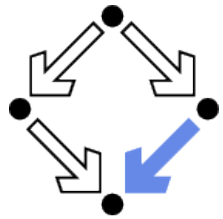
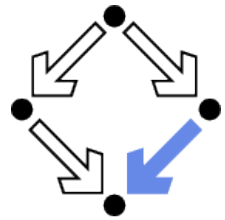
# Specifying and Verifying Programs

Wolfgang Schreiner
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



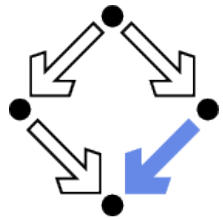
We will discuss two (closely interrelated) calculi.

- 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 $Q$.

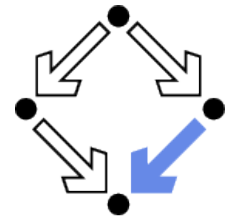
    $\{x = a \wedge y = b\} x := x + y \{x = a + y \wedge y = b\}$
- Predicate Transformers: $\mathrm{wp}(c, Q) = P$
  - If the execution of command $c$ shall yield a post-state with property $Q$, it must be executed in a pre-state with property $P$.

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

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



# The Hoare Calculus



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

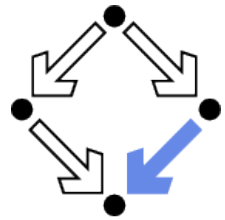
- "Hoare triple": $\{P\}\ c\ \{Q\}$
  - Logical propositions $P$ and $Q$, program command $c$.
  - The Hoare triple is itself a logical proposition.
  - The Hoare calculus gives rules for constructing true Hoare triples.
- Partial correctness interpretation of $\{P\}\ c\ \{Q\}$:

  "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



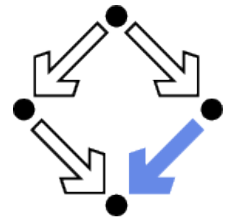
Hoare calculus rules are inference rules with Hoare triples as proof goals.

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

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

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

# Weakening and Strengthening

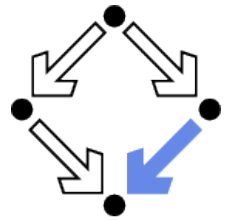


$$\frac{P \Rightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}$$

- Logical derivation: $\dfrac{A_1 \quad A_2}{B}$
  - Forward: If we have shown $A_1$ and $A_2$, then we have also shown $B$.
  - Backward: To show $B$, it suffices to show $A_1$ and $A_2$.
- Interpretation of above 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

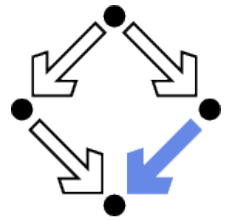


$$\{P\}\ \textbf{skip}\ \{P\} \qquad \{\text{true}\}\ \textbf{abort}\ \{\text{false}\}$$

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

Useful commands for reasoning and program transformations.

# Scalar Assignments

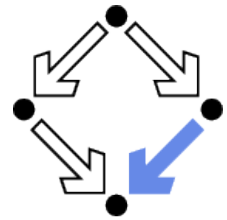


$$\{Q[e/x]\}\ x := e\ \{Q\}$$

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

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

# Array Assignments



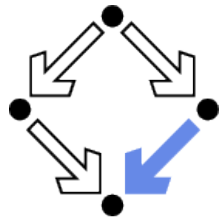
$$\{Q[a[i \mapsto e]/a]\}\ a[i] := e\ \{Q\}$$

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

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

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

# Array Assignments



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

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

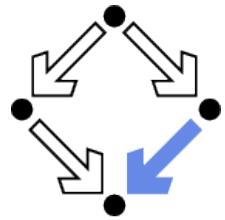
- Array Axioms

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

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

Get rid of "array update terms" when applied to indices.

# Command Sequences



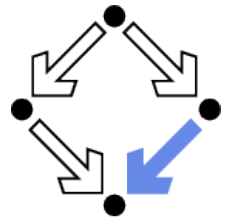
$$\frac{\{P\}\ c_1\ \{R\}\quad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

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

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

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

# Conditionals



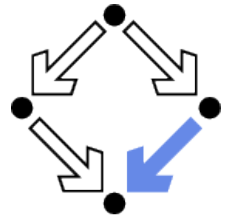
$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \quad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

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

- 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.

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

# Loops



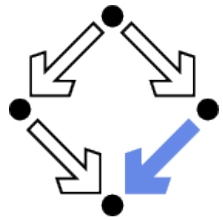
$$\{\text{true}\}\ \textbf{loop}\ \{\text{false}\} \qquad \frac{\{I \wedge b\}\ c\ \{I\}}{\{I\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{I \wedge \neg b\}}$$

- Interpretation:
  - The **loop** command does not terminate and thus trivially satisfies partial correctness.
    - Axiom implies $\{P\}$ **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)

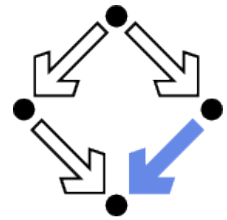


$$\frac{P \Rightarrow I \quad \{I \wedge b\}\ c\ \{I\} \quad (I \wedge \neg b) \Rightarrow Q}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{Q\}}$$

- 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.

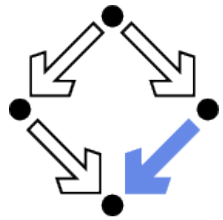
# Example



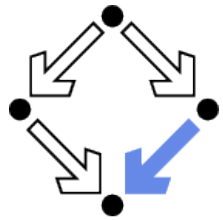
$$I :\Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$$

$$\frac{\begin{array}{c}(n \geq 0 \wedge s = 0 \wedge i = 1) \Rightarrow I \\ \{I \wedge i \leq n\}\ s := s + i; i := i + 1\ \{I\} \\ (I \wedge i \not\leq n) \Rightarrow s = \sum_{j=1}^{n} j\end{array}}{\{n \geq 0 \wedge s = 0 \wedge i = 1\}\ \textbf{while}\ i \leq n\ \textbf{do}\ (s := s + i; i := i + 1)\ \{s = \sum_{j=1}^{n} j\}}$$

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



# A Program Verification



- Verification of the following Hoare triple:

  $\{Input\}$ **while** $i \leq n$ **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:

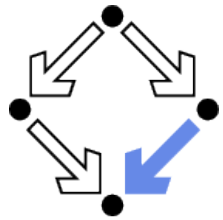
  $A :\Leftrightarrow Input \Rightarrow Invariant$

  $B :\Leftrightarrow Invariant \wedge i \leq n \Rightarrow Invariant[i + 1/i][s + i/s]$

  $C :\Leftrightarrow Invariant \wedge i \not\leq n \Rightarrow Output$

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

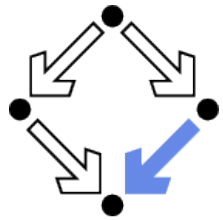
# RISCAL: Checking Program Execution



```
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.

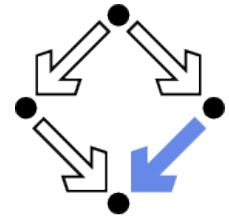
# RISCAL: Checking Verification Conditions



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

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

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

# Another Program Verification



Verification of the following Hoare triple:

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

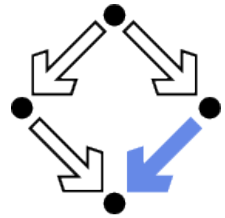
$i := 0; r := -1; n = |a|$
**while** $i < n \wedge r = -1$ **do**
  **if** $a[i] = x$
    **then** $r := i$
    **else** $i := i + 1$

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

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

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

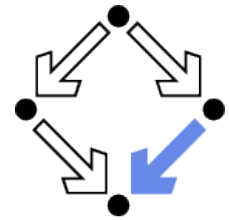
# RISCAL: Checking Program Execution



```
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 ∧ ∀i:index. 0 ≤ i ∧ i < N ⇒ a[i] ≠ x) ∨
          (0 ≤ result ∧ result < N ∧
            a[result] = x ∧ ∀i:index. 0 ≤ i ∧ i < result ⇒ a[i] ≠ x);
{
  var i:index = 0;
  var r:index = -1;
  while i < N ∧ r = -1 do
    invariant 0 ≤ i ∧ i ≤ N ∧ ∀j:index. 0 ≤ j ∧ j < i ⇒ a[j] ≠ x;
    invariant r = -1 ∨ (r = i ∧ i < N ∧ 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



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

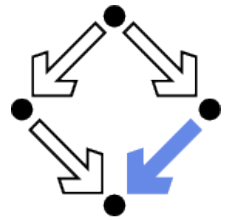
$$\begin{aligned} Output :\Leftrightarrow\ & a = olda \wedge x = oldx \wedge \\ & ((r = -1 \wedge \forall i : 0 \leq i < length(a) \Rightarrow a[i] \neq x) \vee \\ & (0 \leq r < length(a) \wedge a[r] = x \wedge \forall i : 0 \leq i < r \Rightarrow a[i] \neq x)) \end{aligned}$$

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

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

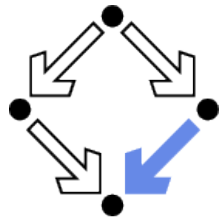
The verification conditions $A, B_1, B_2, C$ must be valid.

