Logic, Checking, and Proving

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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

1. The Language of Logic

2. The RISC Algorithm Language

3. The Art of Proving

4. The RISC Theorem Proving Interface

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 $F_1 \Leftrightarrow F_2$ ("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")

Syntactic Shortcuts

- $\forall x_1, \ldots, x_n : F$
  - $\forall x_1 : \ldots : \forall x_n : F$
- $\exists x_1, \ldots, x_n : F$
  - $\exists x_1 : \ldots : \exists x_n : F$
- $\forall x \in S : F$
  - $\forall x : x \in S \Rightarrow F$ (not: $\wedge$)
- $\exists x \in S : F$
  - $\exists x : x \in S \wedge F$ (not: $\Rightarrow$)

Help to make formulas more readable.

Examples

Terms and formulas may appear in various syntactic forms.

- Terms:

$$\exp(x)$$
$$a \cdot b + 1$$
$$a[i] \cdot b$$
$$\sqrt{\frac{x^2+2x+1}{(y+1)^2}}$$

- Formulas:

$$a^2 + b^2 = c^2$$
$$n \mid 2n$$
$$\forall x \in \mathbb{N} : x \geq 0$$
$$\forall x \in \mathbb{N} : 2|x \vee 2|(x+1)$$
$$\begin{aligned}&\forall x \in \mathbb{N}, y \in \mathbb{N} : x < y \Rightarrow \\ &\qquad \exists z \in \mathbb{N} : x + z = y\end{aligned}$$

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 $F_1 \wedge F_2$ (“F_1 and F_2”)
  - True if and only if F_1 and F_2 are both true.
- Disjunction $F_1 \vee F_2$ (“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.
- $1+1=2 \vee 2+2=2$
  - True.
- $1+1=2 \wedge 2+2=2$
  - 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.

- $\neg\neg F_1 \leftrightsquigarrow F_1$
- $\neg(F_1 \land F_2) \leftrightsquigarrow \neg F_1 \lor \neg F_2$
- $\neg(F_1 \lor F_2) \leftrightsquigarrow \neg F_1 \land \neg F_2$
- $\neg(F_1 \Rightarrow F_2) \leftrightsquigarrow F_1 \land \neg F_2$
- $\neg\forall x : F \leftrightsquigarrow \exists x : \neg F$
- $\neg\exists x : F \leftrightsquigarrow \forall x : \neg F$
- $F_1 \Rightarrow F_2 \leftrightsquigarrow \neg F_2 \Rightarrow \neg F_1$
- $F_1 \Rightarrow F_2 \leftrightsquigarrow \neg F_1 \lor F_2$
- $F_1 \Leftrightarrow F_2 \leftrightsquigarrow \neg F_1 \Leftrightarrow \neg F_2$
- ...

Familiarity with manipulation of formulas is important.

Example

- "All swans are white or black."
  - $\forall x : \mathit{swan}(x) \Rightarrow \mathit{white}(x) \vee \mathit{black}(x)$
- "There exists a black swan."
  - $\exists x : \mathit{swan}(x) \wedge \mathit{black}(x)$.
- "A swan is white, unless it is black."
  - $\forall x : \mathit{swan}(x) \wedge \neg \mathit{black}(x) \Rightarrow \mathit{white}(x)$
  - $\forall x : \mathit{swan}(x) \wedge \neg \mathit{white}(x) \Rightarrow \mathit{black}(x)$
  - $\forall x : \mathit{swan}(x) \Rightarrow \mathit{white}(x) \vee \mathit{black}(x)$
- "Not everything that is white or black is a swan."
  - $\neg \forall x : \mathit{white}(x) \vee \mathit{black}(x) \Rightarrow \mathit{swan}(x)$.
  - $\exists x : (\mathit{white}(x) \vee \mathit{black}(x)) \wedge \neg \mathit{swan}(x)$.
- "Black swans have at least one black parent".
  - $\forall x : \mathit{swan}(x) \wedge \mathit{black}(x) \Rightarrow \exists y : \mathit{swan}(y) \wedge \mathit{black}(y) \wedge \mathit{parent}(y, x)$

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:

  $$\begin{array}{l} x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge y \neq 0 \Rightarrow \\ \quad q \in \mathbb{N} \wedge \exists r \in \mathbb{N} : x = y \cdot q + r \wedge r < y \end{array}$$

- 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: $x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge y \neq 0$
  - Output: q
  - Output condition: $q \in \mathbb{N} \wedge \exists r \in \mathbb{N} : x = y \cdot q + r \wedge r < y$

Problem Specifications

- The specification of a computation problem:
  - Input: variables $x_1 \in S_1, \dots, x_n \in S_n$
  - Input condition ("precondition"): formula $I(x_1, \dots, x_n)$.
  - Output: variables $y_1 \in T_1, \dots, y_m \in T_n$
  - Output condition ("postcondition"): $O(x_1, \dots, x_n, y_1, \dots, y_m)$.
    - $F(x_1, \dots, x_n)$: only $x_1, \dots, x_n$ 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) $f : S_1 \times \ldots \times S_n \to T_1 \times \ldots \times T_m$ such that

$$\begin{array}{l} \forall x_1 \in S_1, \dots, x_n \in S_n : I(x_1, \dots, x_n) \Rightarrow \\ \qquad \textbf{let } (y_1, \dots, y_m) = f(x_1, \dots, x_n) \textbf{ in} \\ \qquad O(x_1, \dots, x_n, y_1, \dots, y_m) \end{array}$$

  - For all arguments that satisfy the input condition, f must compute results that satisfy the output condition.

Basis of all specification formalisms.

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:

  $$\begin{array}{l} \textbf{let } n = \text{length}(a) \textbf{ in} \\ \text{length}(b) = n - l \wedge \\ (\forall i \in \mathbb{N} : i < p \Rightarrow b[i] = a[i]) \wedge \\ (\forall i \in \mathbb{N} : p \leq i < n - l \Rightarrow b[i] = a[i+l]) \end{array}$$

Mathematical theory:

$$\begin{array}{l} T^* := \bigcup_{i \in \mathbb{N}} T^i, T^i := \mathbb{N}_i \to T, \mathbb{N}_i := \{n \in \mathbb{N} : n < i\} \\ \text{length} : T^* \to \mathbb{N}, \text{length}(a) = \textbf{such } i \in \mathbb{N} : a \in T^i \end{array}$$

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?
  - $\forall x : I(x) \Rightarrow \exists y : \neg O(x, y)$.
  - $\exists x : I(x) \wedge \exists y : \neg O(x, y)$.
- 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 Theorem Proving Interface

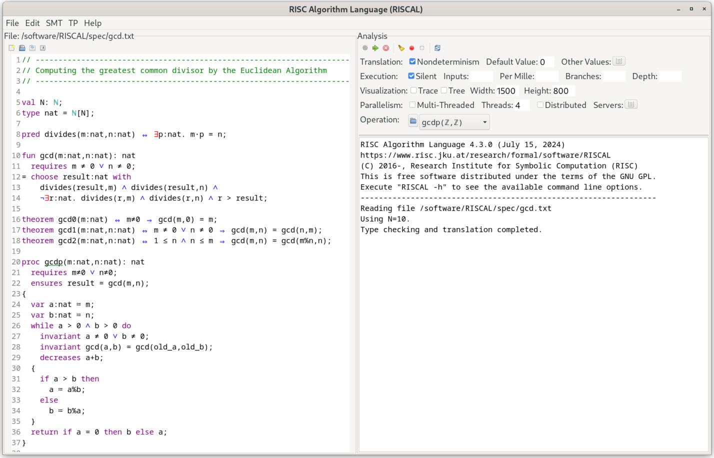
The RISC Algorithm Language (RISCAL)

- A system for formally specifying and checking algorithms.
  - Research Institute for Symbolic Computation (RISC), 2016–.
    https://www.risc.jku.at/research/formal/software/RISCAL.
  - Implemented in Java with the Eclipse 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 def.d 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 &
```



File: /software/RISCAL/spec/gcd.txt

