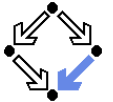
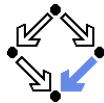


# Computer-Supported Program Verification with the RISC ProofNavigator

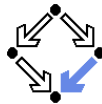
Wolfgang Schreiner  
Wolfgang.Schreiner@risc.uni-linz.ac.at

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



1. An Overview of the RISC ProofNavigator
2. Specifying Arrays
3. Verifying the Linear Search Algorithm
4. Conclusions

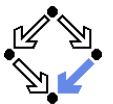
## The RISC ProofNavigator



- **An interactive proving assistant for program verification.**
  - Research Institute for Symbolic Computation (RISC), 2005–:  
<http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator>.
  - Development based on prior experience with PVS (SRI, 1993–).
  - Kernel and GUI implemented in Java.
  - Uses external SMT (satisfiability modulo theories) solver.
    - CVCL (Cooperating Validity Checker Lite) 2.0.
  - Runs under Linux (only); freely available as open source (GPL).
- **A language for the definition of logical theories.**
  - Based on a strongly typed higher-order logic (with subtypes).
  - Introduction of types, constants, functions, predicates.
- **Computer support for the construction of proofs.**
  - Commands for basic inference rules and combinations of such rules.
  - Applied interactively within a sequent calculus framework.
  - Top-down elaboration of proof trees.

Designed for simplicity of use; applied to non-trivial verifications.

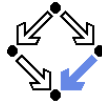
## Using the Software



For survey, see “Program Verification with the RISC ProofNavigator”.  
For details, see “The RISC ProofNavigator: Tutorial and Manual”.

- **Develop a theory.**
  - Text file with declarations of types, constants, functions, predicates.
  - Axioms (propositions assumed true) and formulas (to be proved).
- **Load the theory.**
  - File is read; declarations are parsed and type-checked.
  - Type-checking conditions are generated and proved.
- **Prove the formulas in the theory.**
  - Human-guided top-down elaboration of proof tree.
  - Steps are recorded for later replay of proof.
  - Proof status is recorded as “open” or “completed”.
- **Modify theory and repeat above steps.**
  - Software maintains dependencies of declarations and proofs.
  - Proofs whose dependencies have changed are tagged as “untrusted”.

## Starting the Software



### Starting the software:

ProofNavigator & (32 bit machines at RISC)  
ProofNavigator64 & (64 bit machines at RISC)

### Command line options:

Usage: ProofNavigator [OPTION]... [FILE]

FILE: name of file to be read on startup.

OPTION: one of the following options:

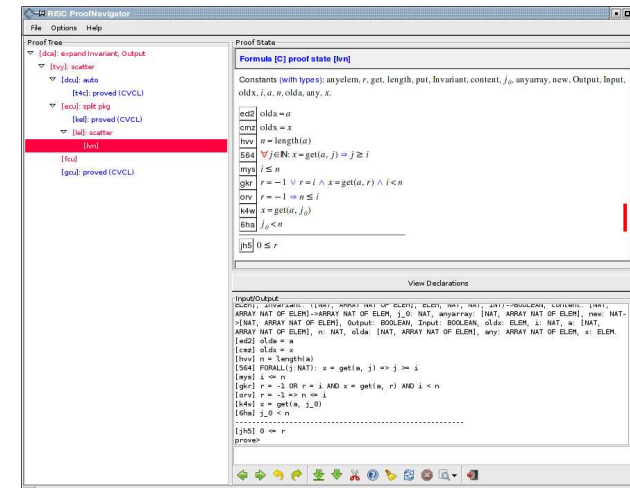
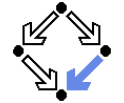
- n, --nogui: use command line interface.
- c, --context NAME: use subdir NAME to store context.
- cvcl PATH: PATH refers to executable "cvcl".
- s, --silent: omit startup message.
- h, --help: print this message.

