CHAPTER 1

A GUIDED TOUR OF THE PROVERBOX

This document provides an overview of the ProverBox in the form of a guided tour, in order to make the reader familiar with the most important features and the basics of the user interface before delving into the more detailed and more technical subsequent chapters. Topics covered in the tour include installation, manipulating formulas, using modules and proving theorems.

The interested reader is encouraged to obtain the application as described in the following section and actually follow the tour in the running program.

1.1 Obtaining and Running the ProverBox

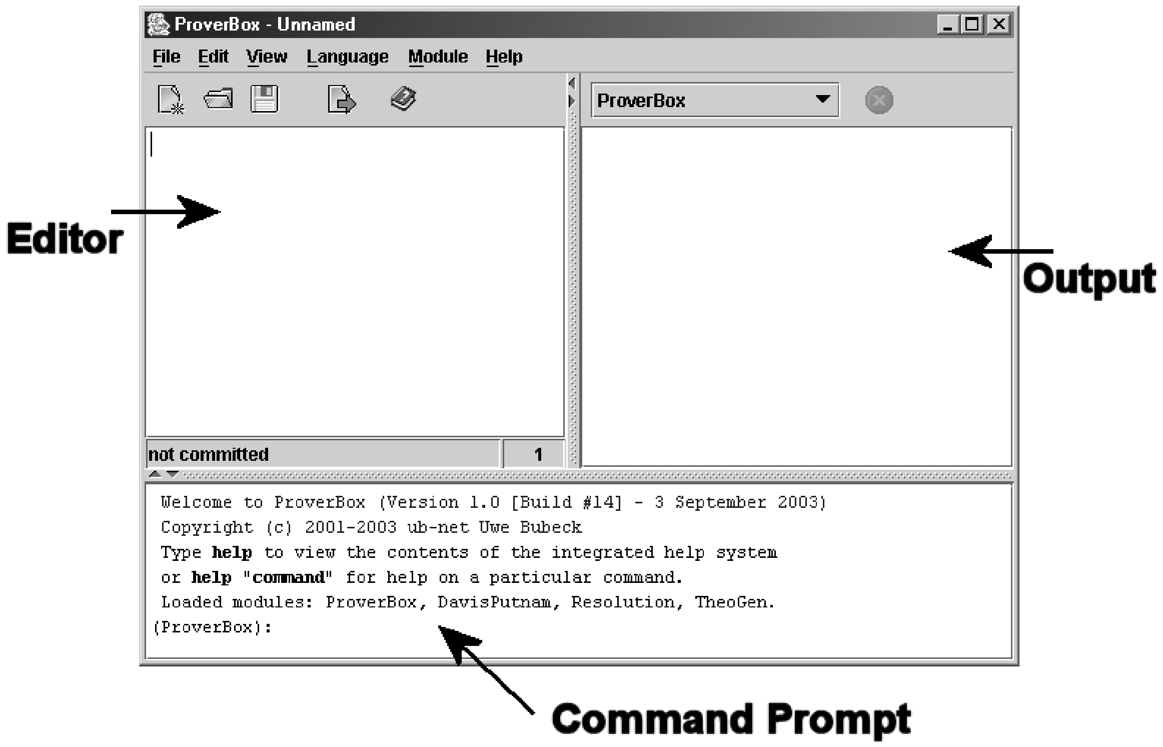
The ProverBox is a pure Java application designed for JDK 5.0 or higher. It is deployed as a self-containing JAR archive and is available from the ProverBox website [1]. A free registration is required to obtain a registration key which the software will be asking for at startup.

The JAR archive should be downloaded into a new directory, and the registration key should be placed into the same directory. The software is then executed with the command `java -jar proverbox.jar`. On many platforms, it is also possible to run the program by (double-)clicking on the archive icon.

Also available on the ProverBox website are some sample specification files to be downloaded separately. Most of these are converted problems from the TPTP Problem Library [2].

1.2 The Workplace

After a short initialization phase, which is visualized by a splash screen, the empty ProverBox workplace awaits the user as shown in Figure 1.1.



```
Welcome to ProverBox (Version 1.0 [Build #14] - 3 September 2003)
Copyright (c) 2001-2003 ub-net Uwe Bubeck
Type help to view the contents of the integrated help system
or help "command" for help on a particular command.
Loaded modules: ProverBox, DavisPutnam, Resolution, TheoGen.
(ProverBox):
```

Figure 1.1: Empty Workplace

On the bottom is the command prompt, above which are the editor (on the left) and the output area (on the right). The user can resize or hide frames to customize the workplace, e.g. minimize the editor when it is not needed.

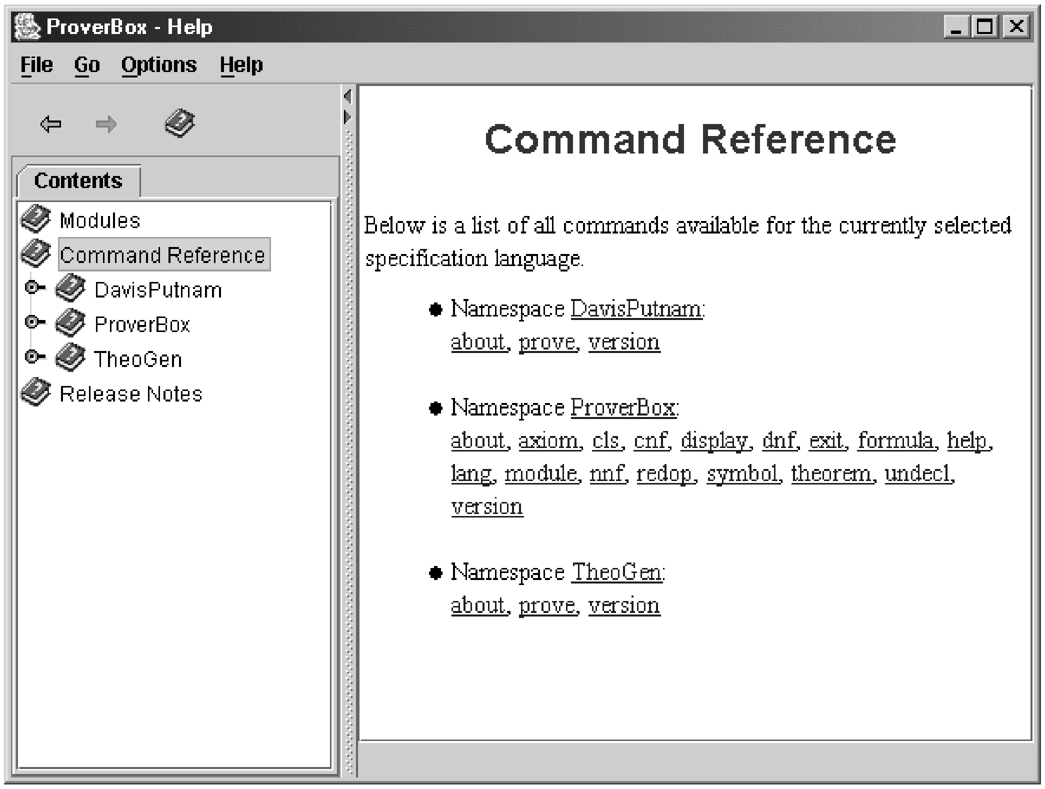
1.3 The Integrated Help System

The ProverBox comes with a multitude of commands to work with formulas or theory specifications, and additional commands are provided by installed prover modules. One of the most important features not to get lost is the integrated help system.

Let us therefore have a short glance at the online help before delving into the depths of the ProverBox. To open it up, type `help` on the command prompt or click on the help icon above the editor. You will then be taken to the list of help topics. To select a topic, just follow the blue hyperlinks as in a web browser, or use the topic tree on the left.

The most important section of the help system is the command listing. To get there, use the `Command Reference` hyperlink from the front page or click on the `Command`

`Reference` node in the tree. As shown in Figure 1.2, the commands are grouped into so-called namespaces for better clarity. Notice that you are only shown commands which are available for the currently selected language.



Command Reference

Below is a list of all commands available for the currently selected specification language.

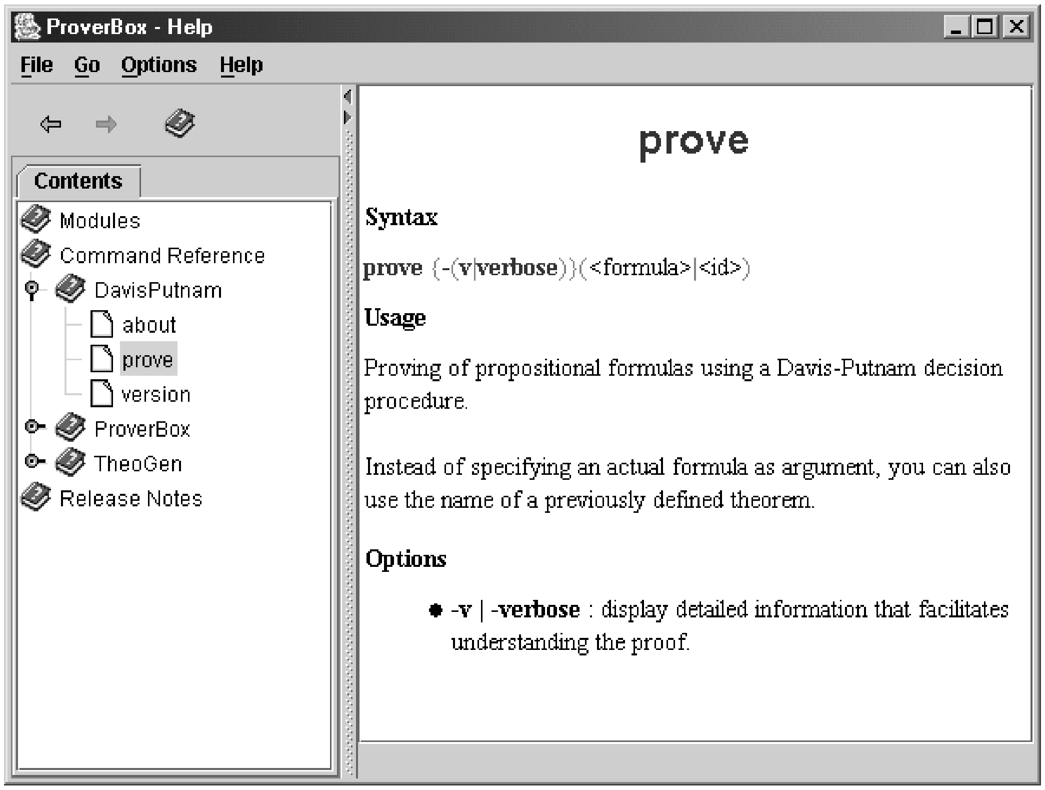
- Namespace DavisPutnam:
  about, prove, version
- Namespace ProverBox:
  about, axiom, cls, cnf, display, dnf, exit, formula, help, lang, module, nnf, redop, symbol, theorem, undecl, version
- Namespace TheoGen:
  about, prove, version

Figure 1.2: The Command Reference

For each command, there exists detailed information on how to use it, including a syntax description. The syntax reference is generated on-the-fly from the underlying grammar. Figure 1.3 shows what the command description looks like for the `prove` command of the DavisPutnam module.

1.4 Working with Formulas

It is now time to demonstrate how to work with formulas. Let us start with some easy examples on propositional formulas. Since the ProverBox is capable of working with multiple logics, we first need to make sure that propositional calculus is selected.



prove

Syntax

prove {-(**v**|**verbose**)}(<formula>|<id>)

Usage

Proving of propositional formulas using a Davis-Putnam decision procedure.

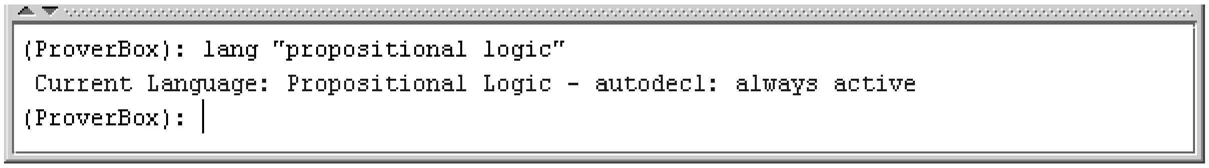
Instead of specifying an actual formula as argument, you can also use the name of a previously defined theorem.