# RISCAL: Checking Verification Conditions



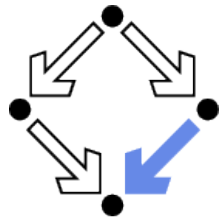
```
pred Input(i:index, r:index) ⇔ i = 0 ∧ r = -1;
pred Output(a:array, x:elem, i:index, r:index) ⇔
  (r = -1 ∧ ∀i:index. 0 ≤ i ∧ i < N ⇒ a[i] ≠ x) ∨
  (0 ≤ r ∧ r < N ∧ a[r] = x ∧ ∀i:index. 0 ≤ i ∧ i < r ⇒ a[i] ≠ x);
pred Invariant(a:array, x:elem, i:index, r:index) ⇔
  0 ≤ i ∧ i ≤ N ∧ (∀j:index. 0 ≤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



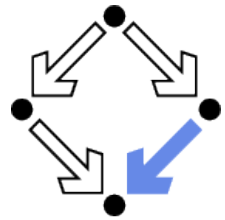
Implication of rule for command sequences and rule for assignments:

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

- 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.

| | | | | |
|---|---|---|---|---|
| $\{P\}$ | $\{P\}$ | $\{P\}$ | $\{P\}$ | $P \Rightarrow x = 4$ |
| 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 | $\{x + y = 15\}$ | $(\Leftrightarrow 3x = 15)$ | | |
| $\{z = 15\}$ | | $(\Leftrightarrow x = 5)$ | | |

# Weakest Preconditions

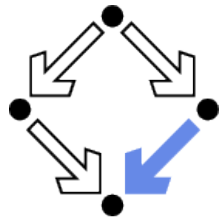


A calculus for "backward reasoning" (E.W. Dijkstra, 1975).

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

Verification is reduced to the calculation of weakest preconditions.

# Weakest Preconditions

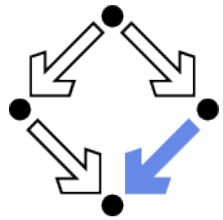


The weakest precondition of each program construct.

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

Loops represent a special problem (see later).

# Example

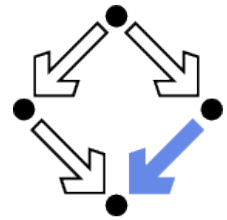


$$
\begin{aligned}
WP &= \mathrm{wp}(\textbf{if } \mathrm{a[i]} < \mathrm{x} \textbf{ then } \{\mathrm{a[i]} := \mathrm{a[i\text{-}1]};\ \mathrm{i} := \mathrm{i\text{-}1}\}, a[i+1] = b) \\
&= (a[i] < x \Rightarrow WP_1) \wedge (\neg(a[i] < x) \Rightarrow a[i+1] = b) \\
&\equiv (a[i] < x \Rightarrow WP_1)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

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

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

# Forward Reasoning



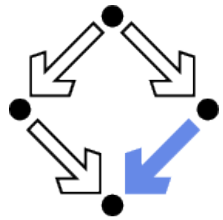
Sometimes, we want to derive a postcondition from a given precondition.

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

- 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 $e[x_0/x]$.

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

# Strongest Postcondition

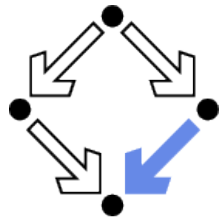


A calculus for forward reasoning.

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

Verification is reduced to the calculation of strongest postconditions.

# Strongest Postconditions

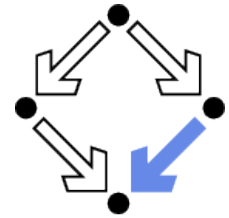


The strongest postcondition of each program construct.

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

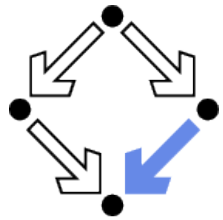
Forward reasoning as a (less-known) alternative to backward-reasoning.

# Example



$$
\begin{aligned}
SP &= \mathrm{sp}(\textbf{if } \mathrm{a[i]} < x \textbf{ then } \{\mathrm{a[i]} := \mathrm{a[i\text{-}1]};\ \mathrm{i} := \mathrm{i\text{-}1}\}, a[i] = b) \\
&= SP_1 \vee (a[i] = b \wedge \neg(a[i] < x)) \equiv SP_1 \vee (a[i] = b \wedge a[i] \geq x) \\
&\equiv SP_1 \vee (b \geq x \wedge a[i] = b) \\
SP_1 &= \mathrm{sp}(\{\mathrm{a[i]} := \mathrm{a[i\text{-}1]};\ \mathrm{i} := \mathrm{i\text{-}1}\}, a[i] = b \wedge a[i] < x) \\
&\equiv \mathrm{sp}(\{\mathrm{a[i]} := \mathrm{a[i\text{-}1]};\ \mathrm{i} := \mathrm{i\text{-}1}\}, a[i] = b \wedge b < x) \\
&= \mathrm{sp}(\mathrm{i{:=}i\text{-}1}, SP_2) \\
SP_2 &= \mathrm{sp}(\mathrm{a[i]{:=}a[i\text{-}1]}, a[i] = b \wedge b < x) \\
&= \mathrm{sp}(\mathrm{a{:=}a[i} \mapsto \mathrm{a[i\text{-}1]]}, a[i] = b \wedge b < x) \\
&= \exists a_0 : a_0[i] = b \wedge b < x \wedge a = a_0[i \mapsto a_0[i-1]] \\
&\equiv b < x \wedge \exists a_0 : a_0[i] = b \wedge a = a_0[i \mapsto a_0[i-1]] \\
&\equiv b < x \wedge a[i] = a[i-1] \\
SP_1 &\equiv \mathrm{sp}(\mathrm{i{:=}i\text{-}1}, b < x \wedge a[i] = a[i-1]) \\
&= \exists i_0 : b < x \wedge a[i_0] = a[i_0 - 1] \wedge i = i_0 - 1 \\
&\equiv b < x \wedge \exists i_0 : a[i_0] = a[i_0 - 1] \wedge i_0 = i + 1 \\
&\equiv b < x \wedge a[i+1] = a[(i+1) - 1] \equiv b < x \wedge a[i+1] = a[i] \\
SP &\equiv \underline{(b < x \wedge a[i+1] = a[i]) \vee (b \geq x \wedge a[i] = b)}
\end{aligned}
$$

# Hoare Calc. and Predicate Transformers



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

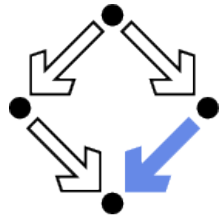
$$\{P\}\ c_1; \textbf{while}\ b\ \textbf{do}\ c; c_2\ \{Q\}$$

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

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

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?

$$\text{wp}(\textbf{loop}, Q) = \text{true}$$
$$\text{wp}(\textbf{while } b \textbf{ do } c, Q) = L_0(Q) \wedge L_1(Q) \wedge L_2(Q) \wedge \ldots$$

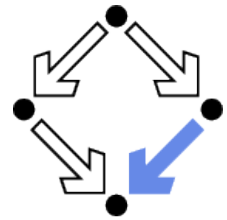
$$L_0(Q) = \text{true}$$
$$L_{i+1}(Q) = (\neg b \Rightarrow Q) \wedge (b \Rightarrow \text{wp}(c, L_i(Q)))$$

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

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

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

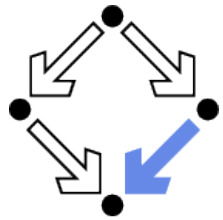
# Example



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

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

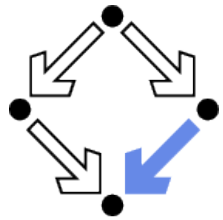
# Weakest Liberal Preconditions for Loops



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

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

# Preconditions for Loops with Invariants

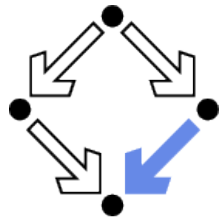


$$\text{wp}(\textbf{while } b \textbf{ do invariant } I;\ c^{x,\cdots}, Q) =$$
$$\quad \textbf{let } oldx = x, \ldots \textbf{ in}$$
$$\quad I \wedge (\forall x, \ldots : I \wedge b \Rightarrow wp(c, I)) \wedge$$
$$\quad (\forall x, \ldots : I \wedge \neg b \Rightarrow Q)$$

- Loop body $c$ only modifies variables $x, \ldots$
- Loop is annotated with invariant $I$.
  - May refer to new values $x, \ldots$ of variables after every iteration.
  - May refer to original values $oldx, \ldots$ 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** $(s := s + i; i := i + 1)$

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

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

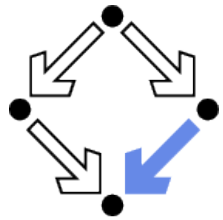
- Weakest precondition:

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

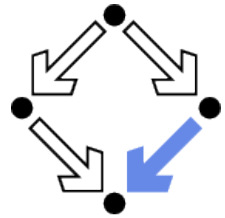
- Verification condition:

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

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



# Termination



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

$$\{\text{false}\}\ \textbf{loop}\ \{\text{false}\}$$

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

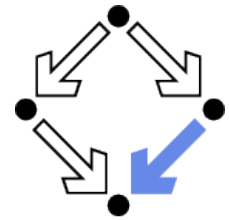
$$\frac{P \Rightarrow I \quad I \Rightarrow t \geq 0 \quad \{I \wedge b \wedge t = N\}\ c\ \{I \wedge t < N\} \quad (I \wedge \neg b) \Rightarrow Q}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{Q\}}$$

- 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

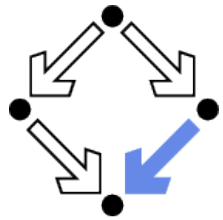


$$I :\Leftrightarrow s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$$
$$t := n - i + 1$$

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

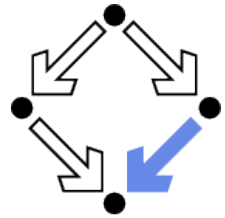
In practice, termination is easy to show (compared to partial correctness).

# Termination in RISCAL



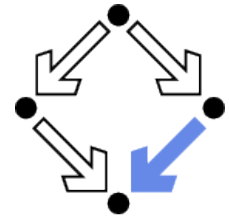
```
while i ≤ n do
  invariant s = ∑j:number with 1 ≤ j ∧ j ≤ i-1. j;
  invariant 1 ≤ i ∧ i ≤ n+1;
  decreases n+1-i;
{
  s := s+i;
  i := i+1;
}