```
// ----------------------------------------------------------------------
// Computing the greatest common divisor by the Euclidean Algorithm
// ----------------------------------------------------------------------

val N: ℕ;
type nat = ℕ[N];

pred divides(m:nat,n:nat) ⇔ ∃p:nat. m·p = n;

fun gcd(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
= choose result:nat with
    divides(result,m) ∧ divides(result,n) ∧
    ¬∃r:nat. divides(r,m) ∧ divides(r,n) ∧ r > result;

theorem gcd0(m:nat) ⇔ m≠0 ⇒ gcd(m,0) = m;
theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0 ⇒ gcd(m,n) = gcd(n,m);
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);

proc gcdp(m:nat,n:nat): nat
  requires m≠0 ∨ n≠0;
  ensures result = gcd(m,n);
{
  var a:nat = m;
  var b:nat = n;
  while a > 0 ∧ b > 0 do
    invariant a ≠ 0 ∨ b ≠ 0;
    invariant gcd(a,b) = gcd(old_a,old_b);
    decreases a+b;
  {
    if a > b then
      a = a%b;
    else
      b = b%a;
  }
  return if a = 0 then b else a;
}
```

Translation: ☑ Nondeterminism Default Value: 0 Other Values:

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

Visualization: □ Trace □ Tree Width: 1500 Height: 800

Parallelism: □ Multi-Threaded Threads: 4 □ Distributed Servers:

Operation: gcdp(ℤ,ℤ)

```
RISC Algorithm Language 4.3.0 (July 15, 2024)
https://www.risc.jku.at/research/formal/software/RISCAL
(C) 2016-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCAL -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /software/RISCAL/spec/gcd.txt
Using N=10.
Type checking and translation completed.
```

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 buttton ❌ to abort execution.

During evaluation all annotations (pre/postconditions, etc.) are checked.

Typing Mathematical Symbols

ASCII String	Unicode Character
Int	ℤ
Nat	ℕ
:=	:=
true	⊤
false	⊥
~	¬
/\	∧
\/	∨
=>	⇒
<=>	⇔
forall	∀
exists	∃
sum	∑
product	∏

ASCII String	Unicode Character
~=	≠
<=	≤
>=	≥
*	·
times	×
{}	∅
intersect	∩
union	∪
Intersect	⋂
Union	⋃
isin	∈
subseteq	⊆
<<	⟨
>>	⟩

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 N
val N: ℕ;
type Num = ℕ[N];

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

// the postcondition, first formulation
pred post1(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m·q + r ∧
  ∀q0:Num, r0:Num.
    n = m·q0 + r0 ⇒ r ≤ r0;

// the postcondition, second formulation
pred post2(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m·q + r ∧ r < m;
```

We will investigate this specification.

Example: Quotient and Remainder

```
// for all inputs that satisfy the precondition
// both formulations are equivalent:
// ∀n:Num, m:Num, q:Num, r:Num.
//   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 postEquiv(ℤ,ℤ,ℤ,ℤ) with all 1296 inputs.
Ignoring inadmissible inputs...
Run 6 of deterministic function postEquiv(0,1,0,0):
Result (0 ms): true
Run 7 of deterministic function postEquiv(1,1,0,0):
Result (0 ms): true
...
Run 1295 of deterministic function 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 postEquiv(ℤ,ℤ,ℤ,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

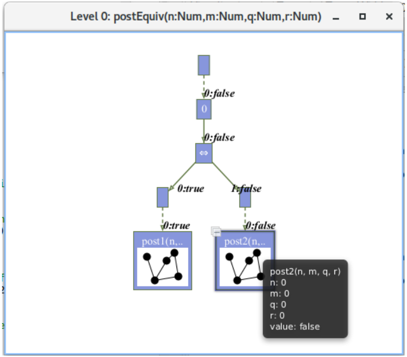
Drop precondition from theorem.

```
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.
```

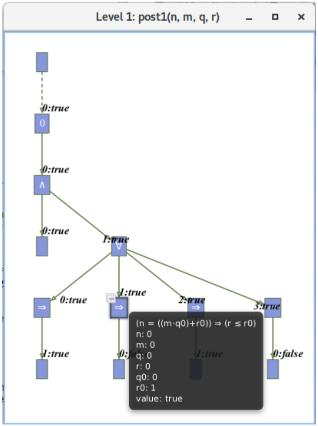
For $n = 0, m = 0, q = 0, r = 0$, the modified theorem is not true.

Visualizing the Formula Evaluation

Select $N = 1$ and visualization option "Tree".



```
post2(n, m, q, r)
n: 0
m: 0
q: 0
r: 0
value: false
```



```
(n = ((m·q0)+r0)) ⇒ (r ≤ r0)
n: 0
m: 0
q: 0
r: 0
q0: 0
r0: 1
value: true
```

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

```
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.

Example: Sorting an Array

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

proc sort(a:array): array
  ...
{
  var b:array = a;
  for var i:index := 1; i < N; i := i+1 do
  {
    var x:elem := b[i];
    var j:index := 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

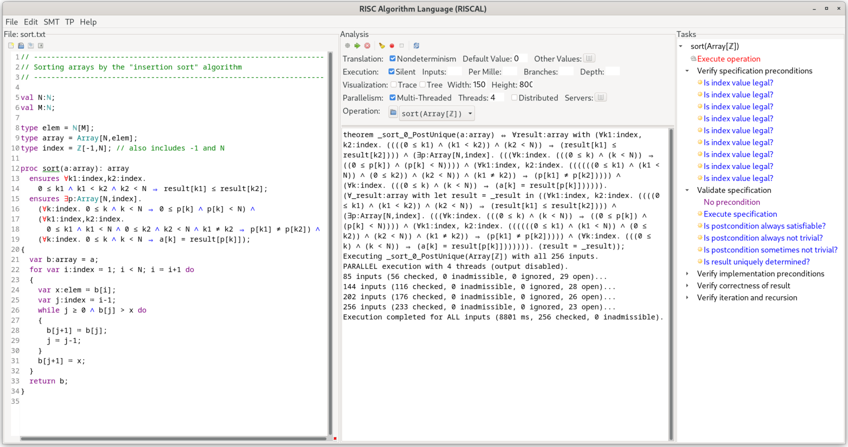
```
proc sort(a:array): array
  ensures ∀k1:index,k2:index.
    0 ≤ k1 ∧ k1 < k2 ∧ k2 < N ⇒ result[k1] ≤ result[k2];
  ensures ∃p:Array[N,index].
    (∀k:index. 0 ≤ k ∧ k < N ⇒ 0 ≤ p[k] ∧ p[k] < N) ∧
    (∀k1:index,k2:index.
      0 ≤ k1 ∧ k1 < N ∧ 0 ≤ k2 ∧ k2 < N ∧ k1 ≠ k2 ⇒ p[k1] ≠ p[k2]) ∧
    (∀k:index. 0 ≤ k ∧ k < N ⇒ a[k] = result[p[k]]);
```

```
Using N=4.
Using M=3.
Computing the value of _tbound_0...
Type checking and translation completed.
Executing sort(Array[ℤ]) with all 256 inputs.
Execution completed for ALL inputs (278 ms, 256 checked, 0 inadmissible).
```

Also this algorithm can be automatically checked.