Options

- **-v** | **-verbose** : display detailed information that facilitates understanding the proof.

Figure 1.3: Description of the `DavisPutnam.prove` Command

To do this, type `lang ‘‘propositional logic’’` on the command prompt. As seen in Figure 1.4, the program will indicate that it has successfully switched language.



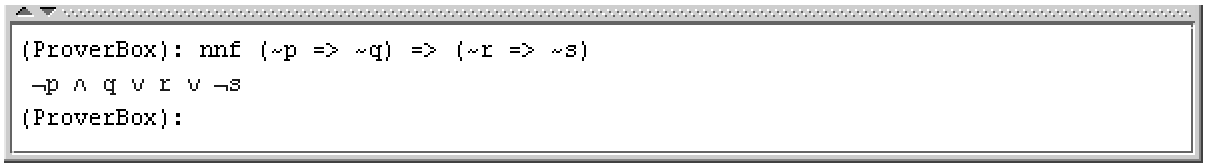
```
(ProverBox): lang "propositional logic"
 Current Language: Propositional Logic - autodecl: always active
(ProverBox): |
```

Figure 1.4: Switching Language on the Command Prompt

It is also possible to accomplish this task without command prompt by clicking on `Language` and then `Select` in the main menu, which will open a language selection dialog.

The formula we are now going to work on is $(\neg p \Rightarrow \neg q) \Rightarrow (\neg r \Rightarrow \neg s)$. ProverBox allows us to write the negation operator $\neg$ either as `not` or as `~`. For the implication $\Rightarrow$, we can write `implies` or `=>` (an equal sign followed by a greater sign).

Let us first transform the formula into Negative Normal Form (NNF). Recall that a formula is in NNF if it contains only operators from the set $\{\neg, \vee, \wedge\}$, with negations only occurring right in front of atoms. The corresponding command is `nnf` (commands are not case-sensitive), so we enter `nnf (~p => ~q) => (~r => ~s)` or `nnf (not p implies not q) implies (not r implies not s)` on the prompt. As expected, the result is $(\neg p \wedge q) \vee r \vee \neg s$, as shown in Figure 1.5. Notice that the ProverBox follows the convention that conjunction ($\wedge$) is of higher priority than disjunction ($\vee$), so the parentheses are redundant and therefore discarded.

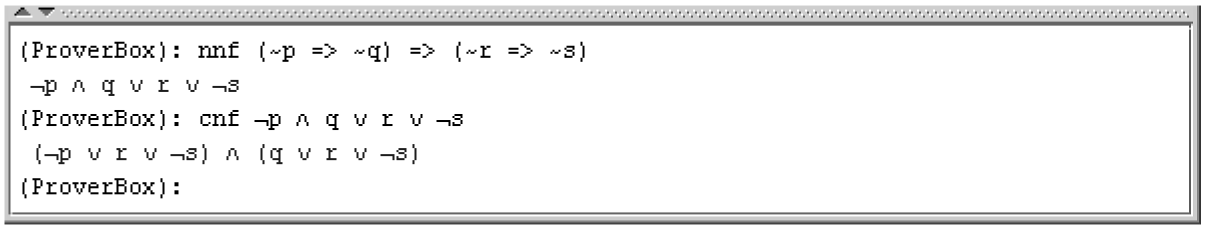


```
(ProverBox): nnf (~p => ~q) => (~r => ~s)
 ¬p ∧ q ∨ r ∨ ¬s
(ProverBox):
```

Figure 1.5: NNF of a Propositional Formula

Be careful: names of user-defined entities, e.g. atoms, names of axioms or symbol names, are always case-sensitive in the ProverBox. So p and P would be considered as two different atoms. However, command names and other keywords are not case-sensitive, therefore `nnf p => q` is equivalent to `NNF p => q`.

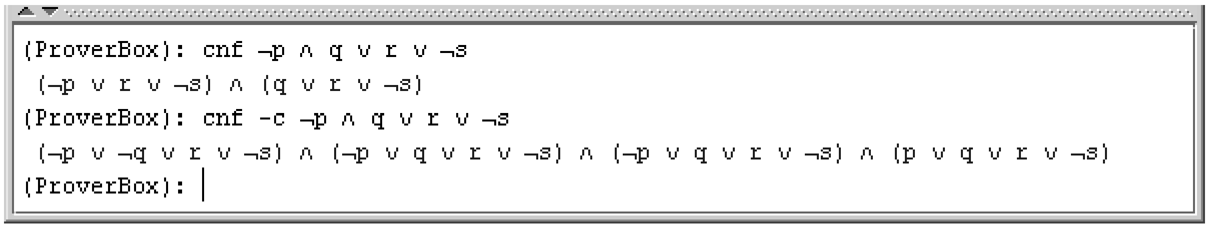
We now want to convert the resulting formula into Conjunctive Normal Form (CNF), i.e. we want to transform it into an expression of the form $\bigwedge_{i=0}^{m} \bigvee_{j=0}^{n} l_{ij}$, where l_{ij} is a literal (= a positive or negative atom). As expected, the command `cnf` does the job, so we might just enter `cnf ~p /\ q \/ r \/ ~s` (the conjunction and disjunction operators are composed of slash `/` and backslash `\`; it is also possible to write `and` resp. `or`). But wait! There is an even easier way to perform the task: type `cnf` followed by a blank, and then press the `TAB`-Key. And voilà - you get the suggested input without having to retype the formula. Whenever you press `TAB` in the command prompt, the ProverBox will copy the last resulting formula into your new input line (provided that such a previous result exists). As shown in Figure 1.6, the result is $(\neg p \vee r \vee \neg s) \wedge (q \vee r \vee \neg s)$.



```
(ProverBox): nnf (~p => ~q) => (~r => ~s)
 ¬p ∧ q ∨ r ∨ ¬s
(ProverBox): cnf ¬p ∧ q ∨ r ∨ ¬s
 (¬p ∨ r ∨ ¬s) ∧ (q ∨ r ∨ ¬s)
(ProverBox):
```

Figure 1.6: CNF of a Propositional Formula

Notice that the CNF from the last paragraph is already simplified, as redundant atoms have been left out (e.g. the first clause does not contain a q). This behavior is not always desirable, and sometimes we might need a so-called complete CNF, where each clause contains all atoms. Hence, the `cnf` command provides an additional option `-c` or `-complete`, which instructs the ProverBox to compute the complete CNF of the argument. Figure 1.7 shows the output for the command `cnf -c ~p /\ q \/ r \/ ~s`.



```
(ProverBox): cnf ¬p ∧ q ∨ r ∨ ¬s
 (¬p ∨ r ∨ ¬s) ∧ (q ∨ r ∨ ¬s)
(ProverBox): cnf -c ¬p ∧ q ∨ r ∨ ¬s
 (¬p ∨ ¬q ∨ r ∨ ¬s) ∧ (¬p ∨ q ∨ r ∨ ¬s) ∧ (¬p ∨ q ∨ r ∨ ¬s) ∧ (p ∨ q ∨ r ∨ ¬s)
(ProverBox): |
```

Figure 1.7: Complete CNF of a Propositional Formula

1.5 Working with Theories

The examples from the last section have shown how to work with single formulas. In this section, we will expand that to working with a collection of formulas, which is called a theory.

1.5.1 Specifying Theories

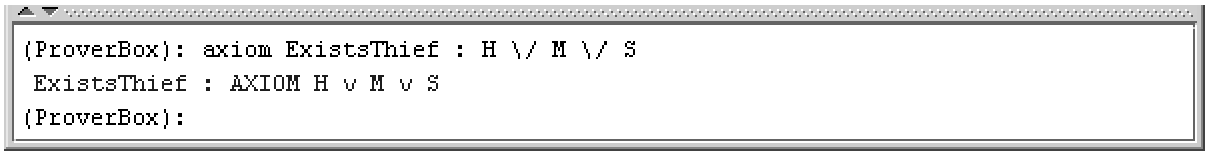
A theory consists of two kinds of formulas: axioms, which are assumed to be true, and theorems, for which it is necessary to prove that they are true. Such a proof can make use of axioms and previously proven theorems.

Axioms are defined with the command `axiom` *`name`* `:` *`formula`*, where *`name`* is the name of the axiom and *`formula`* is the actual formula assumed to be true. Similarly, `theorem` *`name`* `:` *`formula`* defines a theorem.

To demonstrate how these commands are used, let us attempt to translate the following puzzle (based on a problem from [3]) into a propositional theory:

> A valuable item has been stolen. We have three suspects: Harry, Mike and Sue. One of them is the thief, the others are innocent and thus tell the truth. Harry makes the statement that Mike is not the thief, and Sue says that Harry is innocent.

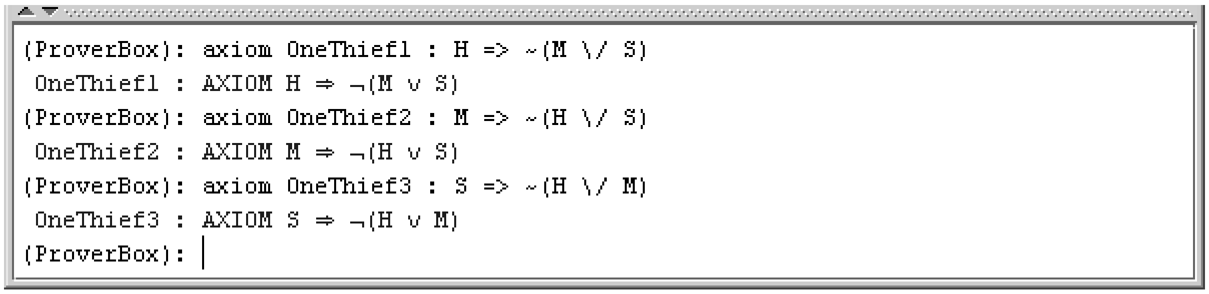
We introduce three atoms, H, M and S, where H is true if and only if Harry is the thief, and analogously for the other persons. To specify that there exists one thief, we enter `axiom ExistsThief : H \/ M \/ S` on the command prompt. As shown in Figure 1.8, the newly added axiom is echoed to indicate successful operation.



```
(ProverBox): axiom ExistsThief : H \/ M \/ S
 ExistsThief : AXIOM H ∨ M ∨ S
(ProverBox):
```

Figure 1.8: Adding a New Axiom

To express that there is at most one thief, three axioms are necessary, each of the form "if one atom is true, then the other two are false", e.g. $H \Rightarrow \neg(M \vee S)$. Figure 1.9 gives the complete input.

