Chapter 2

A Quick Overview

The NUPRL 5 system is organized as a collection of communicating processes that are centered around a *library*, which contains definitions, theorems, inference rules, tactics, structure objects, comments, etc. Inference engines (*refiners*), user interfaces (*editors*), rewrite engines (*evaluators*), and *translators* are started as independent processes that can connect to the library at any time.

2.1 Preparation

We assume that the NUPRL 5 system has already been installed (see Section 3.2.1) and can be found in the directory `/home/nuprl/nuprl5`. We also assume that the NUPRL 5 binaries can be found in `/home/nuprl/bin/` and that the NUPRL fonts are installed in the directory `/home/nuprl/fonts/bdf`.

Make sure that your Unix path includes `/home/nuprl/bin/` and that the X-server has NUPRL's fonts loaded. Create a file `.nuprl.config` in your home directory with the following entries:

```
(libhost "HOSTNAME")
(dbpath "/home/nuprl/nuprl5/NuprlDB")
(libenv "standard")
```

The HOSTNAME for the `libhost` configuration should be the name of the host running the library process. The values for `dbpath` and the `libenv` describe the physical and logical location of the standard library. Optional settings like specific colors and fonts for the NUPRL windows may also be given in that file.

Copy the file `/home/nuprl/nuprl5/mykeys.macro` to your home directory. NUPRL reads the file `~/mykeys.macro` to determine the key bindings that will be used in various windows. You need it to initialize the key combinations described in this manual and to customize them according to your own preferences later.

2.2 Running NUPRL 5

For the basic NUPRL 5 configuration you need to run three processes: a library, an editor, and a refiner. The library (`nulib`) should be started first. The editor (`nuedd`) and the refiner (`nuref`) can then be started in any order. Generally it is a good idea to run these processes in separate emacs frames: there will be interactive top loops, so editing capabilities are sometimes useful.

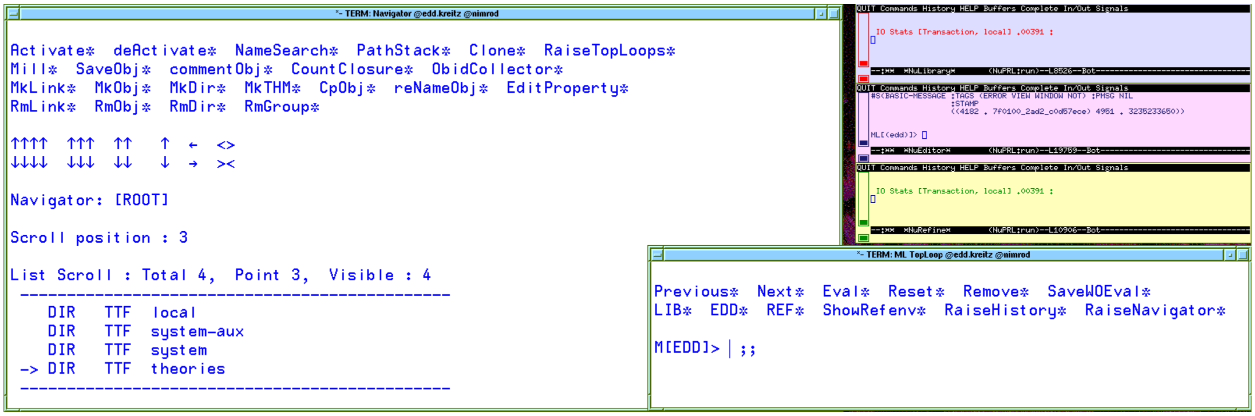


Figure 2.1: Initial NUPRL 5 screen

Once the processes have started, entering (top) at the Lisp prompt will start the ML system.

```
USER(1): (top)
```

Enter go. at the prompt to initialize the corresponding NUPRL process.

```
ML[(ORB)]> go.
```

It is important to initialize the library before the editor and the refiner. The editor process will take few minutes and then pop up two windows: a *navigator* and a *top loop*.

Figure 2.1 shows a typical initial NUPRL 5 screen. The window on the left is the NUPRL 5 navigator. The three emacs windows on the upper right run interactive top loops for the library, editor and the refiner. The NUPRL 5 top loop is shown in the window below. In contrast to the corresponding emacs top loops, the NUPRL 5 top loop incorporates the NUPRL 5 term editor. It is better suited for editing object-level terms but do not support the full editing capabilities of emacs. Unless there is a need to interact with the top ML level, one usually iconifies these four windows to create some space for the windows that will pop up while working with the system.