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

# Termination 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



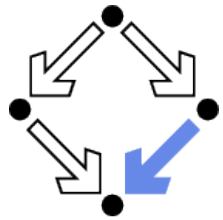
$\text{wp}(\textbf{loop}, Q) = \text{false}$
$\text{wp}(\textbf{while } b \textbf{ do } c, Q) = L_0(Q) \vee L_1(Q) \vee L_2(Q) \vee \ldots$

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

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

  $\text{if}_0 = \textbf{loop}$
  $\text{if}_{i+1} = \textbf{if } b \textbf{ then } (c; \text{if}_i)$

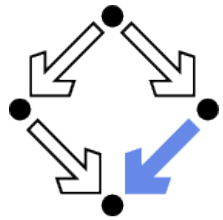
# Example



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

$$\begin{aligned}
L_0(Q) &= \mathrm{false} \\
L_1(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \ wp(i := i + 1, L_0(Q))) \\
&\Leftrightarrow (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \ \mathrm{false}) \\
&\Leftrightarrow i \not< n \wedge Q \\
L_2(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \ wp(i := i + 1, L_1(Q))) \\
&\Leftrightarrow (i \not< n \Rightarrow Q) \wedge \\
&\qquad (i < n \Rightarrow (i + 1 \not< n \wedge Q[i + 1/i])) \\
L_3(Q) &= (i \not< n \Rightarrow Q) \wedge (i < n \Rightarrow \ \mathrm{wp}(i := i + 1, L_2(Q))) \\
&\Leftrightarrow (i \not< n \Rightarrow Q) \wedge \\
&\qquad (i < n \Rightarrow \ ((i + 1 \not< n \Rightarrow Q[i + 1/i]) \wedge \\
&\qquad\qquad (i + 1 < n \Rightarrow (i + 2 \not< n \wedge Q[i + 2/i])))) \\
&\ldots
\end{aligned}$$

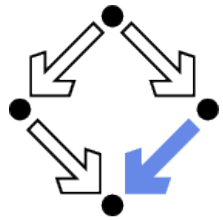
# Weakest Preconditions for Loops



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

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

# Weakest Precondition with Measures

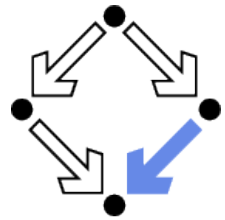


$$\begin{aligned}
&\mathrm{wp}(\textbf{while } b \textbf{ do invariant } I; \textbf{decreases } t;\ c^{x,\dots}, Q) = \\
&\quad \textbf{let } oldx = x, \dots \textbf{ in} \\
&\quad I \wedge (\forall x, \dots : I \wedge b \Rightarrow wp(c, I)) \wedge \\
&\quad (\forall x, \dots : I \wedge \neg b \Rightarrow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

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

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

# Example



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

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

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

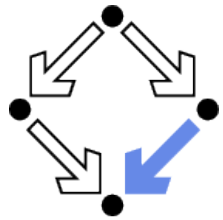
$t := n + 1 - i$

- Weakest precondition:

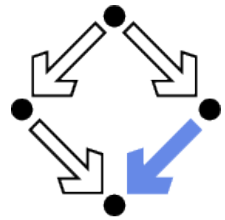
$$\begin{aligned}&\mathrm{wp}(\textbf{while } i \leq n \textbf{ do invariant } I;\ c^{s,i}, Q) = \\ &\quad \textbf{let } olds = s, oldi = i \textbf{ in} \\ &\quad I \wedge (\forall s, i : I \wedge i \leq n \Rightarrow I[s + i/s, i + 1/i]) \wedge \\ &\quad (\forall s, i : I \wedge \neg(i \leq n) \Rightarrow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

- Verification condition:

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



# Abortion



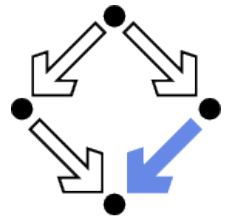
New rules to prevent abortion.

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

- 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.

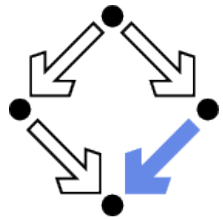
# Definedness of Expressions



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

Assumes that expressions have already been type-checked.

# Abortion



Slight modification of existing rules.

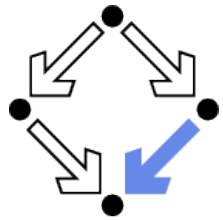
$$\frac{P \Rightarrow D(b) \quad \{P \wedge b\}\ c_1\ \{Q\} \quad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

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

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

Expressions must be defined in any context.

# Abortion

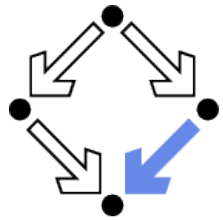


Similar modifications of weakest preconditions.

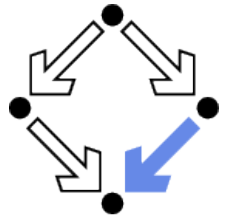
$$\text{wp}(\textbf{abort}, Q) = \text{false}$$
$$\text{wp}(x := e, Q) = Q[e/x] \wedge D(e)$$
$$\text{wp}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, Q) =$$
$$D(b) \wedge (b \Rightarrow \text{wp}(c_1, Q)) \wedge (\neg b \Rightarrow \text{wp}(c_2, Q))$$
$$\text{wp}(\textbf{if } b \textbf{ then } c, Q) = D(b) \wedge (b \Rightarrow \text{wp}(c, Q)) \wedge (\neg b \Rightarrow Q)$$
$$\text{wp}(\textbf{while } b \textbf{ do } c, Q) = (L_0(Q) \vee L_1(Q) \vee L_2(Q) \vee \ldots)$$

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

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



# 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 = ∑j:number with 1 ≤ j ∧ j ≤ n. j;
11 {
12   var s:result = 0;
13   var i:index  = 1;
14   while i ≤ n do
15     invariant s = ∑j:number with 1 ≤ j ∧ j ≤ i-1. j;
16     invariant 1 ≤ i ∧ 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 = ∑j:number with 1 ≤ j ∧ j ≤ n. j;
33
34 pred Invariant(n:number, s:result, i:index) ⇔
35   (s = ∑j:number with 1 ≤ j ∧ j ≤ i-1. j) ∧ 1 ≤ i ∧ i ≤ n+1;
36
37 fun Termination(n:number, s:result, i:index): number
38 = n+1-i;
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)

