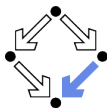


# Specifying and Verifying Programs (Part 2)

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

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



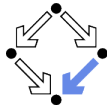


---

# 1. Programs as State Relations

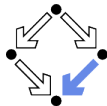
## 2. The RISC ProgramExplorer

# Specification by State Predicates



- Hoare calculus and predicate transformers use **state predicates**.
  - Formulas that talk about a single (pre/post-)state.
  - In such a formula, a reference  $x$  means “the value of program variable  $x$  in the given state”.
- **Relationship** between pre/post-state is not directly expressible.
  - Requires uninterpreted mathematical constants.  
$$\{x = a\}x := x + 1\{x = a + 1\}$$
- **Unchanged variables** have to be explicitly specified.  
$$\{x = a \wedge y = b\}x := x + 1\{x = a + 1 \wedge y = b\}$$
- The **semantics** of a command  $c$  is only **implicitly** specified.
  - Specifications depend on auxiliary state conditions  $P, Q$ .  
$$\{P\}c\{Q\}$$
$$wp(c, Q) = P$$

Let us turn our focus from individual states to pairs of states.



# Specification by State Relations

- We introduce formulas that denote **state relations**.
  - Talk about a pair of states (the pre-state and the post-state).
  - old  $x$ : “the value of program variable  $x$  in the pre-state”.
  - var  $x$ : “the value of program variable  $x$  in the post-state”.
- We introduce the logical judgment  $c : [F]^{x, \dots}$ 
  - If the execution of  $c$  terminates normally, the resulting post-state is related to the pre-state as described by  $F$ .
  - Every variable  $y$  not listed in the set of variables  $x, \dots$  has the same value in the pre-state and in the post-state.

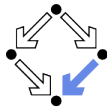
$$c : F \wedge \text{var } y = \text{old } y \wedge \dots$$

$$x := x + 1 : [\text{var } x = \text{old } x + 1]^x$$

$$x := x + 1 : \text{var } x = \text{old } x + 1 \wedge \text{var } y = \text{old } y \wedge \text{var } z = \text{old } z \wedge \dots$$

We will discuss the termination of commands later.

# State Relation Rules



$$\frac{c : [F]^{xs} \quad y \notin xs}{c : [F \wedge \text{var } y = \text{old } y]^{xs \cup \{y\}}}$$

$$\text{skip} : [\text{true}]^{\emptyset} \quad \text{abort} : [\text{true}]^{\emptyset} \quad x = e : [\text{var } x = e']^{\{x\}}$$

$$\frac{c_1 : [F_1]^{xs} \quad c_2 : [F_2]^{xs}}{c_1; c_2 : [\exists ys : F_1[ys/\text{var } xs] \wedge F_2[ys/\text{old } xs]]^{xs}}$$

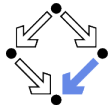
$$\frac{c : [F]^{xs}}{\text{if } e \text{ then } c : [\text{if } e' \text{ then } F \text{ else var } xs = \text{old } xs]^{xs}}$$

$$\frac{c_1 : [F_1]^{xs} \quad c_2 : [F_2]^{xs}}{\text{if } e \text{ then } c_1 \text{ else } c_2 : [\text{if } e' \text{ then } F_1 \text{ else } F_2]^{xs}}$$

$$\frac{c : [F]^{xs} \quad \vdash \forall xs, ys, zs : I[xs/\text{old } xs, ys/\text{var } xs] \wedge e[ys/xs] \wedge F[ys/\text{old } xs, zs/\text{var } xs] \Rightarrow I[xs/\text{old } xs, zs/\text{var } xs]}{\text{while } e \text{ do } \{l, t\} c : [\neg e'' \wedge (I[\text{old } xs/\text{var } xs] \Rightarrow I)]^{xs}}$$

if e then  $F_1$  else  $F_2$   $:\Leftrightarrow (e \Rightarrow F_1) \wedge (\neg e \Rightarrow F_2)$

$e' := e[\text{old } xs/xs], e'' := e[\text{var } xs/xs]$  (for all program variables  $xs$ )



## Example

$$c_1 = y := y + 1;$$

$$c_2 = x := x + y$$

$$c_1 : [\text{var } y = \text{old } y + 1]^y$$

$$c_2 : [\text{var } x = \text{old } x + \text{old } y]^x$$

$$c_1 : [\text{var } y = \text{old } y + 1 \wedge \text{var } x = \text{old } x]^{x,y}$$

$$c_2 : [\text{var } x = \text{old } x + \text{old } y \wedge \text{var } y = \text{old } y]^{x,y}$$

$$c_1; c_2 : [\exists x_0, y_0 :$$

$$y_0 = \text{old } y + 1 \wedge x_0 = \text{old } x \wedge$$

$$\text{var } x = x_0 + y_0 \wedge \text{var } y = y_0]^{x,y}$$

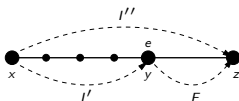
$$c_1; c_2 : [\text{var } x = \text{old } x + \text{old } y + 1 \wedge \text{var } y = \text{old } y + 1]^{x,y}$$

Mechanical translation and logical simplification.

