Logic, Checking, and Proving

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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

http://www.risc.jku.at
1. The Language of Logic

2. The RISC Algorithm Language

3. The Art of Proving

4. The RISC ProofNavigator
The Language of Logic

Two kinds of syntactic phrases.

- **Term** $T$ denoting an object.
  - Variable $x$
  - Object constant $c$
  - Function application $f(T_1, \ldots, T_n)$ (may be written infix)
    - $n$-ary function constant $f$

- **Formula** $F$ denoting a truth value.
  - Atomic formula $p(T_1, \ldots, T_n)$ (may be written infix)
    - $n$-ary predicate constant $p$.
  - Negation $\neg F$ (“not $F$”)
  - Conjunction $F_1 \land F_2$ (“$F_1$ and $F_2$”)
  - Disjunction $F_1 \lor F_2$ (“$F_1$ or $F_2$”)
  - Implication $F_1 \Rightarrow F_2$ (“if $F_1$, then $F_2$”)
  - Equivalence $F_1 \iff F_2$ (“if $F_1$, then $F_2$, and vice versa”)
  - Universal quantification $\forall x : F$ (“for all $x$, $F$”)
  - Existential quantification $\exists x : F$ (“for some $x$, $F$”)

Wolfgang Schreiner

http://www.risc.jku.at
Syntactic Shortcuts

- $\forall x_1, \ldots, x_n : F$
  - $\forall x_1 : \ldots : \forall x_n : F$
- $\exists x_1, \ldots, x_n : F$
  - $\exists x_1 : \ldots : \exists x_n : F$
- $\forall x \in S : F$
  - $\forall x : x \in S \Rightarrow F$
- $\exists x \in S : F$
  - $\exists x : x \in S \land F$

Help to make formulas more readable.
Examples

Terms and formulas may appear in various syntactic forms.

- **Terms:**
  \[
  \begin{align*}
  \exp(x) \\
  a \cdot b + 1 \\
  a[i] \cdot b \\
  \sqrt{\frac{x^2+2x+1}{(y+1)^2}}
  \end{align*}
  \]

- **Formulas:**
  \[
  \begin{align*}
  a^2 + b^2 &= c^2 \\
  n \mid 2n \\
  \forall x \in \mathbb{N} : x \geq 0 \\
  \forall x \in \mathbb{N} : 2|x \lor 2|(x + 1) \\
  \forall x \in \mathbb{N}, y \in \mathbb{N} : x < y \Rightarrow \\
  \exists z \in \mathbb{N} : x + z = y
  \end{align*}
  \]

Terms and formulas may be nested arbitrarily deeply.
The Meaning of Formulas

- **Atomic formula** $p(T_1, \ldots, T_n)$
  - True if the predicate denoted by $p$ holds for the values of $T_1, \ldots, T_n$.
- **Negation** $\neg F$
  - True if and only if $F$ is false.
- **Conjunction** $F_1 \land F_2$ (“$F_1$ and $F_2$”)
  - True if and only if $F_1$ and $F_2$ are both true.
- **Disjunction** $F_1 \lor F_2$ (“$F_1$ or $F_2$”)
  - True if and only if at least one of $F_1$ or $F_2$ is true.
- **Implication** $F_1 \Rightarrow F_2$ (“if $F_1$, then $F_2$”)
  - False if and only if $F_1$ is true and $F_2$ is false.
- **Equivalence** $F_1 \Leftrightarrow F_2$ (“if $F_1$, then $F_2$, and vice versa”)
  - True if and only if $F_1$ and $F_2$ are both true or both false.
- **Universal quantification** $\forall x : F$ (“for all $x$, $F$”)
  - True if and only if $F$ is true for every possible value assignment of $x$.
- **Existential quantification** $\exists x : F$ (“for some $x$, $F$”)
  - True if and only if $F$ is true for at least one value assignment of $x$. 
Example

We assume the domain of natural numbers and the “classical” interpretation of constants 1, 2, +, =, <.

- $1 + 1 = 2$
  - True.
- $1 + 1 = 2 \lor 2 + 2 = 2$
  - True.
- $1 + 1 = 2 \land 2 + 2 = 2$
  - False.
- $1 + 1 = 2 \Rightarrow 2 = 1 + 1$
  - True.
- $1 + 1 = 1 \Rightarrow 2 + 2 = 2$
  - True.
- $1 + 1 = 2 \Rightarrow 2 + 2 = 2$
  - False.
- $1 + 1 = 1 \Leftrightarrow 2 + 2 = 2$
  - True.
Example

- $x + 1 = 1 + x$
  - True, for every assignment of a number $a$ to variable $x$.
- $\forall x : x + 1 = 1 + x$
  - True (because for every assignment $a$ to $x$, $x + 1 = 1 + x$ is true).
- $x + 1 = 2$
  - If $x$ is assigned “one”, the formula is true.
  - If $x$ is assigned “two”, the formula is false.
- $\exists x : x + 1 = 2$
  - True (because $x + 1 = 2$ is true for assignment “one” to $x$).
- $\forall x : x + 1 = 2$
  - False (because $x + 1 = 2$ is false for assignment “two” to $x$).
- $\forall x : \exists y : x < y$
  - True (because for every assignment $a$ to $x$, there exists the assignment $a + 1$ to $y$ which makes $x < y$ true).
- $\exists y : \forall x : x < y$
  - False (because for every assignment $a$ to $y$, there is the assignment $a + 1$ to $x$ which makes $x < y$ false).
Formula Equivalences

Formulas may be replaced by equivalent formulas.