Example: Sorting an Array

Select operation `sort` and press the button “Show/Hide Tasks”.



```
// ----------------------------------------------------------------
// Sorting arrays by the "insertion sort" algorithm
// ----------------------------------------------------------------

val N:ℕ;
val M:ℕ;

type elem = ℕ[M];
type array = Array[N,elem];
type index = ℤ[-1,N]; // also includes -1 and N

proc sort(a:array): array
  ensures ∀k1:index,k2:index.
    0 ≤ k1 ∧ k1 < k2 ∧ k2 < N ⇒ result[k1] ≤ result[k2];
  ensures ∃p:Array[N,index].
    (∀k:index. 0 ≤ k ∧ k < N → 0 ≤ p[k] ∧ p[k] < N) ∧
    (∀k1:index,k2:index.
      0 ≤ k1 ∧ k1 < N ∧ 0 ≤ k2 ∧ k2 < N ∧ k1 ≠ k2 ⇒ p[k1] ≠ p[k2]) ∧
    (∀k:index. 0 ≤ k ∧ k < N → a[k] = result[p[k]]);
{
  var b:array = a;
  for var i:index = 1; i < N; i = i+1 do
  {
    var x:elem = b[i];
    var j:index = i-1;
    while j ≥ 0 ∧ b[j] > x do
    {
      b[j+1] = b[j];
      j = j-1;
    }
    b[j+1] = x;
  }
  return b;
}
```

Translation: ☑ Nondeterminism Default Value: 0 Other Values:

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

Visualization: □ Trace □ Tree Width: 150 Height: 800

Parallelism: ☑ Multi-Threaded Threads: 4 □ Distributed Servers:

Operation: sort(Array[ℤ])

```
theorem _sort_0_PostUnique(a:array) ⇔ ∀result:array with (∀k1:index,
k2:index. ((((0 ≤ k1) ∧ (k1 < k2)) ∧ (k2 < N)) → (result[k1] ≤
result[k2]))) ∧ (∃p:Array[N,index]. (((∀k:index. (((0 ≤ k) ∧ (k < N)) ⇒
((0 ≤ p[k]) ∧ (p[k] < N)))) ∧ (∀k1:index, k2:index. ((((((0 ≤ k1) ∧ (k1 <
N)) ∧ (0 ≤ k2)) ∧ (k2 < N)) ∧ (k1 ≠ k2)) ⇒ (p[k1] ≠ p[k2])))) ∧
(∀k:index. (((0 ≤ k) ∧ (k < N)) → (a[k] = result[p[k]]))))).
(∀_result:array with let result = _result in ((∀k1:index, k2:index. ((((0
≤ k1) ∧ (k1 < k2)) ∧ (k2 < N)) ⇒ (result[k1] ≤ result[k2]))) ∧
(∃p:Array[N,index]. (((∀k:index. (((0 ≤ k) ∧ (k < N)) ⇒ ((0 ≤ p[k]) ∧
(p[k] < N)))) ∧ (∀k1:index, k2:index. ((((((0 ≤ k1) ∧ (k1 < N)) ∧ (0 ≤
k2)) ∧ (k2 < N)) ∧ (k1 ≠ k2)) ⇒ (p[k1] ≠ p[k2])))) ∧ (∀k:index. (((0 ≤
k) ∧ (k < N)) ⇒ (a[k] = result[p[k]]))))))). (result = _result));
Executing _sort_0_PostUnique(Array[ℤ]) with all 256 inputs.
PARALLEL execution with 4 threads (output disabled).
85 inputs (56 checked, 0 inadmissible, 0 ignored, 29 open)...
144 inputs (116 checked, 0 inadmissible, 0 ignored, 28 open)...
202 inputs (176 checked, 0 inadmissible, 0 ignored, 26 open)...
256 inputs (233 checked, 0 inadmissible, 0 ignored, 23 open)...
Execution completed for ALL inputs (8801 ms, 256 checked, 0 inadmissible).
```

- sort(Array[ℤ])
  - Execute operation
  - Verify specification preconditions
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
  - Validate specification
    - No precondition
    - Execute specification
    - Is postcondition always satisfiable?
    - Is postcondition always not trivial?
    - Is postcondition sometimes not trivial?
    - Is result uniquely determined?
  - Verify implementation preconditions
  - Verify correctness of result
  - Verify iteration and recursion

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 (∀k1:index, k2:index.
    ((((0 ≤ k1) ∧ (k1 < k2)) ∧ (k2 < N)) ⇒ (result[k1] ≤ result[k2]))) ∧
    (∃p:Array[N,index].
      (((∀k:index. (((0 ≤ k) ∧ (k < N)) ⇒ ((0 ≤ p[k]) ∧ (p[k] < N)))) ∧
      (∀k1:index, k2:index. ((((((0 ≤ k1) ∧ (k1 < N)) ∧ (0 ≤ k2)) ∧ (k2 < N)) ∧ (k1 ≠ k2)) ⇒
        (p[k1] ≠ p[k2])))) ∧
      (∀k:index. (((0 ≤ k) ∧ (k < N)) ⇒ (a[k] = result[p[k]]))))).
    (∀_result:array with let result = _result in ((∀k1:index, k2:index.
      ((((0 ≤ k1) ∧ (k1 < k2)) ∧ (k2 < N)) ⇒ (result[k1] ≤ result[k2]))) ∧
      (∃p:Array[N,index].
        (((∀k:index. (((0 ≤ k) ∧ (k < N)) ⇒ ((0 ≤ p[k]) ∧ (p[k] < N)))) ∧
        (∀k1:index, k2:index. ((((((0 ≤ k1) ∧ (k1 < N)) ∧ (0 ≤ k2)) ∧ (k2 < N)) ∧ (k1 ≠ k2)) ⇒
          (p[k1] ≠ p[k2])))) ∧
        (∀k:index. (((0 ≤ k) ∧ (k < N)) ⇒ (a[k] = result[p[k]])))))).
          (result = _result));

Executing _sort_0_PostUnique(Array[ℤ]) with all 256 inputs.
PARALLEL execution with 4 threads (output disabled).
85 inputs (56 checked, 0 inadmissible, 0 ignored, 29 open)...
144 inputs (116 checked, 0 inadmissible, 0 ignored, 28 open)...
202 inputs (176 checked, 0 inadmissible, 0 ignored, 26 open)...
256 inputs (233 checked, 0 inadmissible, 0 ignored, 23 open)...
Execution completed for ALL inputs (8801 ms, 256 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 excutions 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).

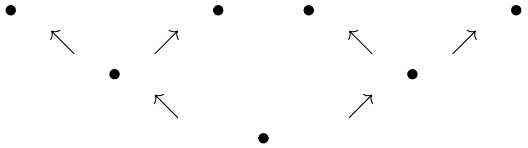
General verification requires the proving of logic formulas.

1. The Language of Logic
2. The RISC Algorithm Language
3. **The Art of Proving**
4. The RISC Theorem Proving Interface

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.

Proof Rules

A proof rules describes how a proof situation can be reduced to zero, one, or more “subsituations”.

$$\frac{\ldots \vdash \ldots \qquad\qquad \ldots \vdash \ldots}{K_1, \ldots, K_n \vdash G}$$

- 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 $F_1 \wedge F_2$

$$\frac{K \vdash G_1 \qquad K \vdash G_2}{K \vdash G_1 \wedge G_2} \qquad\qquad \frac{\ldots, K_1 \wedge K_2, K_1, K_2 \vdash G}{\ldots, K_1 \wedge K_2 \vdash G}$$

- Goal $G_1 \wedge G_2$.
  - Create two subsituations with goals G_1 and G_2.

    We have to show $G_1 \wedge G_2$.
    - *We show G_1: ... (proof continues with goal G_1)*
    - *We show G_2: ... (proof continues with goal G_2)*
- Knowledge $K_1 \wedge K_2$.
  - Create one subsituation with K_1 and K_2 in knowledge.

    We know $K_1 \wedge K_2$. We thus also know K_1 and K_2.
    (proof continues with current goal and additional knowledge K_1 and K_2)

Disjunction $F_1 \lor F_2$

$$\frac{K, \neg G_1 \vdash G_2}{K \vdash G_1 \lor G_2} \qquad \frac{\ldots, K_1 \vdash G \qquad \ldots, K_2 \vdash G}{\ldots, K_1 \lor K_2 \vdash G}$$

- 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: ... (proof continues with current goal and additional knowledge K_1).*
    - *Case K_2: ... (proof continues with current goal and additional knowledge K_2).*

Implication $F_1 \Rightarrow F_2$

$$\frac{K, G_1 \vdash G_2}{K \vdash G_1 \Rightarrow G_2} \qquad \frac{\ldots \vdash K_1 \qquad \ldots, K_2 \vdash G}{\ldots, K_1 \Rightarrow K_2 \vdash G}$$

- Goal $G_1 \Rightarrow G_2$
  - Create one subsituation where G_2 is proved under the assumption that G_1 holds:

    We have to show $G_1 \Rightarrow G_2$. We assume G_1 and show G_2. (proof continues with goal G_2 and additional knowledge G_1)
- Knowledge $K_1 \Rightarrow K_2$
  - Create two subsituations, one with goal K_1 and one with knowledge K_2.

