Theorema:
The Predicate Logic Prover

Implementation in Mathematica 3.0

Bruno Buchberger, Tudor Jebelean
Research Institute for Symbolic Computation
A4232 Schloss Hagenberg, Austria

Bruno.Buchberger@RISC.uni–linz.ac.at

Abstract

This notebook describes an automated prover for predicate logic. The style of the prover is natural deduction but the prover is not just an implementation of any of the natural deduction systems usually considered in textbooks of mathematical logic but, rather, reflects and imitates the author's personal proof style for proofs in (higher order) predicate logic. The prover is written in *Mathematica* 3.0. By initializing the notebook the prover can be used interactively from within the notebook. The prover generates abstract proof objects, i.e. nested expressions that contain all the relevant information on the individual steps in a proof. However, various viewers can be applied to proof objects that generate natural language, easy–to–read versions of proofs. Among these viewers is one that generates representations of proofs in nested *Mathematica* cells that can be clicked open and closed and thus enable the user to inspect parts of proofs to any desired level of details or, alternatively, skip subproofs in which he is not interested.

Note to the reader

This *Mathematica* notebook constitutes at the same time:
– the implementation of the predicate logic prover of the *Theorema* project,
– the documentation of the prover,
– the scientific paper describing the prover.
Therefore, some of the content may be relevant only to a particular usage of the notebook. In particular, the printed form is intended mainly for the third purpose, hence the cells containing the details of the implementation are not shown.

■ Overview of the Prover

■ Quick Introduction by Examples

▪ Purpose of the Examples

We give some examples that should suffice for explaining for which type of problems and how the propositional prover described in this report can be used.

■ Initialization

This notebook also contains the *Mathematica* programs which implement the prover. When the notebook is loaded under *Mathematica* 3.0, the prover can be initialized as follows:

– loading (and initialize) the auxiliary programs ("Language.nb", "Prerequisites.nb"),

– initialize this notebook by clicking "Evaluate Initialization" in the menu "Kernel / Evaluation".

■ Example of a Call of the Prover

Suppose we want to prove that the formula

```
(1)  A ∧ B ∧ C  ⇔ A
```

is a consequence of

```
(2)   A ⇒ B
```

and

```
(3)  C .
```

For this we enter

```
Proof[
⟨"1", A ∧ B ∧ C  ⇔ A⟩,
⟨⟨"2",  A ⇒ B⟩, ⟨"3", C⟩⟩]
```

and we will obtain, in a new notebook that appears on the screen, the proof displayed in the subsequent subsubsection.

Note that the hyphens in compound function names etc must be typed in the form: escape character, hyphen, escape character! Note also that we use the angle brackets "⟨ , ⟩" instead of the curly brackets "{, }"for denoting lists.

- **Example of a Computer−Generated Proof**

  Prove

  (1) $A \wedge B \wedge C \Leftrightarrow A$

  under the assumption(s)

  (2) $A \Rightarrow B$,

  (3) C.

  We prove (1) by natural deduction.

  We prove (1) in both directions.

  Direction from left to right:

  We assume

  (1.LA) $A \wedge B \wedge C$

  and show

  (1.RC) A.

  Formula (1.RC) is true because it is identical to (1.LA.1).

  Direction from right to left:

  We assume

  (1.RA) A

  and show

  (1.LC) $A \wedge B \wedge C$.

  From (1.RA) and (2) we obtain by modus ponens

  (2.C) B.

  We prove the individual conjunctive parts of (1.LC):

  Proof of (1.LC.1) A:

  Formula (1.LC.1) is true because it is identical to (1.RA).

  Proof of (1.LC.2) B:

  Formula (1.LC.2) is true because it is identical to (2.C).

  Proof of (1.LC.3) C:

  Formula (1.LC.3) is true because it is identical to (3).

- **Remarks About the Proof**

  The proof is given in natural language, Proof goals, assumptions and labels are in different color for easy reading. Also, the proof steps are nested in notebook cells so that subparts of the proof can be opened and hidden interactively on request by the user. This is an essential feature of our proof viewer which, however, cannot be demonstrated at this place of the manual because it presupposes that the option "CellGrouping −> Manual" is set for the entire notebook. (The option is set to "CellGrouping −> Automatic" for the notebook you are reading.) Thus, for playing with opening and hiding parts of the proof, you must really enter the above call of the prover and work within the extra notebook produced by the prover.