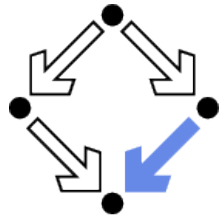
```
at unknown position:
  theorem is not true
ERROR encountered in execution (3 ms).
Executing _summation_0_PostSat(ℤ) with all 5 inputs.
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).
Executing _summation_0_PostNotTrivialAll(ℤ) with all 5 inputs.
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).
Executing _summation_0_PostNotTrivialSome().
Execution completed (0 ms).
Executing _summation_0_PostUnique(ℤ) with all 5 inputs.
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).
Executing _summation_0_CorrOp0(ℤ) with all 5 inputs.
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).
Executing _summation_0_LoopOp0(ℤ) with all 5 inputs.
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).
Executing _summation_0_LoopOp1(ℤ) with all 5 inputs.
Execution completed for ALL inputs (1 ms, 5 checked, 0 inadmissible).
Executing _summation_0_LoopOp2(ℤ) with all 5 inputs.
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).
Executing _summation_0_LoopOp3(ℤ) with all 5 inputs.
Execution completed for ALL inputs (2 ms, 5 checked, 0 inadmissible).
Executing _summation_0_LoopOp4(ℤ) with all 5 inputs.
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).
Executing _summation_0_LoopOp5(ℤ) with all 5 inputs.
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).
Executing _summation_0_PreOp0(ℤ) with all 5 inputs.
Execution completed for ALL inputs (3 ms, 5 checked, 0 inadmissible).
Executing _summation_0_PreOp1(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 trivia
    - 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

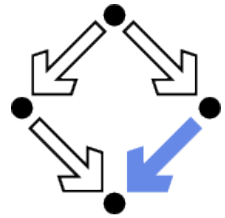


RISCAL splits Dijkstra's single condition $Input \Rightarrow wp(C, Output)$ into many "fine-grained" verification conditions:

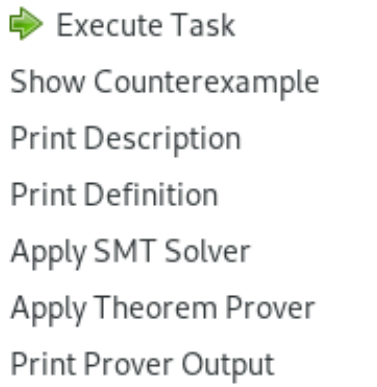
- Implementation preconditions
  - Well-definedness of commands and loop annotations.
  - One condition for every partial function/predicate application.
- Is result correct?
  - One condition for every `ensures` clause.
- Does loop invariant initially hold? Is loop invariant preserved?
  - Partial correctness.
  - 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



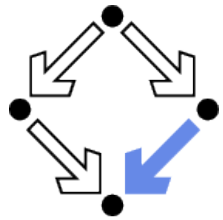
- Double-click a condition to have it checked.
  - Checked conditions turn from red to blue.
- Right-click a condition to see a pop-up menu.
  - Check verification condition (same as double-click)
  - Show variable values that invalidate condition.
  - Print relevant program information (e.g. invariant).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



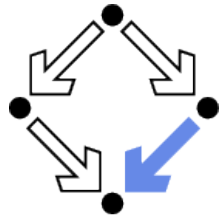

Example: is loop invariant preserved?

```
s = (∑j:number with (1 ≤ j) ∧ (j ≤ (i-1)). j)