2.3 Using the Navigator

The navigator is the main user interface of NUPRL 5. It can be used to browse the library and to create, delete, or edit objects by initiating the appropriate editors. Figure 2.2 shows navigator window shown in its initial state. As in most NUPRL 5 windows, the upper part of the navigator window contains several *buttons*, which are indicated by a * at the end of a word. Clicking a button with the left mouse will trigger some action or pop up a template to be filled in. The lower part of the navigator window shows the *current directory* (here ROOT) and a listing of the *type*, *status*, and *name* of some of the objects in the directory. There is also a distinguished object, the *nav point*), which is marked by an arrow (the *navigation pointer*). When the *edit point* is in the Scroll position field (use the left mouse), the arrow keys on the keyboard can be used to move the through the directory tree.

↑	UP	move navigation pointer one step up
↓	DOWN	move navigation pointer one step down
←	LEFT	move navigation pointer to next higher directory
→	RIGHT	open object at navigation pointer in a new window (enter sub-directory if object is of type DIR)

```
- TERM: Navigator
Activate*  deActivate*  NameSearch*   PathStack*  Clone*  RaiseTopLoops
Mill*  SaveObj*  commentObj*  CountClosure*  ObidCollector*
MkLink*  MkObj*  MkDir*  mkTHM*  CpObj*  reNameObj*  EditProperty*
RmLink*  RmObj*  RmDir*  RmGroup*

↑↑↑↑  ↑↑↑  ↑↑   ↑  ←  <>
↓↓↓↓  ↓↓↓  ↓↓   ↓  →  ><

Navigator: [ROOT]

Scroll position : 0

List Scroll : Total 4,  Point 0,  Visible : 4
 ---------------------------------------------
 -> DIR   TTF  theories
    DIR   TTF  system-aux
    DIR   TTF  local
    DIR   TTF  system
 ---------------------------------------------
```

Figure 2.2: NUPRL 5 Navigator

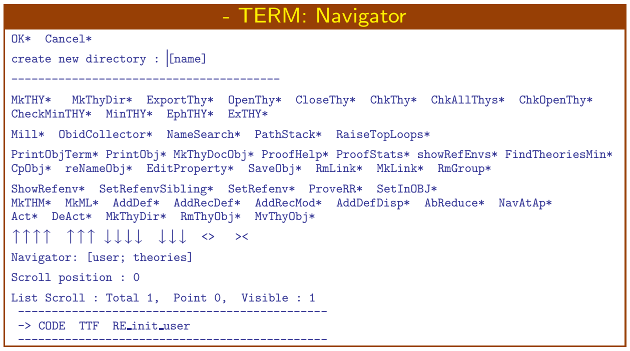
The navigator window also contains arrow buttons for faster navigation through a directory. ↑↑ and ↓↓ scroll half a screen, ↑↑↑ and ↓↓↓ scroll a full screen, and ↑↑↑↑ and ↓↓↓↓ move to the top and bottom of the directory. In addition to buttons and arrow keys, there are also a variety of special key combinations that can be used to manipulate objects in the library. These are described in Chapter 4.

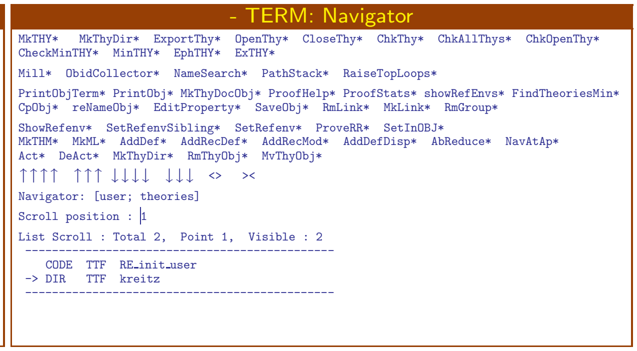
To begin working with the NUPRL system, one will usually move into the `theories` directory. Leaving the initial state will cause additional buttons to become visible.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
Activate*  DeActivate*   MkObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [theories]

