Chapter 4

The Navigator and the Top Loops

The navigator and the interactive ML top loops provide interfaces to the library, the editor, and the refiners. The main interface to the library is the NUPRL 5 *navigator*. It enables the user to

1. browse and search through the library
2. create, delete, and rename library objects of various
3. arrange objects in folders and theories
4. edit library objects by invoking a *term editor* (Chapter 5) or a *proof editor* (Chapter 6)
5. check the validity of objects and theories
6. export and import theories
7. print library objects and theories to text and LaTeX files

The same operations can also be initiated from NUPRL's ML *top loop*, which provides an interactive interface to the ML system of the editor process. The difference is that the navigator provides a visual interface to the library, while the ML top loop requires a user to enter ML commands that will be accepted by the NUPRL 5 editor and communicated to the library.

The ML top loop also provides additional functionality, such as experimenting with NUPRL functions, loading ML files, and exploring the NUPRL state. These functions can also be executed from the ML top loop that was initially started by the editor process. The difference between these two top loops is that NUPRL's ML top loop runs in a *term editor* window and thus supports most of the editing commands described in Chapter 5 work in it. In contrast to that the *process* ML *top loop* runs in a Unix shell that does not support editing NUPRL terms but may support text editing features if run from within emacs. Furthermore, most library functions are not accessible from the process top loop.

NUPRL's ML top loop can also be run as *refiner* ML *top loop* or as *library* ML *top loop*. In these cases the ML top loop interfaces with the refiner process or the library process instead of the editor. This means that a different set of functions will be preloaded and that the commands entered will affect a different process. For instance, all tactics (see Chapter 8) are accessible from the refiner top loop, which enables a user to experiment with new tactics while having access to a term editor. In the library top loop, functions for modifying the library itself (such as loading patches or structural rearrangements) become available. The same functionality will also be provided by the corresponding process ML top loops, yet without term editing support.

Most users will rarely use the ML top loops, because all standard tasks can be performed by using the navigator. More experienced users will occasionally use the refiner or editor top loops. The refiner top loop is usually only required for maintenance.

4.1 The Library

NUPRL's *library* is a mathematical and logical database. All library contents are represented by a common basic data structure called *objects*. There are objects for theorems, definitions, inference rules, tactics and other algorithms, comments and articles, objects that control the visual appearance of the mathematical notation, and objects that are used to organize other objects in theories and directories. A *library table* binds objects to identifiers that are used when referring to them.

In contrast to previous releases of NUPRL, all library contents are kept in a persistent library and are accessible (modulo permission restrictions) as soon as the NUPRL 5 system is started. The library roughly operates like a data base: modifications to theories, such as creating, deleting, or editing objects are immediately committed to the library. However, all changes may be undone if necessary. A backup of all previous versions of an object is kept until is explicitly destroyed in a garbage collection process, which enables a user to recover previous versions if needed.

The navigator shows information on a segment of the library, which is sometimes called the user's *work space*. The format of the navigator window is discussed in Section 4.2.1. Commands for for browsing, searching, editing, and structuring library contents as well as for controlling the navigator window and are discussed in Section 4.3. In the rest of this section we briefly explain the internal structure of NUPRL's library and its relation to the externally visible behavior of NUPRL.

4.1.1 Library Objects

Library objects the common representation of all the contents of NUPRL's library. They are abstract terms that are associated with a *kind*, a variety of *properties*, and possibly with *extra data*.

Abstract terms provide a *uniform data structure* for representing almost any kind of formal content. They consist of an *operator identifier*, a list of *parameters*, and a list of *bound subterms* (see Chapter 5 for a detailed description). The abstract term syntax makes sure that no predefined structure is imposed on the contents of the library and makes parsing unnecessary. All visible structure and notation is generated by the editor process, which consults display forms that describe how to "read" an abstract term. The separation between internal representation and external presentation makes formal notation extremely flexible and expressive, as it supports an almost arbitrary syntax and allows information to be presented differently depending on context and the preferences of the users of NUPRL.

The *kind* of an object is a description of the intended role of the abstract term. It allows making a distinction between theorems, definitions, tactics, comments, etc., and identifying structure information when assembling theories. Currently the following kinds are defined

statement objects contain a proposition and (reference to) a proof. If the proof is complete, the proposition is considered a *theorem* (or a *lemma*). Otherwise it is a *conjecture*. A statement object for a complete theorem also contains the extract term of the theorem

proof objects contain NUPRL *proofs*, i.e. directed acyclic graphs of (references to) inference steps, where the conclusion of a child inference is a premise of its parent inference. A proof is complete if all its leaves are closed by inferences.

inference objects contain records of actual *inference steps*. These may consists of instances of primitive rules, of tactics executions, or more generally of applications of inference engines that are connected to the NUPRL library.

abstraction objects introduce the *abstract definition* of a new term.

display objects define *display forms* for primitive terms and abstractions.

precedence / lattice objects assign *precedences for terms*. Precedences control the automatic parenthesization of terms.

code objects contain the code of *tactics* and other ML code.

rule objects define *primitive rules* of the object logic.

directory objects define NUPRL *theories*. They contain lists of references to other objects and are used to add structure to the library.

comment objects contain *comments*. Comments have no logical significance but can be used to link formal material to informal text.

term objects are used to represent all library objects whose kind is not specified. Inactive (see below) directories are considered term objects.

Statement objects, proofs, and inferences are discussed more in Chapter 6, abstractions in Chapter 7.1, display forms and precedences in Chapter 7.2, rules and tactics in Chapter 8, and ML code in Appendix B.

The *properties* contain status information that is helpful for maintaining the object, tracking dependencies, building justifications etc. The most common properties are

- A *liveness bit*, indicating whether the object is *active* and may be referenced to by others.
- A *sticky bit*, indicating whether the object may be removed from the library table during garbage collection. Most objects are not sticky.
- A *description of clients* to which the object shall be made visible.
- A memnonic *name* which is commonly used for presenting the object identifier.
- The *language* in which a code object is programmed.
- A *reference environment* (Section 4.3.3.1) describing the context of the object.

Extra data are used to collect information that accounts for the validity of an object's content. Statements include a list of (links to) proof objects as extra data, proofs include a tree of inferences, and inferences include primitive inference steps.

4.1.2 The Library Table

In the *library table*, objects are also associated with *abstract identifiers* that are bound to the contents of the object. All references to objects have to use these abstract identifiers, which in turn are linked to names for objects in a user's work space.

Object contents are viewed as non-destructive. To change the content of an object, the library creates a new object content and rebinds the abstract identifier of the object to the new content. To remove the object from the library, it simply removes the binding between the abstract identifier and the content. Object contents are usually not removed from the library except by garbage collection.

All library operations are built from a small collection of primitive operations on object contents and library tables. These operations are performed by the library's *transaction manager*, which ensures that the library is always in a consistent state, and provides mechanisms that make it possible to recover from failures and system crashes. Library transactions also provide a model for controlling the outside access to the actual library contents, which enables the library to certify the correctness of its formal theorems and proofs. The primitive library operations are

- *Binding* an object identifier to an object and *unbinding* an object identifier.
- *Looking up* object contents bound to an abstract identifier.
- Generating *new object identifiers*.
- *(De)activating* an object (changing the liveness bit).
- *(Dis)allowing* garbage collection for the object (changing the sticky bit).

There are also primitives for *creating new object contents* from existing object contents and new data. The most basic primitive creates a new abstract term for the object. Other primitives modify extra data related to building proof structures by changing the list of proofs linked to a statement, modifying the inference tree of a proof, or changing the inference step of an inference object.

4.1.3 User Interaction with the Library

Nuprl users do not directly interact with the library but through an *editor process* that communicates with one of the library's *application interfaces*. The application interface generates the user's work space from the actual library table and communicates the modifications initiated by the user to the transaction manager, which in turn performs the actual modification to the library. This makes it possible to restrict access to certain parts of the library, to hide the abstract representation of data, and to present to the user a consistent view of the library to the user.

In the user's work space, library objects grouped into directories, or *theories*. Every object has a unique name and belongs to exactly one theory, although they may be linked to from other theories or by a different name within the same theory. Theories can be nested like Unix directories and may depend on each other. The dependencies of theories on one-another forms a partial order. Within each theory, objects are ordered linearly.

The Nuprl editor enables users to walk through the directory tree in a visual fashion and to initiate commands for browsing, searching, editing, and structuring library contents through menu buttons. The editor does not execute these commands directly but sends requests to the library's application interface, which in turn will perform the appropriate actions and sends an updated view of the client's work space back to the editor.

For most practical purposes, the distinction between the apparent external behavior of Nuprl and the internal operations that are performed to realize this effect is irrelevant. The subsequent expositions will therefore consider the library to be identical to the directory structure of objects that is presented on the screen.

4.2 Nuprl Windows

A complete Nuprl session (see Section 3.3) usually starts with five windows, shown in Figure 4.1: a *navigator*, an ML *top loop*, and three windows for the library, refiner, and editor processes.[1] Among these, the navigator window is the most important one, as most user interaction goes through the navigator. The ML top loop will only be needed for more advanced tasks that have not (yet) been integrated into the navigator. The process windows receive all system output and error messages and are usually only needed for maintenance and debugging purposes.

[1]Users may also connect to existing library and refiner processes and only see one Lisp process window.

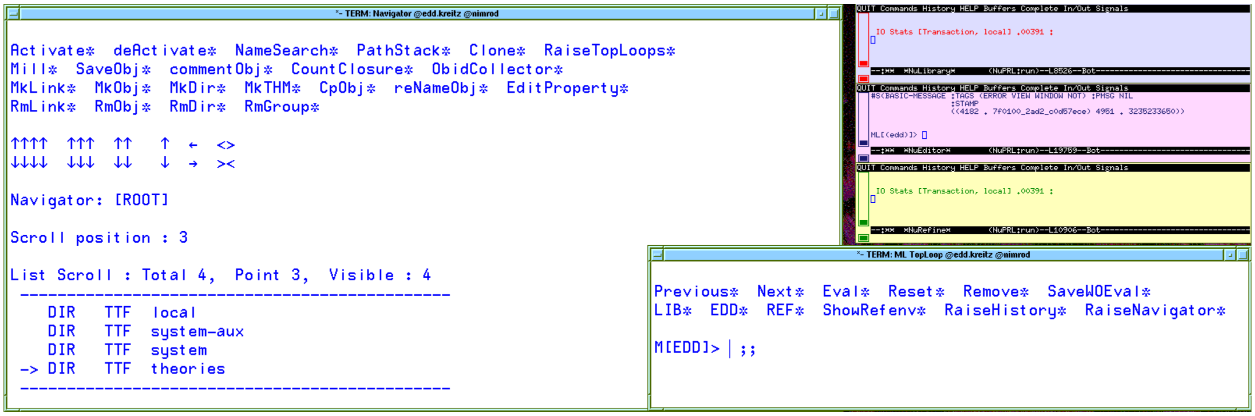


Figure 4.1: Initial NUPRL 5 screen

4.2.1 The Navigator Window

The *navigator window*, shown on the left of the screen in Figure 4.1, is divided into three major zones, a *command* zone, a *library statistics* zone, and a *navigation* zone.

The command zone can be found in the upper part of the navigator window, as in most NUPRL 5 windows. It contains several *buttons*, which are are indicated by a * at the end of a piece of text. Clicking these buttons with the left mouse button will trigger the action described by the text and occasionally pop up a template that needs to be filled in. The arrows in the window (↑↑↑↑, ↓↓↓↓, . . . , ↑, ↓) also operate as buttons that can be clicked for faster scrolling. The commands linked to the navigator buttons are described in detail Section 4.3 below.

Many commands require interaction with the user, for instance typing in the name of an object to be created. The interaction takes place through templates and additional command buttons that will appear on top of the command zone, as illustrated in Section 4.3.2.1. The additional button and slots created depend on the individual command.

It should be noted that the buttons in the command zone may depend on the directory that is currently shown by the navigator. Subsequent snapshots will show, for instance, that the buttons for the standard theories include a variety of theory specific buttons that are not relevant for the root directory. NUPRL allows users to customize the command zone by adding new buttons tailored for specific modes of operation in certain theories.

The statistics zone immediately above the display of library contents shows directory statistics.

- The line beginning with `Navigator` describes the *current directory path* path, beginning with the actual theory and going backward to the root of the directory tree.
- The `Scroll position` field shows the position of the navigation pointer within the current directory. When the *edit point*, which is marked by a thin vertical line, is in this field the arrow keys on the keyboard can be used to move the through the directory tree.
- The `List Scroll` field shows the total number of objects in objects in the current theory, the position of the navigation pointer, and the number of visible objects. The latter us usually 10, or less if there are less than 10 objects in the directory, but can be modified using the `<>` or `><` buttons (Table 4.1).

The navigation zone in the lower part of the navigator window displays a linear segment of the library, one object per line. From left to right each line contains:

The object **kind** is described by a string of three or four characters.

