

# PWSZ Tarnów

Kierunek: Informatyka

Specjalizacja: Inżynieria systemów informatycznych



Projekt z przedmiotu: Techniki kompilacji

## GUI Prover9

Wykonawcy:

Kacper Onak

Krystian Rasławski

Prowadzący:

dr inż. Radosław Klimek

## Spis treści

<b>PROVER9 – DOCUMENTATION (skrótowa dokumentacja w języku angielskim)</b> .....	3
Prover9 - introduction .....	3
FOF-Prover9.....	3
Running Prover9 .....	3
An Input File .....	4
A Basic Prover9 Command .....	4
More Than One Input File .....	4
Time Limit on the Command Line .....	4
Calling Prover9 From Another Program .....	5
Comments and Whitespace .....	5
A Simple Example .....	6
Lists of Objects .....	6
Clauses and Formulas.....	7
Terms.....	8
Clauses.....	9
Formulas.....	9
Infix, Prefix, and Postfix Declarations.....	10
Commands.....	10
Automatic Modes .....	11
Search Prep.....	11
Search Limits .....	11
Output Files .....	12
Evaluable Operations .....	12
<b>Założenia projektu.</b> .....	14
Zamiana w wejścia w języku naturalnym na wejście Prover9.....	14
<b>Informacje o aplikacji i jej wymagania.</b> .....	18
<b>Obsługa GUI</b> .....	19

# PROVER9 - DOCUMENTATION

## Prover9 - introduction

Prover9 is a resolution/paramodulation automated theorem prover for first-order and equational logic. Prover9 is a successor of the OtterProver.

Prover9 is intentionally paired with Mace4, which searches for finite models and counterexamples. Both can be run simultaneously from the same input, with Prover9 attempting to find a proof, while Mace4 attempts to find a (disproving) counter-example.

In July 2006 the LADR/Prover9/Mace4 input language made a major change (which also differentiates it from Otter). The key distinction between "clauses" and "formulas" completely disappeared; "formulas" can now have free variables; and "clauses" are now a subset of "formulas". Prover9/Mace4 also supports a "goal" type of formula, which is automatically negated for proof. Prover9 attempts to automatically generate a proof by default.

Prover9 is the successor of the Otter theorem prover.

**Otter** is an automated theorem prover developed by William McCune at Argonne National Laboratory in Illinois. Otter was the first widely distributed, high-performance theorem prover for first-order logic, and it pioneered a number of important implementation techniques. *Otter* is an acronym for *Organized Techniques for Theorem-proving and Effective Research*.

## FOF-Prover9

FOF (First-Order Formula) reduction is a method of attempting to simplify a problem by reducing it to an equivalent set of independent subproblems. A trivial example is to reduce the conjecture  $A \leftrightarrow B$  to the pair of problems  $A \rightarrow B$  and  $B \rightarrow A$ .

The problem reduction uses a miniscope method [Champeaux-miniscope] that is quite powerful in some cases, but it can easily blow up on complex formulas. Therefore, if the reduction procedure fails to terminate within a few seconds, or if the subproblems it produces are more complex than the original problem, the reduction attempt is aborted (and the user may wish to try the ordinary Prover9 program instead). If the reduction succeeds, each subproblem is given to the ordinary Prover9 search module.

Input files accepted by FOF-Prover9 are the same as those accepted by Prover9, and when FOF-Prover9 runs the search module on a subproblem, it uses all of the options given in the input file.

## Running Prover9

The standard way of running Prover9 is to (1) prepare an input file containing the logical specification of a conjecture and the search parameters, (2) issue a command that runs Prover9 on the input file and produces an output file, (3) look at the output, and (4) maybe run Prover9 again with different search parameters.

## An Input File

Here is an input file; assume it is named subset\_trans.in.

(Use a plain text editor, not a word processor, to create input files.)

```
formulas(sos).  
all x all y (subset(x,y) <-> (all z (member(z,x) -> member(z,y)))).  
end_of_list.  
  
formulas(goals).  
all x all y all z (subset(x,y) & subset(y,z) -> subset(x,z)).  
end_of_list.
```

## A Basic Prover9 Command

Here is a command to run Prover9 on the preceding file and send the output to a file called subset\_trans.out.

```
prover9 -f subset_trans.in>subset_trans.out
```

When you run the preceding command, a message like the following should appear immediately on your screen.

```
----- Proof 1 -----  
THEOREM PROVED  
----- process 3666 exit (max_proofs) -----
```

The output file subset\_trans.out should contain the proof (and a lot of other information about the job).

## More Than One Input File

The input can occur in more than one file:

```
prover9 -f subset.intrans.in>subset_trans.out3
```

All arguments after the "-f" are taken as input filenames, and there can be as many as you like. When multiple filenames are given on the command line, a list of objects (clauses, formulas, or terms) cannot be split across more than one file.

## Time Limit on the Command Line

Prover9 also accepts a time limit, in seconds, on the command line. The following command limits the job to about 10 seconds.

```
prover9 -t 10 -f subset_trans.in>subset_trans.out4
```

If "-t" and "-f" are both in the command, the "-t" must occur first.

## Calling Prover9 From Another Program

If Prover9 is called from another program (e.g., a shell script, a Perl script, or a Python script), Prover9's exit codes can tell the other program the reason Prover9 terminates. The following table shows the exit codes.

ExitCode	Reason for Termination
0 (MAX_PROOFS)	The specified number of proofs ( <b>max_proofs</b> ) was found.
1 (FATAL)	A fatal error occurred (user's syntax error or Prover9's bug).
2 (SOS_EMPTY)	Prover9 ran out of things to do (sos list exhausted).
3 (MAX_MEGS)	The <b>max_megs</b> (memory limit) parameter was exceeded.
4 (MAX_SECONDS)	The <b>max_seconds</b> parameter was exceeded.
5 (MAX_GIVEN)	The <b>max_given</b> parameter was exceeded.
6 (MAX_KEPT)	The <b>max_kept</b> parameter was exceeded.
7 (ACTION)	A Prover9 action terminated the search.
101 (SIGINT)	Prover9 received an interrupt signal.
102 (SIGSEGV)	Prover9 crashed, most probably due to a bug.

The calling program will probably want to look in Prover9's output, for example, to extract a proof.

## Comments and Whitespace

There are two kinds of comment:

- *Line comment.* If the first '%' (percent sign) on a line is not the start of a block comment ('%BEGIN'), everything from that symbol through the end of the line is ignored.
- *Block comment.* If the parser sees the string '%BEGIN', that is not in a line comment, it will ignore everything up through the next occurrence of 'END%'. Line breaks are irrelevant. If there is no 'END%', the rest of the file is ignored, without causing an error.

Comments are not echoed to the output file. Clauses can have label attributes which can serve as different kind of comment which *does* appear in the output file.