theorem _summation_0_LoopOp3(n:number)
requires n ≥ 0;
 ⇔ ∀s:result,i:index. ((((s = (∑j:number with (1 ≤ j)∧(j ≤ (i-1)). j))
    ∧ ((1 ≤ i) ∧ (i ≤ (n+1)))) ∧ (i ≤ n)) ⇒
        (let s = s+i in (let i = i+1 in
          (s = (∑j:number with (1 ≤ j) ∧ (j ≤ (i-1)). j)))));
```

Important: check models with *small* type sizes.

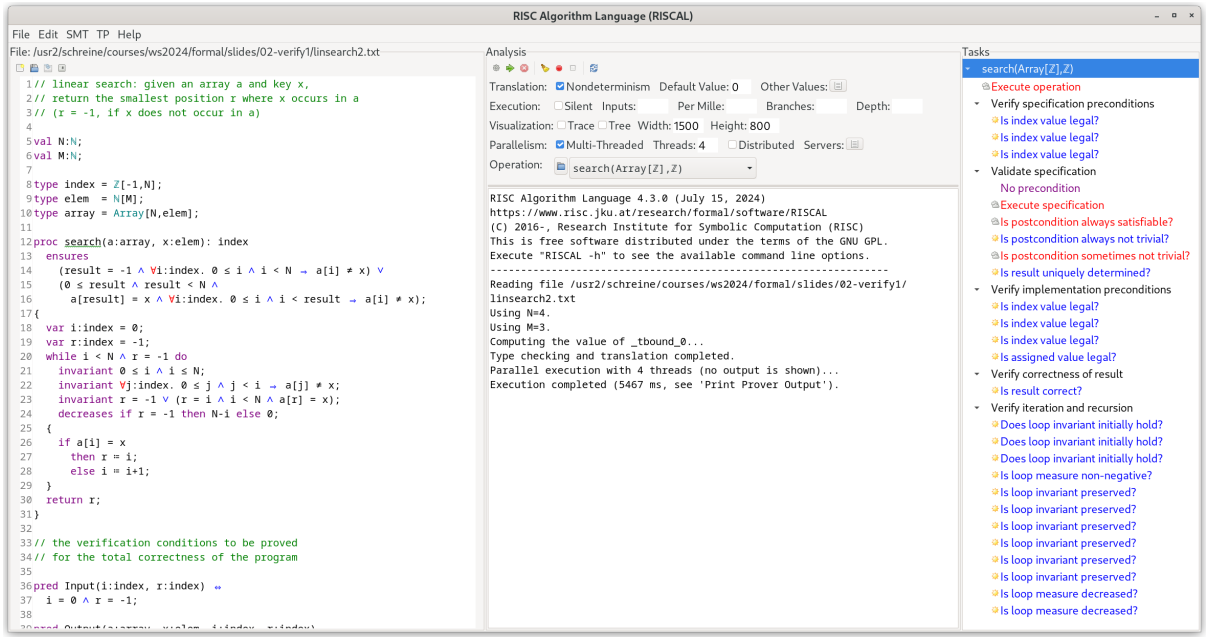


# Proving Verification Conditions



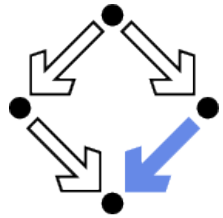
RISCAL also integrates the RISCTP interface to various theorem provers.

- 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.




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

# Example: Linear Search



Does the quantified loop invariant initially hold?

RISC Algorithm Language (RISCAL): RISCTP Theorem Prover

**RISCTP**

Prove | Without Type-Checking Theorems | Method: SMT MESON | Timeout (s): 5 | Multi-Threaded: Threads: 4

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

**Proof Status: Success**

Prover Output

Input File

Proof Problem

[-] **Problem Simplification:**

[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [∀-R | ∃-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [∀-R | ∃-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [def] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [∀-R | ∃-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on the goal)
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [def] on [48])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [def] on [49])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [def] on [53])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on [53])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [def] on [55])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on [55])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on [57])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [~-R | ~-L] on [57])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⇒-R | ∨-R | ∧-L] on [59])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [<≤-L] on [59] and [60])
[-] '_search_0_LoopOp1(Array[ℤ],ℤ)' (rule [⊤-R | ⊥-L] on [59] closes the p

[-] **Subproblems:**

[-] **Clause Forms:**

[-] **Proofs:**

[-] **Proof Search:**

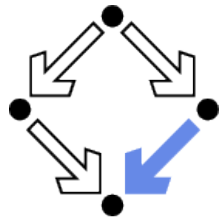
(We hide universally quantified knowledge)

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] 'Nat::type'(N)
49:[M§type] 'Nat::type'(M)
53:['_search_0_LoopOp1(Array[ℤ],ℤ)'.1] 'array::type'(a§)
54:['_search_0_LoopOp1(Array[ℤ],ℤ)'.2.1] 'elem::type'(x§)

goal:['_search_0_LoopOp1(Array[ℤ],ℤ)'.2.2] ∀j:index. ((('-§0'(1) ≤ j) ∧ (j ≤ N)) ⇒ (((0 ≤ j) ∧ (j < 0)) ⇒ (¬'=§0'(a§[j],x§))))

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)

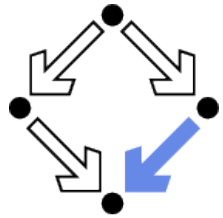
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_LoopOp1(Array[ℤ],ℤ)'.2.1.1] 0 ≤ x§
56:['_search_0_LoopOp1(Array[ℤ],ℤ)'.2.1.2] x§ ≤ M
57:['_search_0_LoopOp1(Array[ℤ],ℤ)'.2.2.1.1] 0 ≤ (j§+1)
58:['_search_0_LoopOp1(Array[ℤ],ℤ)'.2.2.1.2]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

---

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

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

# Example: Linear Search



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

RISC Algorithm Language (RISCAL): RISCTP Theorem Prover

**RISCTP**

Prove | Without Type-Checking Theorems | Method: SMT MESON | Timeout (s): 5 | Multi-Threaded: Threads: 4

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

**Proof Status: Success**

Prover Output

Input File

Proof Problem

[-] **Problem Simplification:**

[+] '_search_0_LoopOp6(Array[ℤ],ℤ)' (rule [∀-R | ∃-L] on the goal)

[-] **Subproblems:**

1. '_search_0_LoopOp6(Array[ℤ],ℤ)'

[-] **Clause Forms:**

1. '_search_0_LoopOp6(Array[ℤ],ℤ)'

[-] **Proofs:**

1. [-] '_search_0_LoopOp6(Array[ℤ],ℤ)' (success)
   - [-] '_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2 (success)
     - [-] '_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.2 (iteration 1) (success)
       - [-] ¬'=§0'([](a§,j§),[](a§,i§)),['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] (success)
         - [-] ≤(0,+(j§,1)) (success)
         - [-] ≤(j§,N§) (success)
         - [-] ≤(0,j§) (success)
         - [-] <(j§,i§) (success)

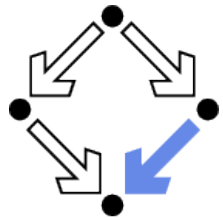
[-] **Proof Search:**

25:[§part1] ∀x:Int,y:Int. ((x ≤ y) ∨ (y < x))
26:[§part2] ∀x:Int,y:Int. (¬((x ≤ y) ∧ (y < x)))
27:[§def1≤] ∀x:Int,y:Int. (((x < y) ∨ '=§0'(x,y)) ⇒ (x ≤ y))
28:[§def2≤] ∀x:Int,y:Int. ((x ≤ y) ⇒ ((x < y) ∨ '=§0'(x,y){*}))
29:[§excl<] ∀x:Int,y:Int. (¬((x < y) ∧ '=§0'(x,y)))
30:[§excl2<] ∀x:Int,y:Int. (¬((y < x) ∧ '=§0'(x,y)))
31:[§+-1<] ∀x:Int,y:Int. ((<(x,y){*}) ⇒ ((¬(y < (x+1))) ∧ (¬((y-1) < x))))
32:[§+1≤] ∀x:Int,y:Int. ((x < y) ⇔ ((x+1) ≤ y))
33:[§+1<] ∀x:Int,y:Int. ((x ≤ y) ⇔ (x < (y+1)))
34:[§-1≤] ∀x:Int,y:Int. ((x < y) ⇔ (x ≤ (y-1)))
35:[§-1<] ∀x:Int,y:Int. ((x ≤ y) ⇔ ((x-1) < y))
36:[§x-1<x] ∀x:Int. ((x-1) < x)
37:[§x<x+1] ∀x:Int. (x < (x+1))
38:[§0<1≤] ∀x:Int. ((0 < x) ⇔ (1 ≤ x))
39:[§≤0<1] ∀x:Int. ((x ≤ 0) ⇔ (x < 1))
40:[§≤0] ∀x:Int. ((0 ≤ x) ⇒ ('-§0'(x) ≤ 0))
41:[§<0] ∀x:Int. ((0 < x) ⇒ ('-§0'(x) < 0))
42:[§x≤y] ∀x:Int,y:Int. ((x ≤ y) ⇒ (0 ≤ (y-x)))
43:[§x<y] ∀x:Int,y:Int. ((x < y) ⇒ (0 < (y-x)))
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
46:[§irrefl<] ∀x:Int. (¬(x < x))
47:[§refl≤] ∀x:Int. (x ≤ x)
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
50:[ext§109] ∀m1:Map[Int,Int],m2:Map[Int,Int]. ((∀k:Int. '=§0'(m1[k],m2[k])) ⇒ '=§1'(m1,m2))
51:[meqstore§112] ∀m:Map[Int,Int],k:Int,k0:Int,v:Int. ('=§0'(k,k0){*} ⇒ '=§0'((m with [k] = v)[k0],v))
52:[mneqstore§113] ∀m:Map[Int,Int],k:Int,k0:Int,v:Int. ((¬'=§0'(k,k0){*}) ⇒ '=§0'((m with [k] = v)[k0],m[k0]))
53:['_search_0_LoopOp6(Array[ℤ],ℤ)'.1.1] ∀x:Int. (((0 ≤ x) ∧ ((x+1) ≤ N)) ⇒ ((0 ≤ a§[x]) ∧ (a§[x] ≤ M)))
54:['_search_0_LoopOp6(Array[ℤ],ℤ)'.1.2] ∀x:Int. ((¬((0 ≤ x) ∧ ((x+1) ≤ N))) ⇒ '=§0'(a§[x],Int§undef))
55:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.1.1] 0 ≤ a§[i§]
56:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.1.2] a§[i§] ≤ M
57:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.1.2] i§ ≤ N
58:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.1] 0 ≤ i§
59:['_search_0_LoopOp6(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_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.2.1] i§ < N
61:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.1.2] j§ ≤ N
62:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] 0 ≤ j§
63:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.2] j§ < i§

goal:['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2] ¬'=§0'(a§[j§],a§[i§])

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: ¬'=§0'([](a§,j§),[](a§,i§)) ['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] (proof depth: 0, proof size: 1)**

Goal: ¬'=§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)'.2.2.2.2.1.1.1.2] with the following instance:

∀j@113:index. ≤(0,+(j@113,1)) ∧ ≤(j@113,N§) ∧ ≤(0,j@113) ∧ <(j@113,i§) ∧ '=§0'([](a§,j@113),[](a§,i§)) ⇒ ⊥

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

j@113 → j§

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

≤(0,+(j§,1)) ∧ ≤(j§,N§) ∧ ≤(0,j§) ∧ <(j§,i§) ⇒ ⊥

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

≤(0,+(j§,1)) ∧ ≤(j§,N§) ∧ ≤(0,j§) ∧ <(j§,i§)

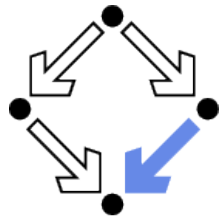
---

SUCCESS: goal ¬'=§0'([](a§,j§),[](a§,i§)) ['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] has been proved with the following substitution:

j@113 → j§

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

# Example: Linear Search



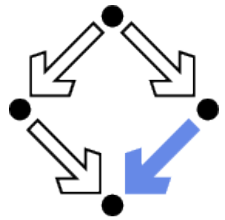
Is the quantified loop invariant preserved by the first conditional branch?

**Goal: ≤(0,+(j§,1)) (proof depth: 1, proof size: 2)**

```
Goal: ≤(0,+(j§,1))

Assumptions:

[1] '=§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:

['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.1.1] ≤(0,j§)
--------------------------------------------------------------------------------
SUCCESS: goal ≤(0,+(j§,1)) has been proved with the following substitution:

  j@113 → 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?

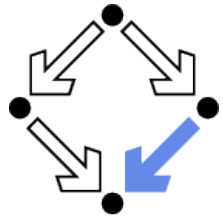
**Proof problem: '_search_0_LoopOp6(Array[ℤ],ℤ)'**