# Loops



$$\frac{c : [F]^{xs} \quad \vdash \forall xs, ys, zs : I[xs/\text{old } xs, ys/\text{var } xs] \wedge e[ys/xs] \wedge F[ys/\text{old } xs, zs/\text{var } xs] \Rightarrow I[xs/\text{old } xs, zs/\text{var } xs]}{\text{while } e \text{ do } \{I, t\} \quad c : [\neg e'' \wedge (I[\text{old } xs/\text{var } xs] \Rightarrow I)]^{xs}}$$



$$w = \text{while } i < n \text{ do } \{I, t\} (s := s + i; i := i + 1)$$

$$I \Leftrightarrow 0 \leq \text{var } i \leq \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{var } i - 1} j$$

$$(s := s + i; i := i + 1) : [\text{var } s = \text{old } s + \text{old } i \wedge \text{var } i = \text{old } i + 1]^{s, i}$$

$$\vdash \forall s_x, s_y, s_z, i_x, i_y, i_z :$$

$$(0 \leq i_y \leq \text{old } n \wedge s_y = \sum_{j=0}^{i_y - 1} j) \wedge i_y < \text{old } n \wedge (s_z = s_y + i_y \wedge i_z = i_y + 1) \Rightarrow$$

$$0 \leq i_z \leq \text{old } n \wedge s_z = \sum_{j=0}^{i_z - 1} j$$

$$w : [\neg(\text{var } i < \text{var } n) \wedge (0 \leq \text{old } i \leq \text{old } n \wedge \text{old } s = \sum_{j=0}^{\text{old } i - 1} j \Rightarrow I)]^{s, i}$$

The loop relation is derived from the invariant (not the loop body); we have to prove the preservation of the loop invariant.

# Example



```
c =  
  if n < 0  
    s := -1  
  else  
    s := 0  
    i := 0  
    while i < n do {l,t}  
      s := s + i  
      i := i + 1
```

$$l \Leftrightarrow 0 \leq \text{var } i \leq \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{var } i-1} j$$
$$t = \text{old } n - \text{old } i$$

$c$  : [if old  $n < 0$   
 then var  $i = \text{old } i \wedge \text{var } s = -1$   
 else var  $i = \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{old } n-1} j$ ] $^{s,i}$

Let us calculate this “semantic essence” of the program.



# Example



$c = \text{if } n < 0 \text{ then } s := -1 \text{ else } b$

$b = (s := 0; i := 0; w)$

$w = \text{while } i < n \text{ do } \{l, t\} (s := s + i; i = i + 1)$

$s := 0 : [\text{var } s = 0]^s$

$s := 0 : [\text{var } s = 0 \wedge \text{var } i = \text{old } i]^{s,i}$

$i := 0 : [\text{var } i = 0]^i$

$i := 0 : [\text{var } i = 0 \wedge \text{var } s = \text{old } s]^{s,i}$

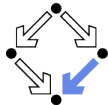
$s := 0; i := 0 : [\exists s_0, i_0 : s_0 = 0 \wedge i_0 = \text{old } i \wedge \text{var } i = 0 \wedge \text{var } s = s_0]^{s,i}$

$s := 0; i := 0 : [\text{var } s = 0 \wedge \text{var } i = 0]^{s,i}$

$w : [\neg(\text{var } i < \text{var } n) \wedge (0 \leq \text{old } i \leq \text{old } n \wedge \text{old } s = \sum_{j=0}^{\text{old } i-1} j \Rightarrow l)]^{s,i}$

$w : [\text{var } i \geq \text{old } n \wedge (0 \leq \text{old } i \leq \text{old } n \wedge \text{old } s = \sum_{j=0}^{\text{old } i-1} j \Rightarrow l)]^{s,i}$

# Example



$c = \mathbf{if} \ n < 0 \ \mathbf{then} \ s := -1 \ \mathbf{else} \ b$

$b = (s := 0; i := 0; w)$

$w = \mathbf{while} \ i < n \ \mathbf{do} \ \{l, t\} \ (s := s + i; i = i + 1)$

$s := 0; i := 0 : [\mathbf{var} \ s = 0 \wedge \mathbf{var} \ i = 0]^{s,i}$

$w : [\mathbf{var} \ i \geq \mathbf{old} \ n \wedge (0 \leq \mathbf{old} \ i \leq \mathbf{old} \ n \wedge \mathbf{old} \ s = \sum_{j=0}^{\mathbf{old} \ i-1} j \Rightarrow l)]^{s,i}$