    We know $K_1 \Rightarrow K_2$.
    - *We show K_1: …(proof continues with goal K_1)*
    - *We know K_2: …(proof continues with current goal and additional knowledge K_2).*

Equivalence $F_1 \Leftrightarrow F_2$

$$\frac{K \vdash G_1 \Rightarrow G_2 \qquad K \vdash G_2 \Rightarrow G_1}{K \vdash G_1 \Leftrightarrow G_2} \qquad \frac{\ldots \vdash (\neg)K_1 \qquad \ldots, (\neg)K_2 \vdash G}{\ldots, K_1 \Leftrightarrow K_2 \vdash G}$$

- Goal $G_1 \Leftrightarrow G_2$
  - Create two subsituations with implications in both directions as goals:

    We have to show $G_1 \Leftrightarrow G_2$.

    - *We show $G_1 \Rightarrow G_2$: ... (proof continues with goal $G_1 \Rightarrow G_2$)*
    - *We show $G_2 \Rightarrow G_1$: ... (proof continues with goal $G_2 \Rightarrow G_1$)*
- Knowledge $K_1 \Leftrightarrow K_2$
  - Create two subsituations, one with goal $(\neg)K_1$ and one with knowledge $(\neg)K_2$.

    We know $K_1 \Leftrightarrow K_2$.

    - *We show $(\neg)K_1$: ... (proof continues with goal $(\neg)K_1$)*
    - *We know $(\neg)K_2$: ... (proof continues with current goal and additional knowledge $(\neg)K_2$)*

Universal Quantification $\forall x : F$

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

- 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 $\exists x : F$

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

- 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) $(\exists x : p(x)) \wedge (\forall x : p(x) \Rightarrow \exists y : q(x,y)) \Rightarrow (\exists x, y : q(x,y))$

We assume

(1) $(\exists x : p(x)) \wedge (\forall x : p(x) \Rightarrow \exists y : q(x,y))$

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) $p(x_0)$

...

Example (Contd)

...

From (3), we know

$$(5)\ p(x_0) \Rightarrow \exists y : q(x_0, y)$$

From (4) and (5), we know

$$(6)\ \exists y : q(x_0, y)$$

From (6), we know for some y_0

$$(7)\ q(x_0, y_0)$$

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

1. The Language of Logic

2. The RISC Algorithm Language

3. The Art of Proving

4. **The RISC Theorem Proving Interface**

The RISC Theorem Proving Interface

- RISCTP: an interface to various theorem proving methods.
  - Research Institute for Symbolic Computation (RISC), 2022–.
    https://www.risc.jku.at/research/formal/software/RISCTP
- Proof Method SMT:
  - Translation to a proof problem in the SMT-LIB language.
  - Application of external provers/SMT solvers Z3, cvc5, Vampire.
  - Fast and effective for problems of moderate complexity.
  - Black box: no human-readable/understandable proofs.
- Proof Method MESON:
  - First proof decomposition/simplification by logical/arithmetical rules.
  - Then application of “Model Elimination, Subgoal-Oriented”.
  - (Optional) support by external SMT solvers for larger efficiency.
  - Transparent: human-readable/understandable proofs.

Developed to provide RISCAL with theorem proving capabilities.

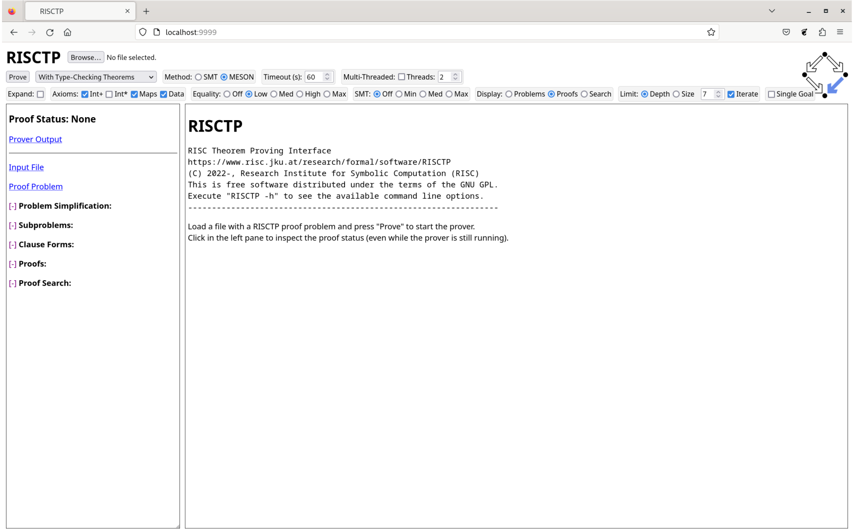
RISCTP as a Standalone Prover

```
> RISCTP -solver z3 -path /software/RISCTP/etc/z3 -web 9999 1
RISC Theorem Proving Interface 1.8.0 (July 15, 2024)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
RISCTP GUI can be browsed at http://localhost:9999/
Press <Enter> to terminate the server.
```

- `-solver z3`: use SMT solver Z3 (default).
- `-path /software/...`: path to executable of SMT solver.
- `-web 9999 1`: show (full) GUI at `http://localhost:9999/`

The RISCTP GUI can be accessed by any web browser.

The RISCTP GUI



RISCTP Browse... No file selected.

Prove | With Type-Checking Theorems | Method: ○ SMT ● MESON | Timeout (s): 60 | Multi-Threaded: □ Threads: 2

Expand: □ | Axioms: ☑ Int+ □ Int* ☑ Maps ☑ Data | Equality: ○ Off ● Low ○ Med ○ High ○ Max | SMT: ● Off ○ Min ○ Med ○ Max | Display: ○ Problems ● Proofs ○ Search | Limit: ● Depth ○ Size 7 ☑ Iterate | □ Single Goal

Proof Status: None

Prover Output

Input File

Proof Problem

[-] **Problem Simplification:**

[-] **Subproblems:**

[-] **Clause Forms:**

[-] **Proofs:**

[-] **Proof Search:**

RISCTP

```
RISC Theorem Proving Interface
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
```

Load a file with a RISCTP proof problem and press "Prove" to start the prover.
Click in the left pane to inspect the proof status (even while the prover is still running).

Proof Method SMT

```
// problem file "fol2a.txt"
type T;
pred p(x:T,y:T);
theorem T ⇔ (∃x:T.∀y:T.p(x,y)) ⇒ (∀x:T.∃y:T.p(y,x));
```

- Button “Browse” `fol2a.txt`.
- Option “Method: SMT”, button “Prove” ⇝ “Proof Status: Success”.
- Link “Prover Output”.

```
=== SMT solving
SMT solver: Z3 version 4.13.0 - 64 bit
Proving theorem T...
SUCCESS: theorem was proved (11 ms).
=== SMT-LIB solver session
(set-logic ALL)
(set-option :produce-unsat-cores true)
(declare-sort T 0)
(declare-fun p ( T  T ) Bool)
(push 1)
(assert (not (=> (exists ((x T)) (forall ((y T)) (p x y))) (forall ((x T)) (exists ((y T)) (p y x))))))
(check-sat)
(pop 1)
(exit)
===
SUCCESS termination (26 ms).
```

SUCCESS: theorem was proved (however, claim is not substantiated).

