Formal Methods for Secure Systems

Master of Science in Computer Engineering

Introduction to PVS (Prototype Verification System) proofs

Andrea Domenici

Department of Information Engineering
University of Pisa, Italy

April 13, 2020

Fundamental rules of sequent calculus (1)

Let us recall the fundamental inference rules. In the following, letters A and B denote formulae, while Γ and Δ denote possibly empty multisets of formulae. In repeated occurrences of a same letter, the corresponding formulae are equal modulo variable renaming, i.e., *syntactically equivalent*. For example, the following formulae are syntactically equivalent:

```
FORALL (x: real): p(x)
FORALL (y: real): p(y)
```

The syntactical equivalence ($\overset{\text{syntax}}{\equiv}$) of the two formulae can be expressed as $\forall_{x\in\mathbb{R}}\mathrm{p}(x) \overset{\text{syntax}}{\equiv} \forall_{y\in\mathbb{R}}\mathrm{p}(y)$.

Note: the $\overset{\text{syntax}}{\equiv}$ symbol belongs to the metalanguage, not to the underlying sequent calculus language.

Fundamental rules of sequent calculus (2)

Axiom: $\dfrac{}{\Gamma, A \vdash A, \Delta}\,\mathrm{axm}$

a sequent is proved if a formula occurs both as an antecedent and a consequent.

Cut: $\dfrac{\Gamma \vdash \Delta, A \qquad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\,\mathrm{cut}$

Cut allows an additional assumption to be introduced and later proved (or *discharged*).

Contraction: $\dfrac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\,\mathrm{ctr\ L} \qquad \dfrac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\,\mathrm{ctr\ R}$

Contraction allows a formula to be replicated. Useful when a lemma is used more than once.

Fundamental rules of sequent calculus (3)

Negation: $\dfrac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}\neg L \quad \dfrac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\neg R$

Negating a formula produces a sequent where the formula is moved to the other side of the turnstile. This is done automatically by the prover.

Conjunction: $\dfrac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\wedge L \quad \dfrac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\wedge R$

In the antecedent, a conjunction is equivalent to the set of its conjuncts as self-standing formulae. In the consequent, proving a conjunction requires proving each conjunct.

Disjunction: $\dfrac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}\vee L \quad \dfrac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}\vee R$

A disjuncion in the antecedent means that the consequent can be proved both from one disjunct and from the other. In the consequent, a disjunction is equivalent to the set of its disjuncts as self-standing formulae.

Fundamental rules of sequent calculus (4)

Implication: $\dfrac{\Gamma\vdash\Delta, A \quad B,\Gamma\vdash\Delta}{A\Rightarrow B,\Gamma\vdash\Delta}{\Rightarrow}L \qquad \dfrac{A,\Gamma\vdash\Delta, B}{\Gamma\vdash A\Rightarrow B,\Delta}{\Rightarrow}R$

Both rules for implication can be obtained using the identity $A \Rightarrow B = \neg A \vee B$ and the rules for conjunction and negation.

Universal: $\dfrac{A[x\leftarrow t],\Gamma\vdash\Delta}{\forall x.A,\Gamma\vdash\Delta}\forall L \qquad \dfrac{\Gamma\vdash\Delta, A[x\leftarrow a]}{\Gamma\vdash\forall x.A,\Delta}\forall R$

Existential: $\dfrac{A[x\leftarrow a],\Gamma\vdash\Delta}{\exists x.A,\Gamma\vdash\Delta}\exists L \qquad \dfrac{\Gamma\vdash\Delta, A[x\leftarrow t]}{\Gamma\vdash\exists x.A,\Delta}\exists R$

In the quantifier rules, $A[x \leftarrow t]$ is the formula obtained by substituting a term t for each free occurrence of x in A, possibly renaming bound variables in A to avoid name clashes with variables in t. Similarly, a denotes a *fresh* constant, i.e., a new constant that does not occur in the conclusion sequent of the rule.

Additional rules: equality

Equality: $\dfrac{}{GG \vdash a{=}b, \Delta}\,eq \qquad \dfrac{a{=}b, \Gamma[b] \vdash \Delta[b]}{a{=}b, \Gamma[a] \vdash \Delta[a]}\,repl$

In the above rules, *a* and *b* are syntactically equivalent terms. In the consequent, a true equality proves the sequent. In the antecedent, an equality allows occurrences of *b* to be replaced by *a* throughout the sequent.

