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

\begin{verbatim}
USER(1): (top)
\end{verbatim}

Enter \texttt{go.} at the prompt to initialize the corresponding Nuprl process.

\begin{verbatim}
ML[(ORB)]> go.
\end{verbatim}

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 \textit{current directory} (here \texttt{ROOT}) and a listing of the \textit{type}, \textit{status}, and \textit{name} of some of the objects in the directory. There is also a distinguished object, the \textit{nav point}, which is marked by an arrow (the \textit{navigation pointer}). When the \textit{edit point} is in the \textit{Scroll position} field (use the left mouse), the arrow keys on the keyboard can be used to move the through the directory tree.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>move navigation pointer one step up</td>
</tr>
<tr>
<td>↓</td>
<td>move navigation pointer one step down</td>
</tr>
<tr>
<td>←</td>
<td>move navigation pointer to next higher directory</td>
</tr>
<tr>
<td>→</td>
<td>open object at navigation pointer in a new window (enter sub-directory if object is of type \texttt{DIR})</td>
</tr>
</tbody>
</table>

Figure 2.1: Initial Nuprl 5 screen
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.

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. The edit point will be in the [name] slot.

As example theorem we take \( \forall A, B : \mathbb{P}. \neg (\neg A \lor \neg B) \Rightarrow \neg (A \land B) \), 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.

---

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

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

Entering the rest of the theorem is straightforward. Typing `implies` generates 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 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.

<table>
<thead>
<tr>
<th>- PRF: not_over_and</th>
</tr>
</thead>
<tbody>
<tr>
<td># top</td>
</tr>
<tr>
<td>∀A,B : P. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))</td>
</tr>
<tr>
<td>BY</td>
</tr>
</tbody>
</table>

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-4ω).

Pressing (C-4ω) 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-4ω) 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.

<table>
<thead>
<tr>
<th>- PRF: not_over_and</th>
</tr>
</thead>
<tbody>
<tr>
<td># top</td>
</tr>
<tr>
<td>∀A,B : P. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))</td>
</tr>
<tr>
<td>BY D 0</td>
</tr>
<tr>
<td># 1</td>
</tr>
<tr>
<td>1. A : P</td>
</tr>
<tr>
<td>⊢ ∀B : P. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))</td>
</tr>
<tr>
<td>BY</td>
</tr>
<tr>
<td># 2</td>
</tr>
<tr>
<td>......wf.......</td>
</tr>
<tr>
<td>P ∈ U'</td>
</tr>
<tr>
<td>BY</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>- PRF: not_over_and</th>
</tr>
</thead>
<tbody>
<tr>
<td># top 1</td>
</tr>
<tr>
<td>1. A : P</td>
</tr>
<tr>
<td>⊢ ∀B : P. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))</td>
</tr>
<tr>
<td>BY</td>
</tr>
</tbody>
</table>

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 theoretical expression. Most goals of this kind can be dealt with automatically. Typing Auto (C-4ω) will complete this subproof: no subgoals are generated and the status marker changes into a *.

<table>
<thead>
<tr>
<th>- PRF: not_over_and</th>
</tr>
</thead>
<tbody>
<tr>
<td>* top 2</td>
</tr>
<tr>
<td>......wf.......</td>
</tr>
<tr>
<td>P ∈ U'</td>
</tr>
<tr>
<td>BY Auto</td>
</tr>
</tbody>
</table>

Pressing the left arrow key will bring you back into the first subgoal. Alternatively you can press (C-M-4ω), which causes the editor to jump to the next unproven subgoal.

Proving the first subgoal requires more efforts. Typing Auto (C-4ω) 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.

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¬ω}_ results in a complete proof of the subgoal.

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

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 \textit{extract term} of the proof (a \( \lambda \)-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 \( \exists ! \) quantifier, which states the existence of a unique element \( x \in T \) that satisfies a property \( P \). A typical definition for this quantifier is the following:

\[
\exists ! x : T. \ P[x] \equiv \exists x : T. \ P[x] \land (\forall y : T. \ P[y] \Rightarrow y = x \in T).
\]

This definition actually presents two aspects of a newly defined term. It first states that a new abstract term, say \( \exists ! x : T \), 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 \( \exists ! x : T. \ P \).

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

To initialize this mechanism, click the \textbf{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 $\exists !$ has two subterms, $T$ and $P$, and binds one variable in the second. To create a template for entering the details, type $\exists \text{uni}(0;1)$. 

This tells the system to create a term called $\exists \text{uni}$, 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.

Enter $T \leftarrow x \leftarrow \text{so_var1} \rightarrow$. 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).

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.

There are two ways to correct that mistake. You may delete the term sovar1 by clicking DELETE 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 DELETE over it and then enter so_var1 ↵.