---

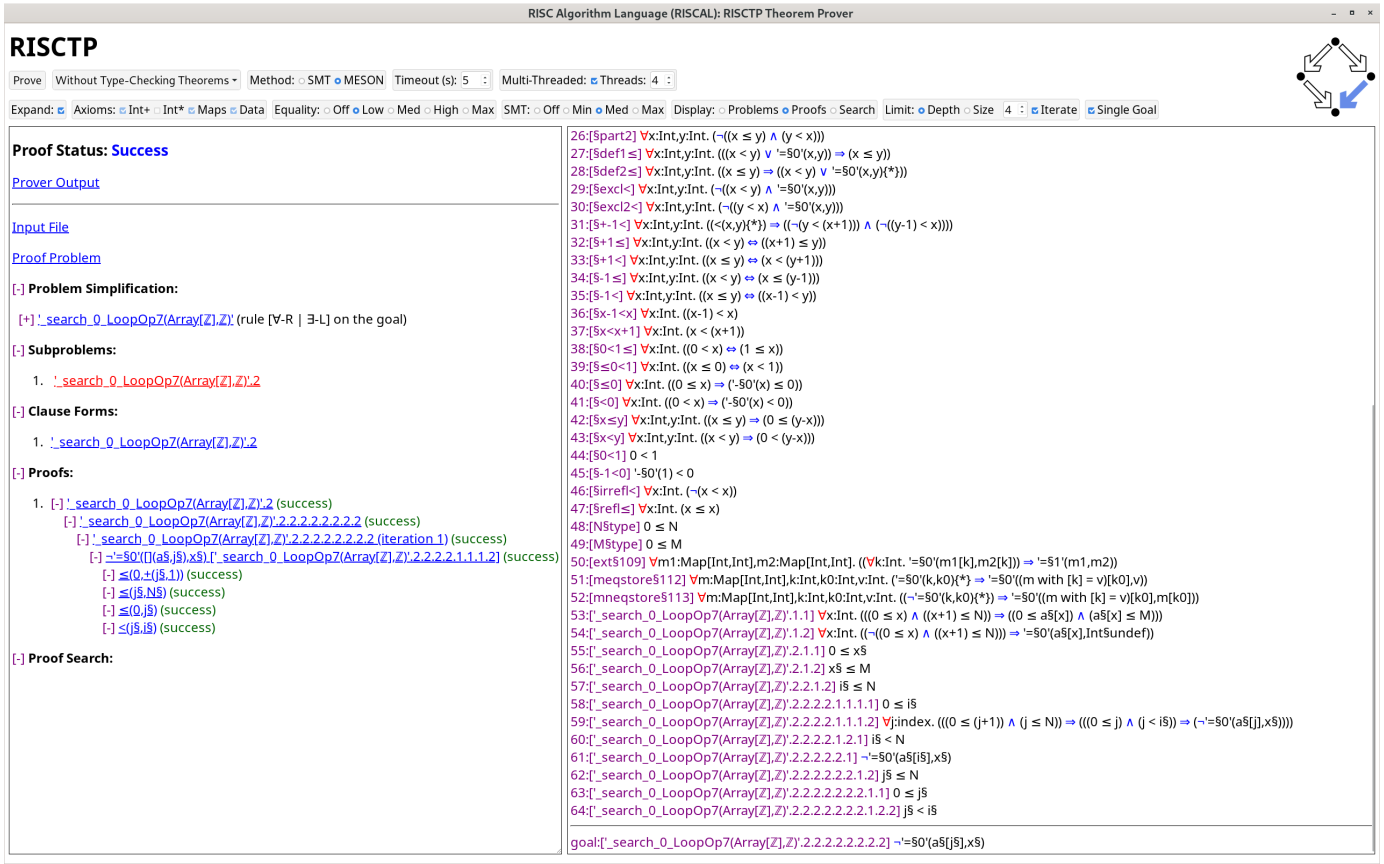
```
The problem has been closed by the SMT solver: the solver states by the output

  unsat

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

['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] ∀j:index. ≤(0,+(j,1)) ∧ ≤(j,N§) ∧ ≤(0,j) ∧ <(j,i§) ∧ '=§0'([](a§,j),[](a§,i§)) ⇒ ⊥
['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.1.2] ≤(j§,N§)
['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] ≤(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§))

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

['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2.1] ≤(0,+(j§,1)) ∧ ≤(j§,N§) ∧ ≤(0,j§) ∧ <(j§,i§) ∧ '=§0'([](a§,j§),[](a§,i§)) ⇒ ⊥
['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.1.2] ≤(j§,N§)
['_search_0_LoopOp6(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] ≤(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

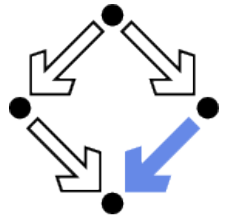


Is quantified loop invariant preserved by the second conditional branch?

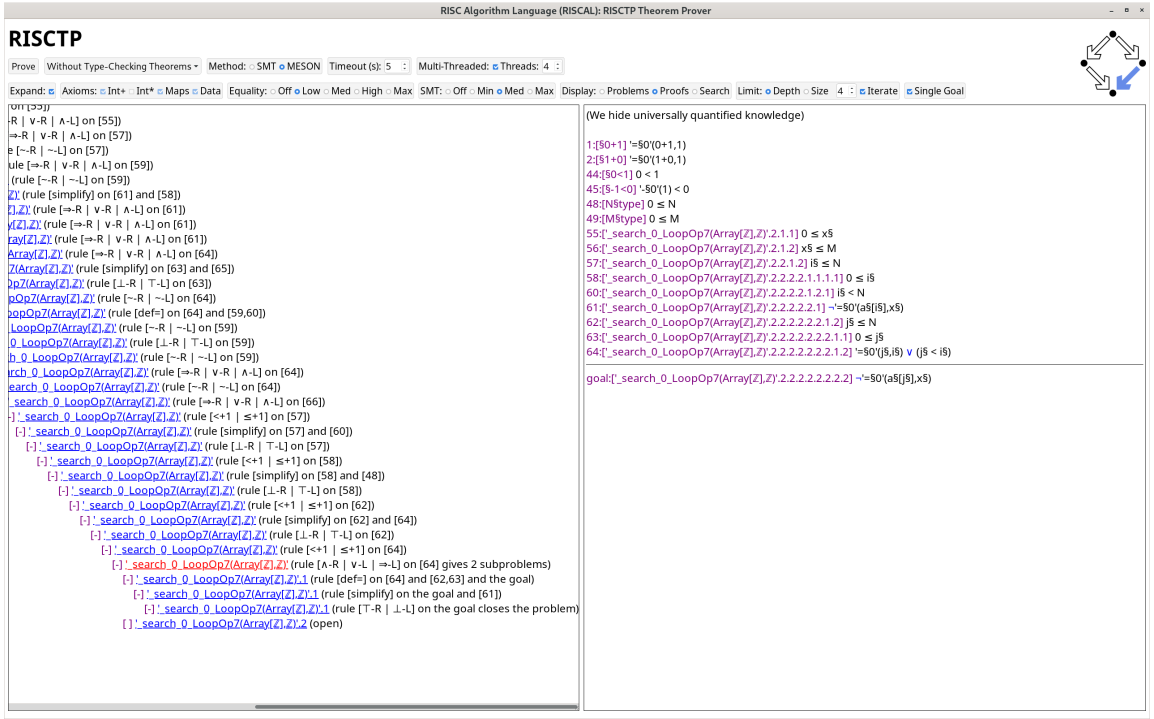



Problem is closed by simplification, proof search, and SMT solving.

# Example: Linear Search

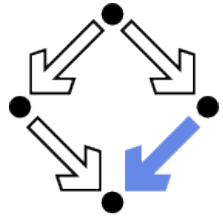


Is quantified loop invariant preserved by the second conditional branch?




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?

RISC Algorithm Language (RISCAL): RISCTP Theorem Prover

**RISCTP**

Prove | Without Type-Checking Theorems | Method: SMT MESON | Timeout (s): 5 | Multi-Threaded: Threads: 4

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

**Proof Status: Success**

Prover Output

Input File

Proof Problem

[-] **Problem Simplification:**

[+] '_search_0_CorrOp0(Array[ℤ],ℤ)' (rule [∀-R | ∃-L] on the goal)

[-] **Subproblems:**

1. '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1.1.2.1
2. '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.2.1.1.2
3. '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1.2
4. '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1
5. '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2

[-] **Clause Forms:**

1. '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1.1.2.1
2. '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.2.1.1.2
3. '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1.2
4. '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1
5. '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2

[-] **Proofs:**

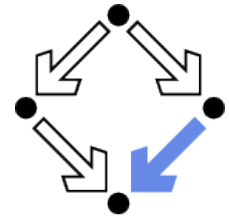
1. [+] '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1.1.2.1 (success)
2. [+] '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.2.1.1.2 (success)
3. [+] '_search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1.2 (success)
4. [+] '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1 (success)
5. [-] '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2 (success)
   - [-] '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.2 (success)
     - [-] '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.2 (iteration 1)
       - [-] ¬'=§0'([](a§,i§0),[](a§,i§)) ['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2
         - [-] ≤(0,+(i§0,1)) (success)
         - [-] ≤(i§0,N§) (success)
         - [-] ≤(0,i§0) (success)
         - [-] <(i§0,i§) (success)

30:[§excl2<] ∀x:Int,y:Int. (¬((y < x) ∧ '=§0'(x,y)))
31:[§+-1<] ∀x:Int,y:Int. ((<(x,y){*}) ⇒ ((¬(y < (x+1))) ∧ (¬((y-1) < x))))
32:[§+1≤] ∀x:Int,y:Int. ((x < y) ⇔ ((x+1) ≤ y))
33:[§+1<] ∀x:Int,y:Int. ((x ≤ y) ⇔ (x < (y+1)))
34:[§-1≤] ∀x:Int,y:Int. ((x < y) ⇔ (x ≤ (y-1)))
35:[§-1<] ∀x:Int,y:Int. ((x ≤ y) ⇔ ((x-1) < y))
36:[§x-1<x] ∀x:Int. ((x-1) < x)
37:[§x<x+1] ∀x:Int. (x < (x+1))
38:[§0<1≤] ∀x:Int. ((0 < x) ⇔ (1 ≤ x))
39:[§≤0<1] ∀x:Int. ((x ≤ 0) ⇔ (x < 1))
40:[§≤0] ∀x:Int. ((0 ≤ x) ⇒ ('-§0'(x) ≤ 0))
41:[§<0] ∀x:Int. ((0 < x) ⇒ ('-§0'(x) < 0))
42:[§x≤y] ∀x:Int,y:Int. ((x ≤ y) ⇒ (0 ≤ (y-x)))
43:[§x<y] ∀x:Int,y:Int. ((x < y) ⇒ (0 < (y-x)))
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
46:[§irrefl<] ∀x:Int. (¬(x < x))
47:[§refl≤] ∀x:Int. (x ≤ x)
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
50:[ext§109] ∀m1:Map[Int,Int],m2:Map[Int,Int]. ((∀k:Int. '=§0'(m1[k],m2[k])) ⇒ '=§1'(m1,m2))
51:[meqstore§112] ∀m:Map[Int,Int],k:Int,k0:Int,v:Int. ('=§0'(k,k0){*} ⇒ '=§0'((m with [k] = v)[k0],v))
52:[mneqstore§113] ∀m:Map[Int,Int],k:Int,k0:Int,v:Int. ((¬'=§0'(k,k0){*}) ⇒ '=§0'((m with [k] = v)[k0],m[k0]))
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)'.2.2.2.2.1.1.1.1] 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)'.2.2.2.2.2.1.2] ¬'=§0'(i§,0-1)
62:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.1.1.2] 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§])

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 ∧ ∀i:index. 0 ≤ i ∧ i < N ⇒ a[i] ≠ x) ∨
    (0 ≤ result ∧ result < N ∧ a[result] = x ∧ ∀i:index. 0 ≤ i ∧ i < result ⇒ a[i] ≠ x);
```