- \neg \neg F_1 \iff F_1
- \neg (F_1 \land F_2) \iff \neg F_1 \lor \neg F_2
- \neg (F_1 \lor F_2) \iff \neg F_1 \land \neg F_2
- \neg (F_1 \Rightarrow F_2) \iff F_1 \land \neg F_2
- \neg \forall x : F \iff \exists x : \neg F
- \neg \exists x : F \iff \forall x : \neg F
- F_1 \Rightarrow F_2 \iff \neg F_2 \Rightarrow \neg F_1
- F_1 \Rightarrow F_2 \iff \neg F_1 \lor F_2
- F_1 \iff F_2 \iff \neg F_1 \iff \neg F_2
- \ldots

Familiarity with manipulation of formulas is important.
Example

- “All swans are white or black.”
  \[ \forall x : \text{swan}(x) \Rightarrow \text{white}(x) \lor \text{black}(x) \]
- “There exists a black swan.”
  \[ \exists x : \text{swan}(x) \land \text{black}(x) \]
- “A swan is white, unless it is black.”
  \[ \forall x : \text{swan}(x) \land \neg \text{black}(x) \Rightarrow \text{white}(x) \]
  \[ \forall x : \text{swan}(x) \land \neg \text{white}(x) \Rightarrow \text{black}(x) \]
  \[ \forall x : \text{swan}(x) \Rightarrow \text{white}(x) \lor \text{black}(x) \]
- “Not everything that is white or black is a swan.”
  \[ \neg \forall x : \text{white}(x) \lor \text{black}(x) \Rightarrow \text{swan}(x). \]
  \[ \exists x : (\text{white}(x) \lor \text{black}(x)) \land \neg \text{swan}(x). \]
- “Black swans have at least one black parent”.
  \[ \forall x : \text{swan}(x) \land \text{black}(x) \Rightarrow \exists y : \text{swan}(y) \land \text{black}(y) \land \text{parent}(y, x) \]

It is important to recognize the logical structure of an informal sentence in its various equivalent forms.
The Usage of Formulas

Precise formulation of statements describing object relationships.

- **Statement:**
  
  If $x$ and $y$ are natural numbers and $y$ is not zero, then $q$ is the truncated quotient of $x$ divided by $y$.

- **Formula:**

  \[ x \in \mathbb{N} \land y \in \mathbb{N} \land y \neq 0 \Rightarrow q \in \mathbb{N} \land \exists r \in \mathbb{N} : x = y \cdot q + r \land r < y \]

- **Problem specification:**

  Given natural numbers $x$ and $y$ such that $y$ is not zero, compute the truncated quotient $q$ of $x$ divided by $y$.

  - **Inputs:** $x, y$
  - **Input condition:** $x \in \mathbb{N} \land y \in \mathbb{N} \land y \neq 0$
  - **Output:** $q$
  - **Output condition:** $q \in \mathbb{N} \land \exists r \in \mathbb{N} : x = y \cdot q + r \land r < y$

Wolfgang Schreiner

http://www.risc.jku.at
Problem Specifications

- The specification of a computation problem:
  - Input: variables \( x_1 \in S_1, \ldots, x_n \in S_n \)
  - Input condition ("precondition"): formula \( I(x_1, \ldots, x_n) \).
  - Output: variables \( y_1 \in T_1, \ldots, y_m \in T_n \)
  - Output condition ("postcondition"): \( O(x_1, \ldots, x_n, y_1, \ldots, y_m) \).
    - \( F(x_1, \ldots, x_n) \): only \( x_1, \ldots, x_n \) are free in formula \( F \).
    - \( x \) is free in \( F \), if not every occurrence of \( x \) is inside the scope of a quantifier (such as \( \forall \) or \( \exists \)) that binds \( x \).

- An implementation of the specification:
  - A function (program) \( f : S_1 \times \ldots \times S_n \rightarrow T_1 \times \ldots \times T_m \) such that
    \[
    \forall x_1 \in S_1, \ldots, x_n \in S_n : I(x_1, \ldots, x_n) \Rightarrow \let (y_1, \ldots, y_m) = f(x_1, \ldots, x_n) \in O(x_1, \ldots, x_n, y_1, \ldots, y_m)
    \]
  - For all arguments that satisfy the input condition, \( f \) must compute results that satisfy the output condition.

Basis of all specification formalisms.

Wolfgang Schreiner
http://www.risc.jku.at
Example: A Problem Specification

Given an integer array \( a \), a position \( p \) in \( a \), and a length \( l \), return the array \( b \) derived from \( a \) by removing \( a[p], \ldots, a[p + l - 1] \).