Scroll position : 0

List Scroll : Total 13,  Point 0,  Visible : 10
 ---------------------------------------------
 -> DIR   TTF  General
    DIR   TTF  pre-utils
    DIR   TTF  initial reference environment
    DIR   TTF  standard
    TERM  TTF  check theories_control
    DIR   TTF  Obvious
    DIR   TTF  user
    DIR   TTF  detritus
    DIR   TTF  utils
    DIR   TTF  .help
 ---------------------------------------------
```

Usually, NUPRL users will work within their own sub-directory within the directory `user` and occasionally browse the `standard` sub-directory, which contains the NUPRL type theory and a few standard libraries of formalized mathematical knowledge.

A new user-directory can be created by clicking the MkThyDir* button. This will open a template for entering the name of the new directory and move the edit point to the [name] slot.

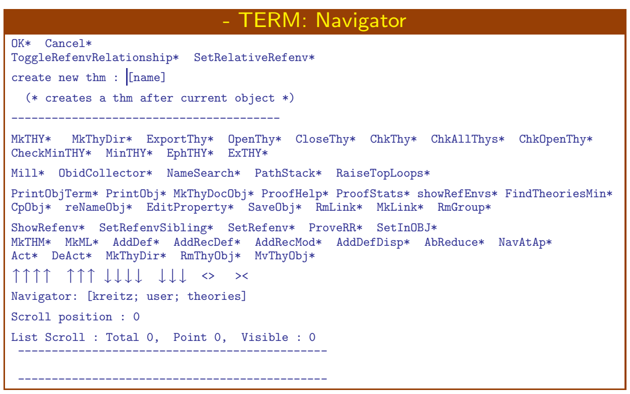


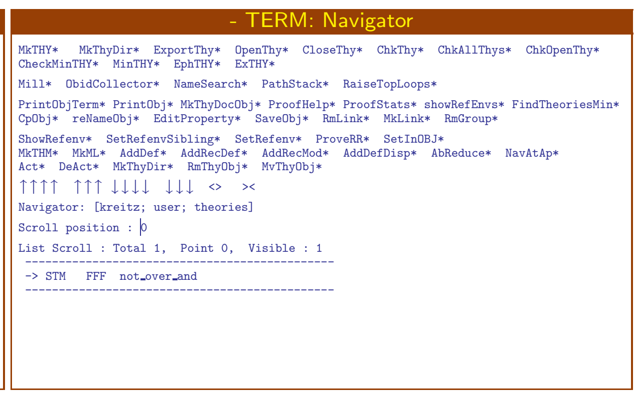


Enter the name of the directory and click OK* (or press ↵ twice). This will create the new directory object, place it immediately below the previous nav point, and move the navigation pointer to it, as shown in the right window.

2.4 Creating Theorem Objects

Before one can prove a theorem in Nuprl one has to create an object that contains it. Clicking the MkTHM* button (after moving into the user directory) will open a template for entering the name and kind of a new library object.[1] The edit point will be in the [name] slot.





As example theorem we take ⌞∀A,B:ℙ. (¬A) ∨ (¬B) ⇒ ¬(A ∧ B)⌟, one of the DeMorgan laws for propositional logic. Enter the name of the theorem not_over_and and click OK*, or press ↵ twice. This will create a new statement object named not_over_and.

2.5 Proving Theorems

To state and prove a theorem, one has to open the corresponding object. Pressing the right arrow key when the nav point is a statement object pops up a new window that shows the contents of this object. If the theorem has not been stated yet, there will be a [goal] slot in the upper part of the window and the *rule slot* next to the keyword BY below will be empty. The # in the upper left corner means that the theorem is not complete yet, while the top next to it indicates that the top node of the proof tree is being displayed.

[1]Right now, the creation of a theorem object requires the current directory to contain at least one object.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*
↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><
Navigator: [kreitz; user; theories]
Scroll position : 0
List Scroll : Total 1,  Point 0,  Visible : 1
 ---------------------------------------------
 -> STM   FFF  not_over_and
 ---------------------------------------------
```

```
- PRF: not_over_and
# top
[goal]
BY
```

Move the mouse cursor into the new window and click left in the `[goal]` slot. This initializes the NUPRL 5 *term editor* in this slot. The goal is now entered in a structural top-down fashion. Entering ␣all↵␣ creates the template for the universal quantifier.

```
- PRF: not_over_and
# top
∀[var]:[type]. [prop]
BY
```

The `[var]` slot of the `all` template is a *text slot* and will not be interpreted. Entering ␣A↵␣ inserts the character '`A`' in the variable slot and moves the edit point to the next slot.

The `[type]` and `[prop]` slots are *term slots*, which means that input will be interpreted and may open new templates. Entering ␣prop↵i↵all↵␣ inserts the *propositional universe* term (whose name `prop` has nothing to do with the `prop` in the `[prop]` place-holder) into the `[type]` slot, and another template for the universal quantifier in the `[prop]` slot.

```
- PRF: not_over_and
# top
∀A:ℙ. ∀[var]:[type]. [prop]
BY
```

Entering ␣B↵prop↵i↵␣ fills the second quantifier. Notice that the two quantifiers get contracted, as `A` and `B` have the same type ℙ.

```
- PRF: not_over_and
# top
∀A,B:ℙ. [prop]
BY
```

Entering the rest of the theorem is straightforward. Typing ␣implies↵␣ creates the implication template. Entering ␣or↵not↵A↵not↵B↵␣ generates `(¬A) ∨ (¬B)` and `¬(A ∧ B)` is generated by ␣not↵and↵A↵B↵␣.