Additional rules: IF

IF operator:: $\dfrac{\Gamma, A, B \vdash \Delta \quad \Gamma, \neg A, C \vdash \Delta}{\Gamma, IF(A, B, C) \vdash \Delta}\,\mathtt{IF}L \qquad \dfrac{\Gamma, A, \vdash B\Delta \quad \Gamma, \neg A, \vdash C\Delta}{\Gamma \vdash IF(A, B, C), \Delta}\,\mathtt{IF}L$

In the above rules, `IF` is of type `[bool, bool, bool -> bool]` and occurs at top level in the sequent. The rules can be derived from the fundamental rules, using the definition of the `IF` operator introduced in the lecture on the PVS language.

An `IF` in the antecedent produces a subgoal to prove the consequent from A and B and a subgoal to prove the consequent from $\neg A$ and C. In the consequent, it produces a subgoal to prove B from A and a subgoal to prove C from $\neg A$.

Lifting: $\dfrac{\Gamma, \mathtt{IF}(A, B[b], B[b) \vdash \Delta}{\Gamma, B(IF(A, b, c) \vdash \Delta}\,liftingL \qquad \dfrac{\Gamma, \vdash \mathtt{IF}(A, B[b], B[b) \; \Delta}{\Gamma, \vdash B(IF(A, b, c) \; \Delta}\,liftingR$

Lifting allows an `IF` expression of any type, occurring as a term in a formula, to be factored out and transformed into a top level sequent formula.

Backward-chaining

The inference rules of sequent calculus are divided into left- and right- *introduction* rules, i.e., in each rule the consequence is a sequent obtained by introducing a connective or quantifier in the left or right side of the turnstile, i.e., in the antecedent or consequent side.

An introduction rule becomes an *elimination* rule when applied *backwards*.

The axiom is equivalent to an inference rule with an empty premise.

A proof can then be carried out backwards, starting from the goal and applying elimination rules until all branches of the proof tree are closed by the empty sequent.

The *prover commands* available in the PVS proof environment are based on the fundamental sequent calculus rules and on other rules (obviously consistent with the fundamental ones) dealing with equality, introduction of lemmas and definitions, and such.

Proof pragmatics

Proving a goal involves the following operations:

- *eliminating quantifiers*
- *disassembling the goal* into simpler subgoals
- *introducing axioms and proved lemmas*, possibly from other theories
- *instantiating axioms and lemmas*
- *expanding definitions*
- *replacing terms* using equality.
- *case reasoning*
- *using induction*

A proof may not need all the above operations.

Any operation may be applied several times in the proof.

Dependencies exist between operations. For example, instantiating lemmas often amounts to quantifier elimination. Quantifier elimination is often the first step in a proof, but not in inductive proofs.

Prover commands

The prover commands are based on the above rules, but they usually do not correspondi directly to single rules, e.g., there is no command called "*left-AND-elimination*". Many commands use different rules depending on whether they are applied to an antecedent or consequent formula.

Prover commands are divided into *primitive rules*, *strategies*, and *defined rules*. Primitive rules produce (if successful) one or more new goals in one step. Strategies produce a new subtree rooted at the current goal. Defined rule are strategies executed as a single step, since only the leaves of the generated subtree are displayed by the prover.

Only a few commands will be shown, and in their simplest form. Please refer to the PVS prover manual for full syntax and semantics.

Most commands refer to formulae by their number (*fnum*). Antecedents have negative numbers, consequents positive numbers. A single '-' or '+' refers to all antecedents or consequents, respectively. An asterisk ('$*$') refers to all formulae.

Thoralf Albert Skolem (1887 – 1963)

Quantifier elimination: consequent (1)

skolem! replaces universally quantified variables with prover-generated fresh constants, e.g.,

```
  |-------                            % consequent FORALL: (skolem!)
{1}   FORALL (x: int): p(x)
Rule? (skolem!)
  |-------
{1}   p(x!1)
```

skosimp applies `skolem!` followed by `flatten` (see below).

skosimp* repeats `skosimp`.

skeep replaces universally quantified variables with constants named as the replaced variables, e.g.,

```
  |-------                            % consequent FORALL: (skeep)
{1}   FORALL (x: int): p(x)
Rule? (skeep)
  |-------
{1}   p(x)
```

Quantifier elimination: consequent (2)

inst replaces existentially quantified variables with terms not containing free variables, e.g.,

```
  |-------                         % consequent EXISTS: (inst)
{1}   EXISTS (x: int): p(x)
Rule? (inst 1 3)
  |-------
{1}   p(3)
```

Quantifier elimination: antecedent (1)

inst replaces universally quantified variables with terms not containing free variables, e.g.,

```
{-1}   FORALL (x: int): p(x)
  |-------                                 % antecedent FORALL: (inst)
{1}   p(3)
Rule? (inst -1 3)
{-1}   p(3)
  |-------
{1}   p(3)
```

inst? finds and applies adequate substitutions to eliminate a universal quantifier.