Here, we just present below one of the many other views of the same proof that can be produced by the user from the above proof by just "closing" (by mouse click) some of the nested cells in the proof notebook:

Prove

(1) $A \wedge B \wedge C \Leftrightarrow A$

under the assumption(s)

(2) $A \Rightarrow B$,

(3) C.

We prove (1) by natural deduction.

We prove (1) in both directions.

Direction from left to right:

Direction from right to left:

We assume

(1.RA) A

and show

(1.LC) $A \wedge B \wedge C$.

From (1.RA) and (2) we obtain by modus ponens

(2.C) B.

We prove the individual conjunctive parts of (1.LC):

- **Generation of the Proof Object**

The proof can also be generated in its "abstract form" as a "proof object", i.e. as a nested *Mathematica* expression, by specifying the additional parameter "proof object":

```
Proof[
⟨"1", A ∧ B ∧ C ⇔ A⟩,
⟨⟨"2", A ⇒ B⟩, ⟨"3", C⟩⟩,
"proof object"]
```

The comman above produces the output:

```
⟨⟨"prove equivalence", "1", "1.RC", A, "1.LA", A ∧ B ∧ C,
  "1.LC", A ∧ B ∧ C, "1.RA", A⟩, ⟨⟨"assumption is conjunction",
   "1.LA", ⟨⟨"1.LA.1", A⟩, ⟨"1.LA.2", B⟩, ⟨"1.LA.3", C⟩⟩⟩,
      ⟨⟨"conclusion is assumption",
        "1.RC", "1.LA.1"⟩, "proved"⟩, "proved"⟩,
     ⟨⟨"modus ponens", "1.RA", "2", "2.C", B⟩,
      ⟨⟨"prove conjunction", "1.LC",
        ⟨⟨"1.LC.1", A⟩, ⟨"1.LC.2", B⟩, ⟨"1.LC.3", C⟩⟩⟩,
       ⟨⟨⟨"conclusion is assumption", "1.LC.1", "1.RA"⟩, "proved"⟩,
        ⟨⟨"conclusion is assumption", "1.LC.2", "2.C"⟩, "proved"⟩,
        ⟨⟨"conclusion is assumption", "1.LC.3", "3"⟩, "proved"⟩⟩,
       "proved"⟩, "proved"⟩,
     "proved"⟩
```

This view of the proof is the one used internally. It can be processed by other programs whereas the nested cells view and the indented printed view are meant to be read be the human user and are normally not processed any further. In fact, the nested cells view and the indented printed view result from the abstract proof object by special post–processing programs.

- **Reference to *Mathematica* 3.0**

For using the prover described in this document, nearly no knowledge about *Mathematica* is necessary except that one must know how to load *Mathematica* and how to type *Mathematica* expressions into *Mathematica* cells.

For understanding (and changing) the program that generates the proof objects, only a very basic knowledge of the *Mathematica* language restricted to the "rewrite" programming style is necessary.

For understanding (and changing) the program that generates the natural language nested cells representation of proofs, in addition, the *Mathematica* language constructs for handling notebooks and cells are necessary.

All this information can be obtained from (Wolfram 1996), which we will also cite as **"the *Mathematica* 3.0 book".** In fact, the knowledge of only a very few sections of this manual is sufficient for understanding (and changing) the programs presented in this document.

■ The Problem

This prover solves the following problem:

Given: F, a formula of predicate logic, and
A, a finite sequence of formulae of predicate logic.

Find: P, a proof of F from A,
if F is a tautological consequence of A;
an indication at which point the proof of F from A fails,
if F is not a tautological consequence of A.

(Note that, of course, by the undecidability of first order predicate logic, we may not expect that the prover always terminates. Furthermore, we do not even claim that the prover is complete. And, also, at certain points the prover may ask the user for interactive help.)