Before a statement can be used in a proof it must be *committed to the permanent library*. This can be done by using the key combination ⟨C-M-g⟩.[2]

```
- PRF: not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY
```

[2] Actually, pressing ⟨C-M-g⟩ is not necessary when entering a statement for the first time, since it will be committed after the first execution of a proof tactic. However, subsequent modifications of the statement will *not* be committed without pressing ⟨C-M-g⟩. Thus it is better to make using this key combination a habit.

To begin with the proof, press the down arrow key once or use the left mouse to move the edit point into the empty rule slot next to the `BY`.

```
- PRF: not_over_and
# top
∀A,B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY |
```

The most common proof tactic is the *single step decomposition* tactic `D` (see chapter 8 for details). It requires as argument the index of the proof hypothesis to which it shall be applied or a zero if it shall be applied to the conclusion. To enter this tactic, type ⌞`D 0` ⟨C-↵⟩⌟.

Pressing ⟨C-↵⟩ when in a rule slot refines the goal at current node with the corresponding tactic. In this (synchronous) mode you have to wait for the refinement process to be complete. Pressing ⟨C-M-↵⟩ instead initializes *asynchronous refinement*, which allows you to continue working while the proof goal is being refined. Once a refinement is completed the proof window gets updated and shows the subgoals that were generated by applying the tactic.

```
- PRF: not_over_and
# top
∀A,B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY D 0
# 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY
# 2
.....wf.....
ℙ ∈ 𝕌'
BY
```

To prove the first subgoal, press the down arrow key. This will move into the first sub-node of the theorem, indicated by a `top 1`.

```
- PRF: not_over_and
# top 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A)  ∨  (¬B))  ⇒  (¬(A  ∧  B)))
BY |
```

To move into the second subgoal, press the right arrow key. As indicated by the `.....wf.....` annotation, the second subgoal is a *well-formedness goal*, stating that ℙ is a well-formed type theoretical expression. Most goals of this kind can be dealt with automatically. Typing ⌞`Auto` ⟨C-↵⟩⌟ will complete this subproof: no subgoals are generated and the status marker changes into a `*`.

```
- PRF: not_over_and
* top 2
.....wf.....
ℙ ∈ 𝕌'
BY Auto
```

Pressing the left arrow key will bring you back into the first subgoal. Alternatively you can press ⟨C-M-j⟩, which causes the editor to jump to the next unproven subgoal.

Proving the first subgoal requires more efforts. Typing ⌞`Auto` ⟨C-↵⟩⌟ will decompose the universal quantifier, the implication, and deal with the corresponding well-formedness subgoals. The result will be a subgoal with two additional hypotheses: a declaration of the variable `B` and the assumption `(¬A) ∨ (¬B)`.

To prove this goal both the conclusion (`D 0`) and the third hypothesis (`D 3`) need to be decomposed. `Auto` does neither of these two steps automatically but can deal with the resulting subgoals. These steps can be combined into a single one by using the tactical `THEN`.

```
- PRF: not_over_and
# top 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY
```

Clicking the left mouse next to the `BY` allows you to enter the tactic into an empty rule slot without having to move into the corresponding sub-node. Typing ⌞`D 0 THEN D 3 THEN Auto` ⟨C-↵⟩⌟ results in a complete proof of the subgoal.

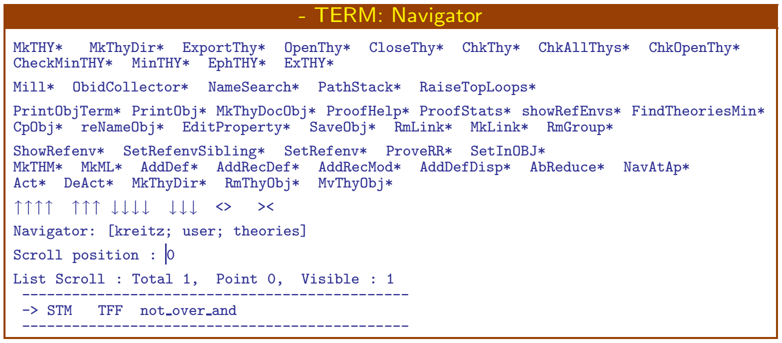
```
- PRF: not_over_and
* top 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto
```

Using the up arrow key will get you back to the parent node, which now shows the complete proof.

```
- PRF: not_over_and
* top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
* 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto

* 2
.....wf.....
ℙ ∈ 𝕌'
BY Auto
```

The proof has already been saved in the library. To close the proof window press ⟨`C-Q`⟩. This key combination will always close the current window.