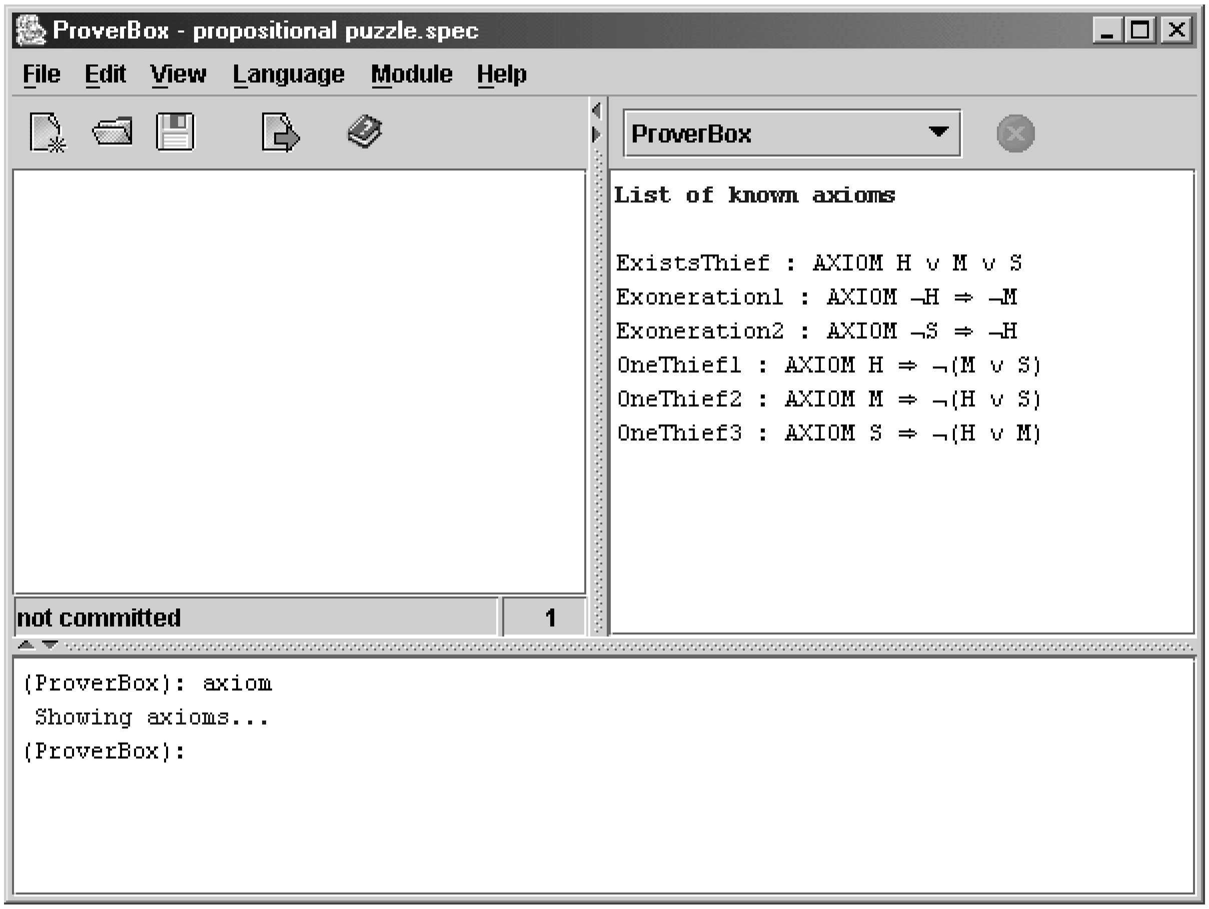


```
(ProverBox): axiom OneThief1 : H => ~(M \/ S)
 OneThief1 : AXIOM H ⇒ ¬(M ∨ S)
(ProverBox): axiom OneThief2 : M => ~(H \/ S)
 OneThief2 : AXIOM M ⇒ ¬(H ∨ S)
(ProverBox): axiom OneThief3 : S => ~(H \/ M)
 OneThief3 : AXIOM S ⇒ ¬(H ∨ M)
(ProverBox): |
```

Figure 1.9: Existence of at most One Thief

Finally, the two exonerations need to be translated. We do this by saying "if Harry is not the thief (i.e. he is innocent and therefore saying the truth), then Mike is not the thief either". This leads to the input `axiom Exoneration1 : ~H => ~M`, and analogously for the second exoneration.

This completes the translation of the puzzle into propositional axioms. We can review the axioms we have just added by using the `axiom` command without argument. We are then given a list of all available axioms in the output window as shown in Figure 1.10.


ProverBox - propositional puzzle.spec
File Edit View Language Module Help
ProverBox
not committed
1

```
List of known axioms

ExistsThief : AXIOM H ∨ M ∨ S
Exoneration1 : AXIOM ¬H ⇒ ¬M
Exoneration2 : AXIOM ¬S ⇒ ¬H
OneThief1 : AXIOM H ⇒ ¬(M ∨ S)
OneThief2 : AXIOM M ⇒ ¬(H ∨ S)
OneThief3 : AXIOM S ⇒ ¬(H ∨ M)
```

```
(ProverBox): axiom
 Showing axioms...
(ProverBox):
```

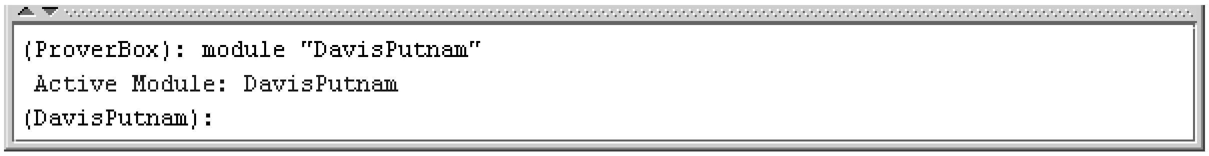
Figure 1.10: List of Known Axioms

1.5.2 Choosing Prover Modules

It is now time to solve the puzzle using one of the prover modules that come with the ProverBox. For this example, we choose the DavisPutnam module. It implements a fast propositional decision procedure and is able to produce counterexamples, which is what we will need to actually solve the puzzle, as will be seen later.

To select the DavisPutnam prover, type `module ‘‘DavisPutnam’’` on the command prompt (see Figure 1.11). Alternatively, choose "DavisPutnam" from the module selection combo box above the output window or click on `Module` and then `Select` in the main menu.

As seen from the figures, the command prompt will change to `(DavisPutnam):` to confirm that the DavisPutnam module is now the current namespace.



```
(ProverBox): module "DavisPutnam"
 Active Module: DavisPutnam
(DavisPutnam):
```

Figure 1.11: Switching Modules on the Command Prompt

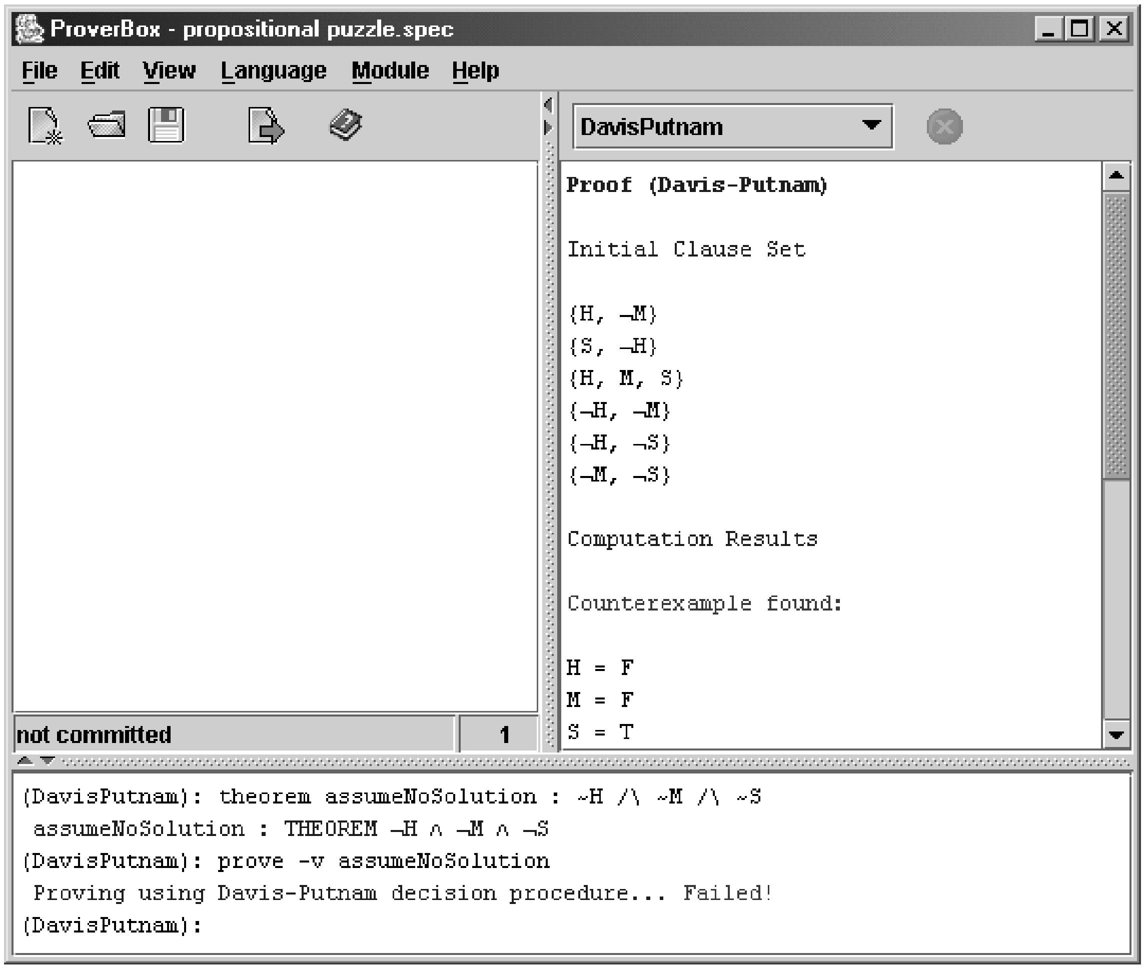
1.5.3 Proving Theorems

Now that we have successfully activated the DavisPutnam prover, how can it be used to solve the puzzle? One might think that a theorem prover is only useful when you already know the solution and want to prove that it is correct. But there is more that a prover can do: as mentioned before, the DavisPutnam module can come up with counterexamples. So the way to solving this puzzle is to assume the contrary, i.e. that there exists no solution, and then have the prover come up with a counterexample or show that there is indeed no solution. In this example, no solution means there is no thief, so all persons are innocent, which leads to $\neg H \wedge \neg M \wedge \neg S$. We give this formula a name and enter `theorem assumeNoSolution : ~H /\ ~M /\ ~S` on the prompt.

Proving (or refuting, as in this example) is now very easy. Naturally enough, the corresponding command is called `prove`, and as an argument, it either expects the name of a theorem or a plain formula to prove, which means that we could have just given it the formula itself instead of defining a theorem. However, using theorems is considered to be better style, plus the system is able to take advantage of proven theorems in subsequent proofs, which is not possible with plain formulas.

There exists an additional option `-v` or `-verbose` to get a more detailed output, including a proof tree, which is quite useful to understand how the prover came to an answer. Try it out and type `prove -v assumeNoSolution` on the command prompt. The resulting output is shown in Figure 1.12.

As seen in Figure 1.12, a counterexample is found, from which we can infer that Sue is the thief. Is this the only solution? Well, for this puzzle to make sense, there better be a unique solution.


ProverBox - propositional puzzle.spec
File Edit View Language Module Help
DavisPutnam
not committed
1

```
Proof (Davis-Putnam)

Initial Clause Set

{H, ¬M}
{S, ¬H}
{H, M, S}
{¬H, ¬M}
{¬H, ¬S}
{¬M, ¬S}

Computation Results

Counterexample found:

H = F
M = F
S = T
```

```
(DavisPutnam): theorem assumeNoSolution : ~H /\ ~M /\ ~S
 assumeNoSolution : THEOREM ¬H ∧ ¬M ∧ ¬S
(DavisPutnam): prove -v assumeNoSolution
 Proving using Davis-Putnam decision procedure... Failed!
(DavisPutnam):
```