Formulae treated by this prover may have one of the following forms:

an atomic formula,
(in fact, any *Mathematica* expression not starting with one of the connectives or quantifiers below is treated as an atomic formula),
true,
false,
not[formula],
Wedge[$\overline{\text{formulae}}$]
(alternative external form: $\text{formula}_1 \wedge ... \wedge \text{formula}_n$),
Vee[$\overline{\text{formulae}}$]
(alternative external form: $\text{formula}_1 \vee ... \vee \text{formula}_n$),
DoubleRightArrow[formula_1, formula_2]
(alternative external form: $\text{formula}_1 \Rightarrow \text{formula}_2$),
DoubleLeftArrow[formula_1, formula_2]
(alternative external form: $\text{formula}_1 \Leftarrow \text{formula}_2$),
DoubleLeftRightArrow[formula_1, formula_2]
(alternative external form: $\text{formula}_1 \Leftrightarrow \text{formula}_2$),
ForAll[variable−list, formula]
(alternative external form: $\forall_{\texttt{variable-list}}$ formula),
Exists[variable−list, formula]
(alternative external form: $\exists_{\texttt{variable-list}}$ formula).

The compound formula have the usual meaning.

For typing special symbols like "⇒" etc in *Mathematica* notebooks, see the respective sections in the *Mathematica* book.

■ The Method

■ The Basic Principle

For each "proof situation", we specify a "proof rule" that reduces the given proof situation to one or several, supposedly simpler, proof situations, until "trivial" proof situations are obtained that can be handled without further reduction.

A proof situation is characterized by the formula to be proved and the current sequence of assumptions. The proof rule to be applied for the reduction of a proof situation is governed by the "outermost connective" in the formula to be proved or, alternatively, by the outermost connective in one of the assumed formulae.

A proof of a formula from a sequence of formulae consists of a trace of the proof steps that were applied for reducing the initial proof situation to trivial ones. The trace of one proof step consists of the name of the proof rule that was applied in this step together with information on the formulae that were involved in this step.

In order to simplify reference to the formulae involved in the proof steps, we provide labels for the initial formulae and systematically generate new labels for all intermediate formulae in a proof.

Typically, a proof is not a linear list of trivial proof steps but, rather, is a deeply nested list of proof steps that results from combining partial proofs for partial proof situations and information on the current proof step in a compound list.

■ Abstract Proof Objects and External Proof Representations

Proofs are internally represented as "abstract proof objects". These are *Mathematica* expressions of the following form:

```
⟨proof-step,
sub-proofs,
proof-result⟩
```

The proof-step is a list of labels, pieces of formulae etc that characterize the proof rule applied in this proof step and the various formulae involved in this application. sub-proofs is the finite sequence of subproofs that are spawned in this proof step and proof-result is the final result of the proof.

For example, the proof object

```
⟨⟨"prove equivalence", "1", "1.RC", A, "1.LA", A∧B∧C,
  "1.LC", A∧B∧C, "1.RA", A⟩, ⟨⟨"assumption is conjunction",
   "1.LA" , ⟨⟨"1.LA.1", A⟩, ⟨"1.LA.2", B⟩, ⟨"1.LA.3", C⟩⟩⟩,
      ⟨⟨"conclusion is assumption",
        "1.RC", "1.LA.1"⟩, "proved"⟩, "proved"⟩,
     ⟨⟨"modus ponens", "1.RA", "2", "2.C", B⟩,
      ⟨⟨"prove conjunction", "1.LC",
        ⟨⟨"1.LC.1", A⟩, ⟨"1.LC.2", B⟩, ⟨"1.LC.3", C⟩⟩⟩,
       ⟨⟨⟨"conclusion is assumption", "1.LC.1", "1.RA"⟩, "proved"⟩,
        ⟨⟨"conclusion is assumption", "1.LC.2", "2.C"⟩, "proved"⟩,
        ⟨⟨"conclusion is assumption", "1.LC.3", "3"⟩, "proved"⟩⟩,
       "proved"⟩, "proved"⟩,
     "proved"⟩
```

generated in the example above consists of the proof-step

```
⟨"prove equivalence", "1", "1.RC", A, "1.LA", A∧B∧C, "1.LC",
 A∧B∧C, "1.RA", A⟩,
```

the two sub-proofs

```
⟨⟨"assumption is conjunction",
  "1.LA", ⟨⟨"1.LA.1", A⟩, ⟨"1.LA.2", B⟩, ⟨"1.LA.3", C⟩⟩⟩,
 ⟨⟨"conclusion is assumption", "1.RC", "1.LA.1"⟩, "proved"⟩,
 "proved"⟩
```