Proof Method SMT

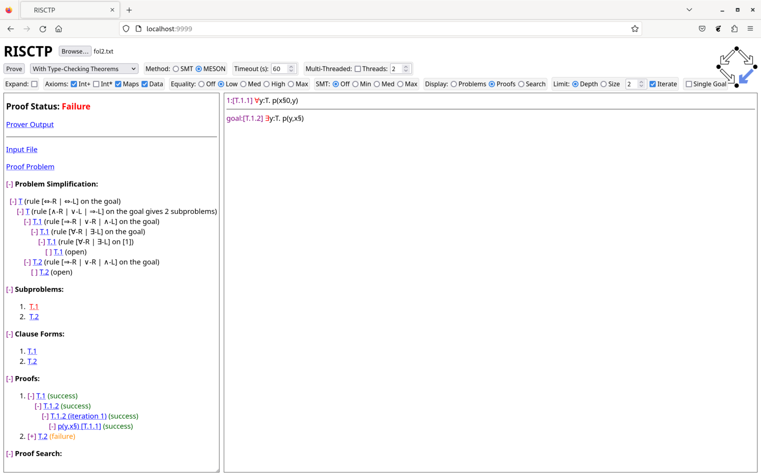
```
// problem file "fol2.txt"
type T;
pred p(x:T,y:T);
// actually, implication only holds from left to right
theorem T ⇔ (∃x:T.∀y:T.p(x,y)) ⇔ (∀x:T.∃y:T.p(y,x));
```

- Button "Browse" `fol2.txt` ⇝ "Proof Status: Failure".

```
=== SMT solving
SMT solver: Z3 version 4.13.0 - 64 bit
Proving theorem T...
FAILURE: theorem was not proved (13 ms).
theorem T ⇔ (∃x:T. (∀y:T. p(x,y))) ⇔ (∀x:T. (∃y:T. p(y,x)));
sat
=== SMT-LIB solver session
(set-logic ALL)
(set-option :produce-unsat-cores true)
(declare-sort T 0)
(declare-fun p ( T  T ) Bool)
(push 1)
(assert (not (= (exists ((x T)) (forall ((y T)) (p x y))) (forall ((x T)) (exists ((y T)) (p y x))))))
(check-sat)
(pop 1)
(exit)
===
FAILURE termination (31 ms).
```

FAILURE: theorem was not proved (however, no indication why this is so).

Proof Method MESON



RISCTP Browse... fol2.txt

Prove | With Type-Checking Theorems | Method: ○ SMT ● MESON | Timeout (s): 60 | Multi-Threaded: □ Threads: 2

Expand: □ | Axioms: ☑ Int+ □ Int* ☑ Maps ☑ Data | Equality: ○ Off ● Low ○ Med ○ High ○ Max | SMT: ● Off ○ Min ○ Med ○ Max | Display: ○ Problems ● Proofs ○ Search | Limit: ● Depth ○ Size 2 ☑ Iterate | □ Single Goal

Proof Status: Failure

Prover Output

Input File

Proof Problem

[-] **Problem Simplification:**

```
[-] T (rule [⇔-R | ⇔-L] on the goal)
  [-] T (rule [∧-R | ∨-L | ⇒-L] on the goal gives 2 subproblems)
    [-] T.1 (rule [⇒-R | ∨-R | ∧-L] on the goal)
      [-] T.1 (rule [∀-R | ∃-L] on the goal)
        [-] T.1 (rule [∀-R | ∃-L] on [1])
          [ ] T.1 (open)
    [-] T.2 (rule [⇒-R | ∨-R | ∧-L] on the goal)
      [ ] T.2 (open)
```

[-] **Subproblems:**

1. T.1
2. T.2

[-] **Clause Forms:**

1. T.1
2. T.2

[-] **Proofs:**

```
1. [-] T.1 (success)
     [-] T.1.2 (success)
       [-] T.1.2 (iteration 1) (success)
         [-] p(y,x§) [T.1.1] (success)
2. [+] T.2 (failure)
```

[-] **Proof Search:**

1:[T.1.1] ∀y:T. p(x§0,y)

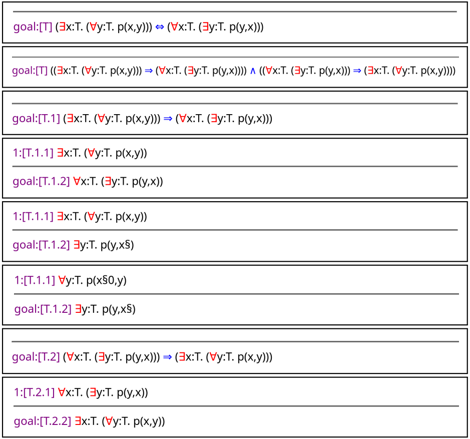
goal:[T.1.2] ∃y:T. p(y,x§)

Problem simplification yields two subproblems of which one can be proved.

Problem Simplification

[-] **Problem Simplification:**

```
[-] T (rule [⇔-R | ⇔-L] on the goal)
    [-] T (rule [∧-R | ∨-L | ⇒-L] on the goal gives 2 subproblems)
        [-] T.1 (rule [⇒-R | ∨-R | ∧-L] on the goal)
            [-] T.1 (rule [∀-R | ∃-L] on the goal)
                [-] T.1 (rule [∀-R | ∃-L] on [1])
                    [ ] T.1 (open)
        [-] T.2 (rule [⇒-R | ∨-R | ∧-L] on the goal)
            [ ] T.2 (open)
```



```
goal:[T] (∃x:T. (∀y:T. p(x,y))) ⇔ (∀x:T. (∃y:T. p(y,x)))

goal:[T] ((∃x:T. (∀y:T. p(x,y))) ⇒ (∀x:T. (∃y:T. p(y,x)))) ∧ ((∀x:T. (∃y:T. p(y,x))) ⇒ (∃x:T. (∀y:T. p(x,y))))

goal:[T.1] (∃x:T. (∀y:T. p(x,y))) ⇒ (∀x:T. (∃y:T. p(y,x)))

1:[T.1.1] ∃x:T. (∀y:T. p(x,y))
goal:[T.1.2] ∀x:T. (∃y:T. p(y,x))

1:[T.1.1] ∃x:T. (∀y:T. p(x,y))
goal:[T.1.2] ∃y:T. p(y,x§)

1:[T.1.1] ∀y:T. p(x§0,y)
goal:[T.1.2] ∃y:T. p(y,x§)

goal:[T.2] (∀x:T. (∃y:T. p(y,x))) ⇒ (∃x:T. (∀y:T. p(x,y)))

1:[T.2.1] ∀x:T. (∃y:T. p(y,x))
goal:[T.2.2] ∃x:T. (∀y:T. p(x,y))
```

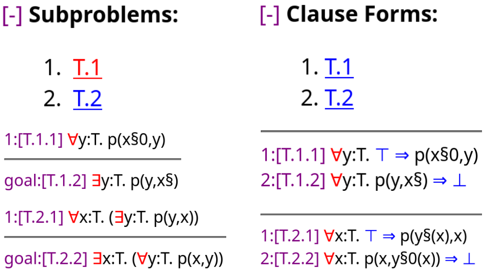
A step-by-step decomposition of the problem into simpler subproblems; each consists of “knowledge” formulas and a “goal” formula.

Clause Transformation

Each formula in a proof (sub)problem is transformed into a set of clauses.

- Clause $\forall x, \ldots. (A_1 \wedge \ldots \wedge A_a) \Rightarrow (B_1 \vee \ldots \vee B_b)$.
  - Closed formula with universally quantified variables $x, \ldots$.
  - The quantifier prefix $\forall x, \ldots$ is usually dropped.
  - Existential variables are replaced by Skolem constants/functions.
- Positive literals (atomic formulas) A_i and B_j.
  - Clause be written as disjunction $(\neg A_1 \vee \ldots \vee \neg A_a \vee B_1 \vee \ldots \vee B_b)$.
    - Negative literals $\neg A_i$, positive literals B_j.
  - Clause is true if some A_i is false or some B_j is true.
    - For some values of the quantified variables.
- Proof problem $K_1, \ldots, K_n \vdash G$:
  - Have to prove validity ("truth") of $(K_1 \wedge \ldots \wedge K_n \Rightarrow G)$.
  - Suffices to prove unsatisfiability ("falseness") of $(K_1 \wedge \ldots \wedge K_n \wedge \neg G)$.
  - Suffices to transform each K_i and $\neg G$ into clauses $\{C_1, \ldots, C_c\}$ and to prove the unsatisfiability of their conjunction $(C_1 \wedge \ldots \wedge C_c)$.
    - Suffices to prove the validity of $(C_1 \wedge \ldots \wedge C_{c-1}) \Rightarrow \neg C_c$.