$b : [\exists s_0, i_0 : s_0 = 0 \wedge i_0 = 0 \wedge$

$\mathbf{var} \ i \geq \mathbf{old} \ n \wedge (0 \leq i_0 \leq \mathbf{old} \ n \wedge s_0 = \sum_{j=0}^{i_0-1} j \Rightarrow l)]^{s,i}$

$b : [\exists s_0, i_0 : s_0 = 0 \wedge i_0 = 0 \wedge$

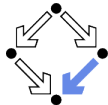
$\mathbf{var} \ i \geq \mathbf{old} \ n \wedge (0 \leq \mathbf{old} \ n \Rightarrow l)]^{s,i}$

$b : [\mathbf{var} \ i \geq \mathbf{old} \ n \wedge$

$(0 \leq \mathbf{old} \ n \Rightarrow 0 \leq \mathbf{var} \ i \leq \mathbf{old} \ n \wedge \mathbf{var} \ s = \sum_{j=0}^{\mathbf{var} \ i-1} j)]^{s,i}$

$b : [\mathbf{var} \ i \geq \mathbf{old} \ n \wedge$

$(0 \leq \mathbf{old} \ n \Rightarrow \mathbf{var} \ i = \mathbf{old} \ n \wedge \mathbf{var} \ s = \sum_{j=0}^{\mathbf{old} \ n-1} j)]^{s,i}$



# Example

$c = \text{if } n < 0 \text{ then } s := -1 \text{ else } b$   
 $b = (s := 0; i := 0; w)$   
 $w = \text{while } i < n \text{ do } \{l, t\} (s := s + i; i = i + 1)$

$s := -1 : [\text{var } s = -1]^s$   
 $s := -1 : [\text{var } i = \text{old } i \wedge \text{var } s = -1]^{s,i}$

$b : [\text{var } i \geq \text{old } n \wedge$   
 $(0 \leq \text{old } n \Rightarrow \text{var } i = \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{old } n-1} j)]^{s,i}$

$c : [\text{if old } n < 0$   
then  $\text{var } i = \text{old } i \wedge \text{var } s = -1$   
else  $\text{var } i \geq \text{old } n \wedge$   
 $(0 \leq \text{old } n \Rightarrow \text{var } i = \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{old } n-1} j)]^{s,i}$

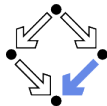
$c : [\text{if old } n < 0$   
then  $\text{var } i = \text{old } i \wedge \text{var } s = -1$   
else  $\text{var } i = \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{old } n-1} j)]^{s,i}$

# Partial Correctness



- **Specification** ( $xs, P, Q$ )
  - Set of program variables  $xs$  (which may be modified).
  - Precondition  $P$  (a formula with “old  $xs$ ” but no “var  $xs$ ”).
  - Postcondition  $Q$  (a formula with both “old  $xs$ ” and “var  $xs$ ”).
- **Partial correctness of implementation**  $c$ 
  1. Derive  $c : [F]^{xs}$ .
  2. Prove  $F \Rightarrow (P \Rightarrow Q)$ 
    - Or:  $P \Rightarrow (F \Rightarrow Q)$
    - Or:  $(P \wedge F) \Rightarrow Q$

Verification of partial correctness leads to the proof of an implication.



# Relationship to Other Calculi

Let all state conditions refer via “*old xs*” to program variables *xs*.

## ■ Hoare Calculus

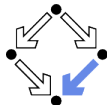
- For proving  $\{P\}c\{Q\}$ ,
- it suffices to derive  $c : [F]^{xs}$
- and prove  $P \wedge F \Rightarrow Q[\text{var } xs/\text{old } xs]$ .

## ■ Predicate Transformers

- Assume we can derive  $c : [F]^{xs}$ .
- If  $c$  does not contain loops, then
$$\text{wp}(c, Q) = \forall xs : F[xs/\text{var } xs] \Rightarrow Q[xs/\text{old } xs]$$
$$\text{sp}(c, P) = \exists xs : P[xs/\text{old } xs] \wedge F[xs/\text{old } xs, \text{old } xs/\text{var } xs]$$
- If  $c$  contains loops, the result is still a valid pre/post-condition but not necessarily the weakest/strongest one.

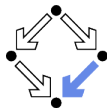
A generalization of the previously presented calculi.

# Termination



- We introduce a judgment  $c \downarrow T$ .  
State condition  $T$  (a formula with “old  $xs$ ” but no “var  $xs$ ”).
  - Starting with a pre-state that satisfies condition  $T$  the execution of command  $c$  terminates.
- **Total correctness** of implementation  $c$ .  
Specification  $(xs, P, Q)$ .
  - Derive  $c \downarrow T$ .
  - Prove  $P \Rightarrow T$ .

Also verification of termination leads to the proof of an implication.