If a theorem shall be used as lemma in other proofs, it has to be *activated*. Many tactics use a list of active theorems which are searched through when attempting to prove a theorem automatically. For this purpose you have to click the `Act*` button, which changes the object status of `not_over_and` from `FFF` to `TFF`.



A theorem can also be activated by closing the proof window with ⟨C-Z⟩ instead of ⟨C-Q⟩. This will cause Nuprl to create the *extract term* of the proof (a λ-term describing its computational content) and to store it along with the theorem object. The status of that theorem will then be `TTF`.

2.6 Adding Definitions

Besides proving theorems, the most common activity in mathematics is introducing new concepts, which are defined in terms of already existing ones. This makes the formulation of theorems crisper and easier to comprehend. Nuprl supports such an enhancement of the formal language through a definition mechanism. This mechanism allows a user to introduce new terms that are definitionally equal to other terms.

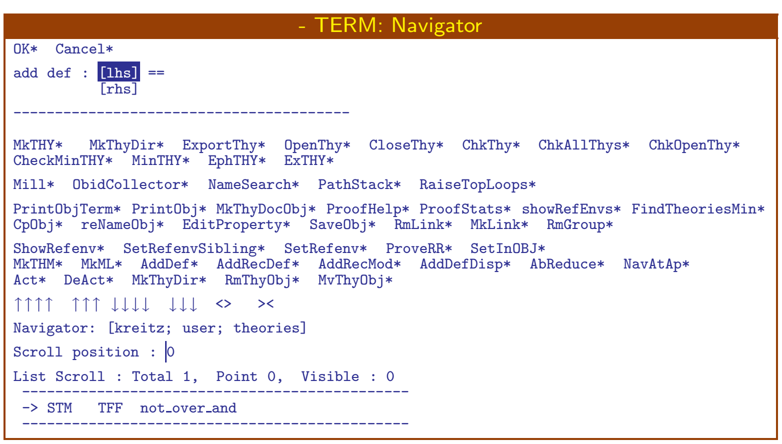
As an example consider the `∃!` quantifier, which states the existence of a unique element `x∈T` that satisfies a property `P`. A typical definition for this quantifier is the following:

`∃!x:T. P[x]   ≡   ∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T).`

This definition actually presents two aspects of a newly defined term. It first states that a new abstract term, say `exists_uni` is to be introduced, which has two subterms (`T` and `P`) and binds occurrences of `x` in `P`. Secondly, it states that the term is to be presented as `∃!x:T. P`.

In Nuprl a formal definition requires the creation of two new objects: an *abstraction*, which defines the abstract term, and a *display form*, which defines its syntactical appearance (see Chapters 7.1 and 7.2). In addition to that, it is advisable to prove a *well-formedness theorem*, which describes the type of the newly introduced term. All three objects can be created with the `AddDef` mechanism.

To initialize this mechanism, click the `AddDef*` button with the left mouse. This will open a template for defining the abstract term.



To enter the new term on the left hand side of the definition, you have to provide its name (or *object identifier*) and a list of subterms. The ∃! has two subterms, `T` and `P`, and binds one variable in the second. To create a template for entering the details, type ⌞`exists_uni(0;1)`⌟.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni\(0;1
          [rhs]
```

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni([term]; [binding].[term]) ==
          [rhs]
```

This tells the system to create a term called `exists_uni`, whose first term has no bound variables and whose second term has one bound variable The template, shown on the right, appears as soon as you have entered the right parenthesis that closes the subterm list. Pressing ↵ then moves the edit point into the first term slot.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni([term]; [binding].[term]) ==
          [rhs]
```

Enter ⌞`T` ↵ `x` ↵ `so_var1` ↵⌟. This puts `T` into the first term slot, makes `x` the binding variable in the second, and states that the second term will be a *second order variable* of arity 1 (see Chapter 7.1).

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.|[variable-id][[term]]) ==
          [rhs]
```

Note that NUPRL's term editor treats any unknown name as variable name, while names that can be linked to (active) object identifiers (and display forms) will cause the corresponding template to appear. Thus `T` will be inserted as variable name, while `so_var1` creates a new template. `x` had been entered into a `binding` slot and is thus viewed as variable.