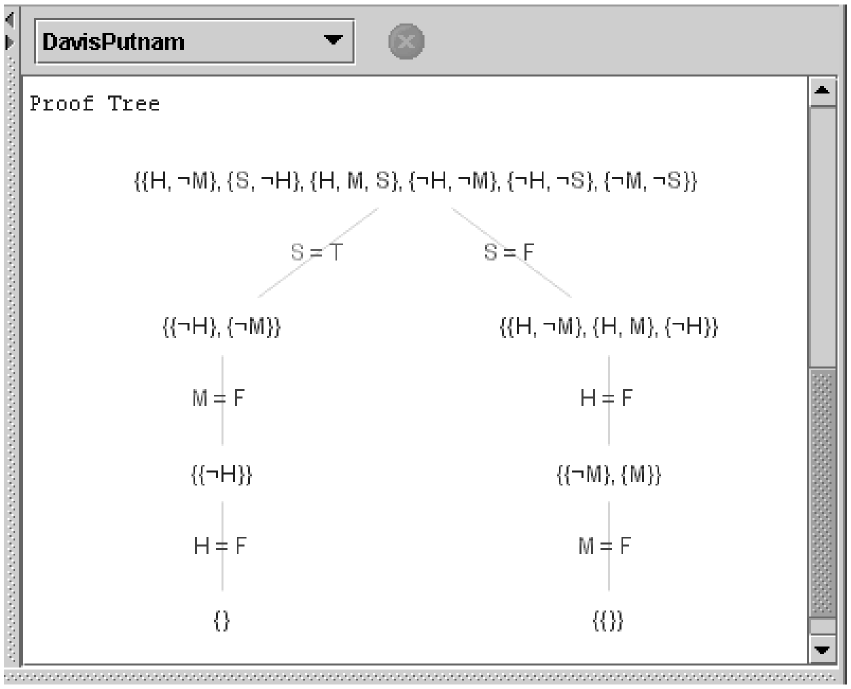
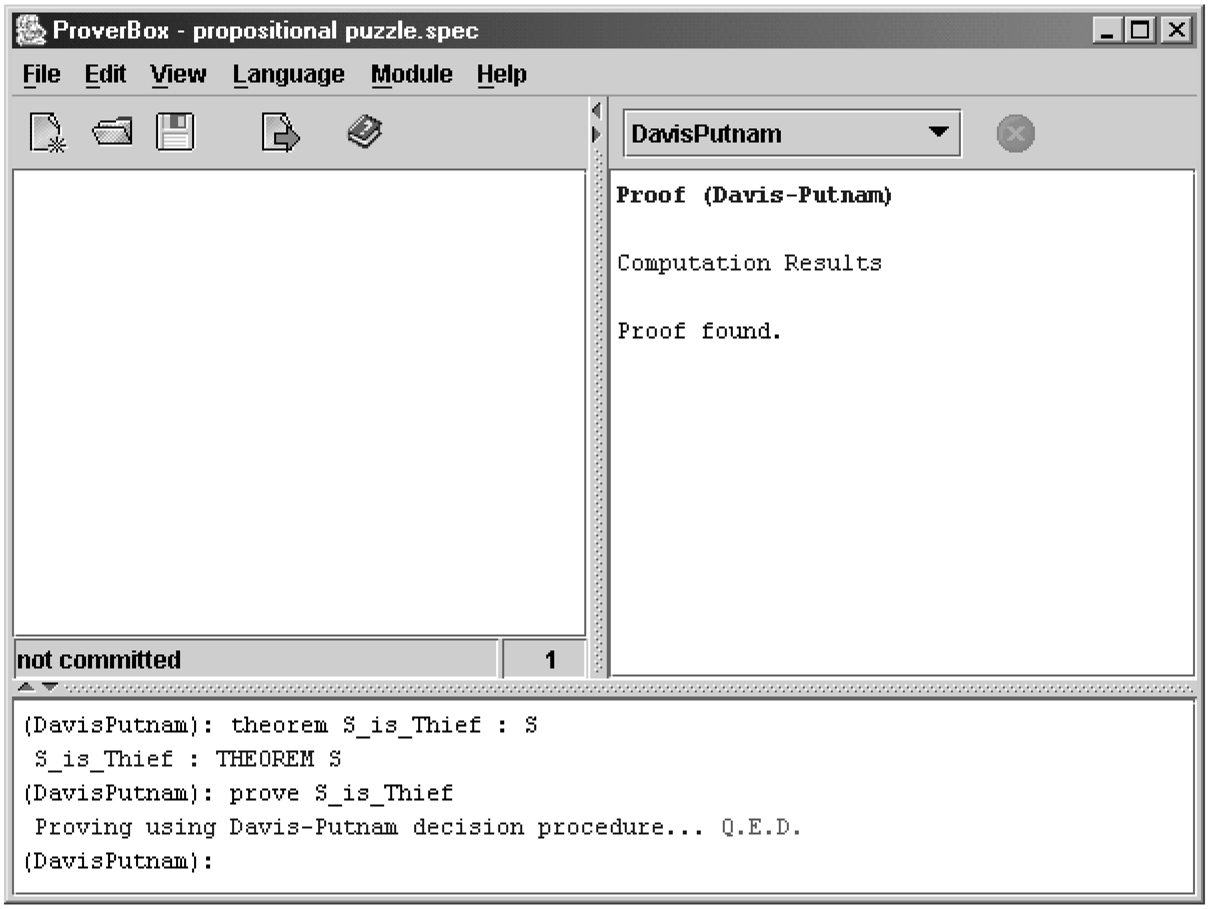


Figure 1.12: Finding a Counterexample with DavisPutnam

Fortunately, we can also show that S is necessarily true by defining a new theorem using `theorem S_is_Thief : S` and then telling the system to do `prove S_is_Thief`. That proof is successful, as Figure 1.13 demonstrates.


ProverBox - propositional puzzle.spec
File Edit View Language Module Help
DavisPutnam
not committed
1

```
Proof (Davis-Putnam)

Computation Results

Proof found.
```

```
(DavisPutnam): theorem S_is_Thief : S
 S_is_Thief : THEOREM S
(DavisPutnam): prove S_is_Thief
 Proving using Davis-Putnam decision procedure... Q.E.D.
(DavisPutnam):
```

Figure 1.13: Proving that the Solution is Unique

1.6 Declarations and the Type System

We now want to move on to first-order predicate logic, because it is more expressive and allows for more interesting examples. However, predicate logic comes with some additional complexity, for example the need for symbol declarations. In this section, we will learn how symbols are declared in the ProverBox, and how to use the so-called typed predicate calculus, which is implemented in the ProverBox.

1.6.1 Typed Predicate Calculus

One difference between propositional calculus and predicate calculus is the fact that in a propositional formula, each symbol name denotes an atom, e.g. in the expression $a \wedge b$, we know that a and b are atoms. Predicate logic has different kinds of symbols: constant symbols, variable symbols, function symbols and predicate symbols. In an expression like $P(a)$, a could be a variable or a constant. The user has to inform the system about this: he has to declare a to be either a variable or a constant.

Having the user declare all symbols also allows the system to find semantic errors. For example, if f is declared to be a unary function, then $f(a, b)$ must be an invalid expression. Now, typisation comes into play: when we assign each symbol a type, e.g. a to be an integer constant and b to be a real constant, further semantic checking is possible, as type mismatches can now be detected. For example, if f is declared to have one integer argument, then $f(a)$ is semantically correct, but $f(b)$ is obviously invalid.

1.6.2 Automatic Declarations

Before we can begin to declare types and symbols, we have to switch to predicate calculus using the `lang` command presented in Section 1.4. There is, however, an optional argument to this command that the reader should be aware of: `autodecl=on/off`. This controls whether automatic declarations are enabled or not.

Automatic declarations allow the user to declare just a few symbols, or even none, and the ProverBox will try to figure out what meaning the remaining symbols have. For example, if *emptySet* is declared to be a constant of type *set*, then from the expression $\neg hasDuplicates(emptySet)$, we can infer that *hasDuplicates* has to be a predicate which takes one argument of type *set*.

Automatic declarations are convenient for formula manipulations or other operations where the actual meaning of the symbols is not important. For working with theory specifications and theorem proving, this option is dangerous, as it can lead to unexpected behavior. For example, a typo like $\neg hasDuplicate(emptySet)$ would not be detected, but lead to another automatic declaration and thus to a different predicate symbol.

In this section, we want to do without automatic declarations, so we enter the command `lang ‘‘predicate logic’’ autodecl=off`.

1.6.3 Declaring Constants and Variables

In predicate logic, there are two basic kinds of symbols to be declared: constant symbols and variable symbols.

Usually, the word *constant* is used to denote a *constant of a given type*, e.g. the empty set being a constant of type “set”. However, in first-order predicate logic, functions and even types can also be viewed as constants. Sometimes, the names *function constants* and *type constants* are used to illustrate that fact. It is therefore reasonable to use a common command, naturally enough called `const`, to declare all these different kinds of constants. The argument to the command varies depending on what kind of constant is to be declared:

- *Elementary types* are declared using the command

  `const` ***`name`*** `: TYPE`

  where ***`name`*** is the name of the type to be declared. The algebraic structure “set” is an example for an elementary type, so we might make the declaration `const SET : TYPE` as shown in Figure 1.14. Notice that we follow the convention of writing type names in upper case.

- It is also possible to declare *Subtypes* from given parent types, using

  `const` ***`name`*** `: TYPE FROM` ***`parent`***

  where the argument ***`parent`*** denotes the name of the parent type. For example, recall that relations are a special kind of sets, which would lead to the declaration `const RELATION : TYPE FROM SET` (Figure 1.14).

- *Constants of a given type* are declared using

  `const` ***`name`*** `:` ***`type`***

  where ***`type`*** is the name of the type to be associated with the newly declared constant. The example from above (the empty set being a constant of type “set”) would be declared as `const emptySet : SET` (Figure 1.14)

- *Functions* can be declared using the command

  `const` ***name*** `: [`***argType1*** `, ...,` ***argTypeN*** `->` ***retType***`]`

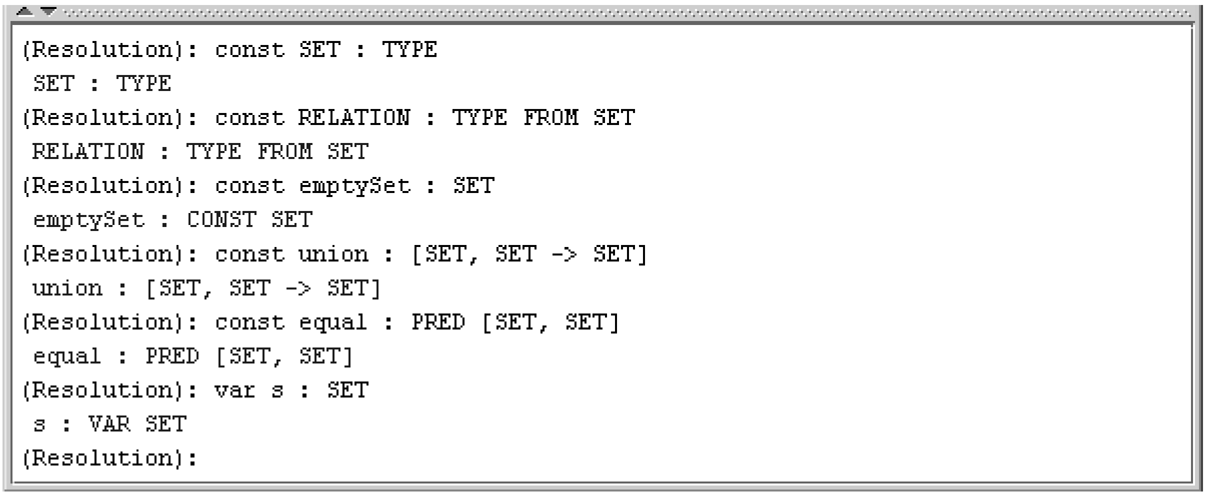
  where ***argType1***, ..., ***argTypeN*** are the types of the arguments, and ***retType*** is the return type. The union operation, for example, takes two sets and returns the unified set, so we could make the declaration `const union : [SET, SET -> SET]` (Figure 1.14).

- For *Predicates*, we use the form

  `const` ***name*** `: PRED [`***argType1*** `, ...,` ***argTypeN***`]`

  As for functions, ***argType1***, ..., ***argTypeN*** are the types of the arguments. For example, we might declare a predicate to test two sets for equality with the command `const equal : PRED [SET, SET]` (Figure 1.14).

In first-order predicate logic, there is only one kind of variables: variables of a given type. Function variables or type variables only occur in higher-order logic. To declare a variable named ***name*** of type ***type***, enter `var` ***name*** `:` ***type*** on the command prompt. For example, `var s : SET` declares a set variable, as shown together with the previous examples in Figure 1.14.