Whitespace (spaces, newlines, tabs, etc.) is optional in most places. The important exception is that whitespace is required around some operations in clauses and formulas

## A Simple Example

The most basic kind of input file consists of list of clauses named "sos" representing the negation of the conjecture, as in the following example.

```
formulas(sos).           % clauses to be placed in the sos list
-man(x) | mortal(x).
man(george).
  -mortal(george).
end_of_list.
```

The preceding example can also be stated in a more natural way by using a non-clausal formula for the man-implies-mortal rule and the goals list for the conclusion, as follows.

The traditional "all men are mortal", "Socrates is a man", prove "Socrates is mortal" can be expressed this way in Prover9:

```
formulas(assumptions).   % synonym for formulas(sos).
man(x) -> mortal(x).    % open formula with free variable x
man(george).
end_of_list.

formulas(goals).         % to be negated and placed in the sos list
mortal(george).
end_of_list.
```

```
formulas(sos).
-man(x) | mortal(x).
man(socrates).
  -mortal(socrates).
end_of_list.
```

## Lists of Objects

Lists of objects start with a type (formulas or terms) and name (sos, goals, weights, etc.), and end with end\_of\_list. The following display show an example of each type of accepted list, with one object in each list.

```
formulas(sos).           p(x).      end_of_list.    % the primary input list
formulas(assumptions).   p(x).      end_of_list.    % synonym for
formulas(sos)
formulas(goals).         p(x).      end_of_list.    % some restrictions (see
Goals)
formulas(usable).        p(x).      end_of_list.    % seldom used
formulas(demodulators).  f(x)=x.    end_of_list.    % seldom used, must be
equalities
formulas(hints).         p(x).      end_of_list.    % should be used more
often (see Hints)

list(weights).           weight(a) = 10.                end_of_list.
% seeWeighting
```

```

list(kbo_weights).      a = 3.                      end_of_list.
% seeTerm Ordering
list(actions).          given = 100 -> set(print_kept).  end_of_list.
% seeActions
list(interpretations).  interpretation(2,[],[relation(p,[1])]). end_of_list.
% seeSemantics

```

If the input contains more than one list of a particular type/name, the lists are simply concatenated by Prover9 as they are read.

## Clauses and Formulas

Here are the important points about clauses and formulas.

- Clauses are a subset of formulas. All input formulas, including clauses, appear in a list headed by formulas(list\_name).
- There is a rule for distinguishing variables from constants, because clauses and other formulas can have free variables (variables not bound by quantifiers). The default rule is that variables start with (lower case) u through z. For example, in the formula  $P(a,x)$ , the term  $a$  is a constant, and  $x$  is a variable. (See also the flag `prolog_style_variables`.)
- Free variables in clauses and formulas are assumed to be universally quantified at the outermost level.
- Prover9's inference rules operate on clauses. If non-clausal formulas are input, Prover9 immediately translates them clauses by NNF, Skolemization, and CNF conversions

Prover9 recognizes several kinds of symbol.

- An *ordinary symbol* is a (maximal) string made from the characters a-z, A-Z, 0-9, \$, and \_.
- A *special symbol* is a (maximal) string made from the *special characters*: `{+*/\^<>=~?@&|!#';}`.
- A *quoted symbol* is any string enclosed in double quotes.
- The *empty list symbol* is `[]`. This is a special case.

Operation	Default Symbol
True	\$T
False	\$F
Negation	-
Disjunction	
Conjunction	&
Implication	->
backward_implication	<-
Equivalence	<->
universal_quantification	all

existential_quantification	exists
Equality	=
negated_equality	!=
Attribute	#

To change the symbol associated with an operation, one uses the following command.

```
redeclare(operation, symbol ). % associate a different symbol with an
operation
```

### Terms

Any term can be written in prefix standard form, for example,  $f(g(x),y)$  and  $*('x),y)$ . If symbols in the term have parsing/printing properties (either built-in) or declared with the `op` command), the term can be written in infix/prefix/postfix form with assumed precedence, for example,  $x*y$ , which represents  $*('x),y)$  under the built-in parsing/printing properties.

A list notation similar to Prolog's can be used to write terms that represent lists. Note that the "cons" operator is ":", instead of "|" as in Prolog.

Term	Standard Prefix Form	What it is
[]	\$nil	the empty list
[a,b,c]	\$cons(a,\$cons(b,\$cons(c,\$nil)))	list of three objects
[a:b]	\$cons(a,b)	first, rest
[a,b:c]	\$cons(a,\$cons(b,c))	first, second, rest



## Clauses

The disjunction (OR) symbol is |, and the negation (NOT) symbol is -. The disjunction symbol has higher precedence than the equality symbol, so equations in clauses do not need parentheses. Every clause ends with a period. Examples of clauses follow (Prover9 adds some extra space when printing clauses).

```
formulas(sos).
p|-q|r.
  a=b|c!=d.
f(x)!=f(y)|x=y.
end of list.
```

## Formulas

Meaning	Connective	Example
negation	-	(-p)
disjunction		(p   q   r)
conjunction	&	(p & q & r)
implication	->	(p -> q)
backwardimplication	<-	(p <- q)
equivalence	<->	(p <-> q)
universalquantification	all	(all x all y p(x,y))
existentialquantification	exists	(exists x exists y p(x,y))

When writing formula, the built-in parsing declarations allow many parentheses to be omitted. For example, the following two formulas are really the same formula.

```
formulas(sos).
all x all y (p <-> -q | r & -s) .
  (all x (all y (p <-> ((-q) | (r & (-s)))))).
end_of_list.
```

## Infix, Prefix, and Postfix Declarations

Several symbols are understood by Prover9 as having special parsing properties that determine how terms involving those symbols can be arranged. In addition, the user can declare additional symbols to have special parsing properties.

The "op" command is used to declare parse types and precedences.

```
op(precedence, type, symbols(s) ). % declare parse type and precedence
```

- $1 \leq \textit{precedence} \leq 998$ .
- *type* is one of { infix, infix\_left, infix\_right, prefix, prefix\_paren, postfix, postfix\_paren, ordinary }.
- *symbol(s)* is either a symbol or a list of symbols. Each multi-character special symbol must be enclosed in double quotes.

The following table shows an example of each type of parsing property (and ignores precedence).

Type	Example	Standard Prefix	Comment
infix	$a*(b*c)$	$*(a,*(b,c))$	likeProlog's xfx
infix_left	$a*b*c$	$*(*(a,b),c)$	likeProlog's yfx
infix_right	$a*b*c$	$*(a,*(b,c))$	likeProlog's xfy
prefix	--p	$-(-(p))$	likeProlog's fy
prefix_paren	-(p)	$-(-(p))$	likeProlog's fx
postfix	a''	$'(a)$	likeProlog's yf
postfix_paren	(a)'	$'(a)$	likeProlog's xf
ordinary	$*(a,b)$	$*(a,b)$	takes away parsing properties

## Commands

Eleven types of command are accepted. Here is an example of each.

```
op(400, infix_right, ["+", "--"]). % declare parse precedence and type
(see Clauses and Formulas)

redeclare(negation, "~").           % change the negation symbol (see
Clauses and Formulas)

set(print_kept).                    % set a flag

clear(auto_inference).              % clear a flag
```

```

assign(max_weight, 40).           % integerparameter
assign(stats, some).             % string parameter
assoc_comm(*).                  % not currently used for Prover9
commutative(g).                  % not currently used for Prover9
predicate_order([=,<=,P,Q).      % predicate symbol precedence (see
Term Ordering)
function_order([0,1,a,b,f,g,*,+]). % function symbol precedence (see Term
Ordering)
lex([0,1,a,b,f,g,*,+]).          % synonym for "function_order"
skolem([a,b,f,g]).               % declare symbols to be Skolem
functions (rarely used)

```

### Automatic Modes

Prover9's automatic mode is set by default. Otter's automatic mode must be explicitly set.