# Termination Condition Rules

**skip**  $\downarrow$  true

**abort**  $\downarrow$  true

$x := e$   $\downarrow$  true

$$\frac{c_1 \downarrow T_1 \quad c_2 \downarrow T_2}{c_1; c_2 \downarrow T_1 \wedge wp(c_1, T_2)}$$

$$\frac{c \downarrow T}{\text{if } e \text{ then } c \downarrow e' \Rightarrow T}$$

$$\frac{c_1 \downarrow T_1 \quad c_2 \downarrow T_2}{\text{if } e \text{ then } c_1 \text{ else } c_2 \downarrow \text{if } e' \text{ then } T_1 \text{ else } T_2}$$

$$c : [F]^{xs} \quad c \downarrow T$$

$\vdash \forall xs, ys, zs :$

$$I[xs/\text{old } xs, ys/\text{var } xs] \wedge e[ys/xs] \wedge F[ys/\text{old } xs, zs/\text{var } xs] \wedge t[ys/\text{old } xs] \geq 0 \Rightarrow T[ys/\text{old } xs] \wedge 0 \leq t[zs/\text{old } xs] < t[ys/\text{old } xs]$$

**while**  $e$  **do**  $\{l, t\}$   $c \downarrow t \geq 0$

In every iteration of a loop, the loop body must terminate and the termination term must decrease (but not become negative).

# Example



```
c =  
  if n < 0  
    s := -1  
  else  
    s := 0  
    i := 0  
    while i < n do {l,t}  
      s := s + i  
      i := i + 1
```

$$I \Leftrightarrow 0 \leq \text{var } i \leq \text{old } n \wedge \text{var } s = \sum_{j=0}^{\text{var } i-1} j$$
$$t = \text{old } n - \text{old } i$$

$c \downarrow$  if old  $n < 0$  then true else ...  
 $c \downarrow$  if old  $n < 0$  then true else old  $n \geq 0$   
 $c \downarrow$  true

We still have to prove the constraint on the loop iteration.



# Example



$s := s + i; i := i + 1 \downarrow \text{true}$

$\forall s_x, s_y, s_z, i_x, i_y, i_z :$

$(0 \leq i_y \leq \text{old } n \wedge s_y = \sum_{j=0}^{i_y-1} j) \wedge$

$i_y < \text{old } n \wedge$

$(s_z = s_y + i_y \wedge i_z = i_y + 1) \wedge$

$\text{old } n - i_y \geq 0 \Rightarrow$

$\text{true} \wedge$

$0 \leq \text{old } n - i_z < \text{old } n - i_y$

Also this constraint is simple to prove.

# Abortion



Also abortion can be ruled out by proving side conditions in the usual way.

Wolfgang Schreiner. *Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs*. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2011.

See the report for the full calculus.



---

## 1. Programs as State Relations

## 2. The RISC ProgramExplorer

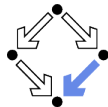
# The RISC ProgramExplorer



- **An integrated environment for program reasoning.**
  - Research Institute for Symbolic Computation (RISC), 2008–.  
<http://www.risc.jku.at/research/formal/software/ProgramExplorer>
  - Integrates the RISC ProofNavigator for computer-assisted proving.
  - Written in Java, runs under Linux (only), freely available (GPL).
- **Programs written in “MiniJava”.**
  - Subset of Java with full support of control flow interruptions.
  - Value (not pointer) semantics for arrays and objects.
- **Theories and specifications written in a formula language.**
  - Derived from the language of the RISC ProofNavigator.
- **Semantic analysis and verification.**
  - Program methods are translated into their “semantic essence”.
    - Open for human inspection.
  - From the semantics, the verification tasks are generated.
    - Solved by automatic decision procedure or interactive proof.

**Tight integration of executable programs, declarative specifications, mathematical semantics, and verification tasks.**

# Using the Software



See “The RISC ProgramExplorer: Tutorial and Manual”.

- **Develop a theory.**
  - File “*Theory.theory*” with a theory *Theory* of mathematical types, constants, functions, predicates, axioms, and theorems.
  - Can be also added to a program file.
- **Develop a program.**
  - File “*Class.java*” with a class *Class* that contains class (static) and object (non-static) variables, methods and constructors.
  - Class may be annotated by a theory (and an object invariant).
  - Methods may be annotated by method specifications.
  - Loops may be annotated by invariants and termination terms.
- **Analyze method semantics.**
  - Transition relations, termination conditions, ... of the method body and its individual commands.
- **Perform verification tasks.**
  - Frame, postcondition, termination, preconditions, loop-related tasks, type-checking conditions.