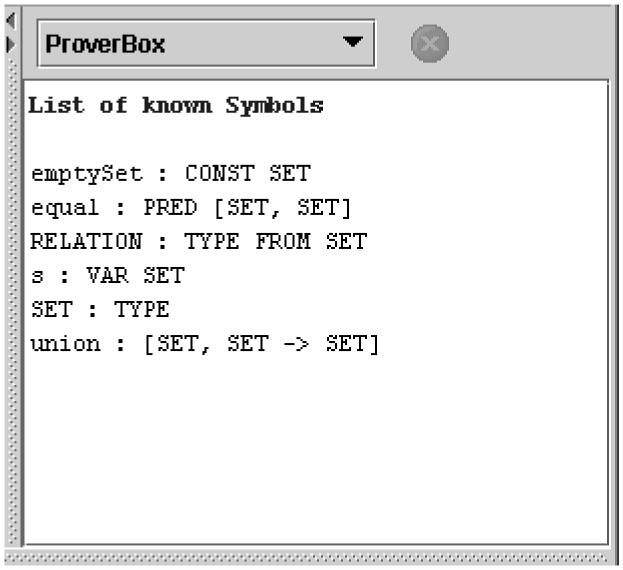


```
(Resolution): const SET : TYPE
 SET : TYPE
(Resolution): const RELATION : TYPE FROM SET
 RELATION : TYPE FROM SET
(Resolution): const emptySet : SET
 emptySet : CONST SET
(Resolution): const union : [SET, SET -> SET]
 union : [SET, SET -> SET]
(Resolution): const equal : PRED [SET, SET]
 equal : PRED [SET, SET]
(Resolution): var s : SET
 s : VAR SET
(Resolution):
```

Figure 1.14: Declaring Constants and Symbols

1.6.4 Working with Symbols

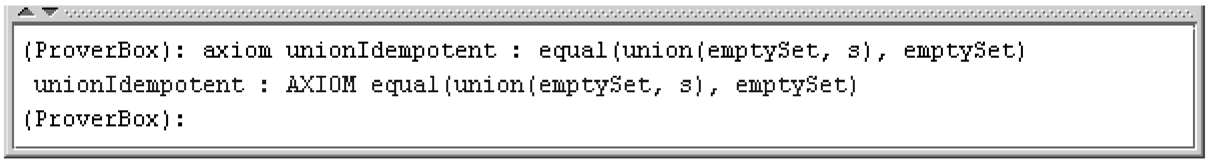
Existing declarations may be reviewed with the command `symbol`. It can be used in two variations: it can either be given the name of a symbol to look up and show its declaration (if a symbol of that name exists). Or, if no argument is specified, an alphabetically sorted list of known symbols is shown. Figure 1.15 shows what the output looks like for the set examples from the previous Section 1.6.3.



```
List of known Symbols

emptySet : CONST SET
equal : PRED [SET, SET]
RELATION : TYPE FROM SET
s : VAR SET
SET : TYPE
union : [SET, SET -> SET]
```

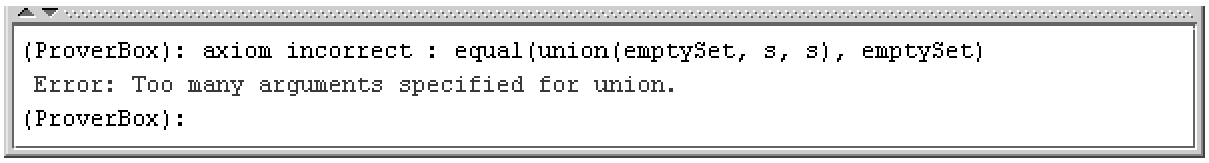
Figure 1.15: List of Known Symbols

As explained in Section 1.6.1, the declarations we have made are used to verify the input which is given to the program. Now that all declarations have been made, we can see this in action. Let us look at two axiom definitions, a correct one and a flawed attempt: in Figure 1.16, all symbols are used according to their declarations, so the axiom is successfully defined. In Figure 1.17, however, there are too many arguments to the binary function *union*. As expected, the mistake is detected, and so the input is rejected.



```
(ProverBox): axiom unionIdempotent : equal(union(emptySet, s), emptySet)
 unionIdempotent : AXIOM equal(union(emptySet, s), emptySet)
(ProverBox):
```

Figure 1.16: Typecheck Successfully Passed



```
(ProverBox): axiom incorrect : equal(union(emptySet, s, s), emptySet)
 Error: Too many arguments specified for union.
(ProverBox):
```

Figure 1.17: Typecheck Not Passed

1.7 Advanced Formula Manipulations

One of the beauties of the ProverBox being an integrated environment for multiple logics is that the different logics are very similar to use. In fact, the basic transformations shown in Section 1.4 are also available for predicate logic, using the same command names. Of course, what is going on under the hood is very different for predicate logic and much harder to compute.

To see how it works, let us use the ProverBox to transform the formula

$$\forall x : T1.\, P(x) \vee \neg(\forall y : T2.\, Q(x, y) \vee R(x, f(y)))$$

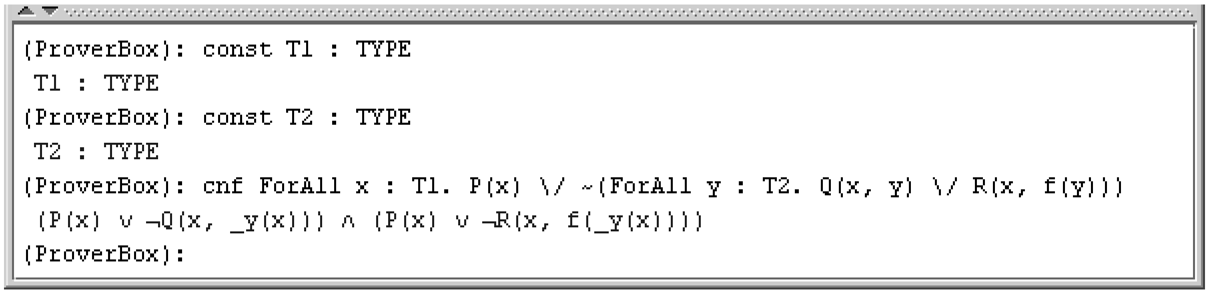
into Conjunctive Normal Form. As can be seen from the formula, quantified variables also need to be given a type, as we are using typed predicate logic.

Before we can enter the formula, we need to make the necessary declarations. However, we can make our lives a lot easier by taking advantage from automatic declarations. As mentioned in Section 1.6.2, this feature is especially convenient for working and experimenting with single formulas, so let us enable it by entering `lang ‘‘predicate logic’’ autodecl=on` on the command prompt. Mouse fans can also click on `Language` and then `Select` to get to the language selection dialog, where the feature can be enabled.

Now, all we have to do is to declare the two types $T1$ and $T2$ using the commands `const T1 : TYPE` and `const T2 : TYPE`, and the ProverBox will figure out the meaning of the remaining symbols.

As we have already seen for propositional logic, the command for transforming a formula into Conjunctive Normal Form is `cnf`, and it expects one argument, which is the formula to work on. Quantifiers are written as `Exists` resp. `ForAll` (using all lower caps is also acceptable, as keywords are not case-sensitive), so we can input `cnf ForAll x : T1. P(x) \/ ~(ForAll y : T2. Q(x, y) \/ R(x, f(y)))`.

Finding the CNF of a formula in predicate logic is much harder than it is in propositional logic. However, the `cnf` command is quite smart and does perform all the necessary steps without user intervention. We immediately get the resulting formula, which is shown in Figure 1.18. Besides the need to declare the two types, this has worked just like finding the CNF of a propositional formula. That is certainly amazing, as almost all the additional complexity of predicate logic has been hidden from us by the ProverBox.

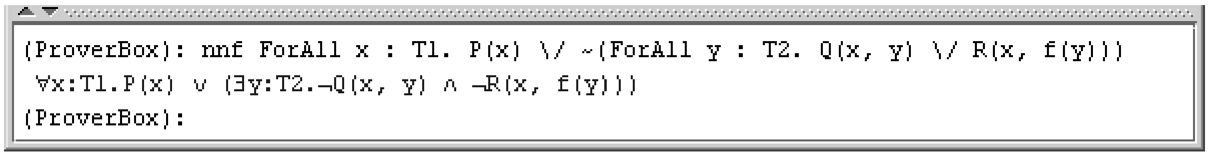


```
(ProverBox): const T1 : TYPE
 T1 : TYPE
(ProverBox): const T2 : TYPE
 T2 : TYPE
(ProverBox): cnf ForAll x : T1. P(x) \/ ~(ForAll y : T2. Q(x, y) \/ R(x, f(y)))
 (P(x) ∨ ¬Q(x, _y(x))) ∧ (P(x) ∨ ¬R(x, f(_y(x))))
(ProverBox):
```

Figure 1.18: CNF of a Formula in Predicate Logic

As this is our first example of working with a formula in predicate logic, it might be necessary to explain what the output actually means and how it was obtained. Fortunately, the ProverBox allows us to get more insight by individually performing the necessary steps instead of using an all-powerful command.

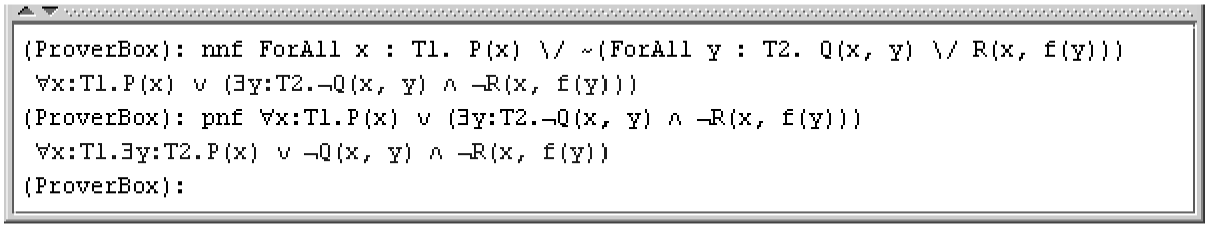
Recall that the first step is to bring the formula into Negative Normal Form (NNF), which is obtained by propagating the negation into the scope of the second quantifier, right in front of the two predicates Q and R. The negation of a universal quantifier can be written as an existentially quantified negation, and the negation of a disjunction is equivalent to a conjunction of negations. As shown in Figure 1.19, this results in the formula $\forall x : T1.\, P(x) \vee (\exists y : T2.\, \neg Q(x, y) \wedge \neg R(x, f(y)))$.



```
(ProverBox): nnf ForAll x : T1. P(x) \/ ~(ForAll y : T2. Q(x, y) \/ R(x, f(y)))
 ∀x:T1.P(x) ∨ (∃y:T2.¬Q(x, y) ∧ ¬R(x, f(y)))
(ProverBox):
```

Figure 1.19: NNF of a Formula in Predicate Logic

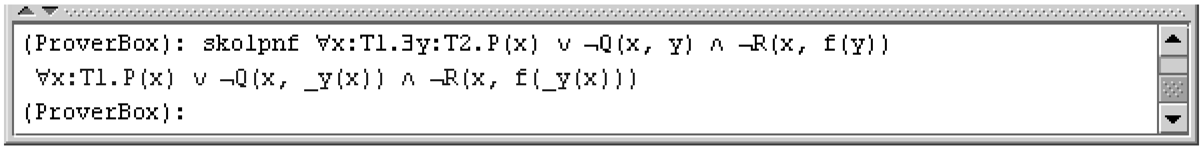
The next step is to transform this formula into Prenex Normal Form (PNF), which is to move all quantifiers to the front. Enter the corresponding command `pnf`, then press the `TAB`-Key to reuse the previous result. As shown in Figure 1.20, the resulting PNF is $\forall x : T1.\, \exists y : T2.\, P(x) \vee (\neg Q(x, y) \wedge \neg R(x, f(y)))$.



```
(ProverBox): nnf ForAll x : T1. P(x) \/ ~(ForAll y : T2. Q(x, y) \/ R(x, f(y)))
 ∀x:T1.P(x) ∨ (∃y:T2.¬Q(x, y) ∧ ¬R(x, f(y)))
(ProverBox): pnf ∀x:T1.P(x) ∨ (∃y:T2.¬Q(x, y) ∧ ¬R(x, f(y)))
 ∀x:T1.∃y:T2.P(x) ∨ ¬Q(x, y) ∧ ¬R(x, f(y))
(ProverBox):
```

Figure 1.20: PNF of a Formula in Predicate Logic

Now it is time to get rid of the existential quantifiers in order to get the so-called Skolemized Prenex Normal Form. To achieve this, all existentially quantified variables need to be replaced by Skolem functions, that is functions which depend on all variables from preceding universal quantifiers. New names not used elsewhere have to be found for those functions. The ProverBox uses the original name of the existentially quantified variable, but prefixed with an underscore. In our example, the y is replaced by a new function $\_y(x)$, and we get $\forall x : T1.\, P(x) \vee (\neg Q(x, \_y(x)) \wedge \neg R(x, f(\_y(x))))$, as seen in Figure 1.21.