Should you mistype `so_var1` and actually enter an identifier that is unknown to NUPRL, say `sovar1`, the identifier will appear as variable name in the term slot.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.sovar1) ==
          [rhs]
```

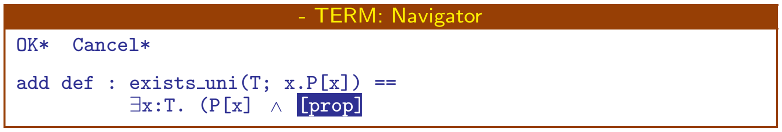
There are two ways to correct that mistake. You may *delete* the term ⌞`sovar1`⌟ by clicking LEFT over it, pressing ⟨M-P⟩ to mark the full term and then ⟨C-K⟩ to cut it. Afterwards you enter `so_var1` ↵ to get the correct template. Note that ⟨C-k⟩ saves the term in a cut buffer and that you can paste this term with ⟨C-Y⟩. To delete a term without saving it, you need to press ⟨C-C⟩.

Alternatively, you may use NUPRL's generic *undo* command ⟨C-_⟩, which will restore the empty term slot. Move the edit cursor into that slot either by pressing ↵ or by clicking LEFT over it and then enter enter `so_var1` ↵.

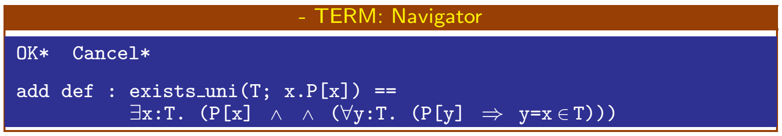
Entering ⌞`P` ↵ `x` ↵⌟ next generates `P[x]` and moves the edit point into the right hand side of the definition.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.P[x]) ==
          [rhs]
```

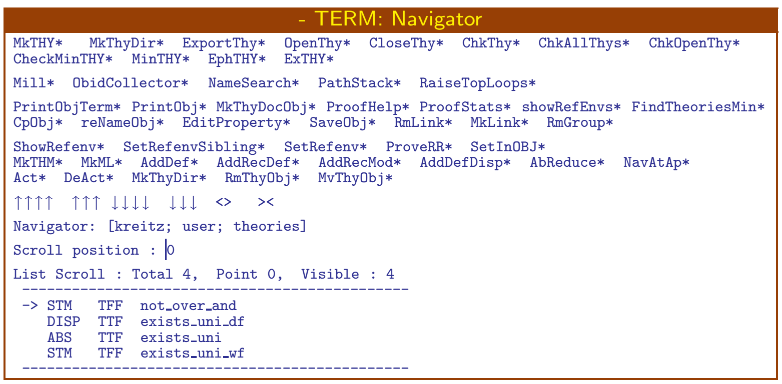
To enter the right hand side of the definition, you have to proceed in a structural top-down fashion. Type ␣exists ↵ x ↵ T ↵ and ↵ so_var1 ↵ P ↵ x ↵␣



and then ␣all ↵ y ↵ T ↵ implies ↵ so_var1 ↵ P ↵ y ↵ equal ↵ x ↵ y ↵ T ↵␣.

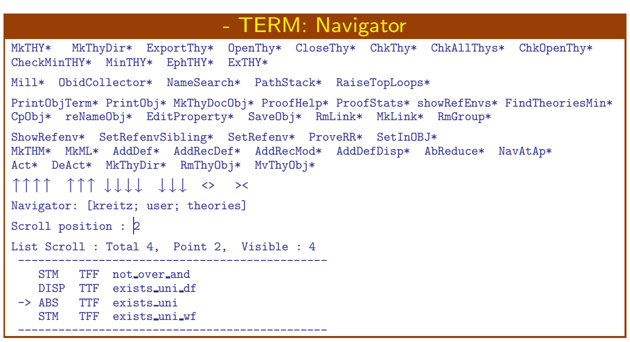


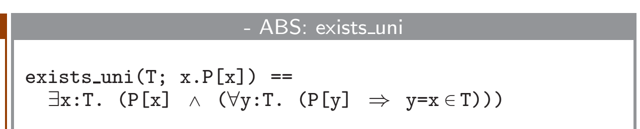
The definition is now complete. To save it to the library click OK* or press ↵ again. This closes the AddDef template and creates a display form exists_uni_df of kind DISP, an abstraction object exists_uni of kind ABS, and a well-formedness theorem exists_uni_wf of kind STM. The navigation pointer is still where it has been before.