If you simply give Prover9 a set of clauses and/or formulas, Prover9 will look at the clauses and decide which inference rules and clause-processing operations to use.

If you don't like the automatic decisions that Prover9 makes, you can clear the flag **auto** or any of the secondary auto flags that depend on it. Prover9 output files show in detail the effects of changing these flags.

### Search Prep

```

set(expand_relational_defs).
clear(expand_relational_defs). % default clear

```

If this flag is set, Prover9 looks for relational definitions in the assumptions and uses them to rewrite all occurrences of the defined relations elsewhere in the input, before the start of the search. The expansion steps are detailed in the output file and appear in proofs with the justification `expand_def`.

Relational definitions must be closed formulas for example,

```

formulas(assumptions).
all x all y all z (A(x,y,z) <-> ((x <= y & y <= z) | (z <= y & y <= x))).
end_of_list.

```

### Search Limits

```

assign(sos_limit, n). % default n=20000, range [-1 ..INT_MAX]

```

This parameter imposes a limit on the size of the sos list ( $n=-1$  means there is no limit). It also activates a method for deleting clauses (in addition to, and after, application of the **max\_weight** parameter).

This is a little bit tricky (and sometimes too clever for its own good). When the sos is half full, it starts being selective about keeping clauses, and as it fills up, it gradually becomes more selective. When it is full, it is very selective about keeping clauses. (The method is not applied to clauses that match hints.) When it decides to keep a clause, and the sos is already full, the "worst" clause in sos is deleted to make room for the new clause.

## Output Files

Even when Prover9 fails to find a proof, its output file usually has lots of valuable information about the search. The output file can suggest many ways of improving the search for subsequent jobs as in the following examples.

- The output shows how equalities are oriented; different term ordering parameters may give better or more intuitive orientations.
- If Prover9 focused the search on uninteresting clauses (see the sequence of given clauses), different inference rules, a different **pick\_given\_ratio**, or a specialized weighting function can be used.
- If Prover9 ran out of time or memory with a huge sos list and small usable list (i.e., few given clauses were used), the **sos\_limit** should be reduced.

## Evaluable Operations

### Functions on Integers

Operation	Comment
+	integer sum
-	integer negation (unary only) (used also for Boolean negation)
/	integerdivision
mod	modulus
Abs	integerabsolutevalue
Min	integer minimum (binary)
Max	integer maximum (binary)

### Relations on Integers

Operation	Comment
<	lessthan
<=	lessorequal
>	greaterthan
>=	greaterorequal
==	equal (used also for non-integers)
!=	not equal (used also for non-integers)

### Properties and Relations on Terms

Operation	Comment
Variable	is the term a variable?
Constant	is the term a constant (includes integers)?
Grodnu	is the term variable-free?
@<	lexically less than
@<=	lexically less orequal
@>	lexicallygreaterthan
@>=	lexicallygreaterorequal
==	identical (used also for integers)
!=	not identical (used also for integers)

### Logic Operations

Operation	Comment
&	Conjunction
&&	conjunction*
	Disjunction
	disjunction*
-	negation (used also for integers)

### ConditionalExpressions

Operation	Comment
If	if(condition, then_part, else_part)

## Założenia projektu.

### Zamiana w wejścia w języku naturalnym na wejście Prover9

GUI do Prover9 będzie umożliwiało wprowadzanie wyrażeń w języku naturalnym (polskim) przy zachowaniu pewnych zasad przedstawionych w tabeli poniżej. Wprowadzane wyrażenia będą konwertowane na wejście dopuszczalne przez Prover9. Możliwe będzie także podanie wejścia w formacie przyjmowanym przez Prover9 (bez konwersji).

Oryginalne wejście	Formuły	Wejście w języku naturalnym
<code>formulas(x)</code>		wyrażenia ( x ) / wzory ( x )
<code>formulas(goals)</code>		wyrażenia ( cele )
<code>formulas(assumption)</code>		wyrażenia ( założenia )
<code>end_of_list.</code>		koniec listy / koniec listy wyrażeń
<code>All x</code>		dla każdego x
<code>Subset(x,y)</code>		podzbiór ( x , y )
<code>Member(x,y)</code>		należy do ( x , y )
<code>x &lt;-&gt; y</code>		x jest równoważne y
<code>x -&gt; y</code>		jeżeli x to y
<code>x &amp; y</code>		x koniunkcja y
<code>x   y</code>		x alternatywa y
Formuły atomiczne		
<code>x + y</code>		x plus y
<code>x - y</code>		x minus y
<code>x * y</code>		x razy y
<code>x / y</code>		x podzielone przez y
<code>x ^ y</code>		x i y
<code>x v y</code>		x lub y
<code>x'</code>		x zanegowane
<code>¬x</code>		nie x / negacja x
<code>Exists x</code>		istnieje takie x / Istnieje x
<code>x &lt; y</code>		x mniejsze od y
<code>x &gt; y</code>		x większe od y
<code>x = y</code>		x równe y
<code>x != y</code>		x różne od y
<code>x &lt;= y</code>		x mniejsze lub równe y
<code>x &gt;= y</code>		x większe lub równe y
Pozostałe		
<code>assign(max_seconds, x).</code>		przypisz ( maksymalny czas , x )
<code>assign(max_megs, x).</code>		przypisz ( maksymalna pamięć , x )
<code>assign(max_given, x).</code>		przypisz ( maksymalnie klauzul , x )

#### Przykład działania translacji:

Wejście w języku naturalnym:

x lub (y lub z) równa się y lub (x lub z) .  
x i y równa się (xzanegowane lub yzanegowane) zanegowane.  
x lub xzanegowane równa się y lub yzanegowane .  
(x lub yzanegowane) i (x lub y) równa się x .

Wejście po konwersji:

```
x v (y v z) = y v (x v z)
x & y = (x' v y')'
x v x' = y v y'
(x v y') ^ (x v y) = x
```

Wejście w języku naturalnym:

```
Wyrażenia (założenia).

(x razy y) razy z równa się x razy (y razy z).

    istnieje e ((dla każdego x (e razy x równa się x)) i
                (dla każdego x istnieje y (y razy x równa się e))).

    istnieje a istnieje b (a razy b różne od b razy a).

Koniec listy wyrażen.
```

Wejście po konwersji:

```
formulas (assumptions).

(x * y) * z = x * (y * z).

exists e ((all x (e * x = x)) &
          (all x exists y (y * x = e))).

exists a exists b (a * b != b * a).

end_of_list.
```

**Pełne przykłady:**

1) Dowód na to, że pierwiastek z liczby 2 jest liczbą niewymierną.