and

```
⟨⟨"modus ponens", "1.RA", "2", "2.C", B⟩, ⟨⟨"prove conjunction",
   "1.LC", ⟨⟨"1.LC.1", A⟩, ⟨"1.LC.2", B⟩, ⟨"1.LC.3", C⟩⟩⟩,
  ⟨⟨⟨"conclusion is assumption", "1.LC.1", "1.RA"⟩, "proved"⟩,
   ⟨⟨"conclusion is assumption", "1.LC.2", "2.C"⟩, "proved"⟩,
   ⟨⟨"conclusion is assumption", "1.LC.3", "3"⟩, "proved"⟩⟩,
  "proved"⟩,
 "proved"⟩,
```

and the proof-result

```
"proved" .
```

The information contained in the proof-step is the key word

```
"prove equivalence",
```

which indicates that we use the proof rule for proving equivalences in this step, the information

```
"1",
```

which indicates that we apply this proof rule to the formula with label "1", and the sequence

```
"1.RC", A, "1.LA", A∧B∧C, "1.LC", A∧B∧C, "1.RA", A
```

of new labels and formulae generated in this proof step as new assumptions and new proof goals. In other words, the information in the proof-step encodes the information that, in the current proof situation, we apply the proof rule "prove equivalence" to the formula with label "1", i.e. we first assume the left–hand side formula A ∧ B ∧ C (for which we generate the new label "1.LA") and show the right–hand side formula A (for which we generate the new label "1.RC"), then we assume the right–hand side formula A (for which we

generate the new label "1.RA") and show the left−hand side formula A∧B∧C (for which we generate the new label "1.LC"),

The proof object generated by the prover can then be used as an input to various viewers, for example, the one that produces a natural language version in nested cell representation.

- **Proof Rules**

The proof rules which we combine in our prover consists of the typical "natural deduction" proof rules described in logic textbooks together with certain additional rules, which are superfluous in principle (i.e. could be omitted without changing the derivation power of the system) but drastically shorten the proofs obtained and, in fact, reflect the way in which the author uses to arrange purely propositional logic proofs of predicate logic formulae.

Here is an example of a proof rule:

"In order to prove an equivalence

$F_1 \Leftrightarrow F_2$

under the assumptions

A

we first add F_1 to the assumptions and prove F_2, then we add F_2 to the assumptions and prove F_1."

In a version in which formulae have labels, this proof rule reads as follows:

"In order to prove an equivalence

(L) $F_1 \Leftrightarrow F_2$

with label (L) under the labeled assumptions

A

we first add

(L . LA) F_1

to the assumptions and prove

(L . RC) F_2

and then we add

(L . RA) F_2

to the assumptions and prove

(L . LC) F_1. "

Labels:

As shown in this example, in addition to the information on the formulae involved in the subproofs, we also generate labels, in a systematic way, for the subformulae involved. From a logic point of view, these labels are not essential. However, they facilitate references in the natural language versions of proofs. The labels generated are mnemonic. For example, "(L.RC)" stands for "the **R**ight−hand side of the formula with label L is the **C**onclusion to be proved".

Survey on the Proof Rules

The proof rules we use are described in the subsubsections of the subsequent sections below. The headings of these subsubsections give a short, informal, description of these rules that should be self−explanatory for the expert. The exact specification of these proof rules is defined by the very *Mathematica* rules given in the respective subsubsections. Alltogether, we use approximately 50 proof rules (which will grow in the future as we will improve the prover).

The proof rules are grouped in three groups:

□ Proof rules that handle *"trivial"* proof situations: For example, if identical assumptions occur in the knowledge base, only one copy of these assumptions is retained in the knowledge base. The most important "trivial" proof situation is the one in which the goal occurs among the assumptions, in which we report of course that the goal is "proved".

□ Proof rules that *expand compound assumptions* in the knowledge base. For example, a conjunction is split into its individual parts. Another example is the case in which the negation of a conjunction occurs in the knowledge base, in which case we expand the formula into the disjunction of the negations of the parts of the conjunction, etc.

□ Proof rules that *reduce the given proof situation* to one or more other proof situations with either stronger assumptions or weaker goals. For example, by the deduction rule, a proof situation whose goal is an implication is handled by adjoining the premise of the implication to the assumptions and proving the conclusion of the implication.

The Sequence in Which the Proof Rules are Applied:

By a general convention of *Mathematica*, the proof rules are applied in the order given in the respective sections below. This sequence is important. A different sequence may render the prover incomplete, or produce unnecessary lengthy proofs. Roughly, we apply the "trivial" proof rules before the proof rules that expand the assumptions and finally we apply the proof rules that reduce the given proof situation.