Logical operator elimination: consequent (1)

split eliminates top-level conjunctions, equivalences, and `IF`'s, e.g.,

```
  |-------                            % consequent AND: (split)
{1}   A AND B
Rule? (split)
  |-------               |-------
{1}   A                {1}   B

----------------------------------

  |-------                            % consequent IFF: (split)
{1}   A IFF B
Rule? (split)
  |-------               |-------
{1}   A IMPLIES B       {1}   B IMPLIES A
```

Logical operator elimination: consequent (2)

```
  |-------                                 % consequent IF: (split)
{1}   IF A THEN B ELSE C ENDIF
Rule?: (split)
  |-------                 |-------
{1}   A IMPLIES B        {1}   NOT A IMPLIES C
```

Logical operator elimination: consequent (3)

flatten eliminates top-level disjunctions and implications, e.g.,

```
  |-------                                % consequent OR: (flatten)
{1}   A OR B
Rule? (flatten)
  |-------
{1}   A
{2}   B

----------------------------------

  |-------                                % consequent IMPLIES: (flatten)
{1}   A IMPLIES B
Rule?: (flatten)
{-1}  A
  |-------
{1}   B
```

Logical operator elimination: antecedent (1)

split eliminates top-level disjunctions, implications, and `IF`'s, e.g.,

```
{-1}  (A OR B)
  |-------                               % antecedent OR: (split)
Rule?: (split)
{-1}  A                  {-1}  B
  |-------                 |-------


-----------------------------------


{-1}  (A IMPLIES B)
  |-------                               % antecedent IMPLIES: (split)
Rule?: (split)
{-1}  B
  |-------                 |-------
                         {1}   A
```

Logical operator elimination: antecedent (2)

```
{-1}  IF A THEN B ELSE C ENDIF
  |-------                                  % antecedent IF: (split)
Rule? (split)
{-1}  A AND B                {-1}  NOT A AND C
  |-------                     |-------
```

Logical operator elimination: antecedent (3)

flatten eliminates top-level conjunctions and equivalences, e.g.,

```
{-1}  (A AND B)
  |-------                          % antecedent AND: (flatten)
Rule?: (flatten)
{-1}  A
{-2}  B
  |-------


----------------------------------


{-1}  (A IFF B)
  |-------                          % antecedent IFF: (flatten)
Rule? (flatten)
{-1}  A IMPLIES B
{-2}  B IMPLIES A
  |-------
```

Logical operator elimination: XOR

The exclusive OR operator is defined in the prelude as a function:

```
A, B: VAR bool
XOR(A, B): bool = (A /= B)
```

Hence, it can be eliminated using **expand** (see below) or introducing Lemma *xor_def* from the prelude:

```
xor_def: LEMMA (A xor B) = IF A THEN NOT B ELSE B ENDIF
```

Logical operator elimination: IF-lifting

lift-if is not really an operator elimination, but it produces a top-level Boolean IF-expression (i.e., a formula) from an IF-expression of any type. For example, given

```
A: bool; T: TYPE+; s,t: T; p, q: pred[T]; x: VAR T;
```

we can prove

```
  |-------
{1}   FORALL (x: T): p(IF A THEN s ELSE t ENDIF) IMPLIES q(x)
Rule? (skeep)
{-1}  p(IF A THEN s ELSE t ENDIF)
  |-------
{1}   q(x)
Rule? (lift-if)
{-1}  IF A THEN p(s) ELSE p(t) ENDIF
  |-------
[1]   q(x)
```

Logical operator elimination: summary table

operator	consequent	antecedent
`AND, /\`	split	flatten
`IFF, <=>`	split	flatten
`IF`	split	split
`OR, \/`	flatten	split
`IMPLIES, =>`	flatten	split
`XOR`	expand, xor_def	
IF-lifting	lift-if	

Definitions and lemmas (1)

expand replaces occurrences of function applications with the function's definition. Expansion may take place throughout the sequent, or in selected formulae, or in selected occurrences in a formula.

lemma introduces an axiom or theorem into the antecedent. The formula may have been declared in the current theory, in a prelude theory, in a theory imported from the context, or in a theory imported from a library.

Name conflicts are resolved by prepending the name of the defining theory and library to the name of the formula.

rewrite takes the specified lemma, finds appropriate substitutions, and uses the resulting formula to rewrite another formula.