- **Input:** \( a \in \mathbb{Z}^* \), \( p \in \mathbb{N} \), \( l \in \mathbb{N} \)
- **Input condition:**
  \[ p + l \leq \text{length}(a) \]
- **Output:** \( b \in \mathbb{Z}^* \)
- **Output condition:**
  
  ```
  \text{let } n = \text{length}(a) \text{ in } \\
  \text{length}(b) = n - l \land \\
  (\forall i \in \mathbb{N} : i < p \Rightarrow b[i] = a[i]) \land \\
  (\forall i \in \mathbb{N} : p \leq i < n - l \Rightarrow b[i] = a[i + l])
  ```

**Mathematical theory:**

\[
T^* := \bigcup_{i \in \mathbb{N}} T^i, \quad T^i := \mathbb{N}_i \rightarrow T, \quad \mathbb{N}_i := \{ n \in \mathbb{N} : n < i \} \\
\text{length} : T^* \rightarrow \mathbb{N}, \quad \text{length}(a) = \textbf{such } i \in \mathbb{N} : a \in T^i
\]
Validating Problem Specifications

Do formal input condition $I(x)$ and output condition $O(x, y)$ really capture our informal intentions?

- Do concrete inputs/output satisfy/violate these conditions?
  - $I(a_1), \neg I(a_2), O(a_1, b_1), \neg O(a_1, b_2)$.
- Is input condition satisfiable?
  - $\exists x : I(x)$.
- Is input condition not trivial?
  - $\exists x : \neg I(x)$.
- Is output condition satisfiable for every input?
  - $\forall x : I(x) \Rightarrow \exists y : O(x, y)$.
- Is output condition for all (at least some) inputs not trivial?
  - $\forall x : I(x) \Rightarrow \exists y : \neg O(x, y)$.
  - $\forall x : I(x) \wedge \exists y : \neg O(x, y)$.
- Is for every legal input at most one output legal?
  - $\forall x : I(x) \Rightarrow \forall y_1, y_2 : O(x, y_1) \wedge O(x, y_2) \Rightarrow y_1 = y_2$.

Validate specification to increase our confidence in its meaning!
1. The Language of Logic

2. The RISC Algorithm Language

3. The Art of Proving

4. The RISC ProofNavigator
The RISC Algorithm Language (RISCAL)

- A system for formally specifying and checking algorithms.
  - Research Institute for Symbolic Computation (RISC), 2016–.
    http://www.risc.jku.at/research/formal/software/RISCAL.
  - Implemented in Java with SWT library for the GUI.
    - Tested under Linux only; freely available as open source (GPL3).
- A language for the defining mathematical theories and algorithms.
  - A static type system with only finite types (of parameterized sizes).
  - Predicates, explicitly (also recursively) and implicitly defined functions.
  - Theorems (universally quantified predicates expected to be true).
  - Procedures (also recursively defined).
  - Pre- and post-conditions, invariants, termination measures.
- A framework for evaluating/executing all definitions.
  - Model checking: predicates, functions, theorems, procedures, annotations may be evaluated/executed for all possible inputs.
  - All paths of a non-deterministic execution may be elaborated.
  - The execution/evaluation may be visualized.

Validating algorithms by automatically verifying finite approximations.
The RISC Algorithm Language (RISCAL)

RISCAL divide.txt &

Wolfgang Schreiner
http://www.risc.jku.at
Using RISCAL

See also the (printed/online) ‘‘Tutorial and Reference Manual’’.

- Press button (or <Ctrl>-s) to save specification.
  - Automatically processes (parses and type-checks) specification.
  - Press button to re-process specification.
- Choose values for undefined constants in specification.
  - Natural number for val const: \( \mathbb{N} \).
  - Default Value: used if no other value is specified.
  - Other Values: specific values for individual constants.
- Select Operation from menu and then press button .
  - Executes operation for chosen constant values and all possible inputs.
  - Option Silent: result of operation is not printed.
  - Option Nondeterminism: all execution paths are taken.
  - Option Multi-threaded: multiple threads execute different inputs.
  - Press button to abort execution.

During evaluation all annotations (pre/postconditions, etc.) are checked.
## Typing Mathematical Symbols

<table>
<thead>
<tr>
<th>ASCII String</th>
<th>Unicode Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>ℤ</td>
</tr>
<tr>
<td>Nat</td>
<td>ℕ</td>
</tr>
<tr>
<td>:=</td>
<td>:=</td>
</tr>
<tr>
<td>true</td>
<td>⊤</td>
</tr>
<tr>
<td>false</td>
<td>⊥</td>
</tr>
<tr>
<td>~</td>
<td>¬</td>
</tr>
<tr>
<td>\u2227</td>
<td>∧</td>
</tr>
<tr>
<td>\u2228</td>
<td>∨</td>
</tr>
<tr>
<td>=&gt;</td>
<td>⇒</td>
</tr>
<tr>
<td>&lt;=</td>
<td>⇔</td>
</tr>
<tr>
<td>\forall</td>
<td>∀</td>
</tr>
<tr>
<td>\exists</td>
<td>∈</td>
</tr>
<tr>
<td>\sum</td>
<td>∑</td>
</tr>
<tr>
<td>\prod</td>
<td>\prod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII String</th>
<th>Unicode Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠</td>
<td>\neq</td>
</tr>
<tr>
<td>≤</td>
<td>\leq</td>
</tr>
<tr>
<td>≥</td>
<td>\geq</td>
</tr>
<tr>
<td>*</td>
<td>\ast</td>
</tr>
<tr>
<td>times</td>
<td>\times</td>
</tr>
<tr>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\cap</td>
<td>\cap</td>
</tr>
<tr>
<td>\cup</td>
<td>\cup</td>
</tr>
<tr>
<td>\langle</td>
<td>\langle</td>
</tr>
<tr>
<td>\rangle</td>
<td>\rangle</td>
</tr>
<tr>
<td>\subseteq</td>
<td>\subseteq</td>
</tr>
<tr>
<td>∈</td>
<td>\in</td>
</tr>
<tr>
<td>\subseteqq</td>
<td>\subseteqq</td>
</tr>
</tbody>
</table>