(We hide universally quantified knowledge)

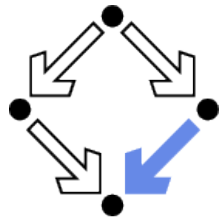
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ x§
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] x§ ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.1] 0 ≤ (i§+1)
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.2] 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)'.2.2.2.2.1.1.1.1] 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] (((0 ≤ r§) ∧ (r§ < N)) ∧ '=§0'(a§[r§],x§)) ∧ (∀i:index. ((('-§0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < r§)) ⇒ (¬'=§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?

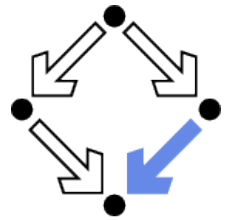
(((0 ≤ r§) ∧ (r§ < N)) ∧ '=§0'(a§[r§],x§)) ∧ (∀i:index. ((('-§0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < r§)) ⇒ (¬'=§0'(a§[i],x§)))))

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.
- $(\forall i : \ldots\ a[i] \neq x) \rightsquigarrow$ 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)

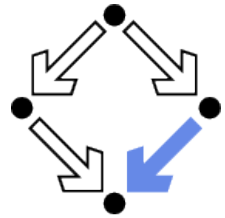
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ x§
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] x§ ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.1] 0 ≤ (i§+1)
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.2] 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)'.2.2.2.2.1.1.1.1] 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)'.2.2.2.2.2.2.2] ∀i:index. ((('-§0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < r§)) ⇒ (¬'=§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)

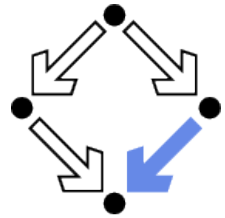
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ x§
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] x§ ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.1] 0 ≤ (i§+1)
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.2] 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)'.2.2.2.2.1.1.1.1] 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§))))))
66:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] 0 ≤ (i§0+1)
67:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.2] i§0 ≤ N
68:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.1.1] 0 ≤ i§0
69:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.1.2] i§0 < r§

---

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

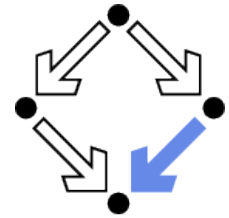
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ x§
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] x§ ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.1] 0 ≤ (i§+1)
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1.2] 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)'.2.2.2.2.1.1.1.1] 0 ≤ i§
63:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.2.2] ('=§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§))))))
66:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] 0 ≤ (i§0+1)
67:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.2] i§0 ≤ N
68:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.1.1] 0 ≤ i§0
69:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.1.2] i§0 < r§

---

goal:['_search_0_CorrOp0(Array[ℤ],ℤ)'.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)

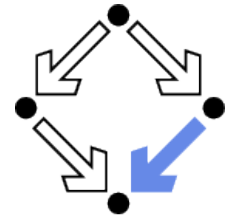
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ a§[i§]
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] a§[i§] ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.1] 0 ≤ (i§+1)
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.2] i§ ≤ N
59:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.1] 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] ¬('=§0'(i§,'-§0'(1)) ∧ (∀i:index. ((('-§0'(1) ≤ i) ∧ (i ≤ N)) ⇒ (((0 ≤ i) ∧ (i < N)) ⇒ (¬'=§0'(a§[i],a§[i§])))))
64:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] 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§])

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)

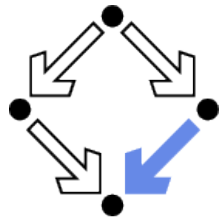
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ a§[i§]
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] a§[i§] ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.1] 0 ≤ (i§+1)
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.2] i§ ≤ N
59:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.1] 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)'.2.2.2.2.2.2.2.1.1] 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)

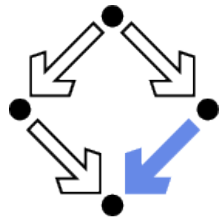
1:[§0+1] '=§0'(0+1,1)
2:[§1+0] '=§0'(1+0,1)
44:[§0<1] 0 < 1
45:[§-1<0] '-§0'(1) < 0
48:[N§type] 0 ≤ N
49:[M§type] 0 ≤ M
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ a§[i§]
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] a§[i§] ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.2] i§ ≤ N
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.1] 0 ≤ 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§])

After further simplification, we have subproblem 5.

# Example: Linear Search



Is result correct?

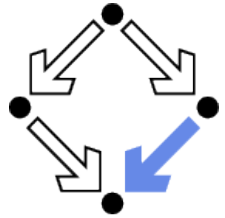
53:['_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1] ∀x:Int. (((0 ≤ x) ∧ ((x+1) ≤ N)) ⇒ ((0 ≤ a§[x]) ∧ (a§[x] ≤ M)))
54:['_search_0_CorrOp0(Array[ℤ],ℤ)'.1.2] ∀x:Int. ((¬((0 ≤ x) ∧ ((x+1) ≤ N))) ⇒ '=§0'(a§[x],Int§undef))
55:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] 0 ≤ a§[i§]
56:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] a§[i§] ≤ M
57:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.2] i§ ≤ N
58:['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.1] 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)'.2.2.2.2.2.2.2.1.1] 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§])

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

# Example: Linear Search



Is result correct?

RISC Algorithm Language (RISCAL): RISCTP Theorem Prover

**RISCTP**

Prove | Without Type-Checking Theorems | Method: SMT MESON | Timeout (s): 5 | Multi-Threaded: Threads: 4

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

**Proof Status: Success**

Prover Output

---

Input File

Proof Problem

[-] **Problem Simplification:**

[+] ' search_0_CorrOp0(Array[ℤ],ℤ)' (rule [∀-R | ∃-L] on the goal)

[-] **Subproblems:**

1. ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1.1.2.1
2. ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.2.1.1.2
3. ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1.2
4. ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1
5. ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2

[-] **Clause Forms:**

1. ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1.1.2.1
2. ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.2.1.1.2
3. ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1.2
4. ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1
5. ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2