### Repository stored in subdirectory of current working directory:

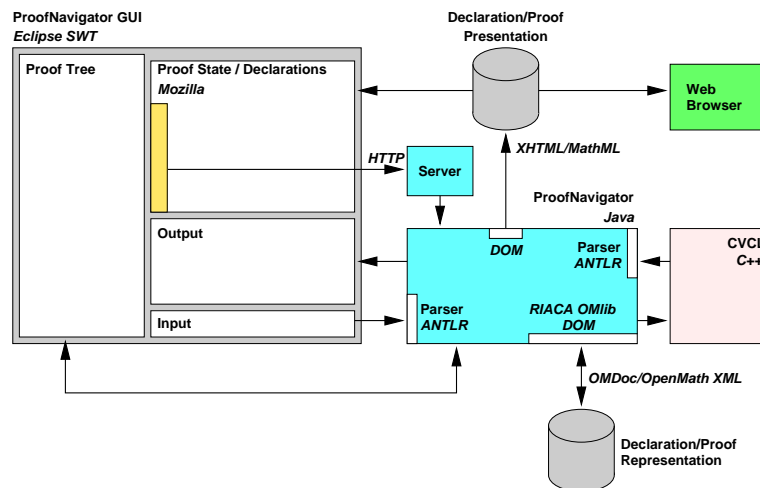
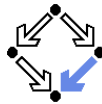
ProofNavigator/

- Option -c *dir* or command newcontext "*dir*" :
  - Switches to repository in directory *dir*.

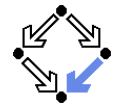
## The Graphical User Interface



## The Software Architecture



## Software Components



### Graphical user interface.

- Display of declarations and proof state.
- Embeds HTML browser as core component.

### Proof engine.

- Commands for navigating the proof.
- Interaction with validity checker to simplify/close proof states.

### Validity checker.

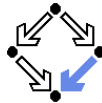
- Simplifies formulas
- Checks the validity of formulas.
- Produces counterexamples for (presumably) invalid formulas.

### Object repository.

- Proof persistence.
- Proof status management.

All data are externally represented in (gzipped) XML.

# A Theory



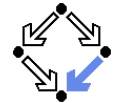
```
% switch repository to "sum"
newcontext "sum";

% the recursive definition of the sum from 0 to n
sum: NAT->NAT;
S1: AXIOM sum(0)=0;
S2: AXIOM FORALL(n:NAT): n>0 => sum(n)=n+sum(n-1);

% proof that explicit form is equivalent to recursive definition
S: FORMULA FORALL(n:NAT): sum(n) = (n+1)*n/2;
```

Declarations written with an external editor in a text file.

# Proving a Formula



When the file is loaded, the declarations are pretty-printed:

```
sum ∈ ℕ → ℕ
axiom S1 ≡ sum(0) = 0
axiom S2 ≡ ∀n ∈ ℕ: n > 0 ⇒ sum(n) = n + sum(n-1)
S ≡ ∀n ∈ ℕ: sum(n) = (n+1)·n / 2
```

The proof of a formula is started by the prove command.

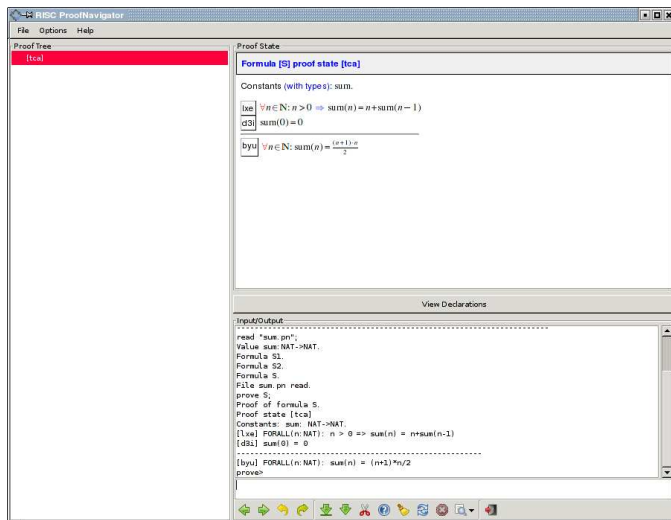
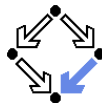
Formula S