Wejście w języku naturalnym:

```
wyrażenia ( założenia ) .

1 razy x równa się x .

x razy 1 równa się x .

x razy ( y razy z ) równa się ( x razy y ) razy z .

x razy y równa się y razy x .

jeżeli ( x razy y równa się x razy z ) to y równa się z .

dzielcSie ( x , y ) jest równoważne ( istnieje z x razy z równa się y ) .

jeżeli dzielcSie ( 2 , x razy y ) to ( dzielcSie ( 2 , x ) alternatywa
dzielcSie ( 2 , y ) ) .
```



```

a razy a równa się ( 2 razy ( b razy b ) ) .
jeżeli ( x różne od 1) to negacja ( dzielicSie ( x , a )
koniunkcjadzielicSie ( x , b ) ) .
2 różne od 1 . koniec listy .

```

Wejście po konwersji:

```

formulas(assumptions).
  1 * x = x.
x * 1 = x. x * (y * z) = (x * y) * z.
x * y = y * x. (x * y = x * z) => y = z.
dzielicsie(x,y) <-> ( existzx * z = y).

dzielicsie(2,x * y) -> (dzielicsie(2,x) | dzielicsie(2,y)).
a * a = (2 * (b * b)).
(x != 1) -> -(dzielicsie(x,a) & dzielicsie(x,b)).
2 != 1.

end_of_list.

```

2 ) Wszyscy ludzie są śmiertelni, Sokrates jest człowiekiem, Sokrates jest śmiertelny

Wejście w języku naturalnym:

```

wyrażenia ( założenia ) .
jeżeli człowiek ( x ) to śmiertelny ( x ) .
człowiek ( sokrates ) .
koniec listy wyrażeń .
wyrażenia ( cele ) .
śmiertelny ( sokrates ) .
koniec listy wyrażeń .

```

Wejście po konwersji:

```

formulas(assumptions).
czlowiek(x) ->smiertelny(x).
czlowiek(sokrates).
end_of_list.
formulas(goals).
smiertelny(sokrates).
end_of_list.

```

## Informacje o aplikacji i jej wymagania.

Aplikacja została napisana w języku „Python” w wersji 3.6.3, a interfejs graficzny wykonano przy użyciu modułu „tkinter”.

Aplikacja przeznaczona jest na system operacyjny Linux.

Do działania aplikacji wymagany jest zainstalowany „Python” w wersji 3.6.3 lub nowszej oraz moduł „tkinter” do Python3.

Aplikacja korzysta z systemu służącego do dowodzenia twierdzeń w logice pierwszego rzędu „Prover9”, który należy zainstalować samodzielnie.

### Instalacja wymaganych komponentów

Instalacja Python 3.6i modułu tkinter

```
sudo apt-get install python3.6  
sudo apt-get install python3-tk
```

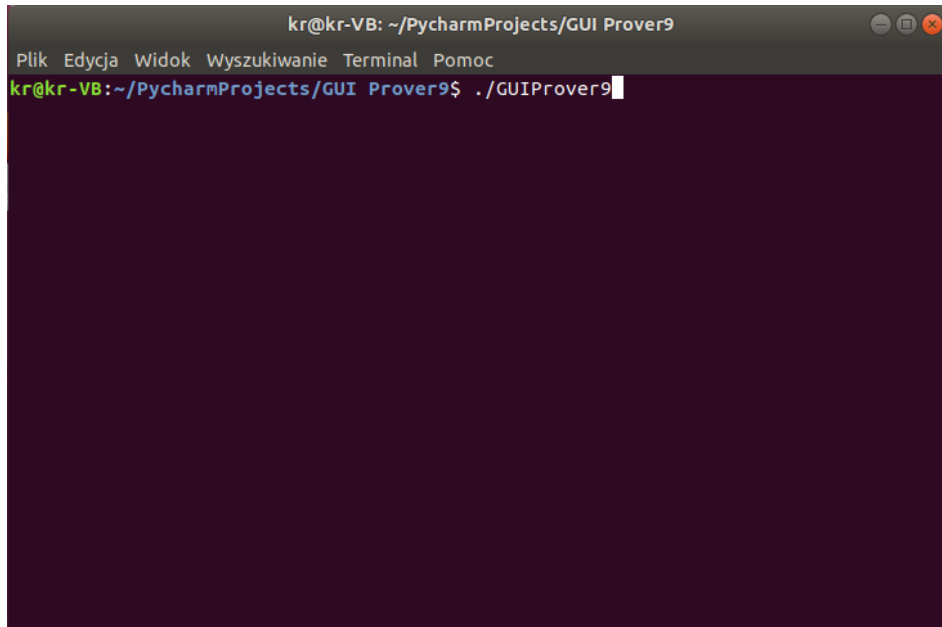
Instalacja Prover9 i niezbędnych komponentów.

```
sudo apt-get install prover 9  
sudo apt-get install laddr4-app
```

## Obsługa GUI

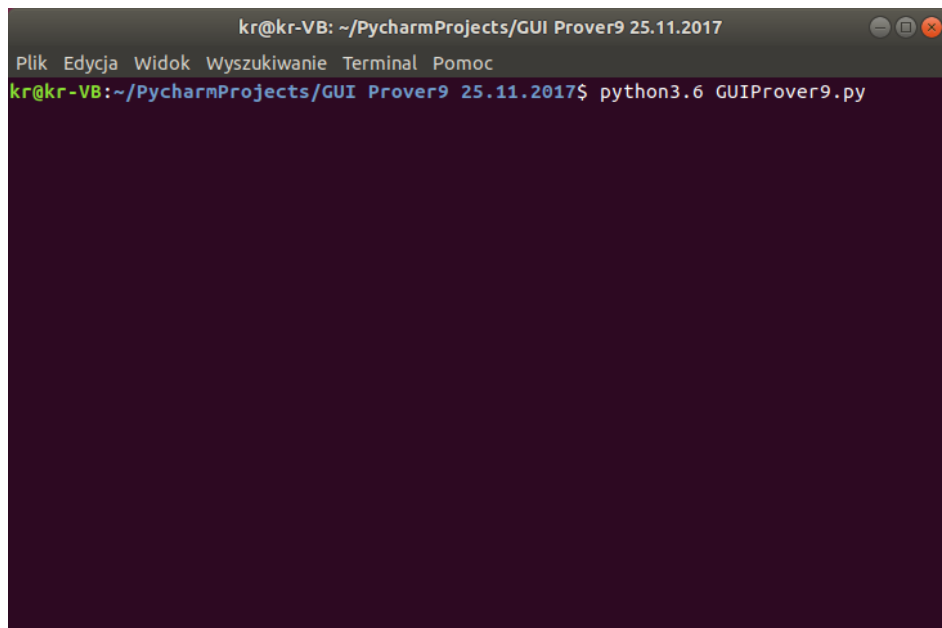
Uruchomienie GUI:

Jeżeli wcześniej nadamy prawa dla pliku (tzn. `chmod +x GUIProver9`), uruchomienie GUI wygląda następująco:



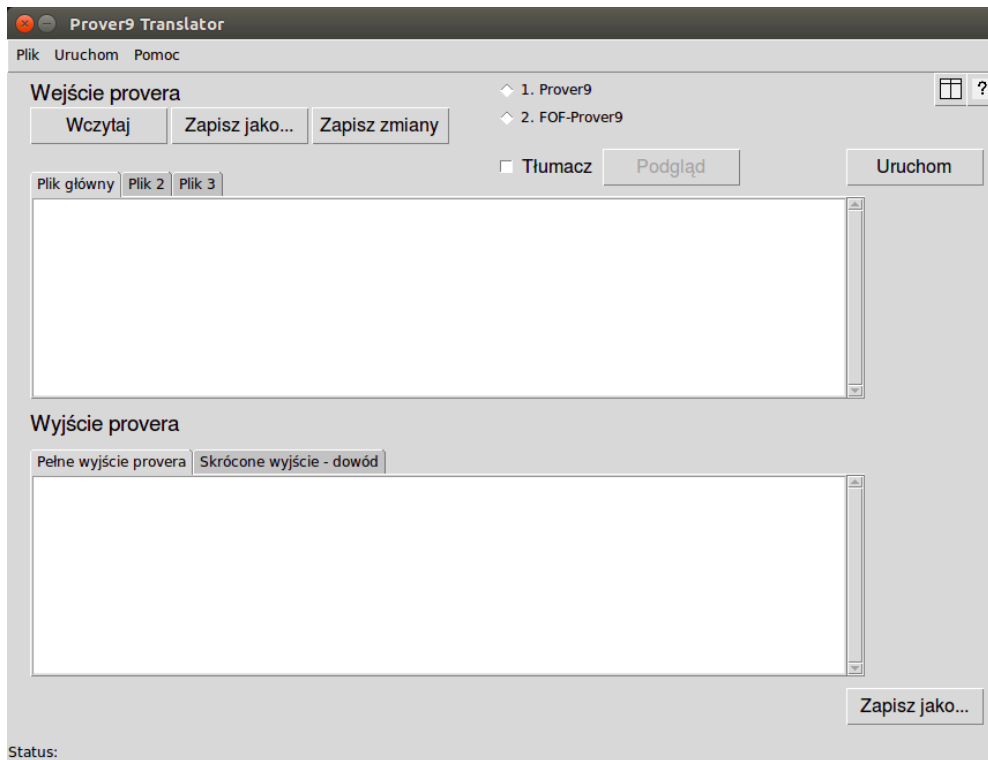
```
kr@kr-VB: ~/PycharmProjects/GUI Prover9
Plik Edycja Widok Wyszukiwanie Terminal Pomoc
kr@kr-VB:~/PycharmProjects/GUI Prover9$ ./GUIProver9
```

W innym przypadku uruchomienie wygląda tak:

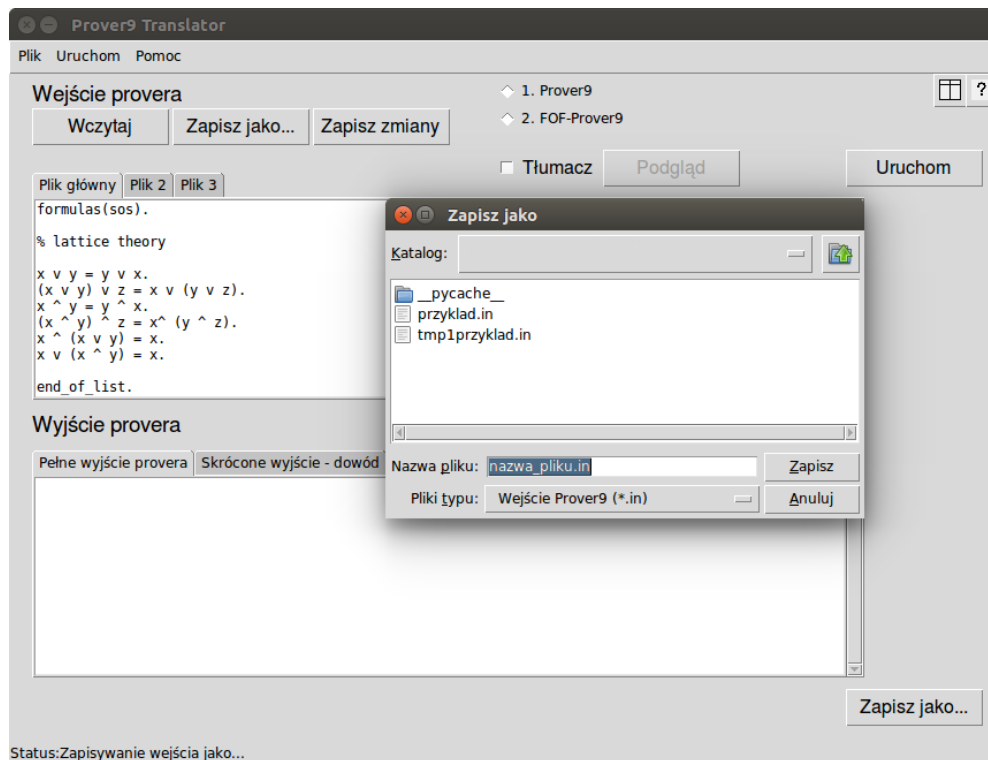


```
kr@kr-VB: ~/PycharmProjects/GUI Prover9 25.11.2017
Plik Edycja Widok Wyszukiwanie Terminal Pomoc
kr@kr-VB:~/PycharmProjects/GUI Prover9 25.11.2017$ python3.6 GUIProver9.py
```

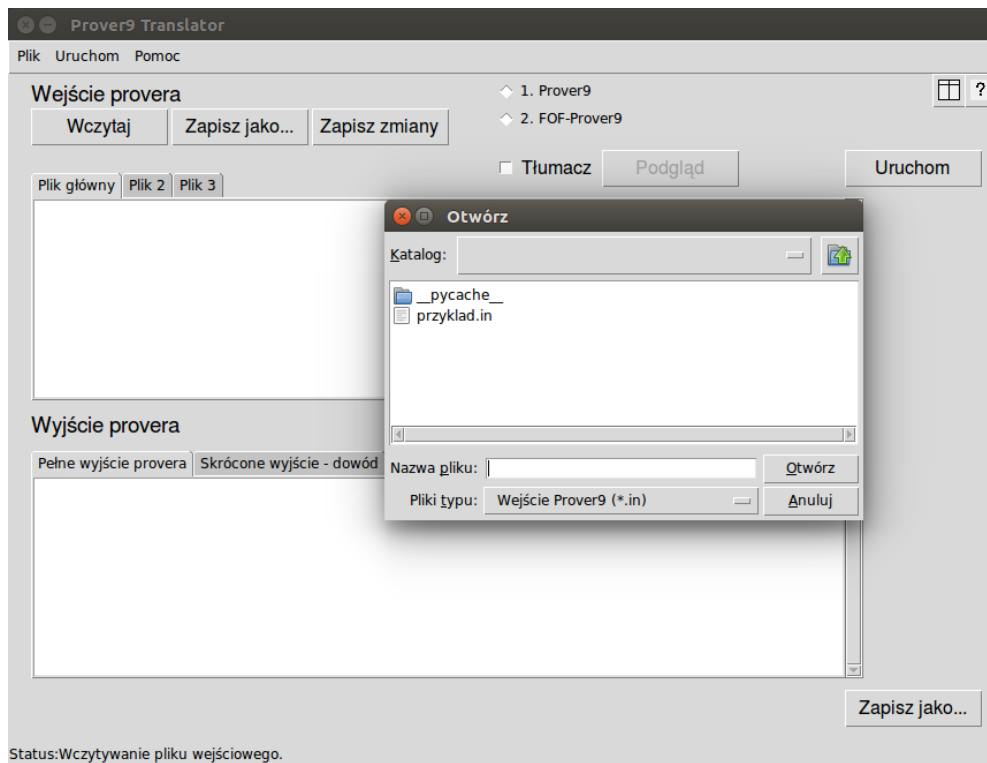
Po uruchomieniu GUI pojawia nam się takie okno:



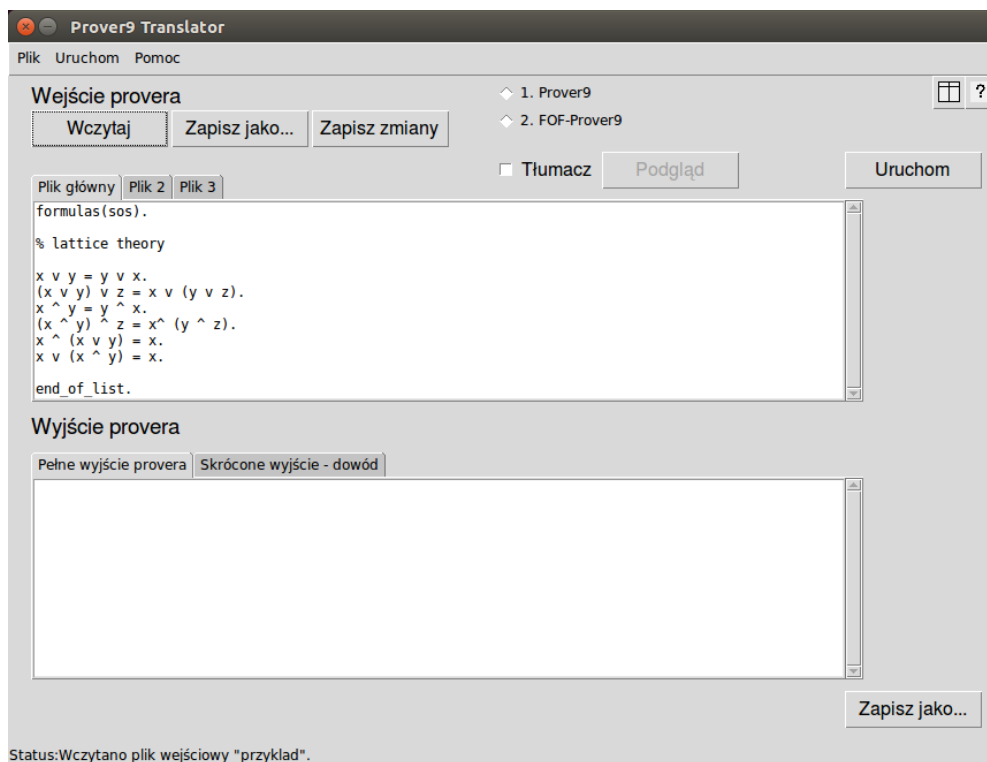
Aby wprowadzić formuły, które mamy do udowodnienia należy je wprowadzić w oknie: Wejście provera->Plik główny/Plik2/Plik3. Po wprowadzeniu wszystkich formuł można je zapisać do plików.



Możemy również wczytać plik za pomocą przycisku „Wczytaj”.



Po wczytaniu pliku w oknie pojawi nam się wczytany plik oraz otrzymamy komunikat o poprawnym wczytaniu pliku (patrz Status).

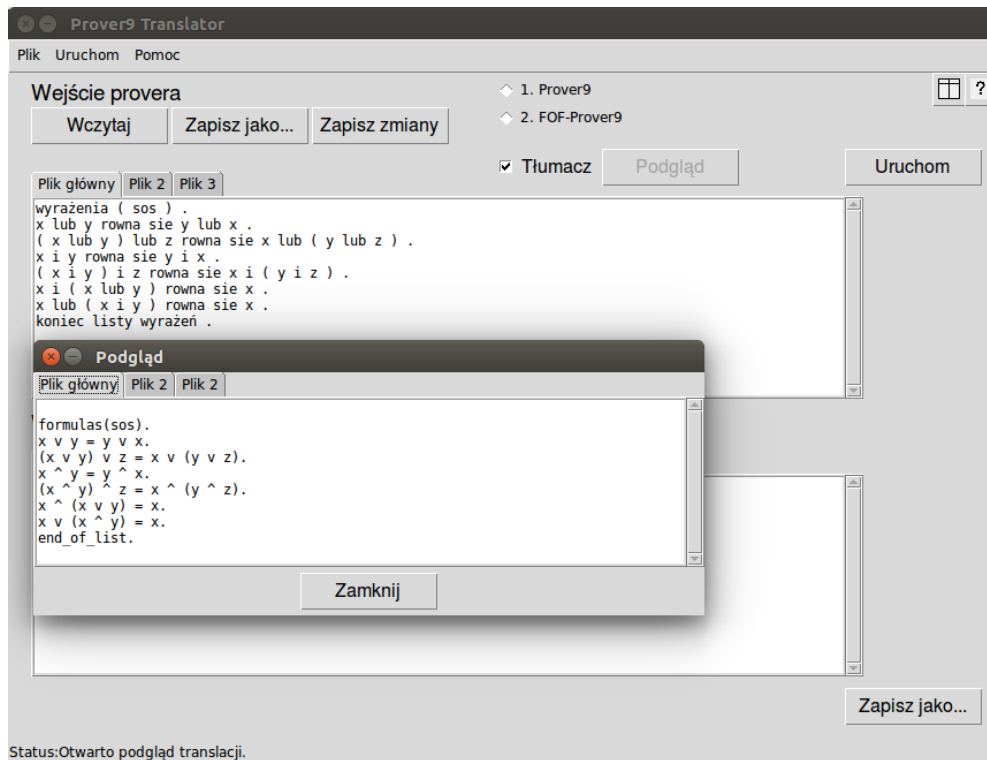


Wprowadzono również możliwość translacji formuł z języka naturalnego (polskiego) na język rozumiany przez Prover9. W tym celu należy zaznaczyć pole przy słowie „Tłumacz”. Podobnie jak punkcie 3. reguły możemy wprowadzić ręcznie lub za pomocą przycisku „Wczytaj”. Należy zapoznać się ze sposobem wprowadzania reguł w języku naturalnym. Tabela translacji oraz informacje potrzebne do zapisu reguł znajdują się w prawym górnym rogu (ikona tabeli oraz ikona pytajnika) i wyglądają one tak:

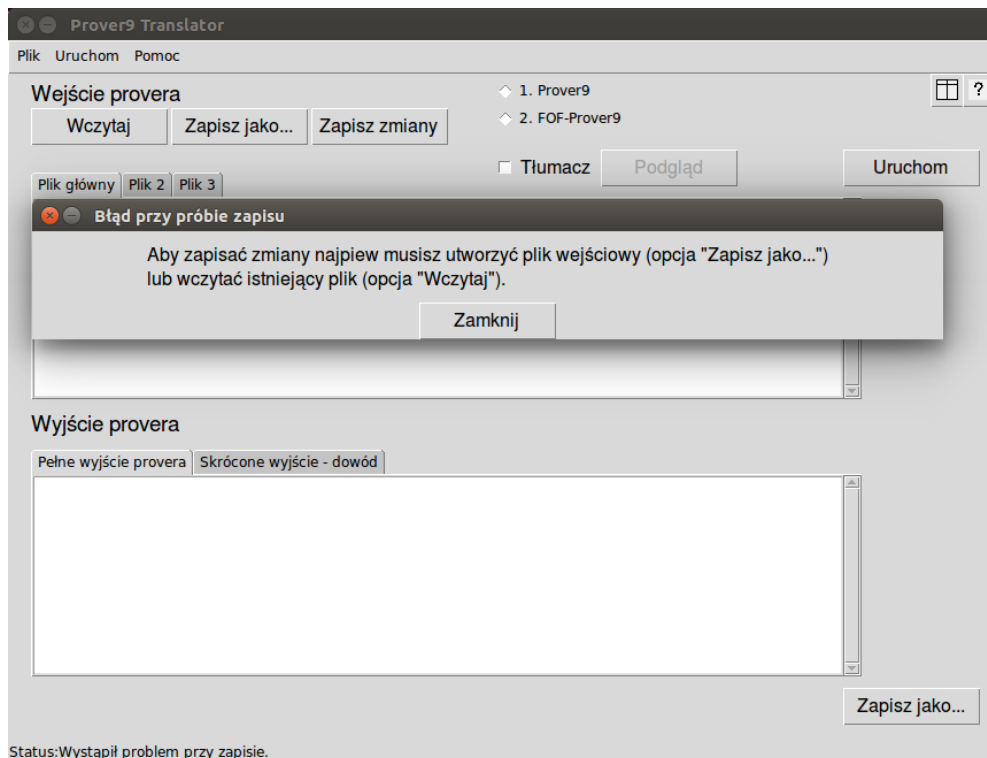
Tabela translacji	
Oryginalne wejście	Wejście w języku naturalnym
Formuły	
<code>formulas(x)</code>	wyrażenia ( x ) / wzory ( x )
<code>formulas(goals)</code>	wyrażenia ( cele )
<code>formulas(assumption)</code>	wyrażenia ( założenia )
<code>end_of_list.</code>	koniec listy / koniec listy wyrażen
<code>All x</code>	dla każdego x
<code>Subset(x,y)</code>	podzbiór ( x , y )
<code>Member(x,y)</code>	należy do ( x , y )
<code>x &lt;-&gt; y</code>	x jest równoważne y
<code>x -&gt; y</code>	jeżeli x to y
<code>x &amp; y</code>	x koniunkcja y
<code>x   y</code>	x alternatywa y
Formuły atomiczne	
<code>x + y</code>	x plus y
<code>x - y</code>	x minus y
<code>x * y</code>	x razy y
<code>x / y</code>	x podzielone przez y
<code>x ^ y</code>	x i y
<code>x v y</code>	x lub y
<code>x'</code>	x zanegowane
<code>¬x</code>	nie x / negacja x
<code>Exists x</code>	istnieje takie x / Istnieje x
<code>x &lt; y</code>	x mniejsze od y
<code>x &gt; y</code>	x większe od y
<code>x = y</code>	x równe y
<code>x != y</code>	x różne od y
<code>x &lt;= y</code>	x mniejsze lub równe y
<code>x &gt;= y</code>	x większe lub równe y
Pozostałe	
<code>assign(max_seconds, x).</code>	przypisz ( maksymalny czas , x )
<code>assign(max_megs, x).</code>	przypisz ( maksymalna pamięć , x )
<code>assign(max_given, x).</code>	przypisz ( maksymalnie klauzul , x )
Zamknij	

Podstawowe informacje
<p>Korzystając z translacji należy trzymać się niżej wymienionych reguł</p> <ul style="list-style-type: none"> <li>*Każda linia powinna być zakończona spacją i kropką</li> <li>*Po każdym nawiasie (otwierającym i zamykającym) należy użyć spacji np.: ( x lub y )</li> <li>*Wyrażenia opisujące założenia muszą znajdować się w wyrażenia ( założenia )</li> <li>*Wyrażenia opisujące cele powinny znajdować się w wyrażenia ( cele )</li> <li>*Tekst wprowadzany w języku naturalnym może, ale nie musi zawierać polskich znaków.</li> </ul> <p>Wymagana jest konsekwencja w pisowni - słowa zawierające kilka polskich znaków muszą być w całości napisane z nimi, lub w całości napisane bez nich - błędne wprowadzenie: x równoważne y</p>
Zamknij

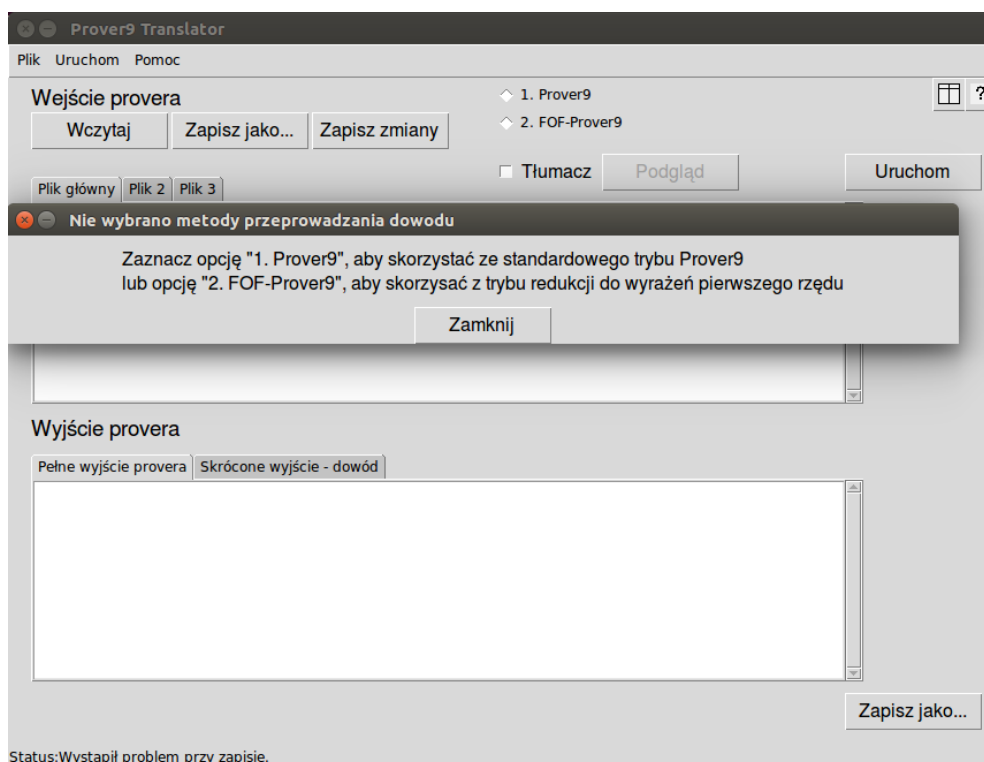
W celu sprawdzenia jak zostanie wykonane tłumaczenie na język rozumiany przez Prover9 należy kliknąć przycisk „Podgląd”.