Type the ASCII string and press `<Ctrl>-#` to get the Unicode character.
Example: Quotient and Remainder

Given natural numbers \( n \) and \( m \), we want to compute the quotient \( q \) and remainder \( r \) of \( n \) divided by \( m \).

// the type of natural numbers less than equal \( \mathbb{N} \)
val N: \( \mathbb{N} \);
type Num = \( \mathbb{N}[\mathbb{N}] \);

// the precondition of the computation
pred pre(n:Num, m:Num) ⇔ m \neq 0;

// the postcondition, first formulation
pred post1(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m \cdot q + r \land
  \forall q0:Num, r0:Num.
  n = m \cdot q0 + r0 \Rightarrow r \leq r0;

// the postcondition, second formulation
pred post2(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m \cdot q + r \land r < m;

We will investigate this specification.
Example: Quotient and Remainder

// for all inputs that satisfy the precondition
// both formulations are equivalent:
// pre(n, m) ⇒ (post1(n, m, q, r) ⇔ post2(n, m, q, r));
theorem postEquiv(n:Num, m:Num, q:Num, r:Num)
  requires pre(n, m);
⇔ post1(n, m, q, r) ⇔ post2(n, m, q, r);

// we will thus use the simpler formulation from now on
pred post(n:Num, m:Num, q:Num, r:Num) ⇔ post2(n, m, q, r);

Check equivalence for all values that satisfy the precondition.
Example: Quotient and Remainder

Choose e.g. value 5 for $N$.

- **Switch option Silent off:**
  
  Executing $\text{postEquiv}(\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$ with all 1296 inputs.
  Ignoring inadmissible inputs...
  Run 6 of deterministic function $\text{postEquiv}(0, 1, 0, 0)$:
  Result (0 ms): true
  Run 7 of deterministic function $\text{postEquiv}(1, 1, 0, 0)$:
  Result (0 ms): true
  ...
  Run 1295 of deterministic function $\text{postEquiv}(5, 5, 5, 5)$:
  Result (0 ms): true
  Execution completed for ALL inputs (6314 ms, 1080 checked, 216 inadmissible).

- **Switch option Silent on:**
  
  Executing $\text{postEquiv}(\mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z})$ with all 1296 inputs.
  Execution completed for ALL inputs (244 ms, 1080 checked, 216 inadmissible).

If theorem is false for some input, an error message is displayed.
Example: Quotient and Remainder

Drop precondition from theorem.

```
theorem postEquiv(n:Num, m:Num, q:Num, r:Num) ⇔
   // requires pre(n, m);
   post1(n, m, q, r) ⇔ post2(n, m, q, r);
```

Executing postEquiv(Z,Z,Z,Z) with all 1296 inputs.
Run 0 of deterministic function postEquiv(0,0,0,0):
ERROR in execution of postEquiv(0,0,0,0): evaluation of
   postEquiv
at line 25 in file divide.txt:
   theorem is not true
ERROR encountered in execution.

For $n = 0, m = 0, q = 0, r = 0$, the modified theorem is not true.
Visualizing the Formula Evaluation

Select $N = 1$ and visualization option “Tree”.

Investigate the (pruned) evaluation tree to determine how the truth value of a formula was derived (double click to zoom into/out of predicates).
Example: Quotient and Remainder

Switch option “Nondeterminism” on.

// 1. investigate whether the specified input/output combinations are as desired
fun quotremFun(n:Num, m:Num): Tuple[Num,Num]
    requires pre(n, m);
= choose q:Num, r:Num with post(n, m, q, r);

Executing quotremFun($\mathbb{Z}$,$\mathbb{Z}$) with all 36 inputs.
Ignoring inadmissible inputs...
Branch 0:6 of nondeterministic function quotremFun(0,1):
Result (0 ms): [0,0]
Branch 1:6 of nondeterministic function quotremFun(0,1):
No more results (8 ms).
...
Branch 0:35 of nondeterministic function quotremFun(5,5):
Result (0 ms): [1,0]
Branch 1:35 of nondeterministic function quotremFun(5,5):
No more results (14 ms).
Execution completed for ALL inputs (413 ms, 30 checked, 6 inadmissible).

First validation by inspecting the values determined by output condition
(nondeterminism may produce for some inputs multiple outputs).
Example: Quotient and Remainder

// 2. check that some but not all inputs are allowed
theorem someInput() ⇔ ∃n:Num, m:Num. pre(n, m);
theorem notEveryInput() ⇔ ∃n:Num, m:Num. ¬pre(n, m);

Executing someInput().
Execution completed (0 ms).
Executing notEveryInput().
Execution completed (0 ms).