prove S: Construct Proof

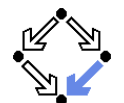
proof S: Show Proof

formula S: Print Formula

# Proving a Formula



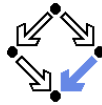
# Proving a Formula



- Proof of formula  $F$  is represented as a **tree**.
  - Each tree node denotes a **proof state (goal)**.
    - Logical sequent:  $A_1, A_2, \dots \vdash B_1, B_2, \dots$
    - Interpretation:  $(A_1 \wedge A_2 \wedge \dots) \Rightarrow (B_1 \vee B_2 \vee \dots)$
  - Initially single node  $Axioms \vdash F$ .
- The **tree must be expanded to completion**.
  - Every leaf must denote an obviously valid formula.
    - Some  $A_i$  is false or some  $B_j$  is true.
- A proof step consists of the **application of a proving rule to a goal**.
  - Either the goal is recognized as true.
  - Or the goal becomes the parent of a number of children (subgoals).  
The conjunction of the subgoals implies the parent goal.

$$\begin{array}{l} \text{Constants: } x_0 \in S_0, \dots \\ [L_1] \quad A_1 \\ \dots \\ [L_n] \quad A_n \\ \hline [L_{n+1}] \quad B_1 \\ \dots \\ [L_{n+m}] \quad B_m \end{array}$$

## An Open Proof Tree



Proof Tree

- ▼ [tca]: induction n in byu
  - [dbj]: proved (CVCL)
    - [ebj]

Formula [S] proof state [dbj]

Constants (with types): sum.

lxe  $\forall n \in \mathbb{N}: n > 0 \Rightarrow \text{sum}(n) = n + \text{sum}(n-1)$

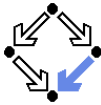
d3i  $\text{sum}(0) = 0$

nfq  $\text{sum}(0) = \frac{(0+1) \cdot 0}{2}$

Parent: [tca]

Closed goals are indicated in blue; goals that are open (or have open subgoals) are indicated in red. The red bar denotes the “current” goal.

## A Completed Proof Tree

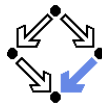


Proof Tree

- ▼ [tca]: induction n in byu
  - [dbj]: proved (CVCL)
    - ▼ [ebj]: instantiate n\_0+1 in lxe
      - [k5f]: proved (CVCL)

The visual representation of the complete proof structure; by clicking on a node, the corresponding proof state is displayed.

## Navigation Commands

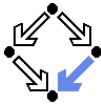


Various buttons support navigation in a proof tree.

- : prev
  - Go to previous open state in proof tree.
- : next
  - Go to next open state in proof tree.
- : undo
  - Undo the proof command that was issued in the parent of the current state; this discards the whole proof tree rooted in the parent.
- : redo
  - Redo the proof command that was previously issued in the current state but later undone; this restores the discarded proof tree.

Single click on a node in the proof tree displays the corresponding state; double click makes this state the current one.

## Proving Commands

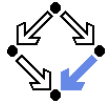


The most important proving commands can be also triggered by buttons.

- (scatter)
  - Recursively applies decomposition rules to the current proof state and to all generated child states; attempts to close the generated states by the application of a validity checker.
- (decompose)
  - Like scatter but generates a single child state only (no branching).
- (split)
  - Splits current state into multiple children states by applying rule to current goal formula (or a selected formula).
- (auto)
  - Attempts to close current state by instantiation of quantified formulas.
- (autostar)
  - Attempts to close current state and its siblings by instantiation.

Automatic decomposition of proofs and closing of proof states.

## Proving Commands

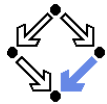


More commands can be selected from the menus.

- **assume**
  - Introduce a new assumption in the current state; generates a sibling state where this assumption has to be proved.
- **case:**
  - Split current state by a formula which is assumed as true in one child state and as false in the other.
- **expand:**
  - Expand the definitions of denoted constants, functions, or predicates.
- **lemma:**
  - Introduce another (previously proved) formula as new knowledge.
- **instantiate:**
  - Instantiate a universal assumption or an existential goal.
- **induction:**
  - Start an induction proof on a goal formula that is universally quantified over the natural numbers.