STM stands for *statement* objects, PRF for *proof* objects, INF for *inference* objects, ABS for *abstractions*, DISP for *display forms*, PRC for *precedence* objects, CODE for *ML* code, RULE for inference *rules*, COM for *comments*, DIR for *directories*, and TERM for *objects* of unspecified kind.

Proof and inference objects are usually not listed in the directory but can be accessed only through the proof editor (Chapter 6.2).

The object **status** is described by three characters, either T or F.

The first character describes whether the object has been *activated* and is T in most cases.

The second character is reserved for theorem objects and describes if the theorem has a complete proof and an extract term. For all other objects it is T.

The third character describes the status of the sticky bit. It is F for most objects.

The object's **name** can have arbitrarily many characters and may include blanks.

One of the displayed objects in the library is also marked by an arrow (->) to the left of its kind. We call this distinguished object the *navigation pointer* (or *nav point*. All navigator commands will be executed relatively to this object.

4.2.2 The ML Top Loop Window

The ML *Top Loop window*, shown on the right bottom of the screen in Figure 4.1, offers a command interface to the editor, refiner, and library processes. It is divided into two major zones, a *command button* zone and a *command line* zone.

The command button zone in the upper part of the ML Top Loop window is similar to the command zone of the navigator. The command buttons, however, do not interact with the library contents but affect the behavior of the editor itself. Most importantly, the buttons LIB*, EDD*, and RED* switch between the processes that the ML Top Loop interacts with and change the command prompt of the command line zone accordingly to M[LIB]>, M[EDD]>, or M[REF]>.

The command line zone between the command line prompt and the double semicolon provides a *term editor* window for entering ML commands that may contain NUPRL terms as arguments. The latter can be inserted by opening a term slot and entering terms as described in Chapter 5.

4.2.3 The Process Top Loop Windows

The Process Top Loop windows are the windows in which the library, refiner, and editor Lisp processes were started. Usually they run as ML top loops for interacting with the corresponding processes. However, it is also possible to switch into Lisp mode, if low-level operations have to be performed. The process Top Loops support most of the commands of the corresponding NUPRL ML top loop, but lack the features for editing NUPRL terms and most of the navigator commands.

These windows should only be used for maintenance and debugging purposes. It is recommended to run them within an emacs shell to have some text editing support.

4.3 Library Commands

Most library commands are best executed from the navigator may also be invoked from the ML top loop (see Section 4.4). In this section we will describe the usual navigator operation and mention the corresponding commands.

Many commands are initiated by clicking the left mouse button on one of the predefined menu buttons in the navigator's command zone. Often this will pop up a template containing one or several slots into which the user has to enter text or terms. When issuing commands, pressing certain key sequences will have the following effects.

- The return key ↵ closes a slot and moves to the next empty slot. If all slots have been filled, it highlights the completed command. Pressing ↵ again then *executes the command*.
- The tabulator key ⇆ usually cancels a command.
- The space key SPC moves the cursor back into the navigation zone but leaves the template open. This is helpful for moving objects or link objects in different theories.
- As SPC is used for the above action, blanks are inserted into a text slot by pressing ⟨S-SPC⟩.
- ⟨C-_⟩ is used to undo an operation (on a fairly fine level)
  ⟨C-+⟩ is used to redo an undone operation.
- Pressing the left mouse button LEFT usually sets the point to that location. Pressing LEFT while over a menu command button executes the corresponding command.
- Pressing the middle mouse button MIDDLE usually raises the display form of a term or the code object containing the definition of an ML function. Within the navigation zone, it opens the corresponding object.
- Pressing the right mouse button MIDDLE raises the abstraction of an object, if there is one.

These bindings are also valid in many other editing contexts.

4.3.1 Browsing the Library

To *browse the library*, a user may move a *navigation pointer* through the current directory by using arrow keys, the mouse, or clicking on one of the arrow buttons ↑↑↑↑, ↓↓↓↓, . . . , ↑, ↓. To move into a directory or to open an object for editing, one uses the right arrow key (or middle-clicks on it with the mouse), to move out of a directory, one moves the navigation pointer to the left. There are also `emacs` like key bindings to substitute for the arrow keys and buttons for changing the number of visible objects, or *screen size*. Table 4.1 lists all the key bindings and buttons for moving through the navigator window and manipulating its size. Users may customize these bindings in their `mykeys.macro` file (see Chapter 3.2.2).

4.3.1.1 Viewing and Editing Objects

In order to view or edit an object, one moves the navigation pointer to it and then opens it using the right arrow (or MIDDLE). If the object is not already being viewed, this will pop up a new window and open the appropriate editor: a *proof editor* (Chapter 6.2) is used on theorem objects, while the *term editor* (Chapter 5) is used for all other objects.

Abstractions and display forms of an abstract term can also be opened when an instance of the term is visible in a term editor. In this case one may click on the term with MIDDLE to view the display form and RIGHT to view the abstraction. Alternatively one may position the term cursor at the term and type ⟨C-`X`⟩`df` or ⟨C-`X`⟩`ab`, respectively. Chapter 5.7 gives a detailed description of these term editor utilities.

Key		Button	
↓	⟨C-`n`⟩		move navigation pointer one step down
⟨C-↓⟩			move navigation pointer 5 steps down
⟨C-M-↓⟩	⟨C-`v`⟩		move navigation pointer 10 steps down
		↓↓↓	move navigation pointer one screen down
		↓↓↓↓	move navigation pointer to bottom
↑	⟨C-`p`⟩		move navigation pointer one step up
⟨C-↑⟩			move navigation pointer 5 steps up
⟨C-M-↑⟩	⟨M-`v`⟩		move navigation pointer 10 steps up
		↑↑↑	move navigation pointer one screen up
		↑↑↑↑	move navigation pointer to top
LEFT			move navigation pointer to mouse point
→	⟨C-`f`⟩		open object at navigation pointer
MIDDLE			open object at mouse point
←	⟨C-`b`⟩		move navigation pointer to next higher directory
		`<>`	increase screen size by 10
		`><`	reduce screen size by 10

Table 4.1: Navigator Motion Commands

Objects can also be viewed by typing the command ⌞`view` *name*⌟ into the editor ML top loop, where *name* is a token indicating the name of the new object. The `view` command was the standard method for viewing objects in the predecessors of NUPRL 5. Its use in NUPRL 5 is discouraged, as the command is ambiguous if the same name use used for multiple objects.

4.3.1.2 Searching for Objects

The navigator provides a utility for a pattern-based search for object names in the library. Name search is initiated by clicking the `NameSearch*` button in the navigator's command zone, which will create a *search command zone* on top of the current command zone and place the edit point into a `[pattern]` slot.

After entering a text string into the pattern slot, a user types ↵ start the search for the next object whose name contains the entered string. By default, the search proceeds forward beginning at root of the library directory tree. Typing ↵ again will search for the next matching name, etc.

The search mechanism can be modified by using the additional buttons of the search command zone. These buttons have the following effects.

- `Hide*`: Hide the search command zone by iconifying it to a button `Search#`.
- `Backward*`: Change the default direction to backward search and search for the next match.
- `Forward*`: Change the default direction to forward search and search for the next match.
- `Global*`: Search within the entire library
- `Tree*`: Restrict search to the subtree beginning in the current directory
- `List*`: Restrict search to the list of objects in the current directory
- `PreviousPattern*`: Replace the current search pattern by the one previously entered. This pattern may be modified but the modified pattern will not be stored in the list of patterns
- `NextPattern*`: Replace the current search pattern by the one entered immediately after it, if there is one.
- `Reset*`: Replace the pattern by an empty `[pattern]` slot.
- `Cancel*`: End the search and remove the search command zone.

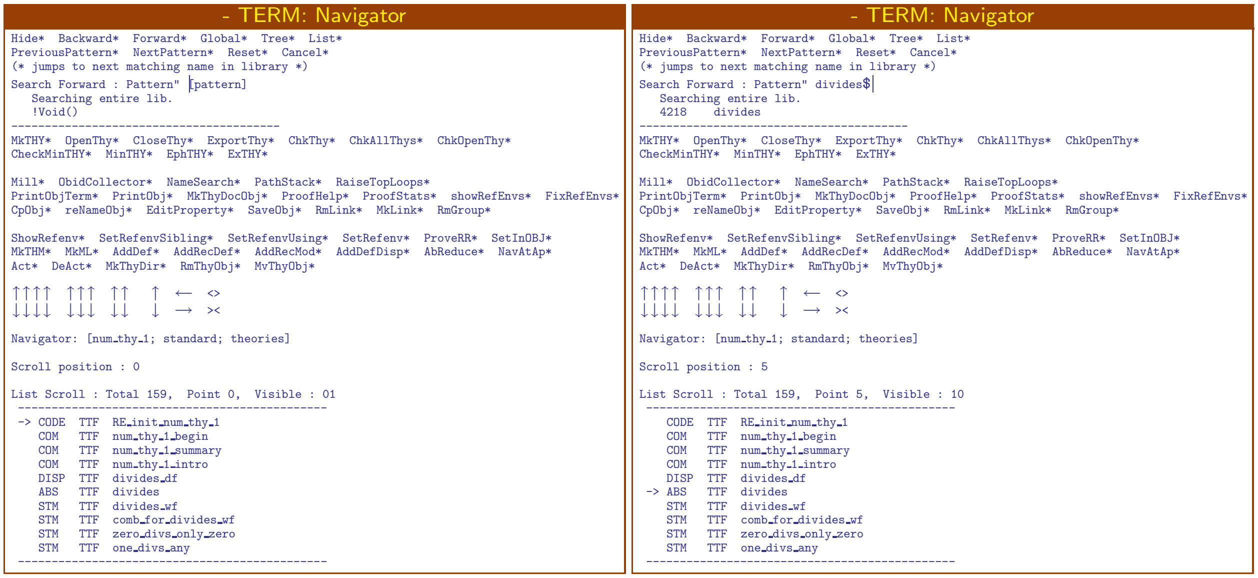


Figure 4.2: Pattern-based name search

Currently, search patterns have to be text strings that can match either a substring of an object's name, its beginning, or its end. To search for the beginning of names, one simply adds a caret (^) before string, to search for the end of names one appends a dollar symbol ($) to its end.

For instance, entering ␣divides␣ into the pattern slot on the left side of Figure 4.2 searches for the object in the library whose name contains the string `divides`. This includes `divides_df`, `divides`, `divides_wf`, `comb_for_divides_wf`, etc. Entering ␣^divides␣ searches only for objects, whose name begins with `divides`, which excludes `comb_for_divides_wf`. Entering ␣divides$␣, as shown on the right side of Figure 4.2, searches only for objects, whose name ends with `divides`, and entering ␣^divides$␣ searches for all objects named `divides`.

4.3.1.3 Advanced Motion: Using Path Stacks

To enable users to jump between commonly used positions in the directory tree, the navigator provides a path stack utility. Clicking the `PathStack*` button will which will create a *path stack command zone* on top of the current command zone and store the current position of the navigation pointer in the path stack. A user may add additional positions to the path stack and jump back to any position stored in it using the additional buttons of the path stack command zone. These buttons have the following effects.

- `Hide*`: Hide the path stack command zone by iconifying it to a button `PathStack#` (see the right of Figure 4.3). This is usually a good idea if one works with several directories at the same time but doesn't jump very often.
- `Yank*`: Jump to the position on top of the path stack.
- `Rot&Yank*`: Rotate the positions in the path stack, moving the top position to the bottom, and jump to the position that is now on top.
- `Push*`: Add the current position of the navigation pointer on top of the path stack.
- `Pop*`: Remove the position on top of the path stack.
- `Swap&Yank*`: Swap the two positions on top of the path stack and jump to the position that is now on top.

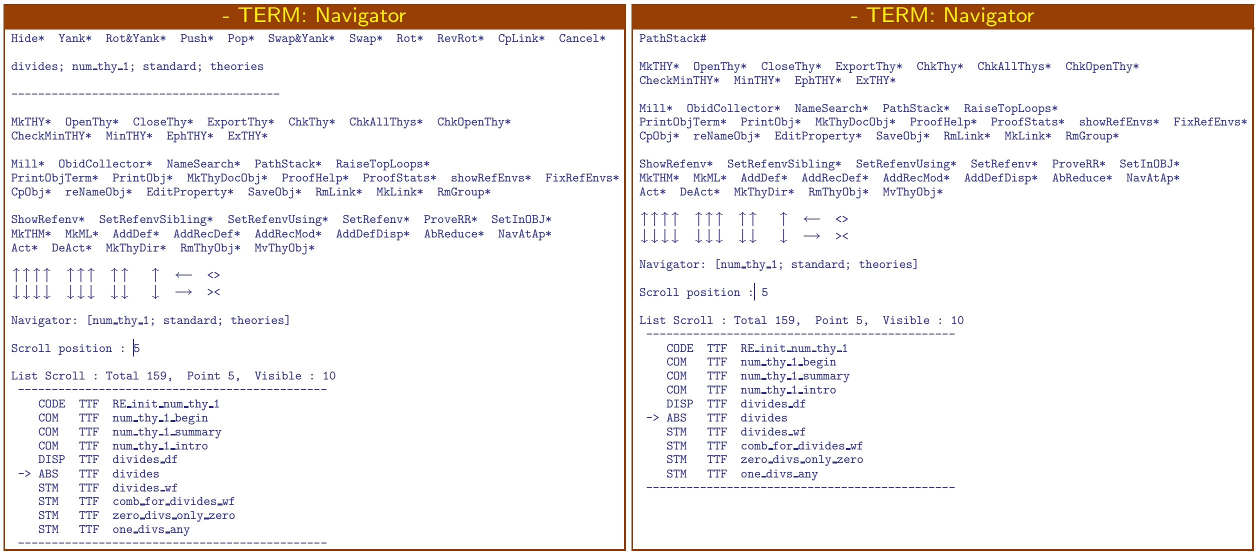


Figure 4.3: Path stack command zone

- `Swap*`: Swap the two positions on top of the path stack.
- `Rot*`: Rotate the positions in the path stack, moving the top position to the bottom.
- `RevRot*`: Rotate the positions in the path stack, moving the bottom position to the top .
- `CpLink*`: Insert a link (Section 4.3.2.5) to the current position of the navigation pointer immediately below the object that is currently on top of the path stack. Links cannot be inserted into the same where the object resides.
- `Cancel*`: Close the path stack and remove the path stack command zone.