The Default Case:

If we reach a proof situation is reached in which none of the proof rules can be applied then we produce the information "not proved" for this proof situation. From there, the prover systematically reports back the failure of the proof so that the user obtains a complete picture about where the proof failed.

Programming the Proof Rules in *Mathematica*

As mentioned above, we do not bother to specify the proof rules used in the usual "sequent style" used in logic text books. Rather, we specify them directly as *Mathematica* rules. With some basic knowledge of (some small part of) the *Mathematica* language, it should be possible to read and understand these rules and, more importantly, in the *Mathematica* form, these rules are directly executable.

We discuss one example rule in more detail. After this discussion, the reader should be able to read and understand all the other rules specificied in the respective subsubsections below.

The proof rule for proving equivalences specified as a *Mathematica* rule:

```
PND[⟨label_, formula1_ ⇔ formula2_⟩,
⟨labeledFormulae___⟩] :=
Block[⟨proof1, proof2, proofValue⟩,
proof1 = PND[
⟨label <> ".RC", formula2⟩,
⟨⟨label <> ".LA", formula1⟩, labeledFormulae⟩];
proof1 = PND[
⟨label <> ".LC", formula1⟩,
⟨⟨label <> ".RA", formula2⟩, labeledFormulae⟩];
proofValue = CombinedValue[⟨proof1, proof2⟩];
⟨⟨"prove equivalence", label,
label <> ".RC", formula2,
label <> ".LA", formula1,
label <> ".LC", formula1,
label <> ".RA", formula2⟩,
proof1, proof2, proofValue⟩
]
```

Explanations: "PND" is the internal name of our propositional prover. This Mathematica rule says the following: For proving a labeled formula (a "goal") of the form

```
⟨label_, formula1_ ⇔ formula2_⟩
```

consisting of the label

```
label_
```

and the equivalence

```
formula1_ ⇔ formula2_
```

given the list

```
⟨labeledFormulae___⟩
```

as assumptions (as "knowledge base") we do the following steps:

1) We generate the partial proof

```
proof1
```

by recursively calling the prover "PND" for the goal

```
⟨label <> ".RC", formula2⟩
```

under the assumptions

```
⟨⟨label <> ".LA", formula1⟩, labeledFormulae⟩.
```

(Concatenation of strings is done by the operation "<>" in *Mathematica*.)

2) We generate the partial proof

```
proof2
```

by recursively calling the prover "PND" for the goal

```
⟨label <> ".LC", formula1⟩
```

under the assumptions

```
⟨⟨label <> ".RA", formula2⟩, labeledFormulae⟩.
```

(Concatenation of strings is done by the operation "<>" in *Mathematica*.)

3) We combine the proof values generated by proof1 and proof2, respectively. The possible proof values generated by a proof are "proved" and "not proved". (In rare cases, when the prover detects that, actually, the goal is not a propositional consequence of the knowledge base, we will generate the proof value "false". In fact, since the propositional prover is complete, generation of the proof value "not proved" already indicates that the goal is not a propositional consequence of the knowledge base. We will discuss this subtle point in more detail elsewhere.) The subroutine "CombinedValue" generates the proof value "proved" iff both argument proofs generate the proof value "proved". More details are given in the subsection on auxiliary programs.

4) Finally, we produce the compound proof object

```
⟨⟨"prove equivalence", label,
    label <> ".RC", formula2,
    label <> ".LA", formula1,
    label <> ".LC", formula1,
    label <> ".RA", formula2⟩,
    proof1, proof2, proofValue⟩
```

as the result of the application of the proof rule. This proof object consists of the description

```
⟨"prove equivalence", label,
    label <> ".RC", formula2,
    label <> ".LA", formula1,
    label <> ".LC", formula1,
    label <> ".RA", formula2⟩,
```

of the proof step (i.e. a list containing all the relevant information that characterizes the present proof step), the subproof

```
proof1,
```

the subproof

```
proof2,
```

and the final proof value

```
proofValue.
```

■ Auxiliary Programs

The purpose of the auxiliary programs is to handle various operations which are not directly corresponding to deduction rules. Most of the auxiliary programs are grouped in the notebook "Prerequisites.nb" because they are used by several provers. In this notebook are included only the programs which are specific to the predicate logic prover.