Equality (1)

replace uses an antecedent equality formula to rewrite terms throughout the sequent or in selected formulae.

beta simplifies various forms of function applications, including

- `LET`-expressions,
- `LAMBDA`-expressions,
- tuple member selection,
- record member selection,

Case reasoning (1)

case introduces one or more assuptions in the antecedent. This produces a new subgoal to prove the consequent under the new assumptions, and other subgoals to prove the consequent under the negation of each assumption. For example, given

```
T: TYPE+; s: T; p, q: pred[T]; x: VAR T;
```

we can prove

```
  |-------
{1}   FORALL (x: T): p(x) IMPLIES q(x)
Rule? (skeep)
{-1}  p(x)
  |-------
{1}   q(x)
Rule? (case "x = s")
{-1}  x = s
[-2]  p(x)                [-1]  p(x)
  |-------                  |-------
[1]   q(x)                {1}   x = s
                           [2]   q(x)
```

Case reasoning (2)

Let us consider another example. Given

```
x: VAR real
```

we can prove

```
case_split_1 :
  |-------
{1}   FORALL (x: real): x * x >= 0
Rule?: (skeep)
case_split_1 :
  |-------
{1}   x * x >= 0
Rule?: (case "x = 0")
this yields 2 subgoals:
```

Case reasoning (3)

```
case_split_1.1 :
{-1}  x = 0                         % case_split_1.1: assume x = 0
  |-------
[1]   x * x >= 0
Rule?: (replace -1 1)
[-1]  x = 0
  |-------
{1}   0 * 0 >= 0
Rule?: (assert)
This completes the proof of case_split_1.1.
```

Case reasoning (4)

```
case_split_1.2 :
  |-------
{1}   x = 0
[2]   x * x >= 0
Rule?: (hide 1)
  |-------
[1]   x * x >= 0
Rule?: (case "x > 0")
this yields 2 subgoals:
```

Case reasoning (5)

```
case_split_1.2.1 :                    % case_split_1.2.1: assume x > 0
{-1}  x > 0
  |-------
[1]   x * x >= 0
Rule?: (lemma pos_times_gt)           % from the prelude
{-1}  FORALL (x, y: real):
        x * y > 0 IFF (0 > x AND 0 > y) OR (x > 0 AND y > 0)
[-2]  x > 0
  |-------
[1]   x * x >= 0
Rule?: (inst?)
{-1}  x * x > 0 IFF (0 > x AND 0 > x) OR (x > 0 AND x > 0)
[-2]  x > 0
  |-------
[1]   x * x >= 0
```

Case reasoning (6)

```
Rule?: (flatten)
{-1}  x * x > 0 IMPLIES (0 > x AND 0 > x) OR (x > 0 AND x > 0)
{-2}  (0 > x AND 0 > x) OR (x > 0 AND x > 0) IMPLIES x * x > 0
[-3]  x > 0
  |-------
[1]   x * x >= 0
Rule?: (hide -1)
[-1]  (0 > x AND 0 > x) OR (x > 0 AND x > 0) IMPLIES x * x > 0
[-2]  x > 0
  |-------
[1]   x * x >= 0
Rule?: (split)
this yields  2 subgoals:
```

Case reasoning (7)

```
case_split_1.2.1.1 :
{-1}  x * x > 0
[-2]  x > 0
  |-------
[1]   x * x >= 0
Rule?: (assert)
This completes the proof of case_split_1.2.1.1.
```

Case reasoning (8)

```
case_split_1.2.1.2 :
[-1]  x > 0
  |-------
{1}   (0 > x AND 0 > x) OR (x > 0 AND x > 0)
[2]   x * x >= 0
Rule?: (flatten)
[-1]  x > 0
  |-------
{1}   (0 > x AND 0 > x)
{2}   (x > 0 AND x > 0)
[3]   x * x >= 0
Rule?: (hide 1)
[-1]  x > 0
  |-------
[1]   (x > 0 AND x > 0)
[2]   x * x >= 0
Rule?: (split)
this yields  2 subgoals:
```

Case reasoning (9)

```
case_split_1.2.1.2.1 :
[-1]  x > 0
  |-------
{1}   x > 0
[2]   x * x >= 0
This completes the proof of case_split_1.2.1.2.1.

case_split_1.2.1.2.2 :
[-1]  x > 0
  |-------
{1}   x > 0
[2]   x * x >= 0
This completes the proof of case_split_1.2.1.2.2
This completes the proof of case_split_1.2.1.2.
This completes the proof of case_split_1.2.1.
```