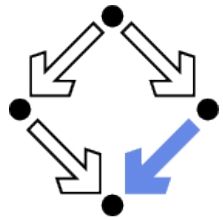
[-] **Proofs:**

1. [+] ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1.1.2.1 (success)
2. [+] ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.2.1.1.2 (success)
3. [+] ' search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1.2 (success)
4. [+] ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.1 (success)
5. [-] ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2 (success)
   - [-] ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.2 (success)
     - [-] ' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.2 (iteration 1) (success)
       - [-] ¬'=§0'([](a§,i§0),[](a§,i§)) [' search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.1.1.2] (success)
         - [-] ≤(0,+(i§0,1)) (success)
         - [-] ≤(i§0,N§) (success)
         - [-] ≤(0,i§0) (success)
         - [-] <(i§0,i§) (success)

```
[§-1<0] ⊤ ⇒ <('-§0'(1),0)
[§irrefl<] ∀x:Int. <(x,x) ⇒ ⊥
[§refl≤] ∀x:Int. ⊤ ⇒ ≤(x,x)
[N§type] ⊤ ⇒ ≤(0,N§)
[M§type] ⊤ ⇒ ≤(0,M§)
[ext§109] ∀m1:Map[Int,Int],m2:Map[Int,Int]. '=§0'([](m1,k§(m1,m2)),[](m2,k§(m1,m2))) ⇒ '=§1'(m1,m
[meqstore§112] ∀m:Map[Int,Int],k:Int,k0:Int,v:Int. '=§0'(k,k0) ⇒ '=§0'([]([]=(m,k,v),k0),v)
[mneqstore§113] ∀m:Map[Int,Int],k:Int,k0:Int,v:Int. ⊤ ⇒ '=§0'(k,k0) ∨ '=§0'([]([]=(m,k,v),k0),[](

The following "negated goals" represent the negation of the theorem to be proved:

['_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.0] ∀x:Int. ≤(0,x) ∧ ≤(+(x,1),N§) ⇒ ≤(0,[](a§,x))
['_search_0_CorrOp0(Array[ℤ],ℤ)'.1.1.1] ∀x:Int. ≤(0,x) ∧ ≤(+(x,1),N§) ⇒ ≤([](a§,x),M§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.0] ∀x:Int. ⊤ ⇒ ≤(0,x) ∨ '=§0'([](a§,x),Int$undef)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.1.2.1] ∀x:Int. ⊤ ⇒ ≤(+(x,1),N§) ∨ '=§0'([](a§,x),Int$undef)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.1] ⊤ ⇒ ≤(0,[](a§,i§))
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.1.2] ⊤ ⇒ ≤([](a§,i§),M§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.1.2] ⊤ ⇒ ≤(i§,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.1] ⊤ ⇒ ≤(0,i§)
['_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
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.2.1.2] ⊤ ⇒ <(i§,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.2] '=§0'(i§,-(0,1)) ⇒ ⊥
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.1.2.1.2] ⊤ ⇒ ≤(i§5,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.1.2.1.2.1] ⊤ ⇒ ≤(0,i§5)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.1.2.2.1.2] ⊤ ⇒ <(i§5,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.1.2.2.2] ⊤ ⇒ '=§0'([](a§,i§5),[](a§,i§))
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.2] ⊤ ⇒ ≤(i§0,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.1.1] ⊤ ⇒ ≤(0,i§0)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.1.2] ⊤ ⇒ <(i§0,i§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2.2] ⊤ ⇒ '=§0'([](a§,i§0),[](a§,i§))

To prove the theorem, we apply the proof strategy MESON (model elimination, subgoal oriented)
to derive from the axioms and negated goals a contradiction. For this,
we prove some (not negated) goal from the "knowledge" represented by the other formulas.
We start the proof with the last goal; if this does not succeed, we also try the previous ones.

According to the 'SMT' setting, we aid MESON by application of an SMT solver:
in every proof situation, we first apply the SMT solver before MESON;
if the MESON proof fails, we apply the SMT solver to the whole proof problem.
--------------------------------------------------------------------------------
SUCCESS: the proof has been completed.
```

The problem is closed by proof search and SMT solving.

# Example: Linear Search



## Is result correct?

**Goal: ¬'=§0'([](a§,i§0),[](a§,i§)) ['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] (proof depth: 0, proof size: 1)**

Goal: ¬'=§0'([](a§,i§0),[](a§,i§))

To prove the goal, we assume its negation

[1] '=§0'([](a§,i§0),[](a§,i§))

and show a contradiction. For this, consider knowledge ['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] with the following instance:

∀j@113:index. ≤(0,+(j@113,1)) ∧ ≤(j@113,N§) ∧ ≤(0,j@113) ∧ <(j@113,i§) ∧ '=§0'([](a§,j@113),[](a§,i§)) ⇒ ⊥

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

j@113 → i§0

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

≤(0,+(i§0,1)) ∧ ≤(i§0,N§) ∧ ≤(0,i§0) ∧ <(i§0,i§) ⇒ ⊥

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

≤(0,+(i§0,1)) ∧ ≤(i§0,N§) ∧ ≤(0,i§0) ∧ <(i§0,i§)

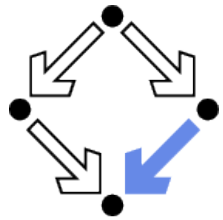
---

SUCCESS: goal ¬'=§0'([](a§,i§0),[](a§,i§)) ['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2] has been proved with the following substitution:

j@113 → i§0

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

# Example: Linear Search



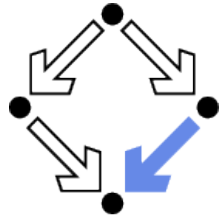
Is result correct?

**Goal: ≤(0,+(i§0,1)) (proof depth: 1, proof size: 2)**

```
Goal: ≤(0,+(i§0,1))

Assumptions:

[1] '=§0'([](a§,i§0),[](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:

['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] ≤(0,i§0)
--------------------------------------------------------------------------------
SUCCESS: goal ≤(0,+(i§0,1)) has been proved with the following substitution:

  j@113 → i§0
```

Option "SMT: Med": the subproblems are closed by the SMT solver.

# Example: Linear Search



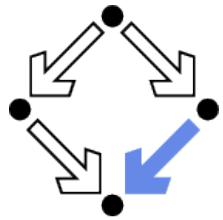
## Is result correct?

**Proof problem: '_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2**

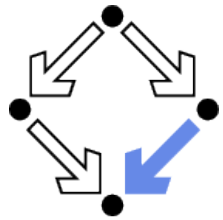
---

```
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)'.2.2.2.2.1.1.1.2] ∀j:index. ≤(0,+(j,1)) ∧ ≤(j,N§) ∧ ≤(0,j) ∧ <(j,i§) ∧ '=§0'([](a§,j),[](a§,i§)) ⇒ ⊥
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.1.2] ≤(i§0,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] ≤(0,i§0)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.2] <(i§0,i§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2] '=§0'([](a§,i§0),[](a§,i§))

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

['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.1.1.1.2.2] ≤(0,+(i§0,1)) ∧ ≤(i§0,N§) ∧ ≤(0,i§0) ∧ <(i§0,i§) ∧ '=§0'([](a§,i§0),[](a§,i§)) ⇒ ⊥
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.1.2] ≤(i§0,N§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.1] ≤(0,i§0)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.1.2] <(i§0,i§)
['_search_0_CorrOp0(Array[ℤ],ℤ)'.2.2.2.2.2.2.2.2] '=§0'([](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.



# Procedure Specifications



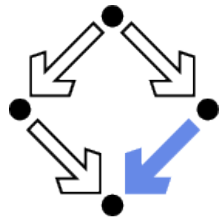
global $g$;
requires $Pre$;
ensures $Post$;
$o := p(i)\ \{\ c\ \}$

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

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

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

# Example



- Procedure specification:

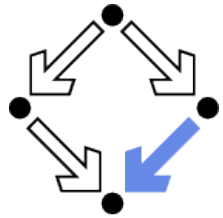
  global $g$
  requires $g \geq 0 \wedge i > 0$
  ensures $g_0 = g \cdot i + o \wedge 0 \leq o < i$
  $o := p(i)$ { $o := g\%i;\ g := g/i$ }

- Proof obligation:

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

A procedure that divides $g$ by $i$ and returns the remainder.

# Procedure Calls



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

$$x := p(e)$$

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

- Original:

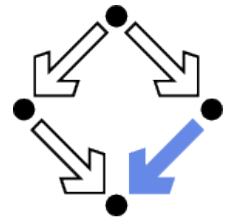
$$\{D(e) \wedge Q[e/x]\}$$
$$x := e$$
$$\{Q\}$$

- Alternative:

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

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

# Procedure Calls



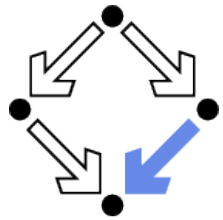
From this, we can derive a rule for the correctness of procedure calls.

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

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

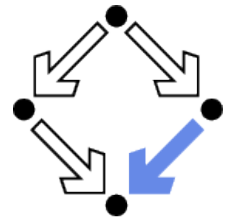


$$\begin{aligned}
&\mathrm{wp}(x = p(e), Q) = \\
&\quad D(e) \wedge Pre[e/i] \wedge \\
&\quad \forall x', g' : \\
&\qquad Post[e/i, x'/o, g/g_0, g'/g] \Rightarrow Q[x'/x, g'/g]
\end{aligned}$$

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

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:

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