4.3.2 Operations on objects

4.3.2.1 Creating Objects

Objects are created by describing their name, their kind, and the position where they shall be inserted into the library. Usually, this is done interactively by clicking the `MkObj*` command button, which will open two templates on top of the current command zone, into which a user may enter the name and kind of a new object, and place the edit point in the `name` slot, as shown on the left of Figure 4.4.

After the name and the kind has been entered into the corresponding slots, a user has to click the `OK*` button (or type ↵ twice), which will close the `new_object` templates and place the corresponding object into the current directory immediately *after* the navigation pointer. The object will have the status `FFF` and no content assigned to it yet.

The name of an object is case sensitive and may contain blanks (enter ⟨S-SPC⟩ to create them) and other special characters. The kind is not case sensitive but is usually displayed in capitals. Typing `not_over_and`↵`stm`↵ ↵ after clicking `MkObj*` in the above context, for instance, leads to the result shown on the right of Figure 4.4.

Nuprl also provides commands and buttons for creating objects of a particular kind. They can be used instead of the more general command, whose button is not shown in most user theories.

Clicking the command button `MkTHM*` creates a statement object and places it into the current directory immediately after the navigation pointer. In the interactive version, one only has to enter

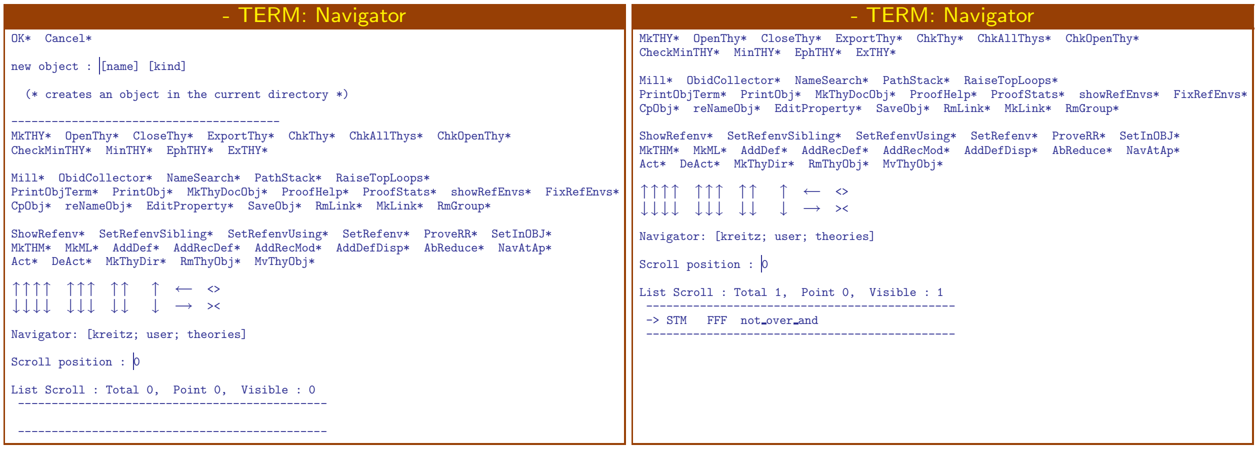


Figure 4.4: Creating Objects: Initial template and resulting update to the library

the name of the theorem. In a similar way MkML* creates a new code object, MkDir* creates a directory object, and MkThyDir* creates a directory object within a theory (see Section 4.3.3.2).

The command button MkTHY* creates a new *theory* within the current directory. Theories are similar to directories, but in addition contain code objects for initializing their *reference environments*. We will discuss them separately in Section 4.3.3.1.

AddDefDisp* creates a display form for a given abstraction. If the navigation pointer is at an abstraction with name *absname*, then clicking the command button AddDefDisp* will create a display form object named ***absname*_df** and places it immediately after the the abstraction object. No new object will be created if an object named ***absname*_df** already exists. Clicking AddDefDisp* while the nav point is not at an abstraction will result in an error.

Currently there are no special command buttons for creating comments, inference rules, or precedence objects. The command for creating abstractions has been subsumed by the mechanism for creating definitions, which is described in Section 4.3.2.2 below.

Objects can also be created by typing the command ⌞dyn_mkobj *kind position directory name*⌟ into the editor ML top loop, where

- *kind* is a token indicating the object's kind.
- *position* is a token indicating the object after which the new object shall be inserted. The empty token, `null_token`, is used to describe a position in an empty directory.
- *directory* is an object identifier indicating the directory in which the new object shall be placed. To create this identifiers, one has to mark the directory object by clicking on it, and yank the corresponding term into the editor top loop by entering ⟨C-y⟩. The term will usually be displayed as `Obid:` *`directory-name`*. To convert this term into an object identifier one has to apply the ML function `ioid`.
- *name* is a token indicating the name of the new object.

Thus, to create the theorem `not_over_and` with an editor command instead of using the interactive command initiated by MkObj* one could alternatively type

⌞`dyn_mkobj ‘stm‘ null_token (ioid Obid: kreitz) ‘not_over_and‘`⌟

However, it is recommended to use the interactive version of the command.

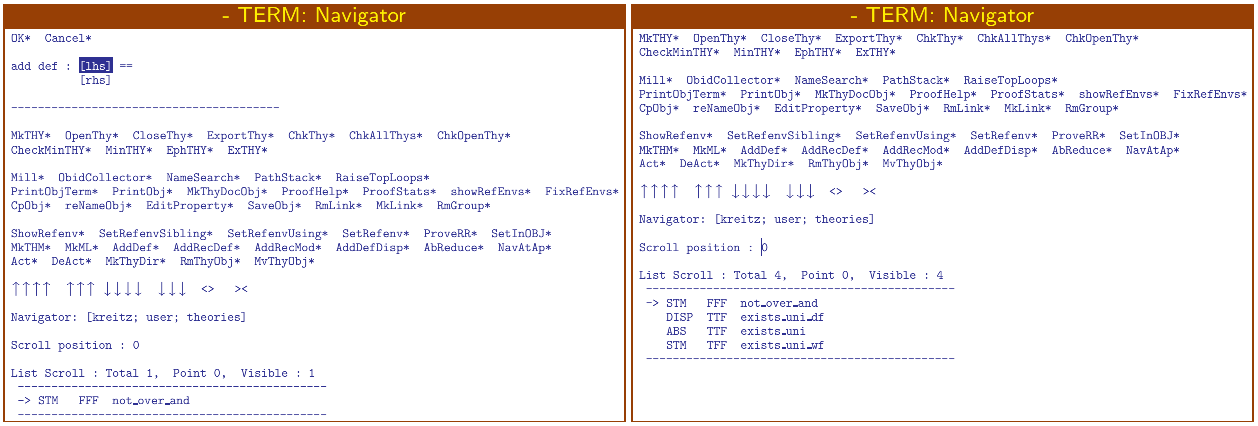


Figure 4.5: Creating Definitions: Initial template and resulting update to the library

4.3.2.2 Creating Definitions

A formal definition adds a new abstract term to the formal language of Nuprl that is defined to be equal to some already existing term. In Nuprl a formal definition requires the creation of two new objects: an *abstraction*, which defines the meaning of the abstract term (see Chapters 7.1), and a *display form*, which defines its syntactical appearance (see Chapter 7.2). In addition, most definitions are accompanied by a *well-formedness theorem*, which proves that the newly introduced term belongs to a certain type and is thus well-formed. The names of these objects follow a certain convention: if the operator identifier of the abstract term is *opid*, then the abstraction object is named *opid*, the display form *opid*`_df`, and the well-formedness theorem *opid*`_wf`.[2]

The `AddDef*` command button provides a convenient way to generate these three objects and a part of their content. Clicking `AddDef*` button will open a template for defining the abstract term.

To enter the abstract term on the left hand side of the definition, one has to provide its *object identifier*, its *parameters*, and a list of its *subterms* together with the variables to be bound in these subterms. Ways to create new terms with the term-editor are described in Sections 5.4.4 and 5.4.5. The term for the right hand side of the definition is entered in the usual structural top-down fashion of the term-editor as explained in Section 5.4.

Closing the `add_def` templates, creates a display form object ***opid*****`_df`**, an abstraction object *opid*, and a statement object ***opid*****`_wf`**, where *opid* is the object identifier of the new abstract term. The abstraction object contains exactly the left and right hand sides of the definition as entered into the `add_def` templates. The display form object contains a display form for the abstract term that makes the term look like the left hand side of the definition but can easily be modified. The statement object is empty, as there are no defaults for initiating a well-formedness theorem. All three objects will be placed immediately after the navigation pointer, which remains at its current position, and are already activated.

Entering ⌞`exists_uni(T; x.P[x])`⌟ and ⌞`∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T)`⌟ into the `add_def` templates, for instance, creates the three objects shown on the right of Figure 4.5.

Definitions can also be created with the command ⌞`lib_thy_add_def` *lhs rhs directory position*⌟, where *lhs* is the left hand side of the definition, *rhs* its right hand side, *directory* the object identifier

[2]In Nuprl 4 this convention made it easier for tactics to access the well-formedness theorems corresponding to a certain abstraction. Although Nuprl 5 offers a more general method for making objects depend on each other, we preserve the convention for compatibility reasons.

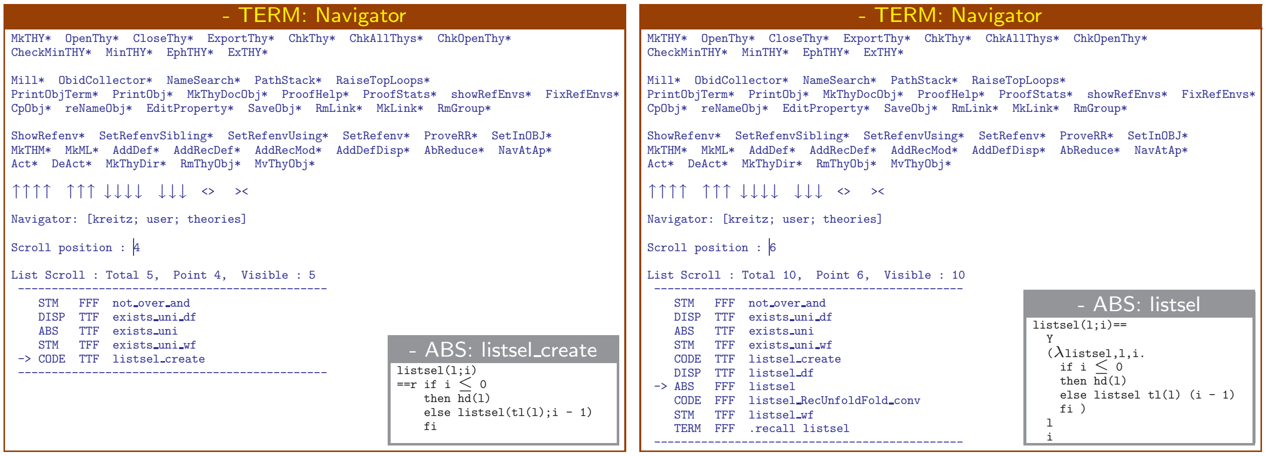


Figure 4.6: Creating Recursive Definitions: code object and created definition objects

of the directory in which the new object shall be placed, and *position* the name of the object after which the definition shall be inserted. Thus, to create the above three objects within the ML top loop one could type

```
lib_thy_add_def  ⌜exists_uni(T; x.P[x])⌝  ⌜∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T)⌝
                 (ioid Obid: kreitz)  `not_over_and`
```

The refiner's `def` utility provides a more advanced method for creating definitions and their well-formedness theorems. This method, however, is less easy to use.