Case reasoning (11)

```
case_split_1.2.2 :
  |-------
{1}   x > 0
[2]   x * x >= 0
Rule?: (case "x < 0")
this yields 2 subgoals:
```

Case reasoning (12)

```
case_split_1.2.2.1 :                      % case_split_1.2.2.1: assume x < 0
{-1}  x < 0
  |-------
[1]   x > 0
[2]   x * x >= 0
Rule?: (lemma pos_times_gt)                     % from the prelude
case_split_1.2.2.1 :
{-1}  FORALL (x, y: real):
        x * y > 0 IFF (0 > x AND 0 > y) OR (x > 0 AND y > 0)
[-2]  x < 0
  |-------
[1]   x > 0
[2]   x * x >= 0
Rule?: (inst?)
```

Case reasoning (13)

```
case_split_1.2.2.1 :
{-1}  x * x > 0 IFF (0 > x AND 0 > x) OR (x > 0 AND x > 0)
[-2]  x < 0
  |-------
[1]   x > 0
[2]   x * x >= 0
Rule?: (flatten)
{-1}  x * x > 0 IMPLIES (0 > x AND 0 > x) OR (x > 0 AND x > 0)
{-2}  (0 > x AND 0 > x) OR (x > 0 AND x > 0) IMPLIES x * x > 0
[-3]  x < 0
  |-------
[1]   x > 0
[2]   x * x >= 0
Rule?: (hide -1)
```

Case reasoning (14)

```
[-1]  (0 > x AND 0 > x) OR (x > 0 AND x > 0) IMPLIES x * x > 0
[-2]  x < 0
  |-------
[1]   x > 0
[2]   x * x >= 0
Rule?: (split)
this yields  2 subgoals:
case_split_1.2.2.1.1 :
{-1}  x * x > 0
[-2]  x < 0
  |-------
[1]   x > 0
[2]   x * x >= 0
Rule?: (assert)
This completes the proof of case_split_1.2.2.1.1.
```

Case reasoning (15)

```
case_split_1.2.2.1.2 :
[-1]  x < 0
  |-------
{1}   (0 > x AND 0 > x) OR (x > 0 AND x > 0)
[2]   x > 0
[3]   x * x >= 0
Rule?: (flatten)
[-1]  x < 0
  |-------
{1}   (0 > x AND 0 > x)
{2}   (x > 0 AND x > 0)
[3]   x > 0
[4]   x * x >= 0
Rule?: (hide 2)
```

Case reasoning (16)

```
[-1]  x < 0
  |-------
[1]   (0 > x AND 0 > x)
[2]   x > 0
[3]   x * x >= 0
Rule?: (split)
this yields  2 subgoals:
case_split_1.2.2.1.2.1 :
[-1]  x < 0
  |-------
{1}   0 > x
[2]   x > 0
[3]   x * x >= 0
Rule?: (assert)
This completes the proof of case_split_1.2.2.1.2.1.
```

Case reasoning (17)

```
case_split_1.2.2.1.2.2 :
[-1]  x < 0
  |-------
{1}   0 > x
[2]   x > 0
[3]   x * x >= 0
Rule?: (assert)
This completes the proof of case_split_1.2.2.1.2.2
This completes the proof of case_split_1.2.2.1.2.
This completes the proof of case_split_1.2.2.1.
```

