The Nuprl Proof Development System, Version 4.2
Reference Manual and User's Guide

Paul Jackson

January 22, 1996

Contents

Chapter 1

Introduction

1.1 Purpose

This manual is a reference manual for version 4.1 of the Nuprl system. It is aimed at beginning and intermediate users of the system. Version 4.1 runs on Unix-based workstations that use the X window system.

Note that this manual is still under development and is incomplete. Most importantly, it is missing information on Nuprl's type theory, and the structure of the primitive rules as perceived by users when executing low-level tactics.

Information on the ML language can be found in a separate Nuprl ML manual. Tutorials on the use of Nuprl's term and proof editor are also available.

1.2 Conventions

We give the conventions we use in this manual for presenting user input and Nuprl output.

Input which you should type is presented typewriter font. For example `this is in typewriter font`. The following symbols are also used:

- SPACE for the space-bar.
- RETURN for the return key (sometimes marked as "enter").
- LINEFEED for the linefeed key.
- TAB for the tab key.
- DELETE for the delete key (sometimes marked as rubout). On some keyboards the BACKSPACE has the same effect.
- MOUSE-LEFT for the left mouse button.
- MOUSE-MIDDLE for the middle mouse button.
- MOUSE-RIGHT for the right mouse button.

Modified keys are presented as follows:

- ⟨C-x⟩ read as "control x". Hold down a control key and simultaneously press key x.
- ⟨M-x⟩ read as "meta x". Hold down a meta key and simultaneously press key x.
- ⟨CM-x⟩ read as "control meta x". Hold down both a control key and a meta key, and simultaneously press key x.
- ⟨S-x⟩ read as "shift x". Hold down a shift key and simultaneously press key x.

Note that x is either a keyboard key *or* a mouse button; for example both ⟨C-`a`⟩ and ⟨M- $\boxed{\text{MOUSE-RIGHT}}$ ⟩ are valid modified keys.. On some keyboard's (for example, those of Sparc-stations) the usual meta keys are the keys marked ◇ either side of the space-bar. The ⟨S-x⟩ modifier is only used with non-printing characters (for example, $\boxed{\text{RETURN}}$).

When we say "click $\boxed{\text{MOUSE-LEFT}}$ " on some part of a window, we mean that the mouse cursor should be pointed at that part, and then the $\boxed{\text{MOUSE-LEFT}}$ button should be pressed.

Be aware that Nuprl can be quite slow to respond to keystrokes, sometimes taking several seconds. Don't hold keys down till you get a response. You might easily make the keys autorepeat, which could be rather annoying.

For clarity when presenting input which a user might type, or output which Nuprl generates, we sometimes enclose the input or output text in special ⌜⌝ quotes. For example ⌜`this is example output`⌝.

Some cursors in Nuprl highlight a part of window. The highlighting is indicated on the screen by swapping the foreground and background colors of the display. For example, if normally the display has black characters on a white background , then a highlighted part has white characters on a black background. In this document we indicate a highlighted region of the screen by drawing an outline around it. For example, in the window

```
ML top loop
-----------------------------------
M> '[int] * [int]' ;;
```

the `[int] * [int]` is considered to be highlighted.

1.3 Practical Details

1.3.1 Getting Set Up

The Nuprl system is written in a combination of Common Lisp and an older dialect of ML. We assume here that your Nuprl administrator has already done the following:

1. Installed the Nuprl directories and compiled the various Nuprl files.
2. Compiled the Nuprl Lisp and ML files and created a Lisp image (disksave) that has these files loaded as well as some initial Nuprl theories.
3. Set up a shell script that both starts up this Lisp image and starts up Nuprl's window system. Below, we assume that the alias `nuprl` has been set up for this script.

4. Installed the Nuprl font files for X in some directory FONT-DIR where your workstation can access them.

To set-up, do the following:

1. Add the following lines to the start of your `.xinitrc` after any initial comments (in particular after the first line if it starts `#! /bin/csh`).

   ```
   xset fp+ FONT-DIR
   xset fp rehash
   ```

   These commands tell `X` the font path to the nuprl fonts. The first time you run Nuprl you should also run these commands interactively in some shell to add the font path to your current `X` environment. The current Nuprl fonts in order of size from smallest to largest are named `nuprl-8x13`, `nuprl-13` and `nuprl-20`. If you want to look at one of the fonts, use the `xfd` command. For example, run in a shell:

   ```
   xfd -font nuprl-13 &
   ```

2. Familiarize yourself with some editor that supports 8-bit fonts and has a capability for starting sub-shells. For example, `lucid-emacs`, `epoch` and `emacs` version 19. Vanilla 18.xx versions of `emacs` do not support 8-bit fonts, although there are several 8-bit patches available. You should run this editor with one of the nuprl fonts.