# Starting the Software



- Starting the software:

```
module load ProgramExplorer (only at RISC)
ProgramExplorer &
```

- Command line options:

```
Usage: ProgramExplorer [OPTION]...
```

```
OPTION: one of the following options:
```

```
-h, --help: print this message.
```

```
-cp, --classpath [PATH]:
```

```
directories representing top package.
```

```
Environment Variables:
```

```
PE_CLASSPATH:
```

```
the directories (separated by ":") representing the
top package (default the current working directory)
```

```
...
```

- Task repository created/read in current working directory:

```
Subdirectory .PETASKS.timestamp (ProgramExplorer tasks)
```

```
Subdirectory .ProofNavigator (ProofNavigator legacy)
```

# The Graphical User Interface



The screenshot displays the Program Explorer GUI with the following components:

- Files/Symbols:** A tree view on the left showing a project structure with packages like java, Arrays, Control, and a class Searching.
- Source Editor:** The central pane shows the source code for `Searching.java`. The `search(int[] a, int x)` method is highlighted, and a tooltip shows its signature: `class method int Searching.search(int[], int) [Searching.java:14:3]`. The code includes a loop and a return statement.
- Console:** The bottom pane shows the execution output, including the version (0.0), copyright (© 2008), and a list of processed classes: `theory Base`, `processing class java.lang.String`, `initialization completed`, `executing tasks..done`, `processing class Searching`, `class Searching was processed with no errors`, and `executing tasks..done`.
- Semantics/Verification/Analysis:** The right pane shows the generated verification conditions for the `Searching` class, including preconditions, loops, and specification validation.

# A Program



```
/*@..  
class Sum  
{  
    static int sum(int n) /*@..  
    {  
        int s;  
        if (n < 0)  
            s = -1;  
        else  
        {  
            s = 0;  
            int i = 1;  
            while (i <= n) /*@..  
            {  
                s = s+i;  
                i = i+1;  
            }  
        }  
        return s;  
    }  
}
```

Markers /\*@.. indicate  
hidden mathematical annotations.



# A Theory



```
/*@
  theory {
    sum: (INT, INT) -> INT;
    sumaxiom: AXIOM
      FORALL(m: INT, n: INT):
        IF n<m THEN
          sum(m, n) = 0
        ELSE
          sum(m, n) = n+sum(m, n-1)
        ENDIF;
  }
  @*/
class Sum
  ...
```

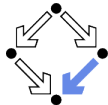
The introduction of a function  $sum(m, n) = \sum_{j=m}^n j$ .

# A Method Specification



```
static int sum(int n) /*@
  requires VAR n < Base.MAX_INT;
  ensures
    LET result=VALUE@NEXT IN
    IF VAR n < 0
      THEN result = -1
      ELSE result = sum(1, VAR n)
    ENDIF;
  @*/
  ...
```

For non-negative  $n$ , a call of program method  $\text{sum}(n)$  returns  $\text{sum}(1, n)$  (and does not modify any global variable).



# A Loop Annotation

---

```
while (i <= n) /*@
  invariant VAR n < Base.MAX_INT
            AND 1 <= VAR i AND VAR i <= VAR n+1
            AND VAR s=sum(1, VAR i-1);
  decreases VAR n - VAR i + 1;
/*@/
{
  s = s+i;
  i = i+1;
}
}
```

The loop invariant and termination term (measure).

# The Specification Language



Derived from the language of the RISC ProofNavigator.

- **State conditions/relations, state terms.**
  - State condition: method precondition (requires).
  - State relation: method postcondition (ensures), loop invariant (invariant).
  - State term: termination term (decreases).
- **References to program variables.**

OLD  $x$ : the value of program variable  $x$  in the pre-state.  
VAR  $x$ : the value of program variable  $x$  in the post-state.

  - In state conditions/terms, both refer to the value in the current state.
  - If program variable is of the program type  $T$ , then then OLD/VAR  $x$  is of the mathematical type  $T'$ .  
 $\text{int} \rightarrow \text{Base.int} = [\text{Base.MIN\_INT}, \text{Base.MAX\_INT}]$ .
- **Function results**

VALUE@NEXT: the return value of a program function.

  - The value of the function call's post-state NEXT.

# The Semantics View



The screenshot shows a software interface with a menu bar (File, Edit, Help) and a toolbar. The main window is titled "Sum.sum" and contains the following code:

```
requires old n < Base.MAX_INT
ensures let result = value@next
in
  (if var n < 0 then result = -1 else result = sum(1, var n) endif )

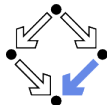
public static int sum(int n) /*@
  requires OLD n < Base.MAX_INT;
  ensures LET result = VALUE@NEXT IN (IF VAR n < 0
  @*/
{
  int s;
  if (n < 0)
    s = -1;
  else
  {
    s = 0;
    int i = 1;
    while (i <= n) /*@
      invariant VAR n < Base.MAX_INT AND 1 <= VAR
      decreases OLD n-OLD i+1;
      @*/
    {
      s = s+i;
      i = i+1;
    }
  }
  return s;
}
```

Below the code is a section titled "Select a statement and define a condition for its prepoststate:" with a text input field and buttons for "Submit", "Reset", "Prestate", and "Poststate".