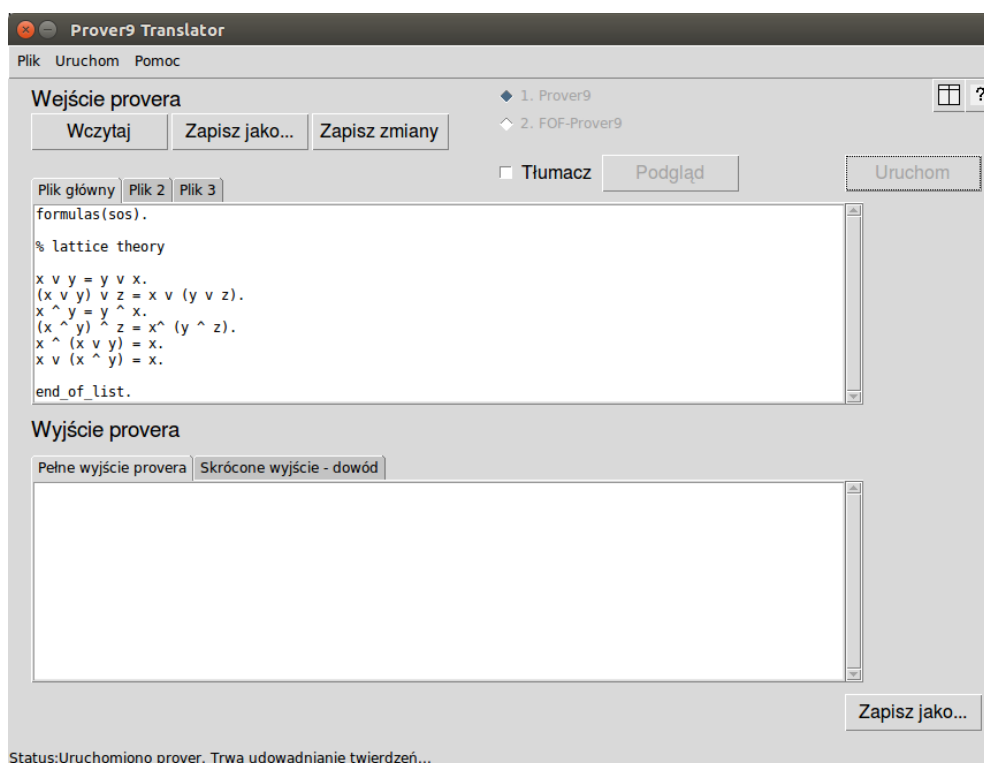
Po utworzeniu pliku wejściowego przyciskiem „Zapisz jako...” użytkownik w szybki sposób może zapisać wprowadzone zamiany w plikach poprzez przycisk „Zapisz zmiany”.



Przed uruchomieniem Prover9 należy wybrać metodę przeprowadzenia dowodu (poszczególne metody zostały opisane przycisku w prawym górnym rogu (ikona pytajnika)). Jeżeli nie zostanie wybrana metoda pojawi się komunikat.

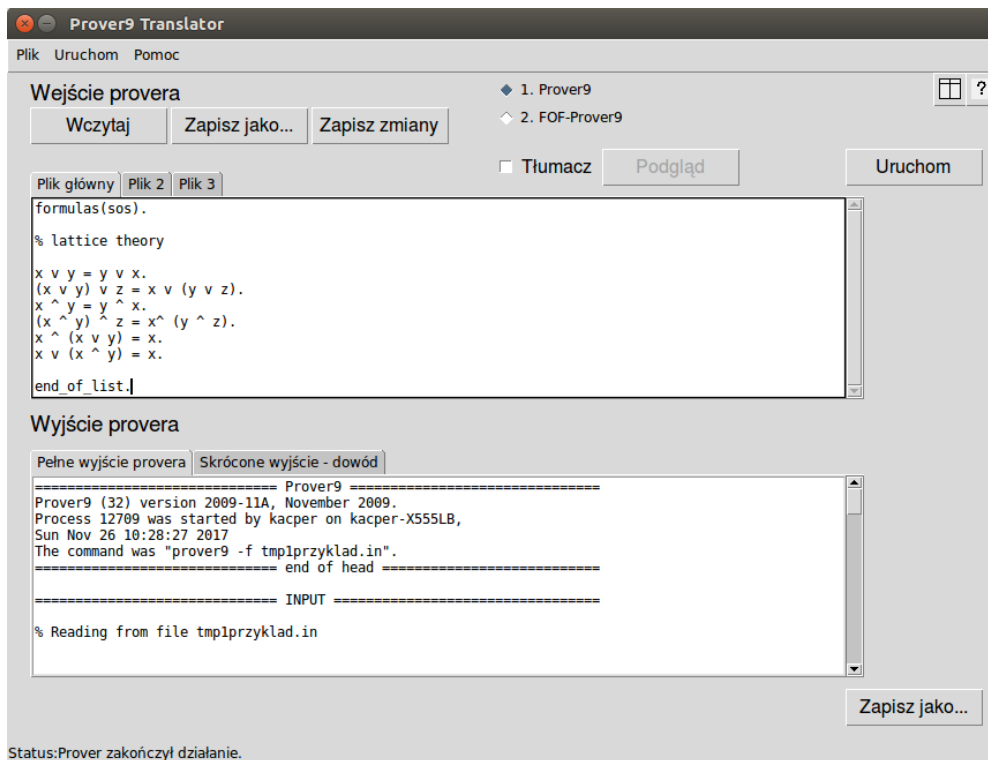


Tak wygląda GUI podczas pracy Prover9:





Po udowodnieniu twierdzeń otrzymamy odpowiedź, którą Prover9 wyprowadził. Została również dodana skrócona wersja wyjścia Prover9 w celu łatwiejszej analizy, wystarczy przejść do okna Skrócone wyjście – dowód. Zostanie również zmieniony aktualny status.



Została dodana możliwość zapisania wyjścia Prover9 do pliku.