Here the creativity of the user is required!

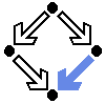
## Proving Strategies







- Initially: semi-automatic proof decomposition.
  - **expand** expands constant, function, and predicate definitions.
  - **scatter** aggressively decomposes a proof into subproofs.
  - **decompose** simplifies a proof state without branching.
  - **induction** for proofs over the natural numbers.
- Later: critical hints given by user.
  - **assume** and **case** cut proof states by conditions.
  - **instantiate** provide specific formula instantiations.
- Finally: simple proof states are yielded that can be automatically closed by the validity checker.
  - **auto** and **autostar** may help to close formulas by the heuristic instantiation of quantified formulas.

Appropriate combination of semi-automatic proof decomposition, critical hints given by the user, and the application of a validity checker is crucial.

## Auxiliary Commands



Some buttons have no command counterparts.

- : **counterexample**
  - Generate a “counterexample” for the current proof state, i.e. an interpretation of the constants that refutes the current goal.
- :
  - Abort current prover activity (proof state simplification or counterexample generation).
- :
  - Show menu that lists all commands and their (optional) arguments.
- :
  - Simplify current state (if automatic simplification is switched off).

More facilities for proof control.

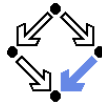
## 1. An Overview of the RISC ProofNavigator

## 2. Specifying Arrays

## 3. Verifying the Linear Search Algorithm

## 4. Conclusions

## A Constructive Definition of Arrays



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

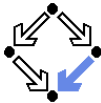
% the types
INDEX: TYPE = NAT;
ELEM:  TYPE;
ARR:   TYPE =
  [INDEX, ARRAY INDEX OF ELEM];

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

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

% the array operations
length: ARR -> INDEX =
  LAMBDA (a:ARR): a.0;
new: INDEX -> ARR =
  LAMBDA (n:INDEX): (n, any);
put: (ARR, INDEX, ELEM) -> ARR =
  LAMBDA (a:ARR, i:INDEX, e:ELEM):
    IF i < length(a)
      THEN (length(a),
            content(a) WITH [i]:=e)
      ELSE anyarray
ENDIF;
get: (ARR, INDEX) -> ELEM =
  LAMBDA (a:ARR, i:INDEX):
    IF i < length(a)
      THEN content(a)[i]
      ELSE anyelem
    ENDIF;
```

## Proof of Fundamental Array Properties



```
% the classical array axioms as formulas to be proved
length1: FORMULA
  FORALL (n:INDEX): length(new(n)) = n;

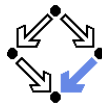
length2: FORMULA
  FORALL (a:ARR, i:INDEX, e:ELEM):
    i < length(a) => length(put(a, i, e)) = length(a);

get1: FORMULA
  FORALL (a:ARR, i:INDEX, e:ELEM):
    i < length(a) => get(put(a, i, e), i) = e;

get2: FORMULA
  FORALL (a:ARR, i, j:INDEX, e:ELEM):
    i < length(a) AND j < length(a) AND
    i /= j =>
      get(put(a, i, e), j) = get(a, j);
```

[adu]: expand length, get, put, content  
[c3b]: scatter  
[qid]: proved (CVCL)

## Proof of a Higher-Level Array Property



```
% extensionality on low-level arrays
extensionality: AXIOM
  FORALL (a, b:ARRAY INDEX OF ELEM):
    a=b <=> (FORALL (i:INDEX): a[i]=b[i]);

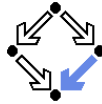
% unassigned parts hold identical values
unassigned: AXIOM
  FORALL (a:ARR, i:INT):
    (i >= length(a)) => content(a)[i]