The `AddDef*` mechanism is sufficient for creating non-recursive extensions of NUPRL's object language. For integers, lists, and recursive data types, NUPRL's type theory also provides expressions that describe *primitive recursion* over these types (see Appendix ch:app-type-theory). The terms $\ulcorner \mathsf{ind}(u,x.f_x;\, s;\, \mathit{base},y.f_y)t \urcorner$, $\ulcorner \mathsf{list\_ind}(s;\, \mathit{base};\, x,l,f_{xl}.t) \urcorner$, and $\ulcorner \mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(e) \urcorner$, however, are insufficient for describing more general forms of recursion and most users find them somewhat awkward to use.

The definition for selecting the `i`-th element of a list `l`, for instance, would typically be expressed as ⌞`l[i]  ≡  if i≤0 then hd(l) else tl(l)[i-1] fi`⌟. This definition, however, involves a simultaneous recursion over both the list `l` and the index `i`. Although it is possible to express this in a primitive recursive fashion,[3] a direct representation of the above definition would certainly be more natural. NUPRL therefore provides a mechanism for a controlled introduction of general recursive definitions using the **Y** combinator. This mechanism proceeds in two separate phases.

In the first phase, clicking the `AddRecDef*` command button will create a code object that contains the ML function `add_rec_def_at`, which will later build the actual definition. For the sake of comprehensibility, the function is encapsulated in a formal definition and initially appears as template ⌞`[lhs] ==r [rhs]`⌟. A user has to provide the left hand side and the right hand side of the recursive definition as arguments to this function. A third argument two the function is the location, where the definition is to be placed. To make sure that the function does not execute every time the object is viewed and closed again, this third argument is initially set to `inl()`.

After the user has entered the left hand side and the right hand side of the recursive definition and closed the code object, clicking the `NavAtAp*` button (Section 4.3.4.3) will create the

[3]One can bypass the simultaneous recursion by using the listind operator to define a *function* on indices, which then is applied to `i`: `l[i]  ≡  (list_ind(l; λj.0; hd,tl,jth-of-tl.λj.if j≤0 then hd else jth-of-tl(i-1) fi))(i)`.

actual definition by executing the function `add_rec_def_at` with the third argument substituted by the location of the code object. This results in 5 additional objects: an abstraction, a display form, a statement object for the well-formedness theorem, a code object that updates the tactics for unfolding and folding definitions, and a recall object, which allows removing all the newly created objects with the `RmGroup*` button (Section 4.3.2.6). For example, entering ⌞`listsel(l;i)`⌟ and ⌞`if i≤0 then hd(l) else listsel(tl(l);i-1`⌟ into the templates of the recursive definition object `listsel_ml` creates the objects shown on the right of Figure 4.6.

Recursive definitions can also be created with the command

⌞`add_rec_def_at` *lhs rhs* `(inr(`*directory, position*`))`⌟,

where *lhs* is the left hand side of the definition, *rhs* its right hand side, *directory* the object identifier of the directory in which the new object shall be placed, and *position* the name of the object after which the definition shall be inserted. Thus, to create the above five objects within the ML top loop one could type

```
add_rec_def_at  ⌜listsel(l;i)⌝ ⌜if i≤0 then hd(l) else listsel(tl(l);i-1⌝
                (inr ((ioid Obid: kreitz), ‘exists_uni_wf‘))
```

This command has to be run in *refiner mode* and will not create the initial code object `listsel_ml`. Again, there is a more advanced version of this command.

Besides creating abstractions, display forms, and well-formedness theorems, introducing a definition may also require updating the tactics that rely on folding and unfolding definitions. As only the user can decide which abstractions should be unfolded automatically and which ones shouldn't, Nuprl provides a mechanism for updating the `Reduce` tactic (see Section 8.6.2) on demand.

Clicking the `AbReduce*` command button will open two templates on top of the command zone. The first is a token template into which a user may enter the name of a new conversion to be added to `Reduce`. The second is a term describing the left hand side of that conversion. Upon clicking `OK*` the right hand side of the conversion will be computed by applying ⌞`UnfoldsC` *opid* `ANDTHENC ReduceC`⌟ (see Section 8.9.3) to this term and the resulting macro-conversion will be added to the list of conversions used by the tactic `Reduce`.

4.3.2.3 Creating Modules

Nuprl provides support for defining *module types*, which are useful for defining abstract data types and algebraic classes. Module types are essentially (dependent) *record types*, where the type of each field can depend on the value of previous fields, and are allowed to have parameters. For instance, an abstract data type for stacks may use the type of stack elements as a parameter. Module types are currently implemented using Nuprl's Σ type.[4] A predefined mechanism helps with setting up new module type definitions, adding projection functions as module component selectors, and updating the `AbReduce` tactic (Section 8.6.2) to recognize applications of these functions. Like adding recursive definitions it proceeds in two phases.

In the first phase, clicking the `AddRecMod*` command button will create a code object that contains the ML function `create_rec_module_at`, which will later build the actual module. Currently, the object contains the function call in its raw from, providing a few slots for the user to describe the module.

[4]A more elegant approach is implementing record types as dependent function types on a type of labels. This approach does not require creating definitions that map field selectors onto projection functions, but is somewhat more complex theoretically and not yet supported by the existing tactics collection.

Figure 4.7: Creating Recursive Modules: code object and created directory

- The first token slot contains the name of the module type, e.g. `‘Stacks‘`.
- The second token slot contains the name of the constructor that builds modules from their individual components.
- The third token slot contains a short name of the module that is prefixed to the names of the abstractions and display forms defining the module's field selectors. This prefix was necessary in previous releases of NUPRL to disambiguate the names of these definitions but has become obsolete because of the directory structure introduced in NUPRL 5. It is retained for compatibility purposes.
- The fourth slot contains the parameters of the module type and their types, which have to be given as list of pairs of NUPRL variables and terms.
- The fifth slot contains the fields of the module type, again as a list of pairs of variables and types. The type of a field may use the variables declared in the previous fields.

In addition to that, the last two function arguments are already filled in. The type of the module is $\mathbb{U}$ and the the location, where the module is to be placed, is initially set to `inl()`.

Clicking the NavAtAp* button (Section 4.3.4.3) will create the actual module by executing the function `create_rec_module_at` with the last argument substituted by the location of the code object. This results in the creation of an object directory for the module that contains definitions for the module type, projection functions, the module constructor and uniform module decomposition operator, (unproven) well-formedness theorems, code for updating the `AbReduce` tactic, a recall object, and two comment objects that serve as delimiters for the module type definition. Figure 4.7 describes a code object for defining an abstract data type of stacks over a type `T` and the object directory created by it.

4.3.2.4 Copying Objects

Copies of existing objects can be created using the the `CpObj*` command button. This will open a template on top of the command zone into which the user may type the name of the object that will contain the copy. The current name of the object already occurs in the template, with the edit cursor at its beginning. To place a copy into a different directory, a user may leave the command zone (by pressing SPC), move to the position where the object should be placed, and then click `OK*`.

Objects may also be copied by typing ⌞`copy_object_after` *directory position name obid*⌟ into the editor ML top loop, where *directory* the object identifier of the directory and *position* the name of the object after which the copy shall be placed, *name* the name of the copy, and *obid* the object identifier of the object to be copied.

4.3.2.5 Links

Links are named references to objects in the library, similar to links or shortcuts in operating systems. Unlike copies of objects, different links refer to the same object and changes to the object will be visible from wherever it is referenced to. For consistency reasons, a directory may not contain duplicate references to the same object.

To create a link, one has to click the `MkLink*` command button. This will open a template on top of the command zone into which the user may type the name of the link to the object. If the link is not placed in a different directory (by leaving the command zone and moving into that directory), creating the link will rename the reference to the object but keep its internal name.

Links may also be created by typing ⌞`dyn_mklink` *directory position name obid rmdup?*⌟ into the editor ML top loop, where *rmdup?* is a boolean flag indicating whether or not to remove duplicate links to the same object from the directory. This flag should usually be set to `true`.

4.3.2.6 Removing Objects and Links

To remove an object in a theory, one simply moves the navigation pointer to it and clicks the `RmThyObj*` button. This will remove the object from the current directory, but preserve external links to it. The same effect can be achieved by typing ⌞`lib_rm_thy_obj` *directory name*⌟ into the *library* ML top loop, where *directory* object identifier of the directory in which the object to be deleted resides and *name* a token describing its name.

Similarly clicking `RmLink*` will remove a reference to an object from the current directory. The effect is almost the same as `RmThyObj*`, but the command will be executed by the editor instead of the library and will not immediately affect proof tactics that refer to the object.

Some theories also provide a `RmDir*` button, which allows to remove a directory and all the objects contained in it. Since this is a dangerous operation, the user is asked for confirmation to avoid that a directory is wiped out accidentally. Directories can also be removed by typing ⌞`delete_tree` *directory name*⌟ into the *editor* ML top loop. In this case there the command is executed without asking for confirmation.

Some editor commands such as `AddRecDef*` and `AddRecMod*` create groups of objects related to each other. NUPRL offers a convenient method for removing all these objects by a single command. For this purpose one has to position the navigation pointer at the recall object of the group (an object of the form '`.recall` *group-name*') and click the `RmGroup*` button. This will remove the objects from the library and update the reference environment (see Section 4.3.3.1) accordingly.

4.3.2.7 Moving Objects

Objects may be moved to different locations within the same directory or to locations in other directories. Clicking the `MvThyObj*` command button will open a template on top of the command zone into which the user may type the name of the object to be moved. The name of the object at the navigation pointer already occurs in the template, with the edit cursor at its beginning. To move the object to a different location one has to leave the command zone, move to the position where the object should be placed, and then click `OK*`.

The same effect can be achieved by typing ⌞`lib_mv_thy_obj` *src-dir name dest-dir position*⌟ into the *library* ML top loop, where *src-dir* and *dest-dir* are the object identifier of the source and destination directories, *name* the name of the object to be moved, and *position* the name of the object after which the definition shall be inserted.

4.3.2.8 Renaming Objects

Renaming an object involves changing both the object's internal name (see Section 4.1.1) and the external reference to the object. Clicking the `reNameObj*` command button will open a template on top of the command zone into which the user may type the new name of the object. Leaving the command zone while renaming an object is not recommended, as renaming will be applied to whatever object the navigation pointer points to at the time the `OK*` button is clicked.

The same effect can also be achieved by using the `EditProperty*` button (Section 4.3.2.10) to change the object's name and `MkLink*` (Section 4.3.2.5) to change the external reference to it.

4.3.2.9 Activating and Deactivating Objects

Usually, a library object is *active* in the sense that it may be referenced by tactics and other objects. Occasionally, users may want to experiment with alternate versions of a definition or theorem and to prevent tactics from using a particular object without having to remove it from the library. This can be done by changing the liveness bit of the object (see Section 4.1.1), indicated by the first character of the object's status information.

To deactivate an object, one moves the navigation pointer to it and clicks the `DeAct*` command button. To activate it again, one clicks the `Act*` command button. Notice that deactivating directories converts them into `TERM` objects and makes their contents (temporarily) inaccessible. Activating a code object will execute its content.

The same effects can be achieved by typing the commands
⌞`lib_thy_deactivate` *directory object*⌟ and ⌞`lib_thy_activate` *directory object*⌟
into the *library* ML top loop, where *directory* the object identifier of the directory of the object to be (de)activated and *object* the object identifier of the object itself.

4.3.2.10 Editing Object Properties

In advanced applications, users may want to change some of the properties of an object (see Section 4.1.1). For instance, when using abstraction objects to represent definitions of the PVS system, it makes sense to make them visible to PVS clients but not to the NUPRL refiner. In rare cases it may be necessary to adjust the reference environment (Section 4.3.3.1) of an object. Therefore, NUPRL provides a simple method for editing the properties of an object directly.

Clicking the `EditProperty*` command button will open a *property command zone* for the object at the navigation pointer on top of the current command zone. It contains a token slot for entering

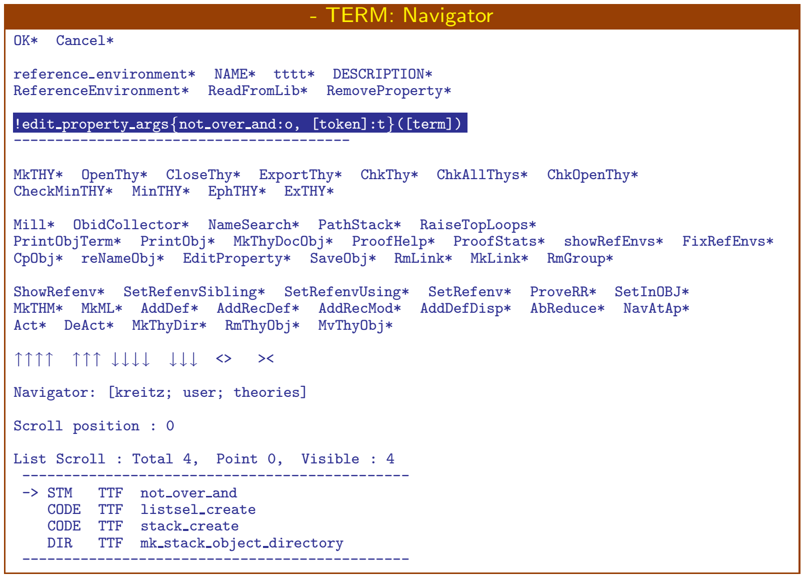


Figure 4.8: Editing Object Properties

the name of the property (e.g. `DESCRIPTION`, `NAME`, or `reference_environment`) and a term slot for entering the value of that property.