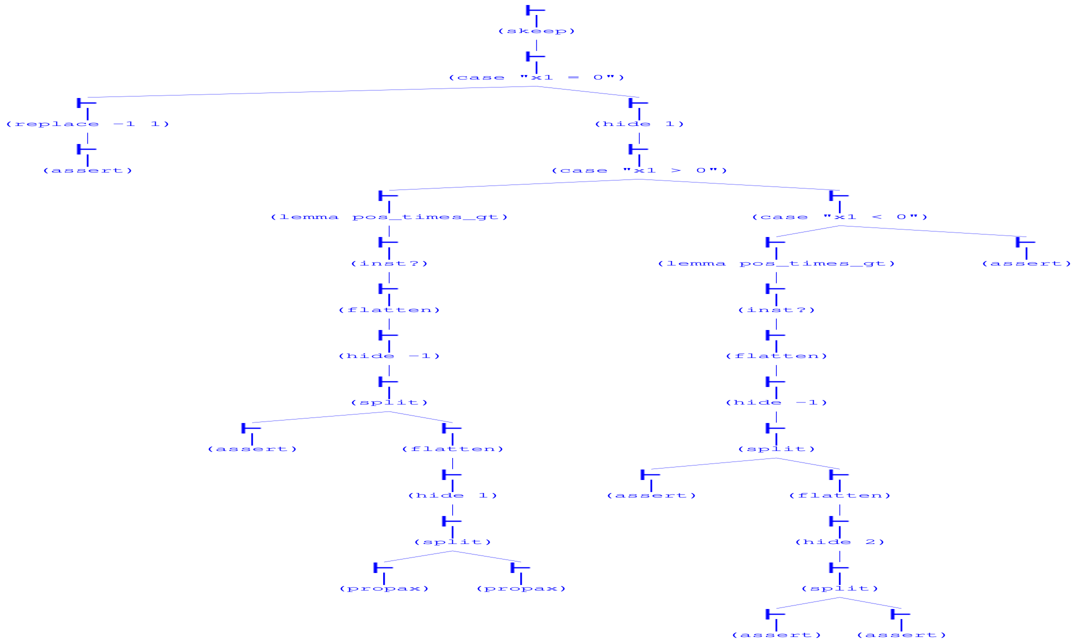
Case reasoning (18)

```
case_split_1.2.2.2 :
  |-------
{1}   x < 0
[2]   x > 0
[3]   x * x >= 0
Rule?: (assert)
This completes the proof of case_split_1.2.2.2
This completes the proof of case_split_1.2.2.
This completes the proof of case_split_1.2.

Q.E.D.
```

Note: the last 18 slides were only meant to show the use of **case**. Actually, the goal can be proved in two steps: **skeep** and **assert**.

Case reasoning (19)



Induction (1)

An *induction schema* is an axiom of this form:

$$\forall_p\,((p(\mathbf{0}) \wedge \forall_x p(x) \Rightarrow p(\mathrm{s}(x))) \\ \Rightarrow \forall_x p(x)) \tag{1}$$

where x belongs to a set T on which a *successor* function 's' is defined, '$\mathbf{0}$' is the element of T that is not successor of any element, and p is a predicate over T.

The best known induction schema is the *mathematical induction* axiom:

$$\forall_p\,((p(0) \wedge \forall_n p(n) \Rightarrow p((n+1))) \\ \Rightarrow \forall_n p(n)) \tag{2}$$

where $n \in \mathbb{N}$.

Several variants exist of the basic induction schema, e.g., *structural induction* on data structures such as lists.

Induction (2)

induct introduces an appropriate induction schema to split a universally quantified goal into the base case and the induction step. For example, given

```
    sum(n): RECURSIVE nat =       % sum of first n numbers
        IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF
    MEASURE n
```

we can prove:

```
sum_closed_form :
  |-------
{1}   FORALL (n: nat): sum(n) = 1 / 2 * n * (n + 1)
Rule?: (induct n)
```

I.e., introduce an induction schema on variable n.

Induction (3)

```
sum_closed_form.1 :                    % base case
  |-------
{1}   sum(0) = 1 / 2 * 0 * (0 + 1)
Rule?: (assert)

  |-------
{1}   sum(0) = 0
Rule?: (grind)

This completes the proof of sum_closed_form.1.
```

Induction (4)

```
sum_closed_form.2 :                % induction step
  |-------
{1}   FORALL j:
        sum(j) = 1 / 2 * j * (j + 1) IMPLIES
         sum(j + 1) = (1 / 2 * (j + 1)) * (j + 1 + 1)
Rule?: (skeep)

{-1}  sum(j) = 1 / 2 * j * (j + 1)
  |-------
{1}   sum(j + 1) = (1 / 2 * (j + 1)) * (j + 1 + 1)
Rule?: (expand sum)

{-1}  IF j = 0 THEN 0 ELSE sum(j - 1) + j ENDIF = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   sum(j) = 1/2 * (j * j) + 1/2 * j
Rule?: (expand sum)
```

Induction (5)

```
{-1}  IF j = 0 THEN 0
      ELSE IF j - 1 = 0 THEN 0 ELSE sum(j - 2) - 1 + j ENDIF + j
      ENDIF
       = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   IF j = 0 THEN 0 ELSE sum(j - 1) + j ENDIF = 1/2 * (j * j) + 1/2 * j
Rule?: (lift-if)

{-1}  IF j = 0 THEN 0 = 1/2 * (j * j) + 1/2 * j
      ELSE IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
           ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j
           ENDIF
      ENDIF
  |-------
{1}   IF j = 0 THEN 0 = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
      ENDIF
Rule?: (split)
```