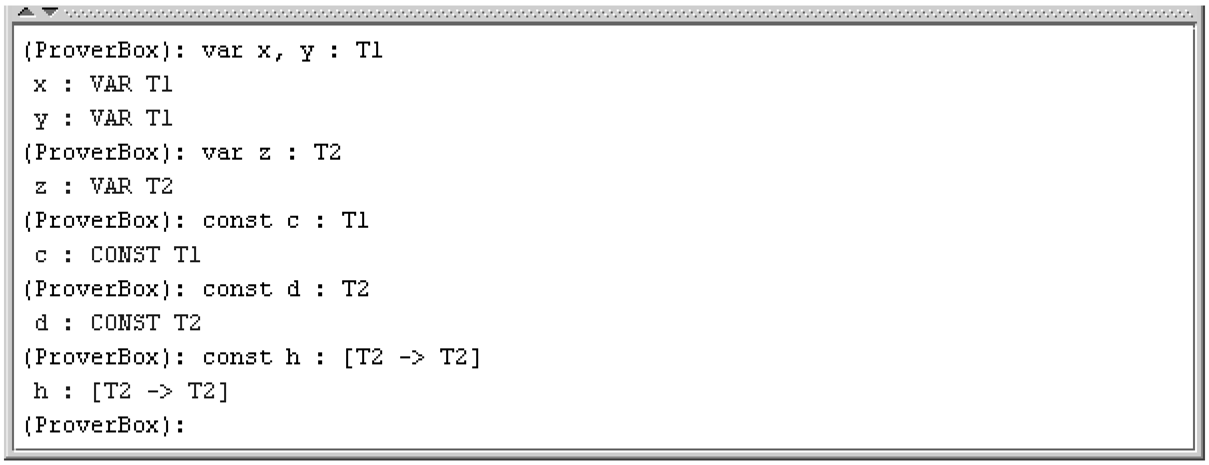
```
(ProverBox): skolpnf ∀x:T1.∃y:T2.P(x) ∨ ¬Q(x, y) ∧ ¬R(x, f(y))
 ∀x:T1.P(x) ∨ ¬Q(x, _y(x)) ∧ ¬R(x, f(_y(x)))
(ProverBox):
```

Figure 1.21: Skolemized PNF of a Formula in Predicate Logic

Finally, the universal quantifiers are left out, so we end up with a quantifier-free expression, which is usually called the matrix of the original formula. The matrix is essentially a propositional expression, so it is easily transformed into CNF, and we get the final result that we already saw in Figure 1.18.

An important application of transforming formulas into CNF is to use the unification algorithm on the terms (the literals) obtained from the CNF. Unification, which is basically the process of finding variable assignments such that two terms are made equal, is the main operation of some of the most important proving algorithms in predicate calculus. In the ProverBox, this can be done by using the `unify` command, which expects two terms to work on as arguments.

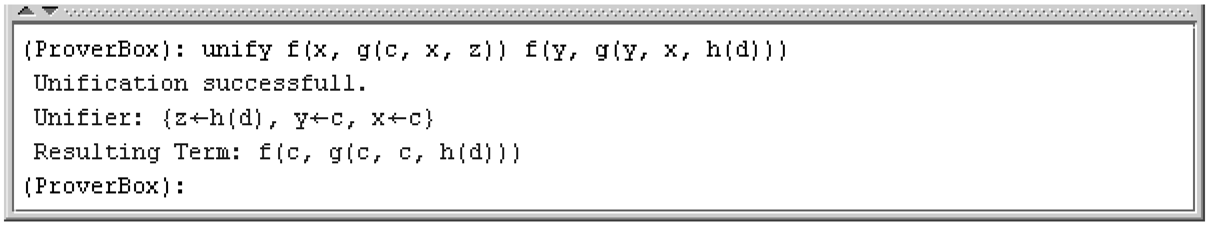
Let us consider an example: assuming that $T1$ and $T2$ are still declared as in the last example, we make the additional declarations `var x, y : T1` (notice that it is actually possible to declare more than one entity at the same time by writing them as a comma-separated list of names as seen here), `var z : T2`, `const c : T1`, `const d : T2` and `const h : [T2 -> T2]` (see Figure 1.22).



```
(ProverBox): var x, y : T1
 x : VAR T1
 y : VAR T1
(ProverBox): var z : T2
 z : VAR T2
(ProverBox): const c : T1
 c : CONST T1
(ProverBox): const d : T2
 d : CONST T2
(ProverBox): const h : [T2 -> T2]
 h : [T2 -> T2]
(ProverBox):
```

Figure 1.22: Declarations for the Unification Example

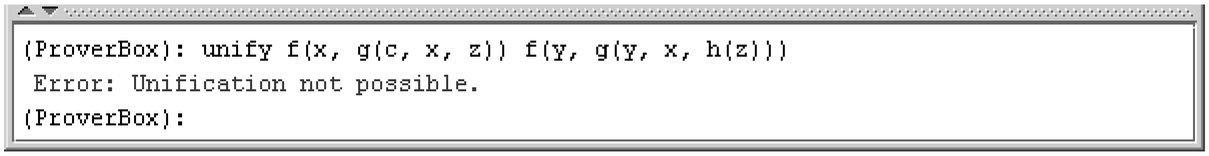
We now want to unify the two terms $f(x, g(c, x, z))$ and $f(y, g(y, x, h(d)))$. Assuming that automatic declarations are still enabled, it is not necessary to declare f and g, as their declaration can be inferred from the description of the other symbols. As can be seen in Figure 1.23, unification is successful if we let z be $h(d)$ and set x and y to c.



```
(ProverBox): unify f(x, g(c, x, z)) f(y, g(y, x, h(d)))
 Unification successfull.
 Unifier: {z←h(d), y←c, x←c}
 Resulting Term: f(c, g(c, c, h(d)))
(ProverBox):
```

Figure 1.23: Example of a Successful Unification

Obviously, unification does not always succeed, for example when the structure of the two terms is different. But there is also a subtle check called the *occurs check* to prevent an assignment to a variable from containing that variable itself. We can get into this situation if we try to unify $f(x, g(c, x, z))$ and $f(y, g(y, x, h(z)))$, which would imply that z has to be $h(z)$. As shown in Figure 1.24, the ProverBox is able to detect this problem.



```
(ProverBox): unify f(x, g(c, x, z)) f(y, g(y, x, h(z)))
 Error: Unification not possible.
(ProverBox):
```

Figure 1.24: Example of a Failed Occurs Check

1.8 Editing and Proving Complex Theories

So far, we have only been using the command prompt of the ProverBox, which is certainly convenient for working with single formulas or short theory specifications. For larger problems, however, this is not suitable, as it is difficult to keep track of the independent commands, and there is no way to save them. Fortunately, the ProverBox offers an integrated editor to be used instead.

The following sample problem (based on problem PUZ012-1 from [2]) will be used to demonstrate how to work with the editor:

> There are three boxes a, b, and c on a table. Each box contains apples or bananas or oranges. No two boxes contain the same thing.
> Each box has a label that says it contains apples or says it contains bananas or says it contains oranges. No box contains what it says on its label. The label on box a says "apples". The label on box b says "oranges". The label on box c says "bananas".
> You pick up box b and it contains apples. What do the other two boxes contain?

Translating real-world problems into logic tends to result in large theory specifications. The problem above is pretty easy, but still results in a surprisingly long translation, as can be seen from Listing 1.1.

The reason for the specification being longer than expected is that we have to include certain facts which are very obvious to a human, but not to the computer: we

Listing 1.1: Theory Specification for the Mislabeled Boxes Problem

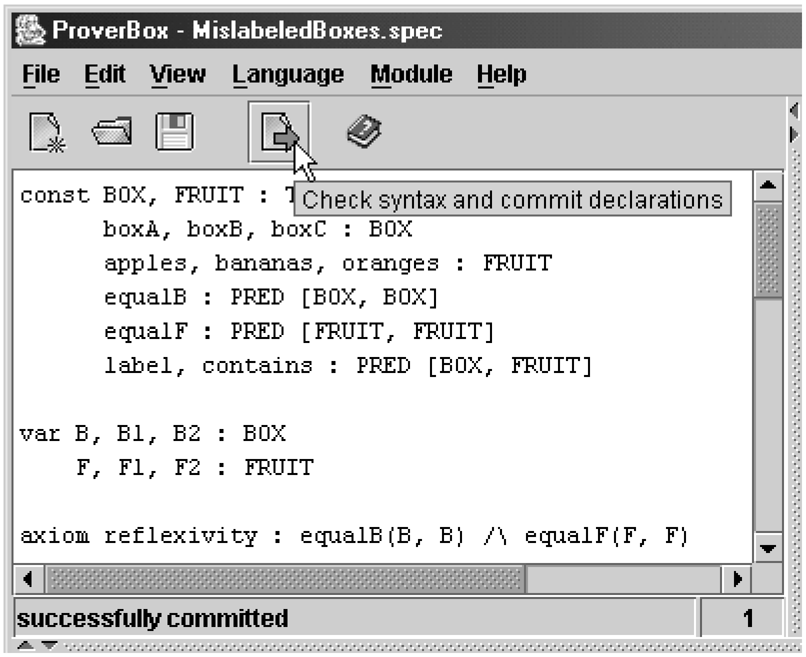
```
const BOX, FRUIT : TYPE
      boxA, boxB, boxC : BOX
      apples, bananas, oranges : FRUIT
      equalB : PRED [BOX, BOX]
      equalF : PRED [FRUIT, FRUIT]
      label, contains : PRED [BOX, FRUIT]

var B, B1, B2 : BOX
    F, F1, F2 : FRUIT

axiom reflexivity : equalB(B, B) /\ equalF(F, F)

axiom symmetry : (equalB(B1, B2) <=> equalB(B2, B1)) /\
                 (equalF(F1, F2) <=> equalF(F2, F1))

axiom equality : ~equalB(boxA, boxB) /\ ~equalB(boxA, boxC) /\
                 ~equalB(boxB, boxC) /\ ~equalF(apples, bananas) /\
                 ~equalF(apples, oranges) /\ ~equalF(bananas, oranges)

axiom labelIncorrect : label(B, F) => ~contains(B, F)

axiom fruitInBox : contains(boxA, F) \/ contains(boxB, F) \/
                   contains(boxC, F)

axiom boxHasFruit : contains(B, apples) \/ contains(B, bananas) \/
                    contains(B, oranges)

axiom wellDefined1 : contains(B, F1) /\ contains(B, F2) => equalF(F1, F2)

axiom wellDefined2 : contains(B1, F) /\ contains(B2, F) => equalB(B1, B2)

axiom givenLabels : label(boxA, apples) /\ label(boxB, oranges) /\
                    label(boxC, bananas)

axiom known : contains(boxB, apples)

theorem solution : exists InA, InC : FRUIT.
                   contains(boxA, InA) /\ contains(boxC, InC)
```

need to tell it about the properties of equality, and we need to state that each kind of fruit is in a box and that each box has fruit in it.

Earlier in this tour of the ProverBox, commands like `const` and `var` as well as `axiom` and `theorem` have already been discussed. Hence, the syntax used in Listing 1.1 should appear very familiar. As a matter of fact, theory files are essentially collections of those single commands.

The theory specification can be downloaded from the ProverBox website [1]. Then select `Open` from the `File` menu and locate the file to load it into the ProverBox. If we were to type it ourselves, we would instead choose `New` from the `File` menu to create an empty theory. Upon creating a new theory, the user is asked to select the desired specification language. For the mislabeled boxes problem, the correct choice is predicate logic, and automatic declarations should be disabled.

Once the specification is loaded or completely typed, it is time to have it activated, just like a computer program needs to be compiled after the programmer has finished editing the source code. In the ProverBox, this process is called committing, and it can be initiated by hitting `ALT-C`, by selecting `Commit` from the `Edit` menu or by clicking on the toolbar icon with the green arrow as shown in Figure 1.25.

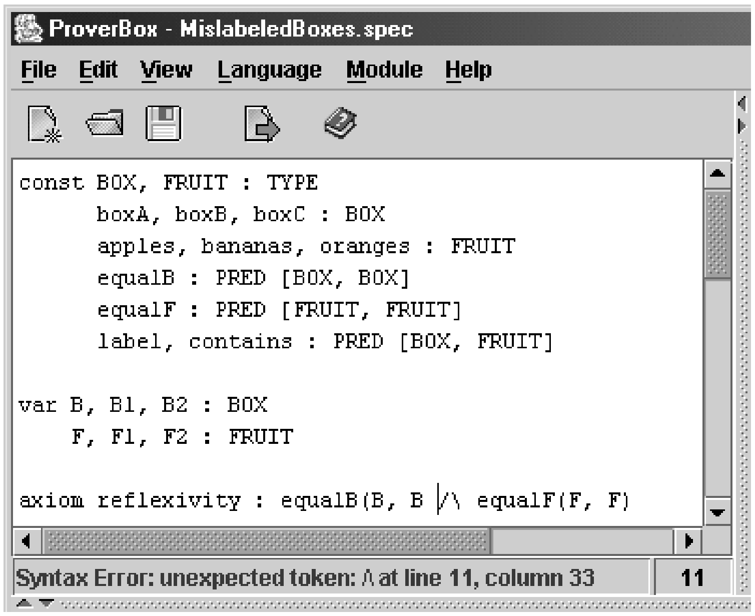

ProverBox - MislabeledBoxes.spec
File Edit View Language Module Help
Check syntax and commit declarations

```
const BOX, FRUIT : T
      boxA, boxB, boxC : BOX
      apples, bananas, oranges : FRUIT
      equalB : PRED [BOX, BOX]
      equalF : PRED [FRUIT, FRUIT]
      label, contains : PRED [BOX, FRUIT]

var B, B1, B2 : BOX
    F, F1, F2 : FRUIT

axiom reflexivity : equalB(B, B) /\ equalF(F, F)
```

```
successfully committed    1
```

Figure 1.25: Committing a Theory Specification

When the user issues the commit command, the specification is first checked for syntactic and semantic errors. If such an error is found, the file cannot be committed, and an error message is shown in the status line at the bottom of the editor panel. Figure 1.26 shows what happens when we intentionally leave out a closing paren.