It is not recommended to enter the value of a particular property directly. Instead, one should make use of the command buttons for editing the important object properties that are immediately above the two slots. Each of the buttons in the top row represents a particular property of the current object, which will be inserted into the slots upon clicking the button. These buttons vary depending on the kind of the object.

Clicking the `NAME*` button, for instance, will insert the token `NAME` into the token slot and the term ⌞*object-name*`:t`⌟ into the term slot, where *object-name* is a token describing the object's name. Clicking `reference_environment*` will insert the token `reference_environment` into the token slot and the term ⌞`Obid:` *object-identifier*⌟ into the term slot, where *object-identifier* is the identifier of the object that is immediately before the current object in the ephemeral reference environment chain. A user may now edit these properties by modifying the corresponding terms.

The buttons in the bottom row are the same for all objects. Clicking `ReferenceEnvironment*` inserts the `reference_environment` property into the two slots, `ReadFromLib*` inserts the stored value of the current property back into the term slot, and `RemoveProperty*` removes that property from the object. Properties, once removed, can only be re-inserted explicitly.

Editing object properties should only be done by advanced users. Most users will rarely find it necessary to edit the properties of an object.

4.3.2.11 Saving Objects

During the development of formal theories, users may occasionally want to save the current version of an object before modifying it. Clicking the `SaveObj*` command button will copy the object at the navigation pointer to a subdirectory `.save` of the current directory. The internal name of the object will be preserved, which makes it easier to move the saved version back into its old location, and the reference to it will include a time stamp in its name. If the subdirectory `.save` doesn't exist yet, it will be created. The same effect can be achieved by typing the command ⌞`copy_to_save` *directory name*⌟ into the *library* ML top loop.

4.3.2.12 Printing Objects

Often users like to create print representations of objects in order to document their formal theories on paper or on the web. Clicking the PrintObj* command button will create a print representation of the object at the navigation pointer and write it into a file ~/nuprlprint/*name*_obj.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/*name*_obj.tex. The directory ~/nuprlprint must already exist. Otherwise clicking the PrintObj* button will result in an error. The same effect can be achieved by typing the command ⌞print_an_object *object*⌟ into the editor ML top loop.

In the conversion to LaTeX PrintObj* is capable of interpreting LaTeX syntax that occurs in a comment object and to re-interpret display forms as LaTeX macros, which allows for a more elegant type setting. In contrast to that, clicking the PrintObjTerm* command button will print the object contents as a single term without interpreting them further.

4.3.2.13 Commenting Objects

In addition to providing comment objects for an online documentation of formal material, NUPRL offers users the opportunity to produce formal articles that blend informal text with direct quotations of the formal material. For this purpose it allows the user to create comment objects that are linked to a specific object.

Figure 4.9: Commenting an object

Clicking the commentObj* button will create this object for the object at the navigation pointer and place it in a subdirectory .comments, which will be created, if it does not exist yet. The object contains a template that allows the user to enter comments that will be printed before (*prefix comments*) and after (*suffix comments*) the object when the theory containing the object is printed (Section 4.3.3.8).

4.3.2.14 Proof Help

NUPRL offers users a minimal form of online help for the development of proofs. Clicking the ProofHelp* button will open a comment object that describes the most important (standard) key bindings for the proof editor. Further online documentation will be added in the future.

Clicking the ProofStats* button while the navigator points to a statement object will pop up a window displaying some statistics about the proof of that statement. This can also be achieved by typing the command ⌞show_stm_stats *object*⌟ into the editor ML top loop.

4.3.3 Theory Operations

Theories are groups of objects that describe the definitions, theorems, and specific methods of reasoning of a mathematical or computational discipline. Formally they are organized like directories, but they contain objects describing their dependencies on other theories and they can also be associated with different sets of command buttons. For structuring purposes theories may be broken into sub-theories. However, these have to be ordinary directories instead of theory objects, since otherwise the dependency tracking mechanism may get confused when sub-theories are moved.

4.3.3.1 Object Dependencies and Reference Environments

While the notion of correctness of a formal proof is easy to define (see Chapter 6.1), the correctness of a formal theory depends on the fact that there is no circular chain of lemma references in its proofs. In principle, this could be guaranteed by requiring that a proof may only depend on lemmata that in some linear ordering of the library occur before the proof that refers to them. While keeping a certain discipline in the development of formal theories certainly helps avoiding circular references, some dependencies are hidden in various reference variables and proof caches employed by some of the tactics. In previous releases of NUPRL this fact often led to major problems when theories were replayed. NUPRL now supports a dependency checking mechanism that adds a layer of indirection between references and their values and allows greater control over these value during the development of formal proofs.

A *reference environment* (often abbreviated as `RefEnv` or `RE`) is an index into a graph of possible values for a set of reference variables. All refinements are parameterized by a reference environment. Reference environments are generally associated with statement and proof object via a `reference_environment` property (see Sections 4.1.1 and 4.3.2.10). Code objects can also have a reference environment property to parameterize reference variables during evaluation of the code.

The specification of a reference environment consists of:

- an *object identifier* describing the index being defined
- a list of *reference environments* to inherit from
- a list of *abstractions* to add
- a list of *lemmas* to add
- a list of *updates* consisting of snippets of code that update the value of a reference variable for the current index. Update code should not itself lookup values of reference variables.

Currently reference environments also contain a list of *additions* to the code of a code object, a method used in previous releases of NUPRL. Additions are preserved for compatibility reasons but they will be phased out in the future.

To contain updates to a reference environment, NUPRL uses code objects that have a `!property{reference environment additions:t}(update:t)` property. There are three varieties of reference environments.

- *Static reference environments* are ML code objects placed in theory directories that evaluate to a reference environment specification.
- *Ephemeral reference environments* are computed at refine time by chaining backwards through `reference_environment` properties of objects until a static reference environment is located. The reference environment specification is then built in a linear depth-first order: it includes the objects "above" it and all the objects in directories above it.
- *Minimal reference environments* are partial specifications of reference environments. Instead of defining an index, they bind an arbitrary temporary index for the scope of some evaluation. The intent is that only objects necessary for a successful evaluation will be listed. A minimal reference environment may be relative to a static one. Thus there are *flavors* of minimal reference environments. Currently, the following flavors are recognized.

  minimal: minimal relative to the empty environment.

  theories minimal: minimal relative to a set of theories, most commonly the standard theories.

  relative minimal: minimal relative to a specific theory. This is commonly used while developing a set of theories.

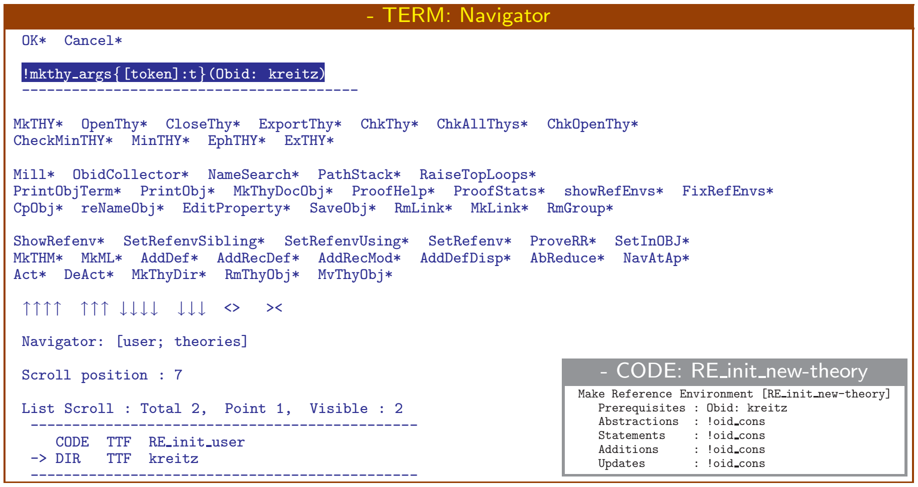


Figure 4.10: Creating Theories: initial template and generated static reference environment

The presence of static reference environments puts certain restrictions on the nesting of theories. Although in principle it would be possible to have theories be objects within other theories, there is no reliable method for moving such a theory within a theory directory without making its static reference environment inconsistent. Conceptually, most sub-theories are not autonomous theories in themselves, but only means for structuring a theory into smaller fragments. Placing them in sub-directories is more appropriate than opening a new theory. Therefore NUPRL does not allow theories to contain other theory objects.

4.3.3.2 Creating Theories and Sub-Theories

Theories are created by clicking the `MkTHY*` command button. This will open a token slot on top of the command zone into which a user may enter the name of the theory. Also visible is a reference to the object after which the theory will be placed. This object will be used for building the reference environment of the theory.

Upon closing the template (by clicking `OK*` or typing ↵) a new directory will be placed immediately after the navigation pointer. This directory contains a code object named `RE_init_`*theory-name*, the specification of the static reference environment of the new theory. An example of such a static reference environment is shown in Figure 4.3.3.2.

Users should not move theory objects or create objects immediately before them, without fixing the static reference environment, since otherwise the static reference environment of that theory would be inconsistent with the visible presentation of the library.

To create sub-theories of a theory, one should use the `MkThyDir*` button instead of `MkTHY*`. Like `MkDir*`, this will create an directory within the theory that does not contain a static reference environment. In addition to that, it adds a `theory` property to the directory object, which will help other commands maintain the reference environment chain within the theory.

Theories can also be created by typing ⌞`lib_mkthy` *preREs directory position name*⌟ into the editor ML top loop, where *preREs* is a list of object identifiers describing the reference environments on which the theory depends, *directory* is an object identifier of the directory in which the theory shall be placed, *position* a token describing the object after which it shall be positioned, and *name* a token describing its name. Theories directories can be created with the command ⌞`lib_mkthy_dir` *directory position name*⌟.

4.3.3.3 Changing Theory Modes

Depending on the reference environments (Section 4.3.3.1) of its objects, a theory can be in one of two *modes*.

- *open & ephemeral*, where all theory objects will have ephemeral reference environments, or
- *closed & minimal*, where all theory objects have some flavor of minimal reference environments and ephemeral reference environments are removed. Instead, the theory has a final reference environment object that summarizes the contents of the entire theory.

In addition to that, theories can also be *static* (or *explicit*), which means that all theory objects have static reference environments. This mode, however, is only needed to maintain older theories that have not yet migrated to be minimal or ephemeral. It will be phased out in the future.

When a theory is created with `MkTHY*`, an initial static reference environment is created and theory will be open and ephemeral until it is explicitly closed. The following command buttons can be used to change the mode of a theory.

- `CloseThy*` closes a theory by creating a finish reference environment named `RE_final_`*theory-name*, which summarizes the contents of the theory.
- `OpenThy*` opens or re-opens a theory by rebuilding its initial reference environment and resetting its ephemeral `reference_environment` property.
- `MinTHY*` will make a theory (relative) minimal and modify the available command buttons for the theory.
- `EphTHY*` will make a theory ephemeral by rechaining ephemeral reference environments and modify buttons for the theory.
- `ExTHY*` will make a theory explicit and modify the available command buttons for the theory.

The above commands can also be executed by typing

⌞`close_theory` *theory*⌟,
⌞`open_theory` *theory*⌟,
⌞`set_theory_relative_minimal` *theory*⌟,
⌞`set_theory_ephemeral` *theory*⌟, or
⌞`set_theory_explicit` *theory*⌟

into the *library* ML top loop, where *theory* is the object identifier of the current theory.

4.3.3.4 Examining and Modifying Reference Environments

To examine the current set of reference environments, one has to click the `showRefEnvs*` command button. This will pop up a new window containing a list if all existing static reference environments. Clicking on one of the terms with MIDDLE (or moving the cursor to it and pressing the right arrow key) will open the corresponding object, which shows the reference environment specification as an association list of reference variables and indices. Clicking MIDDLE on the on a variable/index pair will pop-up some indication of the data that is bound to that variable by that index.

Reference environments may also be examined by typing ⌞`show_ref_environments ()`⌟ into the editor ML top loop.

The normal methods for creating and manipulating theory objects will maintain the chain of ephemeral reference environments in a theory. Occasionally this chain may get corrupted when objects are moved or deleted. To fix this problem, a user has to click the `FixRefEnvs*` button. This will rechaining all the objects in a theory and thus update the ephemeral reference environment.

Reference environments may also be fixed by typing ⌞`reset_ephemeral_refenvs` *directory*⌟ into the *library* ML top loop, where *directory* the object identifier of the theory to be rechained.