The right-hand pane is titled "Body Knowledge" and contains the following information:

- [Show Original Formulas]
- Pre-State Knowledge**  
old n < Base.MAX\_INT
- Effects**  
executes: false, continues: false, breaks: false, returns: true  
variables: -; exceptions:-
- Transition Relation**  
if old n < 0 then  
 returns@next A value@next = -1  
else  
 returns@next  
 A  
 (∃in ∈ Base.int: in = old n + 1 ∧ 1 ≤ in ∧ value@next = sum(1, in - 1))  
 A  
 old n < Base.MAX\_INT  
endif

# The Method Body



## Body Knowledge

[Show Original Formulas]

## Pre-State Knowledge

old  $n < \text{Base.MAX}_{\text{INT}}$

## Effects

**executes:** false, **continues:** false, **breaks:** false, **returns:** true

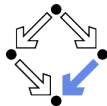
**variables:** -; **exceptions:** -

## Transition Relation

```
if old  $n < 0$  then
  returns@next  $\wedge$  value@next = -1
else
  returns@next
 $\wedge$ 
  ( $\exists \text{in} \in \text{Base.int}: \text{in} = \text{old } n + 1 \wedge 1 \leq \text{in} \wedge \text{value@next} = \text{sum}(1, \text{in} - 1)$ )
 $\wedge$ 
  old  $n < \text{Base.MAX}_{\text{INT}}$ 
endif
```

Select method symbol “sum” and menu entry ”Show Semantics”.

# A Body Command



## Statement Knowledge

[Show Original Formulas]

## Pre-State Knowledge

$\text{old } n < \text{Base.MAX}_{\text{INT}} \wedge \text{old } n \geq 0 \wedge \text{old } s = 0 \wedge \text{old } i = 1$

## Precondition

$\text{old } n < \text{Base.MAX}_{\text{INT}} \wedge 1 \leq \text{old } i \wedge \text{old } i \leq \text{old } n + 1 \wedge \text{old } s = \text{sum}(1, \text{old } i - 1)$

## Effects

**executes:** true, **continues:** false, **breaks:** false, **returns:** false

**variables:**  $s, i$ ; **exceptions:** -

## Transition Relation

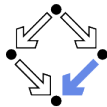
$\text{var } i = \text{old } n + 1 \wedge \text{old } n < \text{Base.MAX}_{\text{INT}} \wedge 1 \leq \text{var } i \wedge \text{var } s = \text{sum}(1, \text{var } i - 1)$

## Termination Condition

$\text{executes}@now \Rightarrow \text{old } n - \text{old } i \geq -1$

Move the mouse pointer  
over the box to the  
left of the loop.

# The Semantics Elements



- **Pre-State Knowledge**

What is known about the pre-state of the command.

- **Precondition**

What has to be true for the pre-state of the command such that the command may be executed.

- **Effects**

Which kind of effects may the command have.

- **variables**: which variables may be changed.

- **exceptions**: which exceptions may be thrown.

- **executes, continues, breaks, returns**: may the execution terminate normally, may it be terminated by a **continue**, **break**, **return**.

- **Transition Relation**

The prestate/poststate relationship of the command.

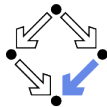
- **Termination**

What has to be true for the pre-state of the command such that the command terminates.

Formulas are shown after simplification (see “Show Original Formulas”).



# Constraining a State



## State Conditions

[[Show Original Formulas](#)]

### Pre-State Condition

$\text{var } i = 1 \wedge \text{var } s = \text{var } i + 2$

### Post-State Condition

$\text{var } s = 3 \wedge \text{var } i = 2$

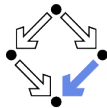
Select the loop body, enter in the box the condition  $\text{VAR } s=2 \text{ AND } \text{VAR } i=1$ , press "Submit", and move the mouse to  $i=i+1$ .

# The Verification Tasks



- class Sum
  - method sum
    - [Sum.sum] effects
    - [Sum.sum] postcondition
    - [Sum.sum] termination
  - preconditions
    - [Sum.sum:0] assignment precondition
    - [Sum.sum:1] while loop precondition
    - [Sum.sum:2] assignment precondition
    - [Sum.sum:3] assignment precondition
  - loops
    - [Sum.sum:qvb] invariant is preserved
    - [Sum.sum:qvb] measure is well-formed
    - [Sum.sum:qvb] measure is decreased
  - type checking conditions
  - specification validation (optional)
    - [Sum.sum] specification is satisfiable
    - [Sum.sum] specification is not trivial

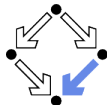
# The Verification Tasks



- **Effects:** does the method only change those global variables indicated in the method's assignable clause?
- **Postcondition:** do the method's precondition and the body's state relation imply the method's postcondition?
- **Termination:** does the method's precondition imply the body's termination condition?
- **Precondition:** does a statement's prestate knowledge imply the statement's precondition?
- **Loops:** is the loop invariant preserved, the measure well-formed (does not become negative) and decreased?
- **Type checking conditions:** are all formulas well-typed?
- **Specification validation:** does for every input that satisfies a precondition exist a result that does (not) satisfy the postcondition?