ProverBox - MislabeledBoxes.spec
File Edit View Language Module Help

```
const BOX, FRUIT : TYPE
      boxA, boxB, boxC : BOX
      apples, bananas, oranges : FRUIT
      equalB : PRED [BOX, BOX]
      equalF : PRED [FRUIT, FRUIT]
      label, contains : PRED [BOX, FRUIT]

var B, B1, B2 : BOX
    F, F1, F2 : FRUIT

axiom reflexivity : equalB(B, B /\ equalF(F, F)
```

```
Syntax Error: unexpected token: /\ at line 11, column 33    11
```

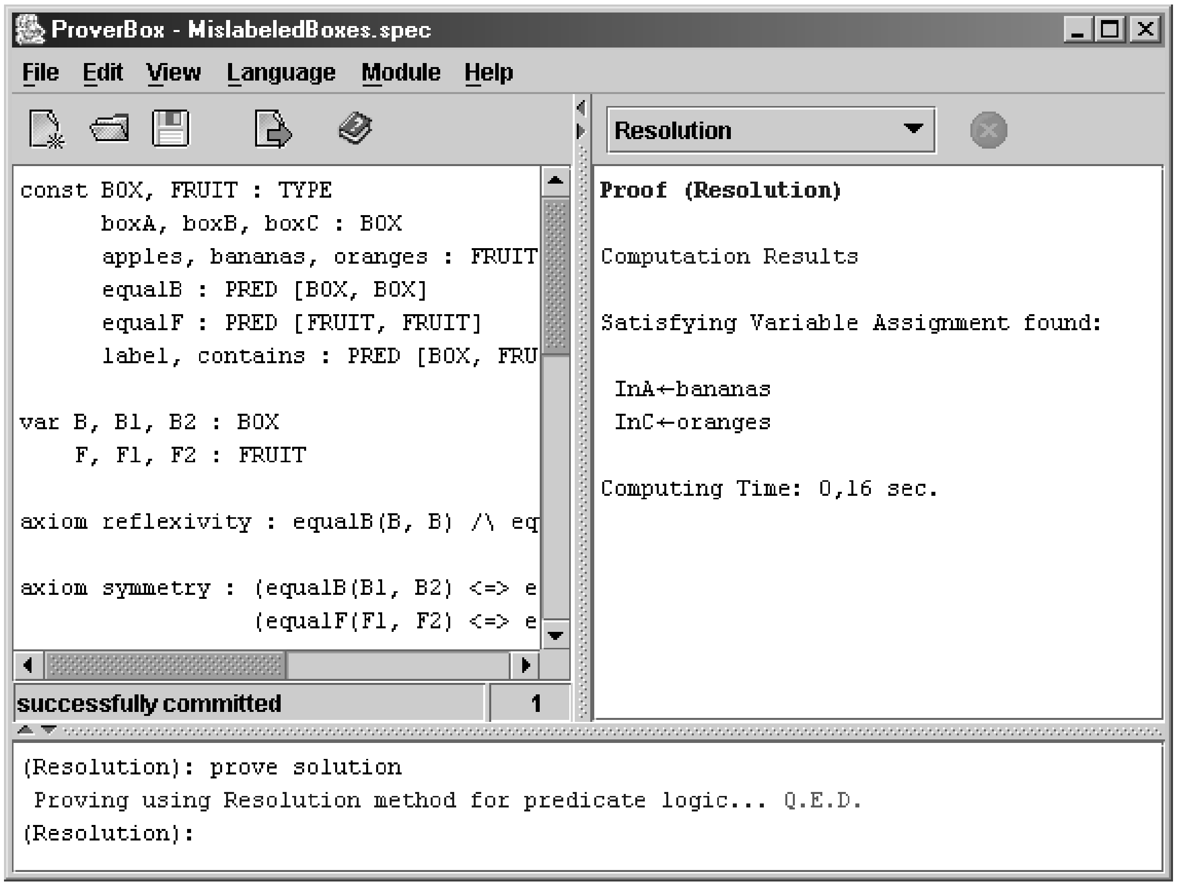
Figure 1.26: Syntax Error in the Specification

After the specification has been successfully committed, its symbol declarations, axioms and theorems are known to the system and can be used on the command prompt as shown in the previous sections of this tour. Notice that committing a theory specification will result in losing all previously declared symbols and formulas, as the system will be brought to a state where it exactly represents the theory specification, with any previous system state being overridden. This behavior can actually be quite useful: for logical reasons, the system does not allow any attempts to prove a theorem more than once. But if you really insist on doing it again (maybe to admire the output of the prover), just hit `ALT-C` to re-commit, and the system will forget that the theorem has been proven.

It is possible to edit a specification after it has been committed. However, the changes won't be effective until it is committed again. This is visualized by the fact that the status line at the bottom of the editor reverts to "not committed" whenever a previously committed specification is edited.

We are now ready to finally solve our puzzle by proving the theorem. This works exactly as it did for the propositional example in Section 1.5.3. For predicate logic, the ProverBox includes a resolution prover, which we are going to use in this example. It is activated with the command `module ‘‘Resolution’’` or by using the module selection combo box above the output pane.

We then invoke the prover with `prove solution` and immediately get the result. As shown in Figure 1.27, box a contains bananas, and box c is filled with oranges.


ProverBox - MislabeledBoxes.spec
File Edit View Language Module Help
Resolution
successfully committed
1

```
const BOX, FRUIT : TYPE
      boxA, boxB, boxC : BOX
      apples, bananas, oranges : FRUIT
      equalB : PRED [BOX, BOX]
      equalF : PRED [FRUIT, FRUIT]
      label, contains : PRED [BOX, FRU

var B, B1, B2 : BOX
    F, F1, F2 : FRUIT

axiom reflexivity : equalB(B, B) /\ eq

axiom symmetry : (equalB(B1, B2) <=> e
                 (equalF(F1, F2) <=> e
```

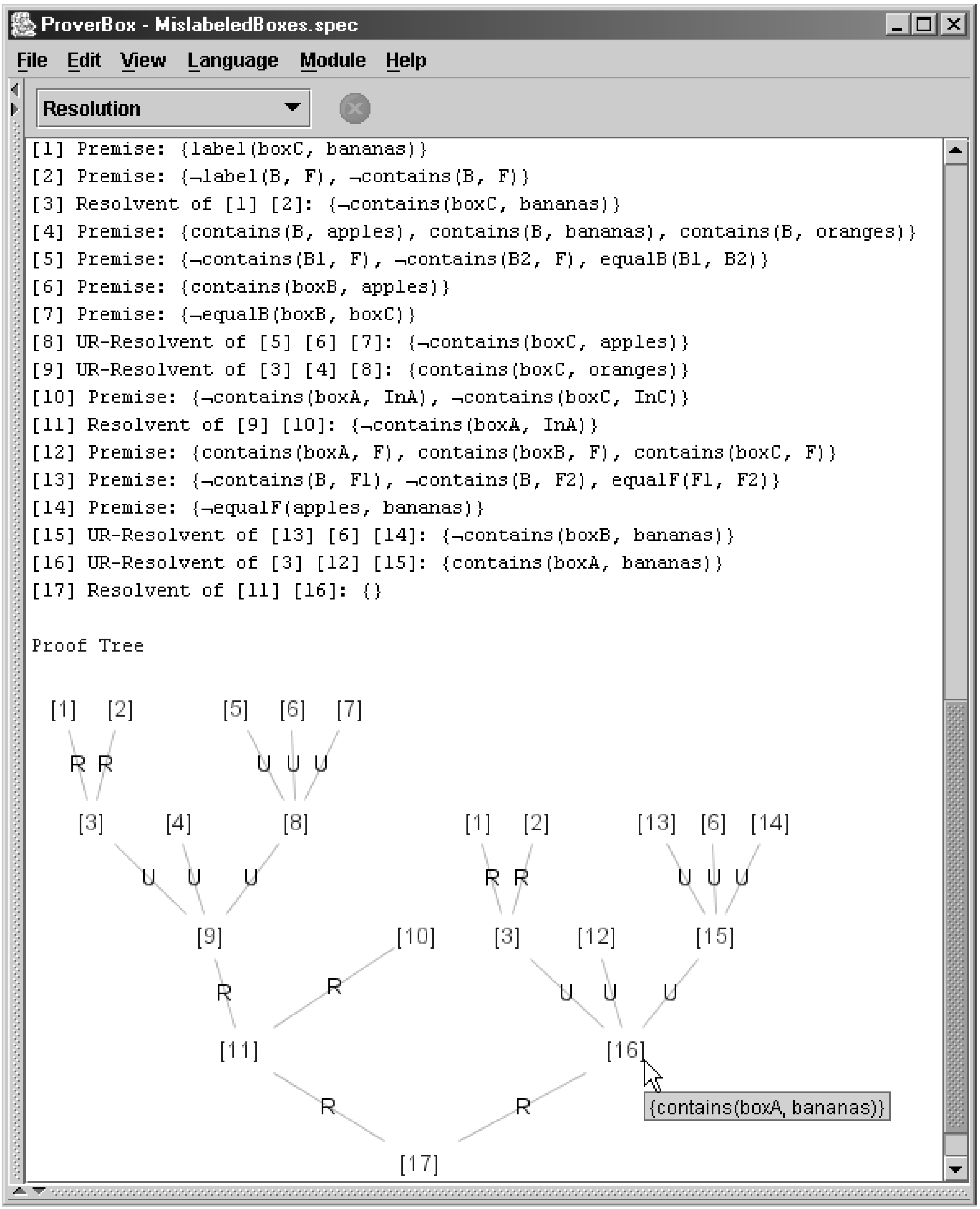
```
Proof (Resolution)

Computation Results

Satisfying Variable Assignment found:

 InA←bananas
 InC←oranges

Computing Time: 0,16 sec.
```

```
(Resolution): prove solution
 Proving using Resolution method for predicate logic... Q.E.D.
(Resolution):
```

Figure 1.27: Solving the Mislabeled Boxes Problem

If you want a verbose output, use `prove -v solution` instead, and you will be given the complete proof plus the nice proof tree displayed in Figure 1.28. As can be seen, a small tooltip appears when the mouse is over one of the tree nodes to display the corresponding formula.

Of course, this was a simple example, which we can easily solve without the computer. But you are now ready to use the ProverBox to continue exploring on your own. You might also want to try the two sample problems from Appendix A, which are a little harder, so that briefly thinking about them is probably not enough to solve them without the computer.


ProverBox - MislabeledBoxes.spec
File Edit View Language Module Help
Resolution

```
[1] Premise: {label(boxC, bananas)}
[2] Premise: {¬label(B, F), ¬contains(B, F)}
[3] Resolvent of [1] [2]: {¬contains(boxC, bananas)}
[4] Premise: {contains(B, apples), contains(B, bananas), contains(B, oranges)}
[5] Premise: {¬contains(B1, F), ¬contains(B2, F), equalB(B1, B2)}
[6] Premise: {contains(boxB, apples)}
[7] Premise: {¬equalB(boxB, boxC)}
[8] UR-Resolvent of [5] [6] [7]: {¬contains(boxC, apples)}
[9] UR-Resolvent of [3] [4] [8]: {contains(boxC, oranges)}
[10] Premise: {¬contains(boxA, InA), ¬contains(boxC, InC)}
[11] Resolvent of [9] [10]: {¬contains(boxA, InA)}
[12] Premise: {contains(boxA, F), contains(boxB, F), contains(boxC, F)}
[13] Premise: {¬contains(B, F1), ¬contains(B, F2), equalF(F1, F2)}
[14] Premise: {¬equalF(apples, bananas)}
[15] UR-Resolvent of [13] [6] [14]: {¬contains(boxB, bananas)}
[16] UR-Resolvent of [3] [12] [15]: {contains(boxA, bananas)}
[17] Resolvent of [11] [16]: {}

Proof Tree
```

Figure 1.28: Verbose Output for the Mislabeled Boxes Problem

BIBLIOGRAPHY

[1] U.M. Bubeck. *Automated Reasoning Environment ProverBox.* `http://www.ub-net.de/proverbox`.