Induction (6)

```
sum_closed_form.2.1 :
{-1}  j = 0 AND 0 = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   IF j = 0 THEN 0 = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
      ENDIF
Rule?: (flatten)

{-1}  j = 0
{-2}  0 = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   IF j = 0 THEN 0 = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
      ENDIF
Rule?: (hide -2)
```

Induction (7)

```
[-1]  j = 0
  |-------
[1]   IF j = 0 THEN 0 = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
      ENDIF
Rule?: (replace -1 1)

[-1]  j = 0
  |-------
{1}   0 = 1/2 * (0 * 0) + 1/2 * 0
Rule?: (assert)

This completes the proof of sum_closed_form.2.1.
```

Induction (8)

```
sum_closed_form.2.2 :
{-1}  NOT j = 0 AND
       IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
       ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j ENDIF
  |-------
[1]   IF j = 0 THEN 0 = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j ENDIF
Rule?: (split)

sum_closed_form.2.2.1 :
[-1]  NOT j = 0 AND
       IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
       ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j ENDIF
  |-------
{1}   j = 0 IMPLIES 0 = 1/2 * (j * j) + 1/2 * j
Rule?: (flatten)

This completes the proof of sum_closed_form.2.2.1.
```

Induction (9)

```
sum_closed_form.2.2.2 :
[-1]  NOT j = 0 AND
       IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
       ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j
       ENDIF
  |-------
{1}   NOT j = 0 IMPLIES sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (flatten)

{-1}  IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j
      ENDIF
  |-------
{1}   j = 0
{2}   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
{3}   j = 0
Rule?: (hide 1 3)
```

Induction (10)

```
[-1]  IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j
      ENDIF
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (split)
sum_closed_form.2.2.2.1 :
{-1}  j - 1 = 0 AND 0 + j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (flatten)

{-1}  j - 1 = 0
{-2}  0 + j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (both-sides + 1 -1)
```

Induction (11)

```
{-1}  j - 1 + 1 = 0 + 1
[-2]  0 + j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (assert)

{-1}  j = 1
{-2}  j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (replace -1 1)

[-1]  j = 1
[-2]  j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   sum(1 - 1) + 1 = 1/2 * (1 * 1) + 1/2 * 1
Rule?: (assert)
```

Induction (12)

```
[-1]  j = 1
[-2]  j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   sum(0) = 0
Rule?: (hide -1 -2)

  |-------
[1]   sum(0) = 0
Rule?: (expand sum)

  |-------
{1}   TRUE

This completes the proof of sum_closed_form.2.2.2.1.
```

Induction (13)

```
sum_closed_form.2.2.2.2 :
{-1}  NOT j - 1 = 0 AND sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (assert)

{-1}  NOT j - 1 = 0 AND sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (flatten)

{-1}  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   j - 1 = 0
[2]   sum(j - 1) + j = 1/2 * (j * j) + 1/2 * j
Rule?: (expand sum 2)
```

Induction (14)

```
[-1]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   j - 1 = 0
{2}   IF j - 1 = 0 THEN 0 ELSE sum(j - 2) - 1 + j ENDIF + j =
       1/2 * (j * j) + 1/2 * j
Rule?: (hide 1)

[-1]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   IF j - 1 = 0 THEN 0 ELSE sum(j - 2) - 1 + j ENDIF + j =
       1/2 * (j * j) + 1/2 * j
Rule?: (lift-if)

[-1]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   IF j - 1 = 0 THEN 0 + j = 1/2 * (j * j) + 1/2 * j
      ELSE sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j ENDIF
Rule?: (split)
```

Induction (15)

```
{-1}  j - 1 = 0
[-2]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   0 + j = 1/2 * (j * j) + 1/2 * j
Rule?: (both-sides + 1 -1)

{-1}  j - 1 + 1 = 0 + 1
[-2]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
[1]   0 + j = 1/2 * (j * j) + 1/2 * j
Rule?: (assert)

This completes the proof of sum_closed_form.2.2.2.2.1.
```