In an open theory, users may insert static reference environments by clicking the `mkRefEnv*` command button. This will create an object named `RE_summary_`*theory-name*_*index* that summarizes all theory objects up to the current one and places it immediately after the current object.

Static reference environments are inserted into a theory mostly for debugging purposes. They enable a user to set the reference environment register (see Section 4.3.3.5 below) to a specific environment and to replay proofs in that environment to analyze dependencies in the proof.

Static reference environments may also be inserted by typing the command ⌞`add_refenv_summary` *directory position*⌟ into the *library* ML top loop, where *position* is a token describing the object up to which the theory should be summarized.

4.3.3.5 The Reference Environment Register

The *reference environment register* (briefly RR) is a global variable in the editor containing a reference environment index. The it is used as an implicit parameter in the some of the navigator commands and also when evaluating refiner top loop commands. The following command buttons can be used to examine or change the contents of the reference environment register.

- `ShowRefEnv*` shows the contents of reference environment register. This will be the empty term until one of the commands below has been applied.
- `SetRefenvSibling*` sets the reference environment register to the reference environment used by the current object.
- `SetRefenvUsing*` sets the reference environment register to the least reference environment that contains the current object. For ephemeral reference environments this command will be phased out, since an object is the least reference environment containing itself.
- `SetRefenv*` sets the reference environment register to the current object.
- `ProveRR*` attempts to replay the proof of the statement at the navigation pointer using the reference environment register instead of the object's reference environment. It allows a user to make a copy of a proof experiment without modifying the original proof. The proof will be attempted asynchronously, so the command will return immediately. When it finishes it will pop-up a window containing the object identifier of the proof generated. Clicking on the object identifier with MIDDLE will pop up the proof.

  Note that the new proof is *not* linked to the statement. It will remain unlinked if the proof is closed with ⟨C-q⟩. If instead one uses ⟨C-z⟩ to exit, the proof will be prepended to the statement's proof list (see Chapter 6).
- `SetInObj*` sets the reference environment register to the `reference_environment` property of the first proof of the statement at the navigation pointer.

The above commands can also be executed by typing

⌞`show_refenv_register ()`⌟,
⌞`set_refenv_register_sibling` *term*⌟,
⌞`set_refenv_register_using` *term*⌟,
⌞`set_refenv_register` *term*⌟,
⌞`prove_using_refenv_register` *term*⌟, or
⌞`set_re_in_first_prf` *term*⌟

into the editor ML top loop, where *term* is a term describing the directory and the position of the current object.

4.3.3.6 Checking Theories

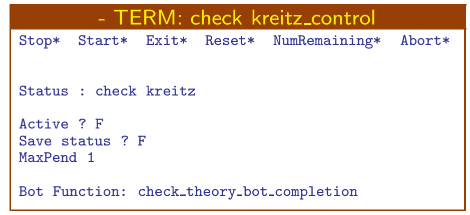


Figure 4.11: Checking a theory

Although NUPRL proof environment guarantees that proofs are correct wrt. the available set of rules, refiners, and lemmata at the time the proof is being constructed, a stored proof may become invalid if the rules and lemmata on which it depends are removed or modified afterwards. The NUPRL library provides a certification mechanism that accounts for the validity of its contents. However, it would be computationally infeasible to recheck these certificates whenever a library object is modified. Instead, the system provides a utility that enables users to explicitly check the consistency of their theories by replay the proofs in a controlled fashion.

To do so, one has to move the navigation pointer to the root of the theory and click the `ChkThy*` command button. This will cause the system to accumulate the object identifiers of the proofs of all the statements in the final reference environment of the theory and then pop up a control window for initiating the checks (Figure 4.3.3.6). Since `ChkThy*` has to determine the final reference environment, it will fail if the theory lacks an initial static reference environment. The command buttons in the control window have the following effects.

- `Stop*` completes the replay of the current proof and then stops the check.
- `Start*` starts the replay of the (remaining) proofs in the theory. (Intermediate) results will be displayed in a separate window.
- `Exit*` close the command window. As proofs are replayed asynchronously this will not stop the ongoing checks.
- `Reset*` resets the list of remaining proofs to be checked to the initial list of proofs.
- `NumRemaining*` displays the remaining number of proofs to be checked.
- `Abort*` aborts the ongoing check and resets the list of remaining proofs to be checked.

Users may also modify the parameters of the check mechanism. The number after `MaxPend` indicates how many should maximally be used for checking proofs. Using more than one refiner is helpful when checking large theories but it takes these refiners away from other tasks. The `Save status ?` bit determines whether to save the status of the checking mechanism when the command window is saved. The `Active ?` bit and the check function after `Bot Function:` should not be changed by a user.

Instead of clicking the `ChkThy*` command button, one may also type the command ⌞`build_check_theory_bot` *directory*⌟ into the editor ML top loop. Although the individual check commands could be issued from the top loop as well, it is not advisable to do so.

NUPRL provides a few variations of the `ChkThy*` command.

- `ChkAllThys*` initiates a check for all the theories in the library. This is the same as clicking `ChkThy*` with the navigation pointer at the root of the `theories` directory.
- `ChkOpenThy*` accumulates *all* the proofs in the theory instead of only proofs of statements in the final reference environment. This command can also be used tom check sub theories, as it does not attempt to build the final reference environment of a theory.
- `ChkMinThy*` initiates a check with a minimal reference environment. Users have to enter one of the flavors `reference_environment_minimal`, `reference_environment_theories_minimal`, or `reference_environment_relative_minimal` into a token slot that appears above the command zone.

4.3.3.7 Exporting and Importing Theories

A capability for exporting and importing theories is important for moving theories between different libraries in a controlled fashion. NUPRL exports theories into files containing raw library data and rebuilds objects from these files when importing theories.

To export a theory one has to click the `ExportTHY*` command button with the navigation pointer at the theory object. This will collect all the objects in the theory and write them into a file `~/nuprlpatch/`*theory-name*`_theory.trm`.

Alternatively, a user may type the command `dump_theory true (`*directory*`,` *theory-object*`)` into the editor ML top loop, where *directory* is the object identifier of the directory where the theory resides and *theory-object* the object identifier of the theory itself.

To import a theory from a file, one has to enter the command `replace_objects` *path-name* into the editor ML top loop, where *path-name* is the complete path name of the theory's dump file. This will create a directory containing all the objects of the dumped theory and place it at the *same location* in the user's work space. If the theory already exists, the objects of the dumped theory will be added to the theory directory. Objects will not be overwritten: in case of name clashes, the existing theory object will be renamed if its content is different from the new theory object. If the two objects are identical, the new object will be ignored.

4.3.3.8 Printing Theories

To print the contents of an entire theory, a user may either click the `PrintThyShort*` or the `PrintThyLong*` command buttons. This will create a print representation of the objects in the theory at the navigation pointer and write it into a file `~/nuprlprint/`*name*`.prl` (or `~/nuprlprint/`*name*`_long.prl`). It will also create a LaTeX-presentation of the theory and write it to `~/nuprlprint/`*name*`.tex` (or `~/nuprlprint/`*name*`_long.tex`).

`PrintThyShort*` provides a less detailed presentation of the theory, which omits the proofs of a theorem and only includes the extract term if a theorem is complete. In contrast to that `PrintThyLong*` adds the complete proof to the presentation of a theorem. Users who are only interested in a listing of all the object names in a theory may do so by clicking the `PrintObj*` button (Section 4.3.2.12).

Theories can also be printed by typing the command `short_print_theory` *theory-object* or `or_print_theory` *theory-object* into the editor ML top loop, where *theory-object* is the object identifier of the theory to be printed.

4.3.3.9 Creating Theory Documentation

The commands for printing theories only create listings of theory contents in linear order, possibly augmented by comment objects as described in Section 4.3.2.13. In addition to these, NUPRL provides a more flexible mechanism for creating formal documentation that enables a user to insert (references to) formal objects into informal text.

Clicking the `MkThyDocObj*` command button creates a comment object `thy_doc-`*timestamp* that contains pointers to all the statement and abstraction objects in the current theory. Users may then edit the object to write formal articles by adding text and rearranging and duplicating the existing pointers. Printing the object with `PrintObj*` will then create a LaTeX article that documents the formal theory. The advantage of this approach is that the formal article is always up to date, even if a user chooses to change the formalization of a theorem or the display form of an abstraction.

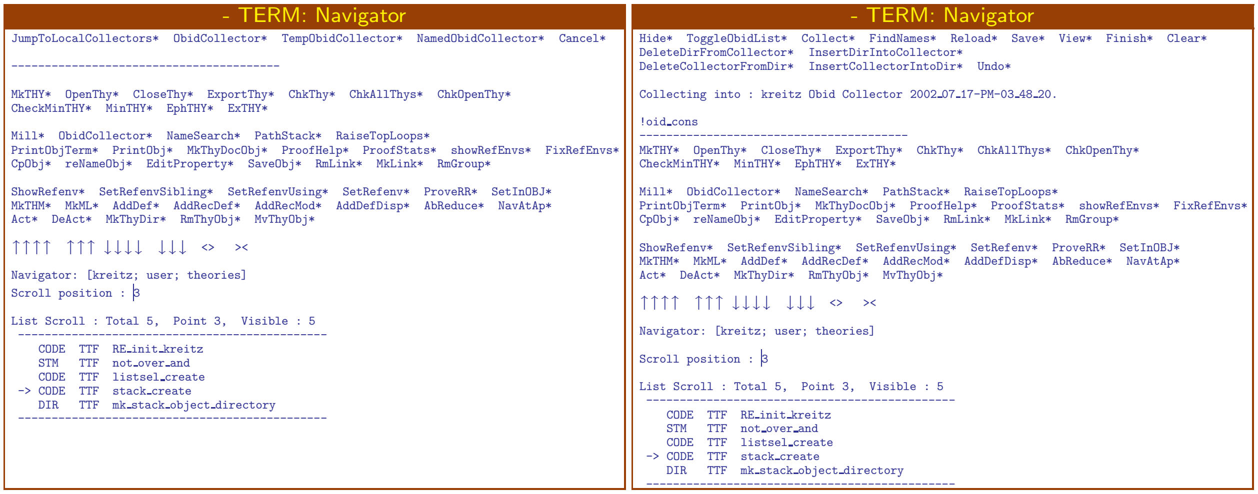


Figure 4.12: Creating Object Collections

An example of an article automatically generated from such an object can be found at `http://www.cs.cornell.edu/home/kreitz/Abstracts/01cucs-HybridProtocol-nuprl.html`.

Theory documentation objects can also be created by typing ⌞`make_thy_doc_object` *directory*⌟ into the editor ML top loop, where *directory* is the object identifier of the directory to be documented.

4.3.4 Miscellaneous Operations

4.3.4.1 Creating Object Collections

An *Obid Collector* is a method of collecting a list of object identifiers to be used as an argument to navigator commands. An Obid Collector persists as an object but the navigator also maintains a cache of the collector's list of object identifiers.

To build an obid collector, one has to click the `ObidCollector*` command button. This will create a *collector command zone* on top of the current command zone, which requires the user to choose between several options, shown on the left of Figure 4.12.

- `JumpToLocalCollectors*` jump the navigator to a directory containing the list of named collectors.
- `ObidCollector*` use the object at the navigation pointer as obid collector. The object has to be of kind `TERM` or `COM`.
- `TempObidCollector*` create an ephemeral collector.
- `NamedObidCollector*` create a new named collector.

Named collectors are stored in the library and will persist from session to session while ephemeral collectors will be discarded at the end of a session.

After the user has created or re-opened an obid collector the collector command zone contains a variety of buttons that modify the collector, shown on the right of Figure 4.12.

- `Hide*` Hide the search collector command zone by iconifying it to a button `ObidCollector#`.
- `ToggleObidList*` hides or shows the list of object identifiers in the collector.
- `Collect*` adds the object at the navigation pointer to the collector.

- FindNames* starts a dialog to search for objects in the library by name. If the string entered by the user matches the name of an object exactly, then its object identifier will be added to the collector. This is useful for finding objects not in directory tree and then adding them to a directory with InsertCollectorIntoDir*.
- Reload* loads the stored object identifiers list into the collector's navigator cache.
- Save* dumps the object identifiers list from the collector's navigator cache to the collector object.
- View* opens the collector object for editing purposes.
- Finish* saves the object identifiers list and then removes the collector from the navigator.
- Clear* clears the collector's navigator cache.
- DeleteDirFromCollector* subtracts the object identifiers of objects in the current navigator directory from the collector.
- InsertDirIntoCollector* adds all object identifiers of objects in the current navigator directory from the collector.
- DeleteCollectorFromDir* removes the object identifiers in the collector from the current navigator directory.
- InsertCollectorIntoDir* adds the object identifiers in the collector from the current navigator directory.
- Undo* undoes last Insert or Delete operation.

Although all obid collector commands could also be issued from the editor ML top loop, it is not advisable to do so.

