(i\$,i\$)$ 
['_search_0_CorrOp0(Array[Z],Z)'.2.2.2.2.2.2.2.2.2]  $=\text{S0}'([](a\$),i\$0),[],(a\$,i\$))$ 
```

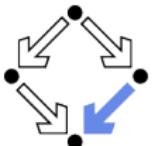
Thus the theorem is valid.

-----  
SUCCESS: goal '\_search\_0\_CorrOp0(Array[Z],Z)'.2.2.2 has been proved.

Option “SMT: Max”: a proof outline is produced by the SMT solver.



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



# Procedure Specifications

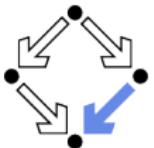
---

```
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

---

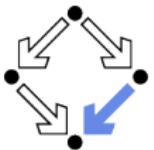
- Procedure specification:

```
global g
requires g ≥ 0 ∧ i > 0
ensures g₀ = g · i + o ∧ 0 ≤ o < i
o := p(i) { o := g%i; g := g/i }
```

- Proof obligation:

$$\begin{aligned}&\{g \geq 0 \wedge i > 0 \wedge i_0 = i \wedge g_0 = g\} \\&o := g \% i; \quad g := g/i \\&\{g_0 = g \cdot i_0 + o \wedge 0 \leq o < i_0\}\end{aligned}$$

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.

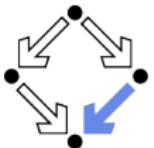
■ Original:

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

■ 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

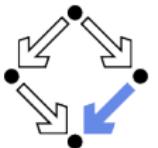
---

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

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

- $Pre[e/i]$  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.
- $Post[\dots]$  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.



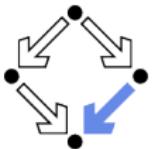
# Corresponding Predicate Transformers

---

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

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

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:

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

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

$\forall x', g' :$

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

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