Partially solved by automatic decision procedure, partially by an interactive computer-supported proof.

# The Task States



The task status is indicated by color (icon).

- **Blue (sun)**: the task was solved in the current execution of the RISC ProgramExplorer (automatically or by an interactive proof).
- **Violet (partially clouded)**: the task was solved in a previous execution by an interactive proof.
  - Nothing has changed, so we need not perform the proof again.
  - However, we may replay the proof to investigate it.
- **Red (partially clouded)**: there exists a proof but it is either not complete or cannot be trusted any more (something has changed).
- **Red (fully clouded)**: there does not yet exist a proof.

Select “Execute Task” to start/replay a proof, “Show Proof” to display a proof, “Reset Task” to delete a proof.

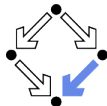
# A Postcondition Proof



The screenshot displays a proof assistant interface with the following components:

- Proof Tree:** A tree structure on the left showing the proof goal `[scat]: scatter` and its sub-goals: `[upf]: proved (CVCL)`, `[vpf]: split jm4`, `[rtb]: proved (CVCL)`, `[stb]: scatter`, and `[mlv]: proved (CVCL)`.
- Proof State:** A window on the right showing the current goal:  $\forall n \in \mathbb{Z}, n \in \mathbb{Z}. \text{if } n < m \text{ then } \text{sum}(m, n) = 0 \text{ else } \text{sum}(m, n) = n + \text{sum}(m, n-1) \text{ endif}$ . The proof state includes a hypothesis `htr` and a goal `if n_ind < 0 then returns_(now_) A value_(now_)+1 = 0 else returns_(now_) A (exists i in int: i = n_ind+1 A 1 <= i A value_(now_) = sum(1, i-1)) A n_ind < MAX_INT`.
- Children:** A list of children for the current goal: `[upf] [vpf]`.
- View Declarations:** A section for viewing declarations, including `newArray`, `y`, `x`, and `i`.
- Input/Output:** A section for input and output, showing the formula `goal_ already has a (skeleton) proof (proof status: trusted, closed, absolute)` and the message `Proof replay successful.`

# Linear Search



```
/*@..  
public class Searching  
{  
    public static int search(int[] a, int x) /*@..  
    {  
        int n = a.length;  
        int r = -1;  
        int i = 0;  
        while (i < n && r == -1) /*@..  
        {  
            if (a[i] == x)  
                r = i;  
            else  
                i = i+1;  
        }  
        return r;  
    }  
}
```

# The Representation of Arrays



The program type `int []` is mapped to the mathematical type `Base.IntArray`.

```
theory Base
{
  ...
  IntArray: TYPE =
    [#value: ARRAY int OF int, length: nat, null: BOOLEAN#];
  ...
}
```

- `(VAR a).length`: the number of elements in array *a*.
- `(VAR a).value[i]`: the element with index *i* in array *a*.
- `(VAR a).null`: *a* is the null pointer.

Program type *Class* is mapped to mathematical type *Class.Class*;  
*Class []* is mapped to *Class.Array*.



```
/*@  
theory uses Base {  
  int: TYPE = Base.int;  
  intArray: TYPE = Base.IntArray;  
  smallestPosition: FORMULA  
    FORALL(a: intArray, n: NAT, x: int):  
      (EXISTS(i:int): 0 <= i AND i < n AND a.value[i] = x) =>  
      (EXISTS(i:int): 0 <= i AND i < n AND a.value[i] = x AND  
        (FORALL(j:int): 0 <= j AND j < n AND a.value[j] = x =>  
          j >= i));  
}  
@*/  
public class Searching  
...
```



# Method Specification



```
public static int search(int[] a, int x) /*@
  requires (VAR a).null = FALSE;
  ensures
    LET result = VALUE@NEXT, n = (VAR a).length IN
    IF result = -1 THEN
      FORALL(i: INT): 0 <= i AND i < n =>
        (VAR a).value[i] /= VAR x
    ELSE
      0 <= result AND result < n AND
      (FORALL(i: INT): 0 <= i AND i < result =>
        (VAR a).value[i] /= VAR x) AND
      (VAR a).value[result] = VAR x
    ENDIF;
  @*/
  ...
```

# Loop Annotation



```
while (i < n && r == -1) /*@
  invariant (VAR a).null = FALSE AND VAR n = (VAR a).length
    AND 0 <= VAR i AND VAR i <= VAR n
    AND (FORALL(i: INT): 0 <= i AND i < VAR i =>
      (VAR a).value[i] /= VAR x)
    AND (VAR r = -1 OR (VAR r = VAR i AND VAR i < VAR n AND
      (VAR a).value[VAR r] = VAR x));
  decreases IF VAR r = -1 THEN VAR n - VAR i ELSE 0 ENDIF;
/*@/
{
  if (a[i] == x)
    r = i;
  else
    i = i+1;
}
```