Printing Object Collections

Users may print the objects in a collection by clicking the PrintCollection* button when the navigation pointer is at a collector object. This will create a print representation of the objects listed in the collector and write it into a file ~/nuprlprint/*collector-name*.prl and a LaTeX-presentation, which will be written to the file ~/nuprlprint/*collector-name*.tex. The objects are printed in the order in which they were added to the collector, i.e. in the reverse order of the object identifier list in the collector object.

Collections may also be printed by typing the command ⌞print_collection *object*⌟ into the editor ML top loop, where*object* is the object identifier of the collector.

4.3.4.2 Milling

Nuprl provides a framework for developing tools for importing and migrating data from external libraries into Nuprl's data repository. This utility can be used for a wide variety of tasks such as searching for objects that contain a specified combination of object identifiers or for objects that have been modified within a given time specification. It is initiated by *milling* a theory.

Clicking Mill* while the navigation pointer is at a directory will open a tag slot above the current command zone into which the user may enter a tag for the milling directory to be created. The system will then create a sub-directory .*tag-name* mill at the beginning of the indicated directory. This directory comes with a variety of new command buttons and examples of code pieces that can be assembled for the tasks the user wants to perform.

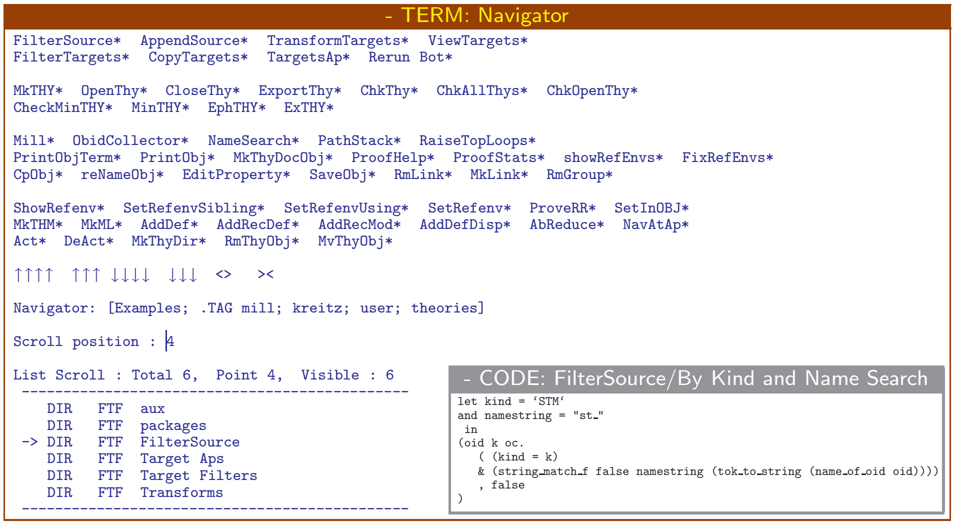


Figure 4.13: Standard Milling Directory

To perform a particular operation, users should copy the corresponding object from one of the sub-directories of the directory `Examples` into the main milling directory, modify the "declaration" part of the code and then click the command button that corresponds to the name sub-directory from where the piece of code was copied.

The object `By Kind and Name Search` in the directory `FilterSource` shown in Figure 4.13, for instance, contains the code for collecting all the objects of the milled directory that have a given kind and name. Changing the let-binding of the variables `kind` and `namestring` to `STM` and `st_` will cause the search to focus on statement object whose name contains the string `st_`. Clicking the button `FilterSource*` will initiate the search and collect all the found objects in a sub-directory `Targets`. If that directory already exists the user will be asked to confirm that its previous contents can be removed. Code pieces are provided in the following categories.

- `FilterSource*` collects all objects of the milled directory that satisfy a given predicate and places them in the sub-directory `Targets`. The search predicate may involve the kind, name, status, or creation time of the object, strings and object identifiers occurring in it, and similar criteria. Existing contents of the directory `Targets` will be overwritten.
- `AppendSource*` filters the milled directory that satisfy a given predicate and *adds* the found objects to the directory `Targets`.
- `TransformTargets*` applies a specific transformation to the contents of all objects in the directory `Targets`.
- `ViewTargets*` opens a window displaying the object identifiers in the directory `Targets`.
- `FilterTargets*` pops up a list of objects in the directory `Targets` that satisfy a given predicate from the directory `Target Filters`. The directory `Targets` itself will not be modified.
- `CopyTargets*` copies the contents of the directory `Targets` to a directory `Copies`.
- `TargetsAp*` applies a specific operation to all objects in the directory `Targets`. Current operations include activating and deactivating, adding properties, and counting.
- `Rerun Bot*` creates two `TERM` objects in the milling directory that allow replaying proofs in the directory `Targets` on a fine level of detail. The purpose of this operation is to safely rebuild proofs that are imported from different (or older) proof environments and fail during replay.

Milling directories can also be built by typing ␣build_mill_dir *directory tag-name*␣ into the editor ML top loop, where *directory* is the object identifier of the directory to be milled and *tag-name* a token describing the the tag for the milling directory.

4.3.4.3 NavAtAp

Clicking the NavAtAp* command button when the navigation pointer is at a code object will replace the final argument of the code in the object with the term inr(*directory, position*), where *directory* the object identifier of the current directory and *position* the name of the object at the navigation pointer, and then evaluate the code.

The main purpose of this command is compatibility of the methods for creating recursive definitions (Section 4.3.2.2) and modules (Section 4.3.2.3) with the ones used in libraries developed with Nuprl 4. The command is likely to be removed in the future.

4.3.4.4 Cloning the Navigator

Users who want to use multiple navigators simultaneously may do so by cloning the navigator. Clicking the Clone* command button will open a new navigator window that is a clone of the current one. Alternatively, a user may type the command ␣dyn_navigator_clone *term*␣ into the editor ML top loop, where *term* is the complete term contained in the current navigator window.

Note that navigator windows, like the ML top loop and the evaluator history window cannot be closed with ⟨C-q⟩ again.

4.3.4.5 Raising the ML top loop window

If the ML top loop is buried under other windows, clicking the RaiseTopLoops* command button will bring the ML top loop window to the foreground. This feature works currently only in the twm window manager.

Table 4.2 summarizes all the navigator command buttons that are described in this manual. The buttons in the upper part of the table occur in all standard user theories, while the other buttons are only present in some of the standard theories.

4.4 The ML Top Loop

The ML Top Loop, shown in Figure 4.14 on the left, provides an interactive interface to Nuprl's editor, refiner, and library ML processes. It can be used to evaluate ML expressions and declarations and to issue commands that change the state of the three processes. Commands have to be entered into the command line zone between the command line prompt and the double semicolon, the termination characters for ML expressions. Commands that have been evaluated are stored in a *command history*, which makes it possible to recall and modify complex commands. The ML Top Loop also contains a command zone with command buttons that affect the behavior of the editor itself.

4.4.1 Top loop command buttons

The buttons in the top line of the Top Loop command zone interact with the contents of the command line zone, the ones below have more global effects

Button	Command	Section
`AbReduce*`	Update the `Reduce` tactic	4.3.2.2
`Act*`	Activate an object	4.3.2.9
`AddDef*`	Create a definition	4.3.2.2
`AddDefDisp*`	Create a display form	4.3.2.1
`AddRecDef*`	Create a recursive definition	4.3.2.2
`AddRecMod*`	Create a (recursive) module	4.3.2.3
`CheckMinTHY*`	Check with theory minimal RefEnv	4.3.3.6
`ChkAllThys*`	Check all library theories	4.3.3.6
`ChkOpenThy*`	Check all proofs in theory	4.3.3.6
`ChkThy*`	Check a theory	4.3.3.6
`CloseThy*`	Close/finalize a theory	4.3.3.3
`CpObj*`	Copy an object	4.3.2.4
`DeAct*`	Deactivate an object	4.3.2.9
`EditProperty*`	Edit object properties	4.3.2.10
`EphTHY*`	Make theory ephemeral	4.3.3.3
`ExTHY*`	Make theory explicit	4.3.3.3
`ExportThy*`	Export theory to file	4.3.3.7
`FixRefEnvs*`	Update ephemeral RefEnv	4.3.3.4
`Mill*`	Mill a theory	4.3.4.2
`MinTHY*`	Make theory (relative) minimal	4.3.3.3
`MkLink*`	Create a link	4.3.2.5
`MkML*`	Create a code object	4.3.2.1
`MkTHM*`	Create a statement object	4.3.2.1
`MkTHY*`	Create a theory	4.3.3.2
`MkThyDir*`	Create sub-theory directory	4.3.3.2
`MkThyDocObj*`	Create theory documentation object	4.3.3.9
`MvThyObj**`	Move an object	4.3.2.7
`NameSearch*`	Search for object names	4.3.1.2
`NavAtAp*`	Apply code to current position	4.3.4.3
`ObidCollector*`	Build Obid Collector	4.3.4.1
`OpenThy*`	(Re-)open a theory	4.3.3.3
`PathStack*`	Advanced motion commands	4.3.1.3
`PrintObj*`	Print an object	4.3.2.12
`PrintObjTerm*`	Print an object as a term	4.3.2.12
`ProofHelp*`	Pop up proof help window	4.3.2.14
`ProofStats*`	Display proof statistics	4.3.2.14
`ProveRR*`	Replay proof using RR	4.3.3.5
`RaiseTopLoops*`	Raising ML top loop window	4.3.4.5
`RmGroup*`	Remove a group of objects	4.3.2.6
`RmLink*`	Remove a link	4.3.2.6
`RmThyObj*`	Remove a library object	4.3.2.6
`SaveObj*`	Save copy of an object	4.3.2.11
`SetInOBJ*`	Set RR to RefEnv of proof	4.3.3.5
`SetRefenv*`	Set RR to object	4.3.3.5
`SetRefenvSibling*`	Set RR to current RefEnv	4.3.3.5
`SetRefenvUsing*`	Set RR to least RefEnv containing object	4.3.3.5
`ShowRefenv*`	Show RR	4.3.3.5
`reNameObj*`	Rename an object	4.3.2.8
`showRefEnvs*`	Display existing RefEnvs	4.3.3.4
`Clone*`	Clone the navigator	4.3.4.4
`MkDir*`	Create a directory object	4.3.2.1
`MkObj*`	Create a library object	4.3.2.1
`PrintCollection*`	Print collection of objects	4.3.4.1
`PrintThyLong*`	Print theory with proofs	4.3.3.8
`PrintThyShort*`	Print theory contents	4.3.3.8
`RmDir*`	Completely remove a directory	4.3.2.6
`commentObj*`	Create comments for an object	4.3.2.13
`mkRefEnv**`	Insert static RefEnv	4.3.3.4

Table 4.2: Navigator command buttons

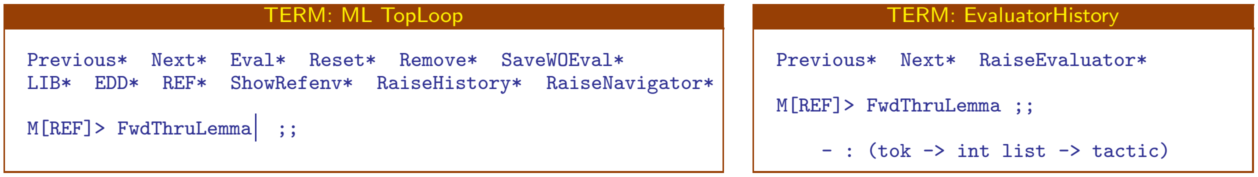


Figure 4.14: The ML Top Loop and the Evaluator History Window

Previous*: Insert the previous command from the command history into the command line zone.

Next*: Insert the next command from the command history into the command line zone.

Eval*: Evaluate the command that is currently in the command line zone.

Reset*: Reset the command line zone.

Remove*: Remove the current command from the history and reset the command line zone.

SaveWOEval*: Save the current command to the command history without evaluating it.

LIB*: switch to interaction with the library ML process.

EDD*: switch to interaction with the editor ML process.

Ref*: switch to interaction with the refiner ML process.

ShowRefenv*: show the contents of reference environment register (Section refsec:nav-RefEnvReg), i.e. the reference environment of the ML expression in the command line (this makes sense only in refiner mode).

RaiseHistory*: open an *evaluator history* window that shows the output from evaluating the ML expression in the command line zone.

The window (shown on the right of Figure 4.14) shows the command prompt of the corresponding process, the ML expression, its value, its type, and a time stamp. It also provides three buttons. `Previous*` and `Next*` are used for going backward and forward in the evaluator history. `RaiseEvaluator*` inserts the current history command back into the ML top loop, provided the command and the ML top loop interact with the same process.

RaiseNavigator*: bring the navigator window to the foreground (works currently only for `twm`).

4.4.2 The command line zone editor

The command line zone provides a *term editor*. The editor is initially in *text mode*, indicated by the text cursor |, which allows the user to enter ML text. NUPRL terms may be inserted into this text by opening a term slot and entering terms as described in Chapter 5. Most of the editor commands described in Chapter 5 will work the same way in the command line zone. The only exception is the return key ↵, which sends the command to the ML evaluator instead of inserting a new line as in the term editor. The commands and key bindings of the command line zone editor that differ from those of the regular term editor are listed in Table 4.3.