% extensionality on arrays to be proved
equality: FORMULA
  FORALL (a:ARR, b:ARR): a = b <=>
    length(a) = length(b) AND
    (FORALL (i:INDEX): i < length(a) => get(a,i) = get(b,i));
```

[adt]: expand length, get, content  
[cw2]: scatter  
[qey]: proved (CVCL)  
[rey]: assume b\_0.1 = a\_0.1  
[zpt]: proved (CVCL)  
[1pt]: instantiate a\_0.1, b\_0.1 in 1fm  
[y51]: scatter  
[ku2]: auto  
[iub]: proved (CVCL)

1. An Overview of the RISC ProofNavigator
2. Specifying Arrays
3. Verifying the Linear Search Algorithm
4. Conclusions

# A Program Verification



Verification of the following Hoare triple:

$$\{olda = a \wedge oldx = x \wedge n = |a| \wedge i = 0 \wedge r = -1\}$$

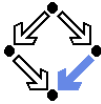
**while**  $i < n \wedge r = -1$  **do**

**if**  $a[i] = x$   
  **then**  $r := i$   
  **else**  $i := i + 1$

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

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

# The Verification Conditions



$A : \Leftrightarrow \text{Input} \Rightarrow \text{Invariant}$

$B_1 : \Leftrightarrow \text{Invariant} \wedge i < n \wedge r = -1 \wedge a[i] = x \Rightarrow \text{Invariant}[i/r]$

$B_2 : \Leftrightarrow \text{Invariant} \wedge i < n \wedge r = -1 \wedge a[i] \neq x \Rightarrow \text{Invariant}[i + 1/i]$

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

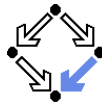
$\text{Input} : \Leftrightarrow olda = a \wedge oldx = x \wedge n = \text{length}(a) \wedge i = 0 \wedge r = -1$

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

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

The verification conditions  $A, B_1, B_2, C$  have to be proved.

# The Verification Conditions

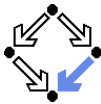


```

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

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

# The Verification Conditions (Contd)



...

A: FORMULA

Input => Invariant(a, x, i, n, r);

B1: FORMULA

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

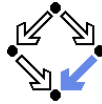
B2: FORMULA

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

C: FORMULA

Invariant(a, x, i, n, r) AND NOT(i < n AND r = -1)  
=> Output;

# The Proofs



A: [bca]: expand Input, Invariant  
[fuo]: scatter  
[bxg]: proved (CVCL)

(2 user actions)

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

(1 user action)

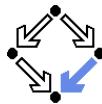
B2: [q1b]: expand Invariant in 6kv  
[six]: scatter  
[a1y]: auto  
[cch]: proved (CVCL)  
[b1y]: proved (CVCL)  
[c1y]: proved (CVCL)  
[d1y]: proved (CVCL)  
[e1y]: proved (CVCL)

(3 user actions)

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

(6 user actions)

# Conclusions



So what does this experience show us?

- Parts of a verification can be handled quite automatically:
  - Top-down proof decomposition.
  - Propositional logic reasoning.
  - Equality reasoning.
  - Linear arithmetic.
- Manual control for crucial “creative steps”
  - Expansion of definitions.
  - Proof cuts by assumptions/case distinctions.
  - Application of additional lemmas.
  - Instantiation of quantified formulas.

Proving assistants can do the essentially simple but usually tedious parts of the proof; the human nevertheless has to provide the creative insight.

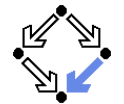
# 1. An Overview of the RISC ProofNavigator

## 2. Specifying Arrays

## 3. Verifying the Linear Search Algorithm

## 4. Conclusions

# Popular Proving Assistants



- **PVS:** <http://pvs.cs1.sri.com>
  - SRI (Software Research Institute) International, Menlo Park, CA.
  - Integrated environment for developing and analyzing formal specs.
  - Core system is implemented in Common Lisp.
  - Emacs-based frontend with Tcl/Tk-based GUI extensions.
- **Isabelle/HOL:** <http://isabelle.in.tum.de>
  - University of Cambridge and Technical University Munich.
  - Isabelle: generic theorem proving environment (aka “proof assistant”).
  - Isabelle/HOL: instance that uses higher order logic as framework.
  - Decisions procedures, tactics for interactive proof development.
- **Coq:** <http://coq.inria.fr>
  - LogiCal project, INRIA, France.
  - Formal proof management system (aka “proof assistant”).
  - “Calculus of inductive constructions” as logical framework.
  - Decision procedures, tactics support for interactive proof development.