■ Prover Calls

The name of the prover is "Proof". It calls the function "PND", which is the actual call of the proof rules. PND calls itself recursively in dependence on the goal formula and assumption formulae involved. We define various versions of "Proof" for special and general arguments. The principal arguments of the prover are the labeled formula to be proved and a list of labeled formulae that form the knowledge base.

If the knowledge base of assumed formulae is omitted, the empty knowledge base is assumed:

```
Clear[Proof];
Proof[labeledFormula_] := Proof[labeledFormula, ⟨⟩];
```

If no indication on the type of output is given then the proof generated is given in nested cells representation:

```
Proof[ labeledFormula_, labeledFormulaeList_] :=
Proof[ labeledFormula, labeledFormulaeList,
"nested cells"];
```

The nested cells presentation of the proof is produced by rendering the notebook expression containing the nested cells:

```
Proof[ labeledFormula_,
labeledFormulaeList_,
"nested cells"] :=
NotebookPut[ Proof[ labeledFormula,
 labeledFormulaeList, "nested cells notebook"]];
```

- **Proof Step**

In order to have more flexible control over the prover, the implementation of the PND is slightly different from the one described in the previous section, subsection "Programming the Proof Rules in *Mathematica*". Namely, "`PND`" on the right–hand side is replaced by "`ProofStep`". For the purpose of the predicate logic prover this function is defined as "`PND`" again, however other provers (e.g. the functor prover) may redefine "`ProofStep`" in a different way.

```
Clear[ProofStep];
ProofStep[x___] := PND[x];
```

- **Display Initial Proof Situation**
- **Initial Proof Step: Natural Deduction**
- **Clear PND**

■ Additional Facts

Besides the proof situation (goal, assumption list), it is necessary to record some additional information about the history of the proof. For instance, if an universally quantified formula is specialized using a certain substitution, then this information is recorded as an additional fact in order to avoid the repeated generation of the same formula.

In the current version the database AF of additional facts is stored as the last element in the assumption list.

This solution has the advantage that AF has to be mentioned only in the rules which actually use it. However, the rules which do not use it have to keep the last element of the assumption list unchanged.

- **Creating AF**
- **Instantiation/Specialization**
- **Back Chaining Using Universal Formulae**

■ Literal Lists

In order to make use of quantified formulae, they have to be instantiated/specialized – then they can be treated by the propositional inference mechanism. We instantiate a quantified formula such that one literal matches an [opposite] ground literal which is already in the assumption list or in the goal. In order to make this in a systematic manner, we create for an universal formula a list of signed literals

- **Signed Literal List**

■ Tracing

For debugging/testing purposes, it is possible to activate/disactivate tracing of some operations of the prover.

- Intermediate Goals
- Matching of Quantified Formulae and New Formulae

Proof Rules for the Trivial Cases

Trivial Terminal Cases

- Conclusion is Assumption: Formula Proved
- Assumption is Instance of Existential Conclusion: Formula Proved
- Conclusion is Instance of Universal Assumption: Formula Proved
- Conclusion Subsumed by Assumption: Formula Proved
- Contradictory Assumptions: Formula Proved
- Contradiction to Universal Assumption: Formula Proved
- Assumption is Empty Disjunction: Formula Proved
- Prove Empty Conjunction: Formula Proved

Trivial Simplification Cases

- Prove Conjunction With Identical Operands: Delete One
- Prove Singleton Conjunction: Delete Conjunction Operator
- Identical Assumptions: Delete One
- Assumption Subsumes Assumption: Delete Second One
- Assumption is RHS of Implication: Delete Implication
- Assumption is Negated LHS of Implication: Delete Implication
- Assumption is Disjunction With Identical Operands: Delete One
- Assumption is Right−to−Left Implication: Reverse
- Assumption is Universal Right−to−Left Implication: Reverse
- Assumption is Tautological Implication: Remove
- Assumption is Universal Implication with Identical Operands: Remove
- Assumption is Equivalence with Identical Operands: Remove
- Assumption is Universal Equivalence with Identical Operands: Remove
- Assumption is Singleton Disjunction: Delete Disjunction Operator
- Prove Singleton Disjunctions: Remove Disjunction Operator

Handling Truth Constants

- Assumption False: Formula Proved
- Assumption (Not True): Formula Proved
- Conclusion True: Formula Proved

- **Assumption True, Conclusion False: Formula False**
- **Conclusion False Without Assumptions: Formula False**
- **Assumption True: Delete**
- **Assumption is (Not False): Delete**