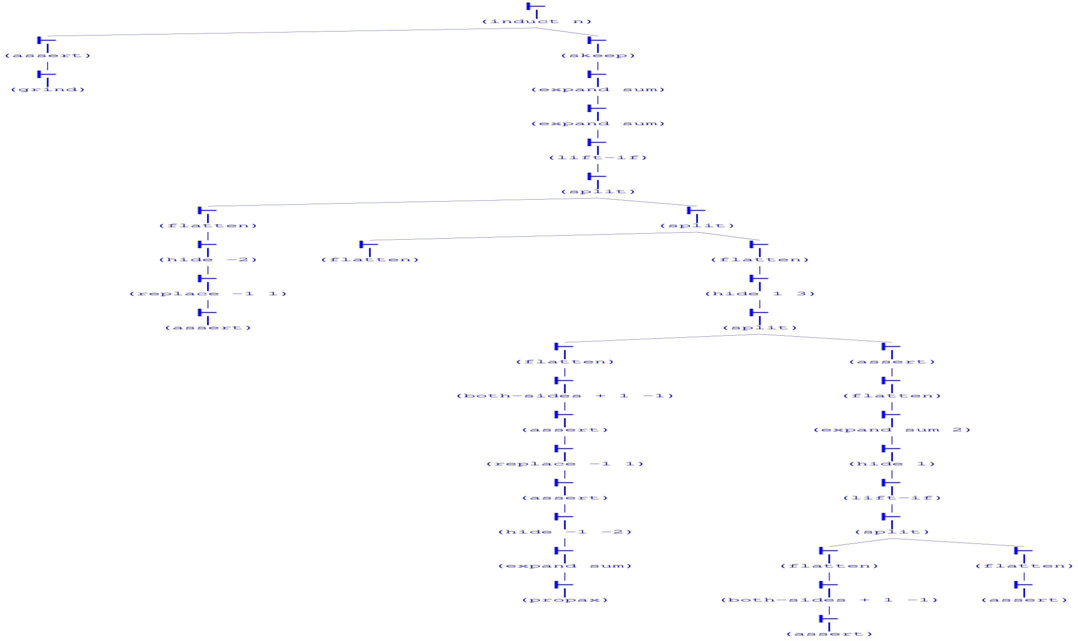
Induction (16)

```
sum_closed_form.2.2.2.2.2 :
[-1]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   NOT j - 1 = 0 IMPLIES sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 *
Rule?: (flatten)

[-1]  sum(j - 2) - 1 + 2 * j = 1/2 * (j * j) + 1/2 * j
  |-------
{1}   j - 1 = 0
{2}   sum(j - 2) - 1 + j + j = 1/2 * (j * j) + 1/2 * j
Rule?: (assert)
This completes the proof of sum_closed_form.2.2.2.2.2.
This completes the proof of sum_closed_form.2.2.2.2.
This completes the proof of sum_closed_form.2.2.2.
This completes the proof of sum_closed_form.2.2.
This completes the proof of sum_closed_form.2.
Q.E.D.
```

Note: try **induct-and-simplify**.

Induction (17)



Decision procedures

Decision procedures are algorithms to simplify and possibly solve a sequent.

assert applies various simplifications to ground formulae. Sometimes it leaves a sequent unchanged, but it still has the effect of producing internal information that facilitates further simplifications.

in particular, **assert** can simplify simple arithmetic expressions.

grind repeatedly applies verious simplifications, including **assert**.

both-sides simplifies arithmetic expressions by applying the same operation (e.g., adding or subtracting a term) to both sides of an equality or inequality.

Copying, hiding, and proof control

copy introduces a duplicate of the specified formula. This is useful before using commands (e.g., **inst**) that replace a formula with another. Some commands (e.g., **inst-cp**) incorporate **copy**.

hide makes the specified formulae invisible to both the user and the prover. This makes the sequent more readable and may save execution time.

reveal reintroduces hidden formulae. The user calls a prover interface command to display a list of hidden formulae, then he/she supplies the number of the selected formulae to the **reveal** command.

undo discards one or more sequents, starting from the current one.

postpone makes the next open subgoal the current one.

quit exits from the prover environment.

Note: use the **x-prove** prover interface command to show a dynamic graphic display of the proof tree. Very useful.

Type checking (1)

The complex PVS type system rules are enforced by the *type checker* component of the prover, which generates *type correctness conditions* (TCC) when it cannot decide if a formula is type-correct. For example, given

```
xx: real
```

we try to prove

```
tcc :
-1  FORALL (n: nat): p(n) IMPLIES q(n)
   |-------

Rule? (inst -1 xx)
this yields  2 subgoals:
```

Type checking (2)

```
tcc.1 :
-1  p(xx) IMPLIES q(xx)
  |-------

tcc.2 (TCC):
  |-------
1   rational_pred(xx) AND integer_pred(xx) AND xx >= 0
```

If `xx` is declared `nat`, no TCC is generated.

Type checking *dynamically* (during the proof, as in the example above) or *statically*, i.e., before a proof is started. Static type checking can be invoked by the user or by the prover. Statically generated TCCs can be shown and discharged with prover environment command. Usually a small number of steps are necessary, often a single **assert**. If proving a TCC takes long, the theory is probably flawed.