Entering $P \leftarrow x \leftarrow$ next generates $P[x]$ and moves the edit point into the right hand side of the definition.
To enter the right hand side of the definition, you have to proceed in a structural top-down fashion. Type \( \exists x : T \) and \( \text{so_var1} \) \( \text{P} \) \( \text{x} \).

\[
\exists x : T. (P[x] \land [\text{prop}])
\]

and then \( \forall y : T \implies \text{so_var1} \) \( P \) \( y \) \( \text{equal} \) \( x \).

\[
\forall y : T. (P[y] \implies y = x)
\]

The definition is now complete. To save it to the library click \( \text{OK} \) or press \( \text{q} \) again. This closes the \text{AddDef} template and creates a display form \( \exists x : T. (P[x] \land (\forall y : T. P[y] \implies y = x)) \).

The abstraction object shows on the right hand side the term that defines the meaning of \( \exists x : T. P[x] \) and on the left hand side the form in which \( \exists x : T. P[x] \) is currently displayed. Right now, this is identical to the abstract term form. To close the abstraction object again press \( \text{C-q} \).
To change the appearance of the term \( \text{exists}_{\text{uni}}(T;x.P[x]) \) to \( \exists!x:T. \ P[x] \) you have to edit the accompanying display form. For this purpose, move the nav point one step up and open the object \( \text{exists}_{\text{uni}}_{\text{df}} \).

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 \text{exist}_{\text{uni}} in the second line and erase the text \text{exist}_{\text{uni}}(T;x.P[x]) using backspace and delete keys. Notice that you can’t delete the slot \(<T:T:*>\) with these keys. Type \( \text{hC-#163!} \) to generate the \( 9 \) 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.

To rearrange the order of the slots click left over \(<T:T:*>\) press \( \text{hM-p} \) to mark the full slot, and \( \text{hC-y} \) to cut it. Then move the mouse to the immediate right of \(<x:\text{var}:*>\) and press \( \text{hC-y} \). Add a colon between \(<x:\text{var}:*>\) and \(<T:T:*>\), a dot and a space between \(<T:T:*>\) and \(<P:P:*>\).

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 \( \text{type} \) instead. In the same way change the second \( P \) in \(<P:P:*>\) to \( \text{prop} \). This makes sure that meaningful descriptions for these slots will show up whenever the template for \text{exists}_{\text{uni}} is opened.

To commit the completed display form to the library, press \( \text{C-Z} \). This will also close the window again. Do not use \( \text{C-Q} \) unless you want all your changes to be ignored. \( \text{C-Q} \) will always close a window without saving its contents to the library.
To check the display form, open the abstraction \texttt{exists uni} again. You will notice that the display of the abstract term \texttt{exists uni}(T;x.P[x]) has now been replaced by \(\exists! x:T. P[x]\).

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 \texttt{exists uni} will involve checking the type of this term. This can be done automatically if \textit{well-formedness theorem} is provided describes the type of the newly defined term.

A unique existence quantifier \(\exists! x:T. P[x]\) is a proposition, provided that \(T\) is a type and \(P\) is a predicate on \(T\). In other words, the type of \(\exists! x:T. P[x]\) is \(P\) if \(T\) is an element of the universe \(U\) of types and \(P \in T \rightarrow P\). Formally, you need to prove \(\forall T:U. \forall P:T \rightarrow P. \exists! x:T. P[x] \in P\).

To state this theorem, open the object \texttt{exists uni} and enter \(\texttt{all} \downarrow T \downarrow \text{univ} \downarrow i \downarrow \text{all} \downarrow P \downarrow \text{fun} \downarrow T \downarrow \text{prop} \downarrow \text{member} \downarrow\) into the goal template.

```
- PRF: exists uni wf

# top
\(\forall T:U. \forall P:T \rightarrow P. ((\text{term}) \in \text{[type]})\)
BY
```

Continue with \texttt{exists uni} \(x \downarrow T \downarrow \text{so} \downarrow \text{apply1} \downarrow P \downarrow x \downarrow \text{prop} \downarrow \text{(C-M-g)}\), 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 \texttt{Auto}. However, since \texttt{Auto} does not unfold a definition unless it is explicitly declared as automatically unfoldable, you need to unfold the definition of \texttt{exists uni} in the first step.

Typing \texttt{Unfold \textquoteleft exists uni	extquoteright 0 THEN Auto} into the rule slot will result in a proof of the well-formedness theorem for \texttt{exists uni}.

```
- PRF: exists uni wf

# top
\(\forall U:U. \forall P:T \rightarrow P. (\exists! x:T. P[x] \in P)\)
BY \texttt{Unfold \textquoteleft exists uni	extquoteright 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 \(\texttt{(C-Q)}\).
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.