[2] G. Sutcliffe and C. Suttner. *The TPTP Problem Library for Automated Theorem Proving.* `http://www.tptp.org`.

[3] R.M. Smullyan. *Alice in Puzzle-Land.* William Morrow and Company Inc, New York, 1982.

[4] L. Wos et al. *Automated Reasoning: Introduction and Applications, 2nd Ed.* McGraw-Hill Inc., New York, 1992.

APPENDIX A

SAMPLE PROBLEMS

SAMPLE PROBLEMS

This appendix contains two sample problems and their corresponding ProverBox specifications, which can easily be solved with the integrated resolution prover. To try for yourself, these specifications are also available from the ProverBox website [1].

Both problems have been obtained from the TPTP problem library [2] and have been converted to the ProverBox syntax.

A.1 Schubert's Steamroller

This is a very famous problem, as it has caused severe problems for many provers during the earlier days of automated theorem proving. Like a steamroller, it has just overrun less powerful provers, because it allows for unbelievably many possible resolvents. The problem is as follows:

> Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants.
> Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves.
> Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants.
> Therefore, there is an animal that likes to eat a grain eating animal.

The translation of this problem is given below in Listing A.1.

Listing A.1: Theory Specification for Schubert's Steamroller

```
CONST OBJ : TYPE
      animal, wolf, fox, bird, caterpillar, snail, grain, plant : PRED[OBJ]
      eats, much_smaller : PRED[OBJ, OBJ]
      caterpillar_food_of, snail_food_of : [OBJ -> OBJ]
      a_wolf, a_fox, a_bird, a_caterpillar, a_snail, a_grain : OBJ

VAR X, Animal, Plant, Small_animal, Other_plant, Caterpillar, Bird,
    Snail, Fox, Wolf, Grain : OBJ

AXIOM wolf_is_an_animal: animal(X) or not wolf(X)

AXIOM fox_is_an_animal: animal(X) or not fox(X)
AXIOM bird_is_an_animal: animal(X) or not bird(X)

AXIOM caterpillar_is_an_animal: animal(X) or not caterpillar(X)

AXIOM snail_is_an_animal: animal(X) or not snail(X)

AXIOM there_is_a_wolf: wolf(a_wolf)

AXIOM there_is_a_fox: fox(a_fox)

AXIOM there_is_a_bird: bird(a_bird)

AXIOM there_is_a_caterpillar: caterpillar(a_caterpillar)

AXIOM there_is_a_snail: snail(a_snail)

AXIOM there_is_a_grain: grain(a_grain)

AXIOM grain_is_a_plant: plant(X) or not grain(X)

AXIOM eating_habits: eats(Animal,Plant) or eats(Animal,Small_animal)
        or not animal(Animal) or not plant(Plant)
        or not animal(Small_animal) or not plant(Other_plant)
        or not much_smaller(Small_animal,Animal)
        or not eats(Small_animal,Other_plant)

AXIOM caterpillar_smaller_than_bird: much_smaller(Caterpillar,Bird)
        or not caterpillar(Caterpillar) or not bird(Bird)

AXIOM snail_smaller_than_bird: much_smaller(Snail,Bird)
        or not snail(Snail) or not bird(Bird)

AXIOM bird_smaller_than_fox: much_smaller(Bird,Fox)
        or not bird(Bird) or not fox(Fox)

AXIOM fox_smaller_than_wolf: much_smaller(Fox,Wolf)
        or not fox(Fox) or not wolf(Wolf)

AXIOM wolf_dont_eat_fox: not wolf(Wolf) or not fox(Fox)
        or not eats(Wolf,Fox)

AXIOM wolf_dont_eat_grain: not wolf(Wolf) or not grain(Grain)
        or not eats(Wolf,Grain)

AXIOM bird_eats_caterpillar: eats(Bird,Caterpillar) or not bird(Bird)
        or not caterpillar(Caterpillar)

AXIOM bird_dont_eat_snail: not bird(Bird) or not snail(Snail)
        or not eats(Bird,Snail)

AXIOM caterpillar_food_is_a_plant: plant(caterpillar_food_of(Caterpillar))
        or not caterpillar(Caterpillar)
```

```
AXIOM caterpillar_eats_caterpillar_food:
        eats(Caterpillar,caterpillar_food_of(Caterpillar))
        or not caterpillar(Caterpillar)

AXIOM snail_food_is_a_plant: plant(snail_food_of(Snail))
        or not snail(Snail)

AXIOM snail_eats_snail_food: eats(Snail,snail_food_of(Snail))
        or not snail(Snail)

THEOREM prove_the_animal_exists: exists Animal:OBJ,Grain:OBJ,
          Grain_eater:OBJ.
          animal(Animal) and animal(Grain_eater) and grain(Grain)
          and eats(Animal,Grain_eater) and eats(Grain_eater,Grain)
```

A.2 The Jobs Puzzle

This problem is also well-known, as it is one of the central puzzles of Larry Wos' classic book on automated reasoning [4]:

> There are four people: Roberta, Thelma, Steve, and Pete. Among them they hold eight different jobs. Each holds exactly two jobs.
> The jobs are: chef, guard, nurse, telephone operator, police officer (either gender), teacher, actor, and boxer. The job of a nurse is held by a male. The husband of the chef is the telephone operator. Roberta is not a boxer. Pete has no education past the ninth grade. Roberta, the chef and the police officer went golfing together.
> Question : Who holds which job?

The following Listing A.2 gives the corresponding ProverBox theory specification.

Listing A.2: Theory Specification for the Jobs Puzzle

```
CONST OBJ : TYPE
      educated : PRED[OBJ]
      equal_jobs : PRED[OBJ,OBJ]
      equal_people : PRED[OBJ,OBJ]
      female : PRED[OBJ]
      has_job : PRED[OBJ,OBJ]
      husband : PRED[OBJ,OBJ]
      male : PRED[OBJ]
      actor : OBJ
      boxer : OBJ
      chef : OBJ
      guard : OBJ
      nurse : OBJ
      operator : OBJ
      pete : OBJ
      police : OBJ
      roberta : OBJ
      steve : OBJ
      teacher : OBJ
      thelma : OBJ

VAR U : OBJ
    X : OBJ
    Y : OBJ
    Z : OBJ

AXIOM reflexivity_for_equal_people: equal_people(X,X)

AXIOM reflexivity_for_equal_jobs: equal_jobs(X,X)

AXIOM symmetry_of_equal_people: not equal_people(X,Y) or equal_people(Y,X)

AXIOM symmetry_of_equal_jobs: not equal_jobs(X,Y) or equal_jobs(Y,X)

AXIOM roberta_not_thelma: not equal_people(roberta,thelma)

AXIOM roberta_not_pete: not equal_people(roberta,pete)

AXIOM roberta_not_steve: not equal_people(roberta,steve)

AXIOM pete_not_thelma: not equal_people(pete,thelma)

AXIOM pete_not_steve: not equal_people(pete,steve)

AXIOM chef_not_guard: not equal_jobs(chef,guard)

AXIOM chef_not_nurse: not equal_jobs(chef,nurse)

AXIOM chef_not_operator: not equal_jobs(chef,operator)

AXIOM chef_not_police: not equal_jobs(chef,police)

AXIOM chef_not_actor: not equal_jobs(chef,actor)

AXIOM chef_not_boxer: not equal_jobs(chef,boxer)

AXIOM chef_not_teacher: not equal_jobs(chef,teacher)

AXIOM guard_not_nurse: not equal_jobs(guard,nurse)

AXIOM guard_not_operator: not equal_jobs(guard,operator)

AXIOM guard_not_police: not equal_jobs(guard,police)

AXIOM guard_not_actor: not equal_jobs(guard,actor)

AXIOM guard_not_boxer: not equal_jobs(guard,boxer)

AXIOM guard_not_teacher: not equal_jobs(guard,teacher)

AXIOM nurse_not_operator: not equal_jobs(nurse,operator)
AXIOM nurse_not_police: not equal_jobs(nurse,police)

AXIOM nurse_not_actor: not equal_jobs(nurse,actor)

AXIOM nurse_not_boxer: not equal_jobs(nurse,boxer)

AXIOM nurse_not_teacher: not equal_jobs(nurse,teacher)

AXIOM operator_not_police: not equal_jobs(operator,police)

AXIOM operator_not_actor: not equal_jobs(operator,actor)

AXIOM operator_not_boxer: not equal_jobs(operator,boxer)

AXIOM operator_not_teacher: not equal_jobs(operator,teacher)

AXIOM police_not_actor: not equal_jobs(police,actor)

AXIOM police_not_boxer: not equal_jobs(police,boxer)

AXIOM police_not_teacher: not equal_jobs(police,teacher)

AXIOM actor_not_boxer: not equal_jobs(actor,boxer)

AXIOM actor_not_teacher: not equal_jobs(actor,teacher)

AXIOM boxer_not_teacher: not equal_jobs(boxer,teacher)

AXIOM nurse_is_male: not has_job(X,nurse) or male(X)

AXIOM actor_is_male: not has_job(X,actor) or male(X)

AXIOM chef_is_female: not has_job(X,chef) or female(X)

AXIOM nurse_is_educated: not has_job(X,nurse) or educated(X)

AXIOM teacher_is_educated: not has_job(X,teacher) or educated(X)

AXIOM police_is_educated: not has_job(X,police) or educated(X)

AXIOM chef_is_not_also_police: not has_job(X,chef)
        or not has_job(X,police)

AXIOM males_are_not_female: not male(X) or not female(X)

AXIOM everyone_male_or_female: male(X) or female(X)

AXIOM husband_is_male: not husband(X,Y) or male(Y)

AXIOM wife_is_female: not husband(X,Y) or female(X)

AXIOM husband_of_chef_is_operator1: not has_job(X,chef)
        or not has_job(Y,operator) or husband(X,Y)
```

```
AXIOM husband_of_chef_is_operator2: not has_job(X,chef)
        or has_job(Y,operator) or not husband(X,Y)

AXIOM each_job_held_once: not has_job(X,Z) or not has_job(Y,Z)
        or equal_people(X,Y)

AXIOM each_has_maximum_of_two_jobs: not has_job(Z,U) or not has_job(Z,X)
        or not has_job(Z,Y) or equal_jobs(U,X) or equal_jobs(U,Y)
        or equal_jobs(X,Y)

AXIOM every_job_is_used: has_job(roberta,X) or has_job(thelma,X)
        or has_job(pete,X) or has_job(steve,X)

AXIOM everyone_works: has_job(X,chef) or has_job(X,guard)
        or has_job(X,nurse) or has_job(X,operator) or has_job(X,police)
        or has_job(X,teacher) or has_job(X,actor) or has_job(X,boxer)

AXIOM pete_is_not_educated: not educated(pete)

AXIOM roberta_is_not_chef: not has_job(roberta,chef)

AXIOM roberta_is_not_boxer: not has_job(roberta,boxer)

AXIOM roberta_is_not_police: not has_job(roberta,police)

AXIOM steve_is_male: male(steve)

AXIOM pete_is_male: male(pete)

AXIOM roberta_is_female: female(roberta)

AXIOM thelma_is_female: female(thelma)

THEOREM find_who_has_each_job: Exists X1:OBJ,X2:OBJ,X3:OBJ,X4:OBJ,X5:OBJ,
          X6:OBJ,X7:OBJ,X8:OBJ.
          (has_job(X1,chef) and has_job(X2,guard) and has_job(X3,nurse)
          and has_job(X4,operator) and has_job(X5,police)
          and has_job(X6,teacher) and has_job(X7,actor)
          and has_job(X8,boxer))
```