A very rough validation of the input condition.
Example: Quotient and Remainder

// 3. check whether for all inputs that satisfy the precondition
// there are some outputs that satisfy the postcondition
theorem someOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. post(n, m, q, r);

// 4. check that not every output satisfies the postcondition
theorem notEveryOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. ¬post(n, m, q, r);

Executing someOutput(\mathbb{Z}, \mathbb{Z}) with all 36 inputs.
Execution completed for ALL inputs (5 ms, 30 checked, 6 inadmissible).
Executing notEveryOutput(\mathbb{Z}, \mathbb{Z}) with all 36 inputs.
Execution completed for ALL inputs (5 ms, 30 checked, 6 inadmissible).

A very rough validation of the output condition.
Example: Quotient and Remainder

// 5. check that the output is uniquely defined
// (optional, need not generally be the case)
theorem uniqueOutput(n:Num, m:Num)
  requires pre(n, m);
⇔
  ∀q:Num, r:Num. post(n, m, q, r) ⇒
  ∀q0:Num, r0:Num. post(n, m, q0, r0) ⇒
    q = q0 ∧ r = r0;

Executing uniqueOutput(\mathbb{Z}, \mathbb{Z}) with all 36 inputs.
Execution completed for ALL inputs (18 ms, 30 checked, 6 inadmissible).

The output condition indeed determines the outputs uniquely.
Example: Quotient and Remainder

// 6. check whether the algorithm satisfies the specification
proc quotRemProc(n:Num, m:Num): Tuple[Num,Num]
    requires pre(n, m);
    ensures let q=result.1, r=result.2 in post(n, m, q, r);
{
    var q: Num = 0;
    var r: Num = n;
    while r ≥ m do
    {
        r := r-m;
        q := q+1;
    }
    return ⟨q,r⟩;
}

Check whether the algorithm satisfies the specification.
Example: Quotient and Remainder

Executing quotRemProc($\mathbb{Z}, \mathbb{Z}$) with all 36 inputs.
Ignoring inadmissible inputs...
Run 6 of deterministic function quotRemProc(0,1):
Result (0 ms): [0,0]
Run 7 of deterministic function quotRemProc(1,1):
Result (0 ms): [1,0]
...
Run 31 of deterministic function quotRemProc(1,5):
Result (1 ms): [0,1]
Run 32 of deterministic function quotRemProc(2,5):
Result (0 ms): [0,2]
Run 33 of deterministic function quotRemProc(3,5):
Result (0 ms): [0,3]
Run 34 of deterministic function quotRemProc(4,5):
Result (0 ms): [0,4]
Run 35 of deterministic function quotRemProc(5,5):
Result (1 ms): [1,0]
Execution completed for ALL inputs (161 ms, 30 checked, 6 inadmissible).

A verification of the algorithm by checking all possible executions.
Example: Quotient and Remainder

```plaintext
proc quotRemProc(n:Num, m:Num): Tuple[Num,Num]
    requires pre(n, m);
    ensures post(n, m, result.1, result.2);
{
    var q: Num = 0;
    var r: Num = n;
    while r > m do // error!
    {
        r := r-m;
        q := q+1;
    }
    return ⟨q,r⟩;
}

Executing quotRemProc(Z,Z) with all 36 inputs.
ERROR in execution of quotRemProc(1,1): evaluation of
    ensures let q = result.1, r = result.2 in post(n, m, q, r);
at line 65 in file divide.txt:
    postcondition is violated by result [0,1]
ERROR encountered in execution.

A falsification of an incorrect algorithm.
```

Wolfgang Schreiner  
http://www.risc.jku.at
Example: Sorting an Array

val N:Nat; val M:Nat;
type nat = Nat[M]; type array = Array[N,nat]; type index = Nat[N-1];

proc sort(a:array): array
  ensures ∀i:nat. i < N-1 ⇒ result[i] ≤ result[i+1];
  ensures ∃p:Array[N,index].
    (∀i,index,j:index. i ≠ j ⇒ p[i] ≠ p[j]) ∧
    (∀i:index. a[i] = result[p[i]]);
{
  var b:array = a;
  for var i:Nat[N]:=1; i<N; i:=i+1 do {
    var x:nat := b[i];
    var j:Int[-1,N] := i-1;
    while j ≥ 0 ∧ b[j] > x do {
      b[j+1] := b[j];
      j := j-1;
    }
    b[j+1] := x;
  }
  return b;
}
Example: Sorting an Array

Using N=5.
Using M=5.
Type checking and translation completed.
Executing sort(Array[ℤ]) with all 7776 inputs.
1223 inputs (1223 checked, 0 inadmissible, 0 ignored)...
2026 inputs (2026 checked, 0 inadmissible, 0 ignored)...
...
5114 inputs (5114 checked, 0 inadmissible, 0 ignored)...
5467 inputs (5467 checked, 0 inadmissible, 0 ignored)...
5792 inputs (5792 checked, 0 inadmissible, 0 ignored)...
6118 inputs (6118 checked, 0 inadmissible, 0 ignored)...
6500 inputs (6500 checked, 0 inadmissible, 0 ignored)...
6788 inputs (6788 checked, 0 inadmissible, 0 ignored)...
7070 inputs (7070 checked, 0 inadmissible, 0 ignored)...
7354 inputs (7354 checked, 0 inadmissible, 0 ignored)...
7634 inputs (7634 checked, 0 inadmissible, 0 ignored)...
Execution completed for ALL inputs (32606 ms, 7776 checked, 0 inadmissible).
Not all nondeterministic branches may have been considered.