The abstraction object contains exactly what has been typed into the AddDef template and usually does not have to be edited anymore. The display form has been generated from the left hand side of the AddDef template and currently causes the term exists_uni to be displayed in exactly this way. The well-formedness theorem is empty but already activated, which enables the general tactics to access it whenever they have to deal with exists_uni.

To view the abstraction, move the navigation pointer down 2 steps then open the object exists_uni by pressing the right arrow key.





The abstraction object shows on the right hand side the term that defines the meaning of exists_uni(T;x.P[x]) and on the left hand side the form in which exists_uni(T;x.P[x]) is currently displayed. Right now, this is identical to the abstract term form. To close the abstraction object again press ⟨C-q⟩.

To change the appearance of the term `exists_uni(T;x.P[x])` to `∃!x:T. P[x]` you have to edit the accompanying display form. For this purpose, move the nav point one step up and open the object `exists_uni_df`.

- TERM: Navigator

```
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 1

List Scroll : Total 4,  Point 1,  Visible : 4
 ---------------------------------------------
    STM   TFF  not_over_and
 -> DISP  TTF  exists_uni_df
    ABS   TTF  exists_uni
    STM   TFF  exists_uni_wf
 ---------------------------------------------
```

- DISP: exists_uni_df

```
EdAlias exists_uni ::
exists_uni(<T:T:*>;<x:var>.<P:P:*>) ==
  exists_uni(<T>; <x>.<P>)
```

The display form consists of a list of attributes (in this case only the name that can be used to open the template), a template that determines the outer appearance of a term, and the term that is to be represented by that template. Both the template and the term contain slots that are marked with `<..>` and describe the name of a placeholder, a description that will appear whenever the template is initiated, and information about parenthesizing. Chapter 7.2 describes how to create display forms from scratch, but usually copying and pasting is sufficient.

To edit the template click mouse over the `exist_uni` in the second line and erase the text `exist_uni(` using backspace and delete keys. Notice that you can't delete the slot `<T:T:*>` with these keys. Type ⌞⟨C-#⟩163!⌟ to generate the ∃ symbol (see table 5.1 on page 73 for a list of all special characters) and the exclamation mark. Delete the semicolon and the dot between the term slots and also the right parenthesis.

- DISP: exists_uni_df

```
EdAlias exists_uni ::
∃!<T:T:*><x:var><P:P:*> ==
  exists_uni(<T>; <x>.<P>)
```

To rearrange the order of the slots click left over `<T:T:*>` press ⟨M-p⟩ to mark the full slot, and ⟨C-k⟩ to cut it. Then move the mouse to the immediate right of `<x:var:*>` and press ⟨C-y⟩. Add a colon between `<x:var:*>` and `<T:T:*>`, a dot and a space between `<T:T:*>` and `<P:P:*>`.

- DISP: exists_uni_df

```
EdAlias exists_uni ::
∃!<x:var>:<T:T:*>. <P:P:*> ==
  exists_uni(<T>; <x>.<P>)
```

In principle, the display form is now complete. However it is advisable to edit the slot description of `T` and `P`. Click right of the second `T` in `<T:T:*>`, remove it and enter ⌞type⌟ instead. In the same way change the second `P` in `<P:P:*>` to `prop`. This makes sure that meaningful descriptions for theses slots will show up whenever the template for `exists_uni` is opened.

- DISP: exists_uni_df

```
EdAlias exists_uni ::
∃!<x:var>:<T:type:*>. <P:prop:*> ==
  exists_uni(<T>; <x>.<P>)
```

To commit the completed display form to the library, press ⟨C-Z⟩. This will also close the window again. *Do not use* ⟨C-Q⟩ unless you want all your changes to be ignored. ⟨C-Q⟩ will always close a window without saving its contents to the library.

To check the display form, open the abstraction `exists_uni` again. You will notice that the display of the abstract term `exists_uni(T;x.P[x])` has now been replaced by `∃!x:T. P[x]`.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*
Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*
PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*
ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*
↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><
Navigator: [kreitz; user; theories]
Scroll position : 2
List Scroll : Total 4,  Point 2,  Visible : 4
 ---------------------------------------------
    STM   TFF  not_over_and
    DISP  TTF  exists_uni_df
 -> ABS   TTF  exists_uni
    STM   TFF  exists_uni_wf
 ---------------------------------------------
```