Clause Transformation



[-] **Subproblems:**

1. T.1
2. T.2

1:[T.1.1] ∀y:T. p(x§0,y)

goal:[T.1.2] ∃y:T. p(y,x§)

1:[T.2.1] ∀x:T. (∃y:T. p(y,x))

goal:[T.2.2] ∃x:T. (∀y:T. p(x,y))

[-] **Clause Forms:**

1. T.1
2. T.2

1:[T.1.1] ∀y:T. ⊤ ⇒ p(x§0,y)
2:[T.1.2] ∀y:T. p(y,x§) ⇒ ⊥

1:[T.2.1] ∀x:T. ⊤ ⇒ p(y§(x),x)
2:[T.2.2] ∀x:T. p(x,y§0(x)) ⇒ ⊥

- Subproblems:
  - Above line: knowledge formulas.
  - Below line: goal.
- Clause Forms:
  - Above line: clauses from theory axioms (here none).
  - Below line: clauses from theorem (knowledge and negation of goal).

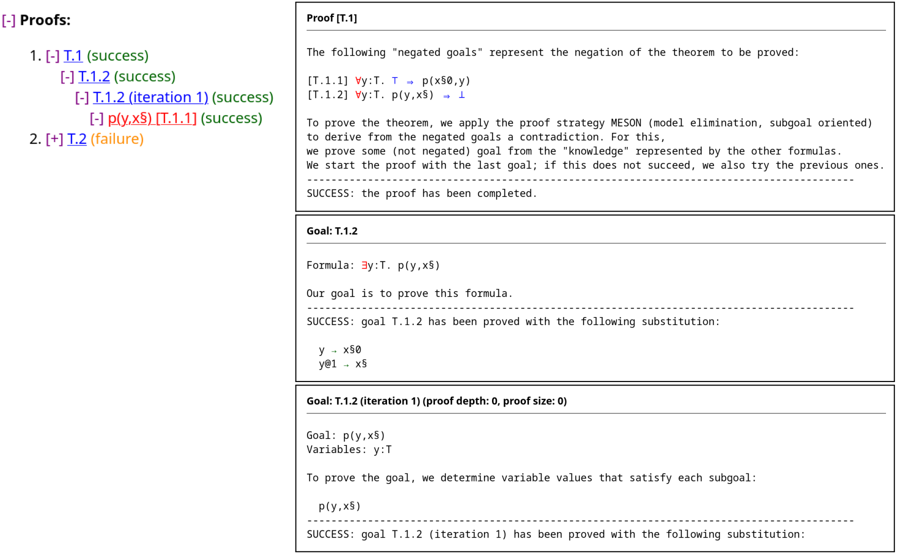
It suffices to prove the negation of the last clause from the other clauses.

Proof Method MESON

- MESON: Model Elimination, Subgoal-Oriented (Loveland, 1968).
  - A (Prolog-like) “backchaining strategy” for proving.
- Current goal: literal G (initially from the goal clause).
  - Current variable substitution σ.
- Pick clause $(L_1 \lor \ldots \lor L_i \lor \ldots \lor L_n)$.
  - Goal $G\sigma$ can be unified with $L_i\sigma$ by new substitution σ'.
  - Equivalent to $(\neg L_1 \land \ldots \land \neg L_{i-1} \land \neg L_{i+1} \land \ldots \land \neg L_n \Rightarrow L_i)$.
- New goals: $(\neg L_1, \ldots, \neg L_{i-1}, \neg L_{i+1}, \ldots, \neg L_l)$.
  - New variable substitution $\sigma\sigma'$.
  - Goal G is replaced by negations of clause literals other than L_i.
- Assumptions: literals $A_1, \ldots, A_a$.
  - G may be also proved from the current set of assumptions.
  - If not, we add $\neg G$ to the assumptions for the proof of the new goal.

During the proof search, the method must attempt every literal in every clause that can be unified with the goal literal; furthermore, the proof search must start from every clause arising from the theorem.

Proof Method MESON



[-] **Proofs:**

1. [-] T.1 (success)
   - [-] T.1.2 (success)
     - [-] T.1.2 (iteration 1) (success)
       - [-] p(y,x§) [T.1.1] (success)
2. [+] T.2 (failure)

Proof [T.1]

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

[T.1.1] ∀y:T. ⊤ ⇒ p(x§0,y)
[T.1.2] ∀y:T. p(y,x§) ⇒ ⊥

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

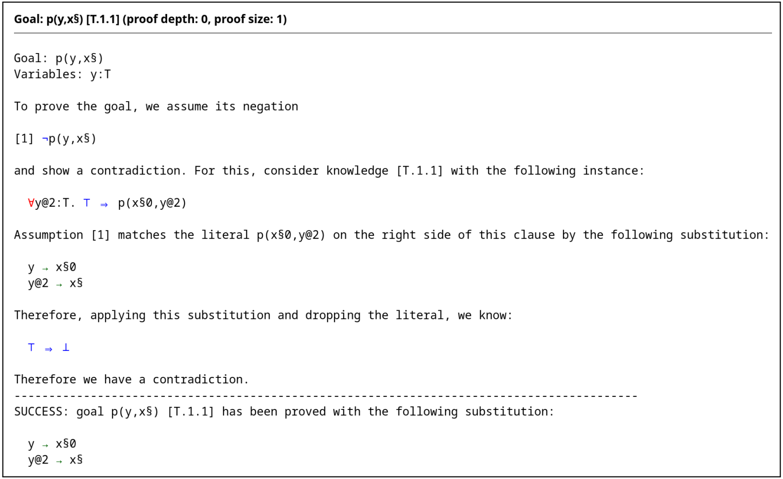
Goal: T.1.2

```
Formula: ∃y:T. p(y,x§)

Our goal is to prove this formula.
-----------------------------------------------------------------------------------------
SUCCESS: goal T.1.2 has been proved with the following substitution:

  y → x§0
  y@1 → x§
```

Goal: T.1.2 (iteration 1) (proof depth: 0, proof size: 0)

```
Goal: p(y,x§)
Variables: y:T

To prove the goal, we determine variable values that satisfy each subgoal:

  p(y,x§)
-----------------------------------------------------------------------------------------
SUCCESS: goal T.1.2 (iteration 1) has been proved with the following substitution:
```

Proof Method MESON



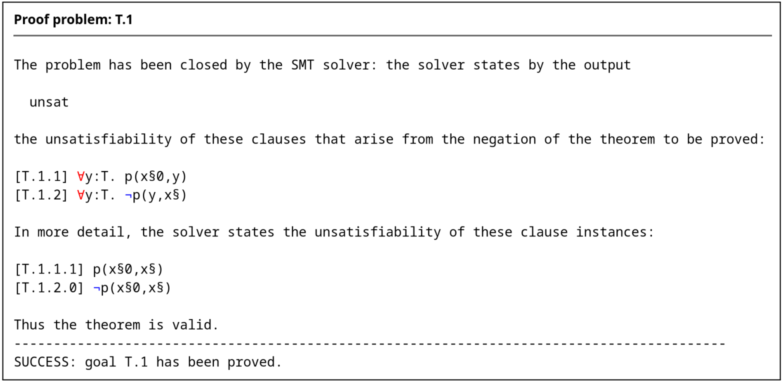
Goal: p(y,x§) [T.1.1] (proof depth: 0, proof size: 1)

```
Goal: p(y,x§)
Variables: y:T

To prove the goal, we assume its negation

[1] ¬p(y,x§)

and show a contradiction. For this, consider knowledge [T.1.1] with the following instance:

  ∀y@2:T. ⊤ ⇒ p(x§0,y@2)

Assumption [1] matches the literal p(x§0,y@2) on the right side of this clause by the following substitution:

  y → x§0
  y@2 → x§

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

  ⊤ ⇒ ⊥

Therefore we have a contradiction.
------------------------------------------------------------------------------------------
SUCCESS: goal p(y,x§) [T.1.1] has been proved with the following substitution:

  y → x§0
  y@2 → x§
```

The problem is closed by substituting in the first clause variable y with constant $x\S 0$ and in the second clause variable y with constant $x\S$.

Proof Method MESON with “SMT: Max”

We attempt the proof with the help of the external SMT solver first.