Also this algorithm can be automatically checked.

Wolfgang Schreiner http://www.risc.jku.at
33/67
Example: Sorting an Array

Select operation sort and press the button "Show/Hide Tasks".

Automatically generated formulas to validate procedure specifications.
Example: Sorting an Array

Right-click to print definition of a formula, double-click to check it.

For every input, is postcondition true for only one output?

theorem _sort_0_PostUnique(a:array) ⇐
∀result:array with

(∀i:index. ((i < (N-1)) ⇒ (result[i] ≤ result[i+1]))) ∧
(∃p:Array[N,index]. (∀i:index, j:index. ((i ≠ j) ⇒ (p[i] ≠ p[j]))) ∧
(∀i:index. (a[i] = result[p[i]]))).

(∀_result:array with let result = _result in #
(∀i:index. ((i < (N-1)) ⇒ (result[i] ≤ result[i+1]))) ∧
(∃p:Array[N,index]. (∀i:index, j:index. ((i ≠ j) ⇒ (p[i] ≠ p[j]))) ∧
(∀i:index. (a[i] = result[p[i]])))).

(result = _result));

Using N=3.
Using M=3.
Type checking and translation completed.
Executing _sort_0_PostUnique(Array[Z]) with all 64 inputs.
Execution completed for ALL inputs (529 ms, 64 checked, 0 inadmissible).

The output is indeed uniquely defined by the output condition.
Model Checking versus Proving

Two fundamental techniques for the verification of computer programs.

- **Checking Program Executions**
  - Enumeration of all possible executions and evaluation of formulas (e.g. postconditions) on the resulting states.
  - Fully automatic, no human interaction is required.
  - Only possible if there are only finitely many executions (and finitely many values for the quantified variables in the formulas).
  - State space explosion: “finitely many” means “not too many”.

- **Proving Verification Conditions**
  - Logic formulas that are valid if and only if program is correct with respect to its specification.
  - Also possible if there are infinitely many executions and infinitely many values for the quantified variables.
  - Many conditions can be automatically proved (automated reasoners); in general interaction with human is required (proof assistants).

General verification requires the proving of logic formulas.
1. The Language of Logic

2. The RISC Algorithm Language

3. The Art of Proving

4. The RISC ProofNavigator
Proofs

A proof is a structured argument that a formula is true.

- A tree whose nodes represent proof situations (states).

```
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
  ↖   ↗   ↖   ↗   ↖
  *   *   *   *   *
```

- Each proof situation consists of knowledge and a goal.
  - $K_1, \ldots, K_n \vdash G$

  - Knowledge $K_1, \ldots, K_n$: formulas assumed to be true.
  - Goal $G$: formula to be proved relative to knowledge.

- The root of the tree is the initial proof situation.
  - $K_1, \ldots, K_n$: axioms of mathematical background theories.
  - $G$: formula to be proved.
A proof rules describes how a proof situation can be reduced to zero, one, or more “subsituations”.

\[
\begin{array}{c}
\vdash \ldots \vdash \ldots \vdash \ldots \\
K_1, \ldots, K_n \vdash G
\end{array}
\]

- Rule may or may not close the (sub)proof:
  - Zero subsituations: $G$ has been proved, (sub)proof is closed.
  - One or more subsituations: $G$ is proved, if all subgoals are proved.

- **Top-down rules:** focus on $G$.
  - $G$ is decomposed into simpler goals $G_1, G_2, \ldots$

- **Bottom-up rules:** focus on $K_1, \ldots, K_n$.
  - Knowledge is extended to $K_1, \ldots, K_n, K_{n+1}$.

In each proof situation, we aim at showing that the goal is “apparently” true with respect to the given knowledge.
Conjunction $F_1 \land F_2$

\[
\begin{align*}
K \vdash G_1 & \quad K \vdash G_2 \\
\hline
K \vdash G_1 \land G_2
\end{align*}
\]

- **Goal** $G_1 \land G_2$.
  - Create two subsituations with goals $G_1$ and $G_2$.
    - *We have to show* $G_1 \land G_2$.
      - *We show* $G_1$: $\ldots$ *(proof continues with goal $G_1$)*
      - *We show* $G_2$: $\ldots$ *(proof continues with goal $G_2$)*

- **Knowledge** $K_1 \land K_2$.
  - Create one subsituation with $K_1$ and $K_2$ in knowledge.
    - *We know* $K_1 \land K_2$. *We thus also know* $K_1$ and $K_2$.
      * (proof continues with current goal and additional knowledge $K_1$ and $K_2$)
Disjunction $F_1 \lor F_2$

\[
\begin{align*}
K, \neg G_1 & \vdash G_2 \\
\hline
K & \vdash G_1 \lor G_2
\end{align*}
\quad
\begin{align*}
\ldots, K_1 & \vdash G \\
\ldots, K_2 & \vdash G \\
\hline
\ldots, K_1 \lor K_2 & \vdash G
\end{align*}
\]

- **Goal** $G_1 \lor G_2$.
  - Create one subsituation where $G_2$ is proved under the assumption that $G_1$ does not hold (or vice versa):
    
    We have to show $G_1 \lor G_2$. We assume $\neg G_1$ and show $G_2$.