```
- ABS: exists_uni

∃!x:T. P[x] ==
  ∃x:T. (P[x] ∧ (∀y:T. (P[y] ⇒ y=x∈T)))
```

The abstraction and the display form are sufficient for using a newly defined term in formal theorems and other abstractions. However, many proofs involving an instance of `exists_uni` will involve checking the type of this term. This can be done automatically if *well-formedness theorem* is provided describes the type of the newly defined term.

A unique existence quantifier `∃!x:T. P[x]` is a proposition, provided that `T` is a type and `P` is a predicate on `T`. In other words, the type of `∃!x:T. P[x]` is ℙ if `T` is an element of the universe 𝕌 of types and `P ∈ T→ℙ`. Formally, you need to prove `∀T:𝕌. ∀P:T→ℙ. ∃!x:T.P[x] ∈ ℙ`.

To state this theorem, open the object `exists_uni_wf` and enter

⌞`all` ↵ `T` ↵ `univ` ↵ `i` ↵ `all` ↵ `P` ↵ `fun` ↵ `T` ↵ `prop` ↵ `member` ↵⌟

into the goal template.

```
- PRF: exists_uni_wf
# top
∀T:𝕌. ∀P:T→ℙ. ([term] ∈ [type])
BY
```

Continue with ⌞`exists_uni` ↵ `x` ↵ `T` ↵ `so_apply1` ↵ `P` ↵ `x` ↵ `prop` ↵ ⟨C-M-g⟩⌟ to complete the goal and commit it to the library.

The proof of this theorem is fairly simple, since it can be handled almost completely by Nuprl's tactic `Auto`. However, since `Auto` does not unfold a definition unless it is explicitly declared as automatically unfoldable, you need to unfold the definition of `exists_uni` in the first step.

Typing ⌞`Unfold ‘exists_uni‘ 0 THEN Auto`⌟ into the rule slot will result in a proof of the well-formedness theorem for `exists_uni`.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*
Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*
PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*
ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*
↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><
Navigator: [kreitz; user; theories]
Scroll position : 3
List Scroll : Total 4,  Point 3,  Visible : 4
 ---------------------------------------------
    STM   TFF  not_over_and
    DISP  TTF  exists_uni_df
    ABS   TTF  exists_uni
 -> STM   TFF  exists_uni_wf
 ---------------------------------------------
```

```
- PRF: exists_uni_wf
# top
∀T:𝕌. ∀P:T→ℙ.  (∃!x:T. P[x] ∈ ℙ)
BY Unfold ‘exists_uni‘ 0 THEN Auto
```

This completes all necessary steps for adding a unique existence quantifier to the formal language of `Nuprl`. Close the well-formedness theorem with ⟨C-Q⟩.

2.7 Printing Snapshots

To print a snapshot of a particular object, simply click the `PrintObj*` button. This will create a print representation of the object at the nav point and write it into a file `~/nuprlprint/OBJECT-NAME.prl`. This file can be inspected with any 8bit capable editor that has the Nuprl fonts loaded. It will also create a LaTeX-version and write it to `~/nuprlprint/OBJECT-NAME.tex`. The directory `~/nuprlprint` must already exist. Otherwise clicking the `PrintObj*` button will result in an error.

The button `PrintCollection` will create a print representation of a whole range of objects, while the buttons `PrintThyLong` and `PrintThyShort` will create a print representation of the directory at the nav point. The long version shows the complete proofs of theorems while the short version only prints the theorem statement and the extract of the proof, that is a term representing its computational content.

2.8 Troubleshooting

Since the Nuprl library never destroys information, typos and almost all commands can be undone by entering the key combination ⟨C-_⟩. The undo history, however, is limited and recovering from an error that was made many steps ago is more difficult.

Should you accidentally close the navigator window you may open it again by typing the command `win.` into the editor process top loop. This will open all the windows of the initial screen.

If the editor hangs for an unusually long time, one of the three main processes (editor, refiner, or library) may be broken. In this case there will be an error message in the corresponding (emacs) top loop. Often, evaluating the expression `(fooe)` will recover the process. Sometimes, typing the Lisp command `:cont` will help as well. In the worst case, kill the process and then restart it. The library process will detect the link going dead and clean it up automatically.

2.9 Shutting Nuprl down

In principle, there is no need to shutdown Nuprl, as all data are saved immediately and updates may be integrated by by loading patches into the running process. However the Nuprl 5 are quite demanding as far as cpu and memory are concerned.

To shutdown gracefully you should first close the refiner and the editor and then the library. Enter `stop.` at the corresponding ML prompt.

```
ML[(ORB)]> stop.
```

As a result, the editor and refiner will communicate to the library that they will disconnect now and then stop the respective ML and Lisp processes. The library process will cleanly shut down the knowledge base and then stop as well.

Instead of shutting down gracefully you may also simply kill all three processes to stop Nuprl 5, although this is not recommended.