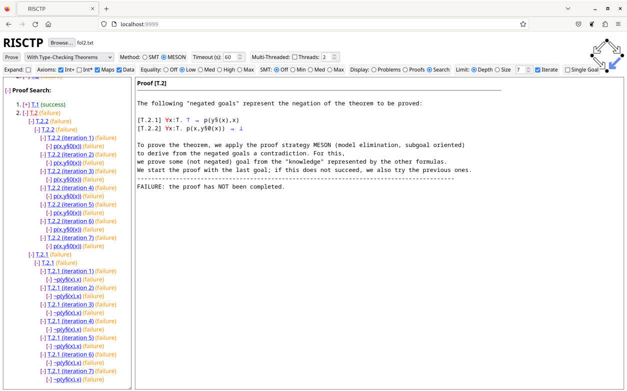


Proof problem: T.1

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

  unsat

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

[T.1.1] ∀y:T. p(x§0,y)
[T.1.2] ∀y:T. ¬p(y,x§)

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

[T.1.1.1] p(x§0,x§)
[T.1.2.0] ¬p(x§0,x§)

Thus the theorem is valid.
-------------------------------------------------------------------------------------------
SUCCESS: goal T.1 has been proved.
```

First we determine the clauses needed to close the proof problem, then we determine the actual instances of the clauses needed.

Proof Method MESON: Failed Proofs



RISCTP Browse... fol2.txt

Prove | With Type-Checking Theorems | Method: ○ SMT ● MESON | Timeout (s): 60 | Multi-Threaded: □ Threads: 2

Expand: □ | Axioms: ☑ Int+ □ Int* ☑ Maps ☑ Data | Equality: ○ Off ● Low ○ Med ○ High ○ Max | SMT: ● Off ○ Min ○ Med ○ Max | Display: ○ Problems ○ Proofs ● Search | Limit: ● Depth ○ Size 7 ☑ Iterate | □ Single Goal

```
[-] Proof Search:

    1. [+] T.1 (success)
    2. [-] T.2 (failure)
       [-] T.2.2 (failure)
         [-] T.2.2 (failure)
           [-] T.2.2 (iteration 1) (failure)
             [-] p(x,y§0(x)) (failure)
           [-] T.2.2 (iteration 2) (failure)
             [-] p(x,y§0(x)) (failure)
           [-] T.2.2 (iteration 3) (failure)
             [-] p(x,y§0(x)) (failure)
           [-] T.2.2 (iteration 4) (failure)
             [-] p(x,y§0(x)) (failure)
           [-] T.2.2 (iteration 5) (failure)
             [-] p(x,y§0(x)) (failure)
           [-] T.2.2 (iteration 6) (failure)
             [-] p(x,y§0(x)) (failure)
           [-] T.2.2 (iteration 7) (failure)
             [-] p(x,y§0(x)) (failure)
       [-] T.2.1 (failure)
         [-] T.2.1 (failure)
           [-] T.2.1 (iteration 1) (failure)
             [-] ¬p(y§(x),x) (failure)
           [-] T.2.1 (iteration 2) (failure)
             [-] ¬p(y§(x),x) (failure)
           [-] T.2.1 (iteration 3) (failure)
             [-] ¬p(y§(x),x) (failure)
           [-] T.2.1 (iteration 4) (failure)
             [-] ¬p(y§(x),x) (failure)
           [-] T.2.1 (iteration 5) (failure)
             [-] ¬p(y§(x),x) (failure)
           [-] T.2.1 (iteration 6) (failure)
             [-] ¬p(y§(x),x) (failure)
           [-] T.2.1 (iteration 7) (failure)
             [-] ¬p(y§(x),x) (failure)
```

Proof [T.2]

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

[T.2.1] ∀x:T. ⊤ → p(y§(x),x)
[T.2.2] ∀x:T. p(x,y§0(x)) ⇒ ⊥

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

- Limit: Depth *D* Iterate: iteratively search for a proof up to depth *D*.
- Display: Search: generate a proof tree also for a failed search.

We may also investigate failed proof attempts.

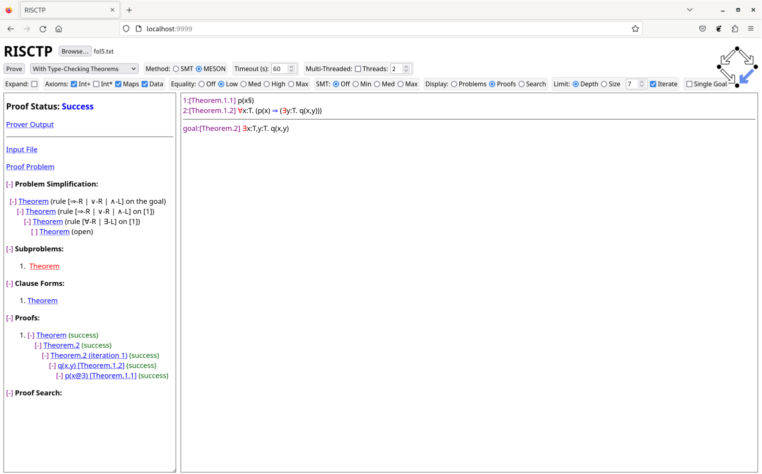
Another Proof Problem

```
// problem file "fol5.txt"
type T;
pred p(x:T);
pred q(x:T,y:T);
theorem Theorem ⇔
  (∃x:T. p(x)) ∧ (∀x:T. p(x) ⇒ ∃y:T. q(x,y)) ⇒ (∃x:T,y:T. q(x,y));
```

```
=== SMT solving
SMT solver: Z3 version 4.13.0 - 64 bit
Proving theorem Theorem...
SUCCESS: theorem was proved (9 ms).
=== SMT-LIB solver session
(set-logic ALL)
(set-option :produce-unsat-cores true)
(declare-sort T 0)
(declare-fun p ( T ) Bool)
(declare-fun q ( T  T ) Bool)
(push 1)
(assert (not (=> (and (exists ((x T)) (p x)) (forall ((x T))
  (=> (p x) (exists ((y T)) (q x y))))) (exists ((x T)) (exists ((y T)) (q x y))))))
(check-sat)
(pop 1)
(exit)
===
SUCCESS termination (15 ms).
```

Proof succeeds with Method SMT.

Another Proof Problem (Continued)



RISCTP | localhost:9999

RISCTP Browse… fol5.txt

Prove | With Type-Checking Theorems | Method: ○ SMT ● MESON | Timeout (s): 60 | Multi-Threaded: □ Threads: 2

Expand: □ | Axioms: ☑ Int+ □ Int* ☑ Maps ☑ Data | Equality: ○ Off ● Low ○ Med ○ High ○ Max | SMT: ● Off ○ Min ○ Med ○ Max | Display: ○ Problems ● Proofs ○ Search | Limit: ● Depth ○ Size 7 ☑ Iterate | □ Single Goal

Proof Status: Success

Prover Output

Input File

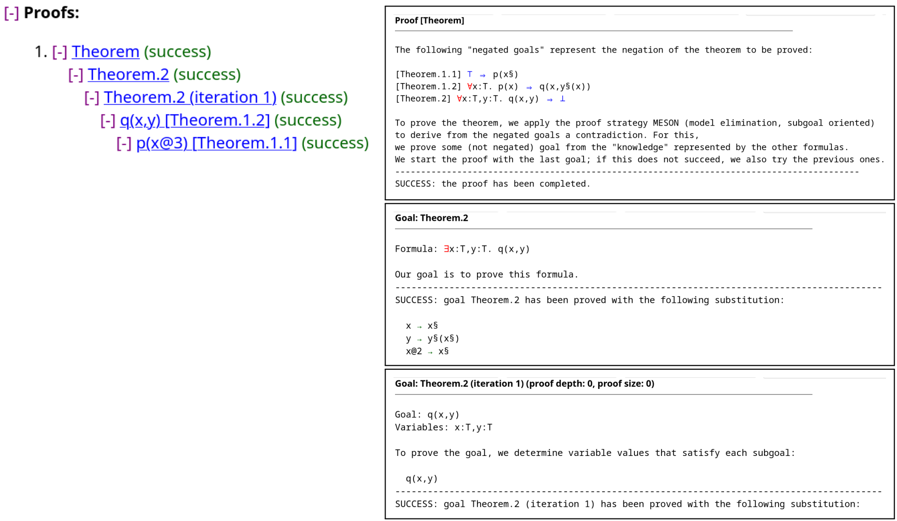
Proof Problem