    (proof continues with goal $G_2$ and additional knowledge $\neg G_1$)

- **Knowledge** $K_1 \lor K_2$.
  - Create two subsituations, one with $K_1$ and one with $K_2$ in knowledge.
    
    We know $K_1 \lor K_2$. We thus proceed by case distinction:
    
    - **Case** $K_1$: \ldots (proof continues with current goal and additional knowledge $K_1$).
    - **Case** $K_2$: \ldots (proof continues with current goal and additional knowledge $K_2$).
Implication $F_1 \Rightarrow F_2$

\[
\frac{K, G_1 \vdash G_2}{K \vdash G_1 \Rightarrow G_2} \quad \frac{\ldots \vdash K_1 \quad \ldots, K_2 \vdash G}{\ldots, K_1 \Rightarrow K_2 \vdash G}
\]

- **Goal** $G_1 \Rightarrow G_2$
  - Create one subsituation where $G_2$ is proved under the assumption that $G_1$ holds:
    
    \[\text{We have to show } G_1 \Rightarrow G_2. \text{ We assume } G_1 \text{ and show } G_2.\]
    
    \[(\text{proof continues with goal } G_2 \text{ and additional knowledge } G_1)\]

- **Knowledge** $K_1 \Rightarrow K_2$
  - Create two subsituation, one with goal $K_1$ and one with knowledge $K_2$.
    
    \[\text{We know } K_1 \Rightarrow K_2.\]
    
    \[\text{We show } K_1: \ldots (\text{proof continues with goal } K_1)\]
    
    \[\text{We know } K_2: \ldots (\text{proof continues with current goal and additional knowledge } K_2).\]
Equivalence $F_1 \iff F_2$

\[
\frac{\begin{array}{c}
K \vdash G_1 \Rightarrow G_2 \\
K \vdash G_2 \Rightarrow G_1
\end{array}}{K \vdash G_1 \iff G_2}
\frac{\begin{array}{c}
\ldots \vdash (\neg)K_1 \\
\ldots, (\neg)K_2 \vdash G
\end{array}}{\ldots, K_1 \iff K_2 \vdash G}
\]

- **Goal** $G_1 \iff G_2$
  - Create two subsituations with implications in both directions as goals:
    - We have to show $G_1 \iff G_2$.
    - We show $G_1 \Rightarrow G_2$: \ldots (proof continues with goal $G_1 \Rightarrow G_2$)
    - We show $G_2 \Rightarrow G_1$: \ldots (proof continues with goal $G_2 \Rightarrow G_1$)

- **Knowledge** $K_1 \iff K_2$
  - Create two subsituations, one with goal $(\neg)K_1$ and one with knowledge $(\neg)K_2$.
    - We know $K_1 \iff K_2$.
    - We show $(\neg)K_1$: \ldots (proof continues with goal $(\neg)K_1$)
    - We know $(\neg)K_2$: \ldots (proof continues with current goal and additional knowledge $(\neg)K_2$)
Universal Quantification $\forall x : F$

\[
\frac{K \vdash G[x_0/x]}{K \vdash \forall x : G} \quad \text{(}x_0\text{ new for } K, G) \quad \frac{\ldots, \forall x : K, K[T/x] \vdash G}{\ldots, \forall x : K \vdash G}
\]

- **Goal $\forall x : G$**
  - Introduce new (arbitrarily named) constant $x_0$ and create one subsituation with goal $G[x_0/x]$.
  
    *We have to show $\forall x : G$. Take arbitrary $x_0$. We show $G[x_0/x]$. (proof continues with goal $G[x_0/x]$)*

- **Knowledge $\forall x : K$**
  - Choose term $T$ to create one subsituation with formula $K[T/x]$ added to the knowledge.
  
    *We know $\forall x : K$ and thus also $K[T/x]$. (proof continues with current goal and additional knowledge $K[T/x]$)*
Existential Quantification $\exists x : F$

$$\begin{array}{c}
K \vdash G[T/x] \\
\hline
K \vdash \exists x : G
\end{array} \quad \begin{array}{c}
\ldots, K[x_0/x] \vdash G \\
\hline
\ldots, \exists x : K \vdash G
\end{array} \quad (x_0 \text{ new for } K, G)$$

- **Goal $\exists x : G$**
  - Choose term $T$ to create one subsituation with goal $G[T/x]$.
    
    *We have to show $\exists x : G$. It suffices to show $G[T/x]$. (proof continues with goal $G[T/x]$)*

- **Knowledge $\exists x : K$**
  - Introduce new (arbitrarily named constant) $x_0$ and create one subsituation with additional knowledge $K[x_0/x]$.
    