## Transition Relation

$(\exists in \in \text{Base.int}, n \in \text{Base.int}:$

$n = \text{old } a.\text{length} \wedge (in \geq n \vee \text{value@next} \neq -1) \wedge 0 \leq in \wedge in \leq n$

$\wedge$

$(\forall i \in \mathbb{Z}: 0 \leq i \wedge i < in \Rightarrow \text{old } a.\text{value}[i] \neq \text{old } x)$

$\wedge$

$(\text{value@next} = -1$

$\vee$

$\text{value@next} = in \wedge in < n \wedge \text{old } a.\text{value}[\text{value@next}] = \text{old } x)) \wedge \neg \text{old } a.\text{null}$

$\wedge$




















$\text{returns@next}$

## Termination Condition

$\text{executes@now} \Rightarrow \text{old } a.\text{length} \geq 0$

# Verification Tasks



- ▼  method search
  -  [Searching.search] effects
  -  [Searching.search] postcondition
  -  [Searching.search] termination
- ▼  preconditions
  -  [Searching.search:0] declaration precondition
  -  [Searching.search:1] declaration precondition
  -  [Searching.search:2] while loop precondition
  -  [Searching.search:3] conditional precondition
  -  [Searching.search:4] assignment precondition
- ▼  loops
  -  [Searching.search:rb] invariant is preserved
  -  [Searching.search:rb] measure is well-formed
  -  [Searching.search:rb] measure is decreased
- ▼  type checking conditions
  -  [Searching.(local):p3x] value is in interval
  -  [Searching.(local):smu] value is in interval
  -  [Searching.(local):unx] value is in interval
- ▶  specification validation (optional)

# Invariant Proof



File Options Help

Semantics Verification Analysis

Proof Tree

- [l0]: decompose
  - [upf]: split ofx
    - [1j]: scatter
      - [elz]: auto
        - [kaw]: proved (CVCL)
      - [2j]: scatter
        - [2xa]: proved (CVCL)
        - [3xa]: proved (CVCL)
        - [4xa]: auto
          - [j43]: proved (CVCL)

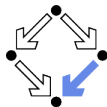
Declarations

$$\begin{aligned} & \wedge \\ & \quad r_{\text{new}} = r \wedge i_0 < n_{\text{old}} \\ & \quad \wedge \\ & \quad \quad i_{\text{new}} = i_0 + 1 \\ & \quad \text{endif} \\ \Rightarrow & \quad \neg a_{\text{old}}.\text{null} \wedge n_{\text{old}} = a_{\text{old}}.\text{length} \wedge 0 \leq i_{\text{new}} \\ & \quad \wedge \\ & \quad \quad i_{\text{new}} \leq n_{\text{old}} \\ & \quad \wedge \\ & \quad \quad (0 \leq i_1 \wedge i_1 < i_{\text{new}} \Rightarrow a_{\text{old}}.\text{value}[i_1] \neq x_{\text{old}}) \\ & \quad \wedge \\ & \quad \quad ( r_{\text{new}} = -1 \\ & \quad \quad \vee \\ & \quad \quad \quad r_{\text{new}} = i_{\text{new}} \wedge i_{\text{new}} < n_{\text{old}} \wedge a_{\text{old}}.\text{value}[r_{\text{new}}] = x_{\text{old}}) \end{aligned}$$

View Declarations

Input/Output

```
[#null:BOOLEAN, new:INT#, length:[0..MAX_INT], null:BOOLEAN#].
Value x:[0..MAX_INT].
Value y:[#value:ARRAY [MIN_INT..MAX_INT] OF [#null:BOOLEAN, new:INT#],
length:[0..MAX_INT], null:BOOLEAN#].
Value i:[0..MAX_INT].
Formula goal_ already has a (skeleton) proof (proof status: trusted, closed,
absolute)
Proof state [kaw] is closed by decision procedure.
Proof state [2xa] is closed by decision procedure.
Proof state [3xa] is closed by decision procedure.
Proof state [j43] is closed by decision procedure.
Proof replay successful.
Use 'proof goal_' to see proof.
```



# Working Strategy

---

- Develop theory.
  - Introduce interesting theorems that may be used in verifications.
- Develop specifications.
  - Validate specifications, e.g. by showing satisfiability and non-triviality.
- Develop program with annotations.
  - Validate programs/annotations by investigating program semantics.
- Prove postcondition and termination.
  - Partial and total correctness.
  - By proofs necessity of additional theorems may be detected.
- Prove precondition tasks and loop tasks.
  - By proofs necessity of additional theorems may be detected.
- Prove mathematical theorems.
  - Validation of auxiliary knowledge used in verifications.

The integrated development of theories, specifications, programs, annotations is crucial for the design of provably correct programs.