```
[-] Problem Simplification:

 [-] Theorem (rule [⇒-R | ∨-R | ∧-L] on the goal)
   [-] Theorem (rule [⇒-R | ∨-R | ∧-L] on [1])
     [-] Theorem (rule [∀-R | ∃-L] on [1])
       [ ] Theorem (open)

[-] Subproblems:

    1. Theorem

[-] Clause Forms:

    1. Theorem

[-] Proofs:

    1. [-] Theorem (success)
         [-] Theorem.2 (success)
           [-] Theorem.2 (iteration 1) (success)
             [-] q(x,y) [Theorem.1.2] (success)
               [-] p(x@3) [Theorem.1.1] (success)

[-] Proof Search:
```

```
1:[Theorem.1.1] p(x§)
2:[Theorem.1.2] ∀x:T. (p(x) ⇒ (∃y:T. q(x,y)))

goal:[Theorem.2] ∃x:T,y:T. q(x,y)
```

Proof succeeds with Method MESON.

Another Proof Problem (Continued)



[-] **Proofs:**

1. [-] Theorem (success)
   - [-] Theorem.2 (success)
     - [-] Theorem.2 (iteration 1) (success)
       - [-] q(x,y) [Theorem.1.2] (success)
         - [-] p(x@3) [Theorem.1.1] (success)

Proof [Theorem]

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

[Theorem.1.1] ⊤ ⇒ p(x§)
[Theorem.1.2] ∀x:T. p(x) ⇒ q(x,y§(x))
[Theorem.2] ∀x:T,y:T. q(x,y) ⇒ ⊥

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

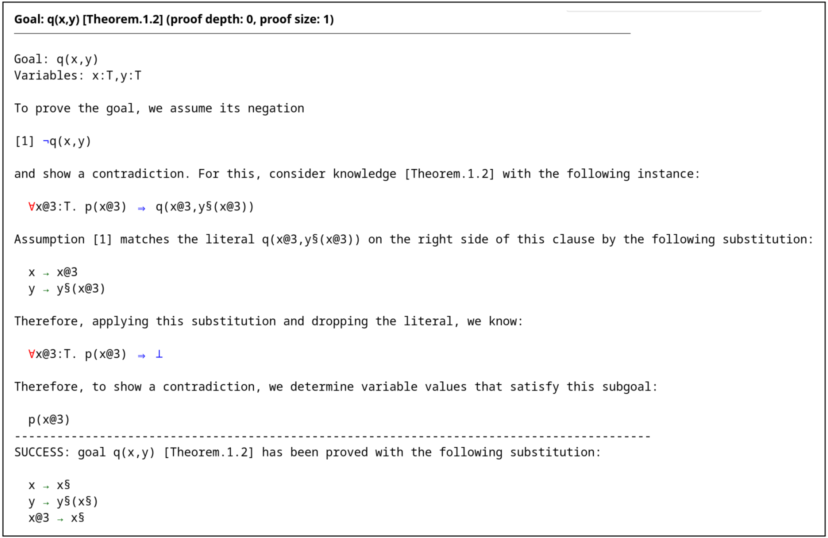
Goal: Theorem.2

```
Formula: ∃x:T,y:T. q(x,y)

Our goal is to prove this formula.
------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.2 has been proved with the following substitution:

  x → x§
  y → y§(x§)
  x@2 → x§
```

Goal: Theorem.2 (iteration 1) (proof depth: 0, proof size: 0)

```
Goal: q(x,y)
Variables: x:T,y:T

To prove the goal, we determine variable values that satisfy each subgoal:

  q(x,y)
------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.2 (iteration 1) has been proved with the following substitution:
```

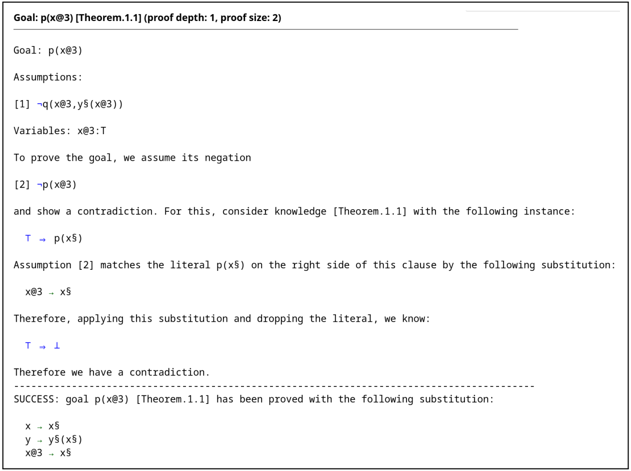
Another Proof Problem (Continued)



Goal: q(x,y) [Theorem.1.2] (proof depth: 0, proof size: 1)

```
Goal: q(x,y)
Variables: x:T,y:T

To prove the goal, we assume its negation

[1] ¬q(x,y)

and show a contradiction. For this, consider knowledge [Theorem.1.2] with the following instance:

  ∀x@3:T. p(x@3) ⇒ q(x@3,y§(x@3))

Assumption [1] matches the literal q(x@3,y§(x@3)) on the right side of this clause by the following substitution:

  x → x@3
  y → y§(x@3)

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

  ∀x@3:T. p(x@3) ⇒ ⊥

Therefore, to show a contradiction, we determine variable values that satisfy this subgoal:

  p(x@3)
-----------------------------------------------------------------------------------------
SUCCESS: goal q(x,y) [Theorem.1.2] has been proved with the following substitution:

  x → x§
  y → y§(x§)
  x@3 → x§
```

Another Proof Problem (Continued)



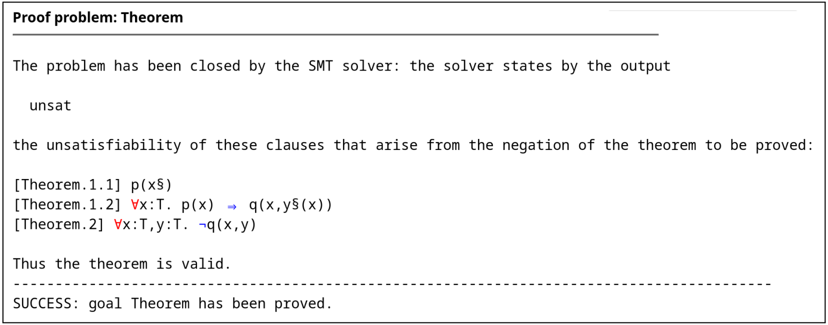
Goal: p(x@3) [Theorem.1.1] (proof depth: 1, proof size: 2)

```
Goal: p(x@3)

Assumptions:

[1] ¬q(x@3,y§(x@3))

Variables: x@3:T

To prove the goal, we assume its negation

[2] ¬p(x@3)

and show a contradiction. For this, consider knowledge [Theorem.1.1] with the following instance:

  ⊤ ⇒ p(x§)

Assumption [2] matches the literal p(x§) on the right side of this clause by the following substitution:

  x@3 → x§

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

  ⊤ ⇒ ⊥

Therefore we have a contradiction.
------------------------------------------------------------------------------------------
SUCCESS: goal p(x@3) [Theorem.1.1] has been proved with the following substitution:

  x → x§
  y → y§(x§)
  x@3 → x§
```

Proof succeeds by instantiating in clause 2 variable x with constant $x§$ and in clause 3 variables x and y with constants $x§$ and $y§(x§)$, respectively.

Anoter Proof Problem (Continued)

Set option “SMT: Max”.



Proof problem: Theorem

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

  unsat

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

[Theorem.1.1] p(x§)
[Theorem.1.2] ∀x:T. p(x) ⇒ q(x,y§(x))
[Theorem.2] ∀x:T,y:T. ¬q(x,y)

Thus the theorem is valid.
-------------------------------------------------------------------------------------------
SUCCESS: goal Theorem has been proved.
```

Here the actual clause instances could not be determined (a simple strategy is applied that attempts only instantiations with variable-free terms that appear in the proof problem).