    *We know $\exists x : K$. Let $x_0$ be such that $K[x_0/x]$. (proof continues with current goal and additional knowledge $K[x_0/x]$)*
Example

We show

(a) \((\exists x : \forall y : P(x, y)) \Rightarrow (\forall y : \exists x : P(x, y))\)

We assume

(1) \(\exists x : \forall y : P(x, y)\)

and show

(b) \(\forall y : \exists x : P(x, y)\)

Take arbitrary \(y_0\). We show

(c) \(\exists x : P(x, y_0)\)

From (1) we know for some \(x_0\)

(2) \(\forall y : P(x_0, y)\)

From (2) we know

(3) \(P(x_0, y_0)\)

From (3), we know (c). QED.
Example

We show

\[(a) \ (\exists x : p(x)) \land (\forall x : p(x) \Rightarrow \exists y : q(x, y)) \Rightarrow (\exists x, y : q(x, y))\]

We assume

\[(1) \ (\exists x : p(x)) \land (\forall x : p(x) \Rightarrow \exists y : q(x, y))\]

and show

\[(b) \ \exists x, y : q(x, y)\]

From (1), we know

\[(2) \ \exists x : p(x)\]
\[(3) \ \forall x : p(x) \Rightarrow \exists y : q(x, y)\]

From (2) we know for some \(x_0\)

\[(4) \ p(x_0)\]

\[\ldots\]
Example (Contd)

\[ p(x_0) \Rightarrow \exists y : q(x_0, y) \]

From (4) and (5), we know

\[ \exists y : q(x_0, y) \]

From (6), we know for some \( y_0 \)

\[ q(x_0, y_0) \]

From (7), we know (b). QED.
Indirect Proofs

\[
\begin{align*}
K, \neg G & \vdash \text{false} \\
\hline
K & \vdash G
\end{align*}
\]

\[
\begin{align*}
K, \neg G & \vdash F \\
K, \neg G & \vdash \neg F
\end{align*}
\]

\[
\begin{align*}
\ldots, \neg G & \vdash \neg K \\
\hline
\ldots, K & \vdash G
\end{align*}
\]

- Add \( \neg G \) to the knowledge and show a contradiction.
- Prove that “false” is true.
- Prove that a formula \( F \) is true and also prove that it is false.
- Prove that some knowledge \( K \) is false, i.e. that \( \neg K \) is true.
  - Switches goal \( G \) and knowledge \( K \) (negating both).

Sometimes simpler than a direct proof.
Example

We show

\[(a) \ (\exists x : \forall y : P(x, y)) \Rightarrow (\forall y : \exists x : P(x, y))\]

We assume

\[(1) \ \exists x : \forall y : P(x, y)\]

and show

\[(b) \ \forall y : \exists x : P(x, y)\]

We assume

\[(2) \ \neg \forall y : \exists x : P(x, y)\]

and show a contradiction.

\[\ldots\]
Example

... From (2), we know

(3) \( \exists y : \forall x : \neg P(x, y) \)

Let \( y_0 \) be such that

(4) \( \forall x : \neg P(x, y_0) \)

From (1) we know for some \( x_0 \)

(5) \( \forall y : P(x_0, y) \)

From (5) we know

(6) \( P(x_0, y_0) \)

From (4), we know

(7) \( \neg P(x_0, y_0) \)

From (6) and (7), we have a contradiction. QED.
1. The Language of Logic

2. The RISC Algorithm Language

3. The Art of Proving

4. The RISC ProofNavigator
The RISC ProofNavigator

- An interactive proving assistant for program verification.
  - Research Institute for Symbolic Computation (RISC), 2005–.
    http://www.risc.jku.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, CVC3, CVC4 1.4.
  - 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”.

Wolfgang Schreiner http://www.risc.jku.at
Starting the Software

Starting the software:
module load ProofNavigator  (users at RISC)
ProofNavigator &

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
% 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:

\[
\begin{align*}
\text{sum} &\in \mathbb{N} \rightarrow \mathbb{N} \\
\text{axiom } S1 &\equiv \text{sum}(0) = 0 \\
\text{axiom } S2 &\equiv \forall n \in \mathbb{N}: n > 0 \Rightarrow \text{sum}(n) = n + \text{sum}(n-1) \\
S &\equiv \forall n \in \mathbb{N}: \text{sum}(n) = \frac{(n+1) \cdot n}{2}
\end{align*}
\]

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

\[
\forall n \in \mathbb{N}; \quad n > 0 \implies \text{sum}(n) = n + \text{sum}(n - 1) \\
\text{sum}(0) = 0 \\
\forall n \in \mathbb{N}; \quad \text{sum}(n) = \frac{(n+1) \cdot n}{2}
\]
Proving a Formula

- Proof of formula \( F \) is represented as a tree.
  - Each tree node denotes a proof state (goal).
    - Logical sequent: \( A_1, A_2, \ldots \vdash B_1, B_2, \ldots \).
    - Interpretation: \( (A_1 \land A_2 \land \ldots) \Rightarrow (B_1 \lor B_2 \lor \ldots) \).
  - Initially single node \textit{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.

Constants: \( x_0 \in S_0, \ldots \)

\[
\begin{array}{c}
[\mathcal{L}_1] & A_1 \\
\vdots & \\
[\mathcal{L}_n] & A_n \\
[\mathcal{L}_{n+1}] & B_1 \\
\vdots & \\
[\mathcal{L}_{n+m}] & B_m
\end{array}
\]
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!
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.
Proving Strategies

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