■ Proof Rules for the Expansion of Compound Assumptions

■ Simplification Proof Rules for Assumption

- **Assumption Is Double Negation: Delete Double Negation**
- **Assumption Is Universal Double Negation: Delete Double Negation**
- **Assumption Is Negated Conjuction: Assume Disjunction of Negations**
- **Assumption Is Universal Negated Conjuction: Assume Universal Disjunction of Negations**
- **Assumption Is Negated Disjunction: Assume Conjunction of Negations**
- **Assumption Is Universal Negated Disjunction: Assume Conjunction of Universal Negations**
- **Assumption Is Negated Implication: Assume Premise and Negated Conclusion**
- **Assumption Is Universal Negated Implication: Assume Universal Premise and Universal Negated Conclusion**
- **Assumption is Negated Equivalence: Assume Transformed Equivalence**
- **Assumption is Universal Negated Equivalence: Assume Universal Transformed Equivalence**
- **Assumption Is Negated Universal Formula: Assume Negated Constant Fromula**
- **Assumption Is Negated Existential Formula: Assume Universal Negation**
- **Assumption is Conjunction: Assume Individual Parts**
- **Assumption is Universal Conjunction: Assume Individual Parts**

■ Resolution and Modus Ponens

- **Simplify Disjunction**
- **Assumption is Disjunction With a Literal Complimentary to an Assumption: Resolution**
- **Assumption is Implication With Assumed Hypothesis: Apply Modus Ponens**
- **Assumption is Equivalence With Assumed Operand: Apply Modus Ponens**
- **Assumption is Implication With Assumed Negated Conclusion: Apply Inverse Modus Ponens**
- **Assumption is Equivalence With Assumed Negated Operand: Apply Inverse Modus Ponens**

■ Assumption is Existential Formula: Skolem Constant

■ Proof Rules for the Reduction of Proof Situations

■ Proof Rules for Compound Conclusions

- **Prove Negation: Proof by Contradiction**

- **Prove Conjunction: Prove Each Part Individually**
- **Prove Disjunction: Assume Negation of All Other Alternatives**
- **Prove Existential Disjunction: Transform Conclusion into Disjunction of Existential Formulae**
- **Prove Implication: Apply Deduction Rule**
- **Prove Right−to−Left Implication: Reverse**
- **Prove Existential Implication: Transform Conclusion into Implication**
- **Prove Equivalences: From Left to Right and Vice Versa**
- **Prove Existential Equivalence: Transform Conclusion into Implication**
- **Prove Universal Formula: Arbitrary But Fixed**

■ Case Distinction

- **Assumption is Disjunction: Case Distinction**

■ Back Chaining

If the conclusion of an implication is identical to the goal, then we try first to prove the right−hand side of the implication. If the proof fails, then the implication is transformed into disjunction. A similar scheme is used for equivalence.
Moreover, a similar principle is used even if the RHS is not the goal, this leads to a more natural use of the assumed implications and equivalences.

However, the back−chaining rules should be used with care in the case of predicate logic, because they could generate infinite loops.

- **Assume Implication, Remaining Case: Back Chaining**
- **Assume Equivalence, Remaining Case: Back Chaining**
- **Assume Implication, Remaining Case: Prove sub−goal**
- **Assume Equivalence, Remaining Case: Prove sub−goal**

■ Transform Implication and Equivalence into Disjunction

Note: These rules will be applied only when the corresponding back−chaining rules are disactivated!

At this point, the "natural style" use of implications is exausted, i.e. using modus ponens, inverse modus ponens and back−chaining we are not able to produce a proof. Therefore, we transform now an implication into a disjunction, which will enable the resolution mechanism and the proof by cases. This is less natural, but insures success when the natural style fails.

It is important that this rule is applied after the rule "Assume Implication, Remaining Case: Prove sub−goal". This is insured by using "goal_" instead of "⟨label_, formula_⟩".

- **Implication**
- **Equivalence**

■ Specialization

- **Prove Existential Formula: Match with Goal**
- **Assume Universal Formula: Match with Goal**

- **Assume Universal Formula: Match with Ground Literal**

■ Default Case/Special Prover

This is the last rule of PND, hence it will be called if the proof fails. In order to facilitate the interaction with other provers, a "SpecialProver" is called. If the special prover returns "not proved", **PND** might discard the special proof and try an alternative proof (e.g. in the case of back–tracking – see subsection "Back–chaining" above).