↵	EVALUATOR_EVAL	call ML evaluator
⟨S-↵⟩	INSERT-NEWLINE	add line-break
⟨C-R⟩	EVALUATOR_PREVIOUS	scroll back through history

Table 4.3: Command line zone editor commands and bindings

To evaluate an expression, type it in at a text cursor after the command prompt and then use either the `Eval*` button or the return key ↵. You may edit the expression using the term-editor commands described in Chapter 5. To break an expression into several lines, use ⟨S-↵⟩. Output from evaluating the ML expression in the command line zone is usually directed to the evaluator history window (although it is possible to have it appear below the command line zone in the ML top loop window as well).

Nuprl error messages appear in a separate window that describes the nature of the error and some debugging information. The window can be closed by either clicking it's Quit* button or by typing ⟨C-`Q`⟩. Errors can come from various sources. In most cases the ML expression you typed doesn't parse or type-check properly or has been sent to the wrong process.

Occasionally you can get the ML top loop into an unexpected state. In this case undo the previous steps using ⟨C-_⟩ until a stable state has been recovered. If that doesn't work, the editor process itself may have reached an unrecoverable state. It is best to close all other Nuprl windows, saving their contents if needed, to kill the editor process, and to start it again.

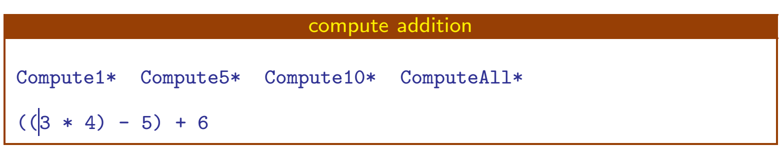
4.4.3 Top Loop Commands

Depending on the command prompt at the beginning of the command line zone top loop commands are sent by the editor to either the library, editor, or refiner processes. Refiner commands usually involve evaluating existing or newly developed tactics (Chapter 8) and related ML functions, or analyzing proof details that are not shown by the proof editor (Chapter 6). Library commands affect the contents of the library as permitted by the corresponding library application interface. Editor commands change the visible contents of the navigator (which my also affect the library), modify the behavior of the editor itself, or open new windows that invoke specific applications such as a proof editor or the Nuprl term evaluator.

Most of the editor and library top loop commands have been described in Section 4.3. In this section we briefly summarize other commands that may be of interest for a user.

4.4.3.1 Invoking the Nuprl term evaluator

The ML top loop provides the means to evaluate expressions of Nuprl's meta language ML. Nuprl's object language, however, comes with its own notion of evaluation (see Table A.2 in Appendix A.2.1), which is supported by a separate *term evaluator*. To invoke this evaluator, a user has to type the command ⌞`view_showc` *name term*⌟ into the editor ML top loop, where the token *name* will be a suffix to the name of a new window and *term* is a Nuprl term to be evaluated. This will open a new window with the name '`compute` *name*' that contains the Nuprl term *term* and four buttons that initiate the evaluation of the term.



Clicking `Compute1*` once will perform one top-level reduction step on the Nuprl term. Clicking it again will perform the next step, and so on. For the term ⌜`((3*4)-5)+6`⌝, for instance, this will result in the reduction sequence ⌜`((3*4)-5)+6`⌝ ⟶ ⌜`(12-5)+6`⌝ ⟶ ⌜`7+6`⌝ ⟶ ⌜`13`⌝.

The other three buttons proceed in larger steps. `Compute5*` and `Compute10*` perform 5, respectively 10 computation steps at once. `ComputeAll*` continues with the evaluation until no further reduction is possible. To undo a computation step, simply use the undo key combination ⟨C-_⟩.

Note that evaluation in NUPRL is lazy: evaluating the term ⌜⟨((3*4)-5)+6⟩7⌝ will leave it unchanged. Using `ComputeAll*` on terms like ⌜`Y (λx.x)`⌝, whose evaluation does not terminate will cause the library and refiner processes to loop indefinitely. A user will have to interrupt these processes and bring them back into a stable state (see Section 4.6 below).

With a similar command, a user may also invoke the NUPRL term evaluator on the extract term of a theorem (see Section 6.3.3). Typing ⌞`view_show_co` *obid*⌟ into the editor ML top loop will open a NUPRL term evaluator window that contains as its term argument the extract term of the theorem object denoted by the term *obid*. This, however, requires that the extract term of the theorem has been made available to the editor. If this hasn't been done already, one has to enter the command ⌞`require_termof (ioid` *obid*`)`⌟ before invoking the evaluator.

4.4.3.2 Loading and compiling ML code

NUPRL allows users to load files containing ML code into the library, editor, or refiner processes. To do so, a user has to type the command ⌞`loadt_system` *root-dir* `[`$path_1$, $files_1$ `; ...;` $path_n$, $files_n$`]`⌟ into the respective ML top loop. *root-dir* is a string describing the root directory of the user's NUPRL files. *path* is a list of strings describing the sub-directory in which the specific files reside, and *files* is a list of file names (without the `.ml` extension) in that directory. The command will compile all the named files and load the compiled code into the current process. For instance,

```
loadt_system "/home/nuprl/lib/ml"
             [ ["standard"],["a";"b"]; ["testing";"new"],["d";"e";"f"] ]
```

will compile and load files in the following order

```
"/home/nuprl/lib/ml/standard/a.ml"
"/home/nuprl/lib/ml/standard/b.ml"
"/home/nuprl/lib/ml/testing/new/d.ml"
"/home/nuprl/lib/ml/testing/new/e.ml"
"/home/nuprl/lib/ml/testing/new/f.ml"
```

Compiled files will be stored in a sub-directory `mlbin/`*os*`/`*lisp-version* of the directory in which the ML files reside, where *os* is currently either `linux` or `solaris`, and *lisp-version* is the name of the Lisp dialect that runs the process, e.g. `allegro61` or `cmucl`. If these directories do not yet exists, an error message will be created.

4.4.3.3 Importing Text

One of the advantages of having code reside within the NUPRL library instead of in external files is that NUPRL links ML functions to the code object in which they are defined. Clicking the middle mouse button MIDDLE on a piece of ML text will raise the code object where the corresponding ML function is defined, provided its definition is stored within the library.

One way to migrate an ML file into the system is to import its text. Entering the command ⌞`import_text` *filename*⌟ into the editor ML top loop will open a new window containing the contents of the file described by the string *filename*. Users may then copy and paste pieces of the text into any term editor that is in text mode such as the ML top loop or code and comment objects. This feature also simplifies the on-line documentation of theories, as it allows importing previously written text and turning it into comment objects.

4.5 Process Top Loops

The Process Top Loops are NUPRL's interface to the system processes that run the editors, refiners, or the library. They represent the top loops of the corresponding ML interpreters and do not provide any editing features. Their main purpose is to display system output and error messages and to execute maintenance and debugging commands. Usually, they are run within an emacs shell to have some text editing support.

Most users will hardly ever use the process top loops except for monitoring the process in case of long delays (see Section 4.6 below). There are, however, a few useful commands that advanced users may want to take advantage of.

Most commands have to be entered as conventional ML expressions, which must be terminated explicitly by a double semicolon. Users may also switch to Lisp mode and enter low-level system commands in Lisp. These commands need to be terminated by a double semicolon as well, but will be forwarded to the Lisp interpreter. For both ML and Lisp there are also a few *dotted commands*. These are expressions without arguments that are terminated by a period. Below we list some of the most commonly used commands.

- Editor Commands:
  - ⌞`nuprl_oed_suspend ();;`⌟ closes all NUPRL windows.
  - ⌞`nuprl_oed_resume ();;`⌟ reopens the NUPRL windows.
  - ⌞`nuprl_oed_reset ();;`⌟ : kills all NUPRL windows.
  - ⌞`win.`⌟ opens the navigator and ML top loop windows.
  - ⌞`set_xhost "`*hostname*`"` *display_index*`;;`⌟ redirects all NUPRL windows to the specified display after the next suspend/resume cycle. *hostname* must be a string describing the display host and *display_index* a number (usually 0) specifying the display terminal on that host.
  - ⌞`nuprl_oed_rehash ();;`⌟ rehashes the macros and bindings in the user's `mykeys.macro` file (see Section 3.2.2).

- Commands for the Editor or Refiner:
  - ⌞`setup_connect` *socket*$_1$ *socket*$_2$ `"`*hostname*`";;`⌟ sets up a connection to the library at host *hostname* using the indicated socket numbers.
  - ⌞`dc ();;`⌟ try to establish the connection that was set up.
  - ⌞`dd ();;`⌟ disconnect the process.
  - ⌞`open_lib '`*lib-memnonic*`';;`⌟ opens the connection to the library environment called *lib-memnonic*.
  - ⌞`close_lib '`*lib-memnonic*`';;`⌟ closes the connection to the library environment *lib-memnonic*.

  The difference between the commands `dc`/`dd` and `open_lib`/`close_lib` is that the former establish the low-level TCP/IP connection to the library's object request broker, while the latter link to a client work space provided by the library (see Section 4.1).

  The above commands are implicitly executed when the editor and refiner processes are started, using the data contained in the user's `.nuprl.config` file (see Section 3.2.2).

- Library commands:
  - ⌞`nosa` *socket1*`;;`⌟ Opens a connection to a client using the indicated socket number.

– ␣library_open ``*lib-memnonic*``;;␣ opens the library environments *lib-memnonic* for external connections. Usually, one opens the library environment that was stored the last time the library was closed, but there may be reasons to re-open older library environments.
– ␣library_open_as ``*lib-memnonic*`` `*new_memnonic*`;;␣ opens the library environments *lib-memnonic* under the alias *new_memnonic*.
– ␣library_close `*libenv*`;;␣ closes the library environment *libenv*.
– ␣library_close_gc `*libenv*`;;␣ closes the library environment *libenv*, performing garbage collection first. Unlinked library objects will not be included in the stored environment. However, they will not be removed from the data base and may be recovered by opening an older environment.
– ␣db_envs_print ``*memnonic-match*``;;␣ Print a list of all existing library environments that match all the tokens in the list *memnonic-match*.

Library Lisp commands:

– ␣(stop-db-buffering);;␣ stops buffering data base information. This is useful when one sees buffering messages like ␣WBI 23␣ or ␣LBI 25␣ building up to high numbers and never decreasing.

- Commands for all processes:
  – ␣envs ();;␣ prints a list of the environments currently accessible by the process.
  – ␣l.␣ switches to Lisp mode.
  – ␣ml.␣ switches to ML mode.
  – ␣stop.␣ terminates the process.

4.6 Recovering from Errors

Most Nuprl errors relate to commands that were entered into the navigator, the ML top loop, or the proof editor. Quite often they have to do with misspelled tactics or ML functions, type errors, or unsuccessful executions of the command. In these cases, error messages appear in a separate window that describes the nature of the error and some debugging information. Usually, it suffices to re-enter the command after correcting the mistake.

Many library commands that were executed erroneously, like removing an object or unlinking a directory, can be undone by entering the key combination ⟨C-_⟩ (see Section 4.3). Since the library never destroys information, it is possible to retrieve the contents of every object that was previously accessible. The undo history is limited, though. Recovering from an error that was made many steps ago is more difficult.

If a user enters text into the navigator while the cursor is at the "scroll position" the navigator will show an error after the next operation. Using the undo operation until the entered text is removed and moving the cursor to where it is supposed to be solves the problem.

Sometimes the contents of a window are not updated after resizing it. Scrolling down and up with ⟨C-v⟩ and ⟨M-v⟩ usually forces the window to be updated.

If a user has messed up the contents of a window and cannot undo the error, closing the window with ⟨C-q⟩ and opening it again will often solve the problem. ⟨C-q⟩ closes the window without saving the modifications to the object, so reopening the object will show the state after the last save operation (usually ⟨C-z⟩). Note that the navigator, the ML top loop, and the evaluator history cannot be closed without using the editor commands from Section 4.5.

If the system appears to be inexplicably stuck, check the ML process loops. It is possible that one of them is *garbage-collecting*, which may take up to several minutes depending on processor speed and available memory.

In rare cases, one of the three Lisp processes crashes and ends up in debug mode, which offers several restart actions to the user. Entering the Lisp command ⌞`(fooe)`⌟ after the prompt usually brings the process back to a stable state.

If a user has initiated a non-terminating computation, for instance by entering a recursive ML expression into the ML top loop or by applying the NUPRL term evaluator to a term containing the Y combinator, the corresponding process must be interrupted explicitly. Typing ⟨C-c⟩ repeatedly will eventually break the Lisp process, which then can be restarted with `(fooe)`.

If everything else fails, one may have to restart the editor, the refiner, or even all three NUPRL processes. Interrupt the Lisp process with ⟨C-c⟩, type ⌞`:exit`⌟ (or kill the process from Unix), and then start it again as described in Chapter 3. If all three NUPRL processes have to be shut down, it is best to stop those that are still alive using the ⌞`stop.`⌟ command, shutting down the library last.