- **Default Case**
- **Special Prover: Not Proved**

For the purpose of the predicate logic prover, it just returns "not proved".

```
SpecialProver[labeledFormula_, labeledFormulaeList_] :=
 ⟨⟨"default case", First[labeledFormula]⟩, "not proved"⟩;
```

By replacing the definition of **SpecialProver** with a new one, the new special prover will be called whenever proof by **PND** fails.
The new special prover has to return a proof object having "proved" or "not proved" on the last position.

■ Proof Cells Generation

■ Systematic Test Examples

■ Demo

This section contains examples which demonstrate the capabilities of the prover.

■ Introductory Examples

- **A Simple Example**
- **Antisymmetry**

  - ```
    antiSymm =
    ⟨⟨ "antiSymm", ∀⟨x, y⟩ x ≺ y ⇒ not @ (y ≺ x)⟩⟩;
    Proof[⟨"antiRef", ∀⟨x⟩ not @ (x ≺ x)⟩, antiSymm]
    ```

- **Distributivity of ForAll over Implication**

  - ```
    Proof[⟨"da", (∀⟨x⟩ (P[x] ⇒ Q)) ⇔ ((∃⟨x⟩ P[x]) ⇒ Q)⟩]
    ```

This proof is shown in [Buchberger et. al. *A survey of the Theorema project*, ISSAC'97].

- **Nested Quantifiers**

  - ```
    Proof[⟨"G", ∃⟨x⟩ (P[x] ∧ S[x])⟩, ⟨⟨"F1", ∀⟨x⟩ (P[x] ⇒ not@Q[x])⟩,
      ⟨"F2", ∀⟨x⟩ ((R[x] ∧ not@Q[x]) ⇒ ∃⟨y⟩ (S[y] ∧ T[x, y]))⟩,
      ⟨"F3", ∃⟨x⟩ (P[x] ∧ R[x] ∧ ∀⟨y⟩ (T[x, y] ⇒ P[y]))⟩⟩]
    ```

This proof is shown in the appendix.

■ Theorems on Implication
    ```

■ Quantified Theorems on Implication

■ Distributivity of implication

- ```
  Proof[⟨"da",
  ((∃⟨x⟩ P[x]) ⇒ Q) ⇔ (∀⟨x⟩ (P[x] ⇒ Q))⟩]
  ```

- ```
  Proof[⟨"de",
  ((∀⟨x⟩ P[x]) ⇒ Q) ⇔ (∃⟨x⟩ (P[x] ⇒ Q))⟩]
  ```

- ```
  Proof[⟨"DA", (∀⟨x,y⟩ (P[x] ⇒ Q[y])) ⇔ ((∃⟨x⟩ P[x]) ⇒ (∀⟨y⟩ Q[y]))⟩]
  ```

- ```
  Proof[⟨"DE",
    (∃⟨x,y⟩ (P[x, y] ⇒ Q[x, y])) ⇔ ((∀⟨x,y⟩ P[x, y]) ⇒ (∃⟨x,y⟩ Q[x, y]))⟩]
  ```

■ Equivalence

- ```
  Proof[⟨"E",
  (∃⟨x,y⟩ (P[x, y] ⇔ Q[x, y])) ⇔
     ((∀⟨x,y⟩ (P[x, y] ∨ Q[x, y])) ⇒ (∃⟨x,y⟩ (P[x, y] ∧ Q[x, y])))⟩]
  ```

- ```
  Proof[⟨"E", (∀⟨x⟩ (not@(P[x] ⇔ Q[x]))) ⇔
     (∀⟨x⟩ ((P[x] ∧ not@Q[x]) ∨ (not@P[x] ∧ Q[x])))⟩]
  ```

- ```
  Proof[⟨"E", (∀⟨x⟩ not@(P[x] ⇔ Q[x])) ⇔
     ((∀⟨x⟩ (P[x] ∨ Q[x])) ∧ (∀⟨x⟩ ((not@P[x]) ∨ (not@Q[x]))))⟩]
  ```

- ```
  Proof[⟨"E", (∀⟨x⟩ not@(P[x] ⇔ Q[x])) ⇔
     ((∀⟨x⟩ (P[x] ∨ Q[x])) ∧ (∀⟨x⟩ ((not@P[x]) ∨ (not@Q[x]))))⟩]
  ```

■ Propositional Logic is Boolean Algebra