   Such an editor is not strictly necessary, but is a good idea for several reasons:

   - Nuprl frequently writes output to Lisp's 'standard output' which is almost invariably the same window as that which Nuprl is started up from. If Nuprl is started up from an editor sub-shell, it becomes easy to review this output and save portions of it to files.
   - This output is in Nuprl's 8-bit font.
   - Listings of theory files use Nuprl's 8-bit font. These files contain definitions, theorems and proofs, and it is often useful to be able to browse them.

1.3.2 Starting Up

We assume you have set things up as described in the previous section.

- Start up the 8-bit emacs you have chosen to use.
- Start a sub-shell. For emacs-related editors type

  ⟨M-x⟩`shell` RETURN

- In the sub-shell, start up nuprl. Type:

  `nuprl` RETURN

It will take a few seconds for Nuprl to start up. When it does, Nuprl's two main windows, the ML-Top-Loop window and the Library window should open up on your display. The ML-Top-Loop window should look like

```
ML Top Loop
-----------

M> | ;;
```

The Lisp underlying Nuprl is running in the window in which you typed `nuprl`. Since output from the ML Top Loop and error messages are written to this window, it is a good idea to keep it visible.

Nuprl is now ready for use.

1.3.3 Hints on Using the System

Nuprl's windows are at the "top-level" in the X environment. The windows can be managed (positioned, sized, etc.) in the same way as other top-level applications such as X-terminals. Creation and destruction of Nuprl windows, and manipulation of window contents, is done solely via commands interpreted by Nuprl. Nuprl will receive mouse clicks and keyboard strokes whenever the the input focus is on any of its windows. Exactly one window is "active" at any given time; this window is identified by the presence of Nuprl's cursor. This appears either as a vertical bar or as a highlighted region. The specific location of the cursor determines the semantics of keyboard strokes and mouse clicks, and is independent of the location of the mouse cursor.

The two main windows — the ML-Top-Loop window and the library window — remain throughout the session and you cannot create new versions of them. Chapter 2 describes use of the ML-Top-Loop window and Chapter 3 describes the format of the library window. Chapter 3 also describes the kinds of objects that can be found in the library.

There are two other kinds of windows; *term editor* windows and *proof editor* windows. Both are used for editing objects in the library. Terms and the term editor is described in Chapter 4 and the proof editor is described in Chapter 7.

Most Lisps allow computations to be interrupted. This is usually done by sending ⟨C-C⟩ to the Lisp process. (If Lisp is started up from an emacs sub-shell, you usually can do this by typing ⟨C-C⟩⟨C-C⟩ to the sub-shell window). This will cause Lisp to enter its debugger, from which the computation can be resumed or aborted. Aborting Nuprl is almost always safe. When Nuprl is restarted, the state should be exactly as it was when Nuprl was killed, except that any computations within Nuprl will have been aborted.

See Chapter A for how to use the Lisp debugger, and in particular, for what to if Nuprl crashes. Nuprl is a continually-evolving experimental research system, and it is inevitable that it will contain bugs. Please report any behavior you think is due to a bug, or inconsistencies between the operation of the system and the documentation. Also report any break-points that you hit; they have either been left in the code accidentally, or they are there to help track down the source of bugs. We welcome suggestions for improvement. Send e-mail to `nuprlbugs@cs.cornell.edu`.

If the system appears to be inexplicably stuck, check the window running Lisp; it is very possible that Lisp is garbage-collecting. This sometimes takes a few minutes.

1.3.4 Exiting

When you are ready to stop, click MOUSE-LEFT in the ML-Top-Loop window, and enter ⟨C-z⟩. This should give you a Lisp prompt (>) in the shell from which you started up Nuprl. To exit Lisp, enter

(quit) RETURN

in this window. It is important that you explicitly type `quit`, rather than just for example quit out of the editor Nuprlis running under. In the latter case, the Lisp process can be left floating around in a hung state, hogging memory resources. (This could also happen if your editor crashes). You can use the Unix command `ps` to check for a hung Lisp and the command `kill` to kill it.

1.3.5 Alternative Setups

Intermediate and experienced users will probably want to create their own initialization procedures for Nuprl. These could allow customizations such as:

- Changing the initialization of Nuprl's X windows.
- Changing key-bindings for the term and proof editors.
- Loading more / different theories.
- Loading more / different tactics.

Depending on how significant the changes are, these initialization procedures could be run after starting up some pre-prepared disksave, or after starting up an plain Lisp process. Users can of course too make their own disksaves for future use.

To get an idea of how you might set up an initialization procedure, look at the files in the `sys/utils/` directory. You probably will want to put all your initialization commands into a Lisp file that is automatically loaded whenever a plain Lisp image or disksave is started up.

Note that Nuprl runs in the *nuprl* package. All symbols entered in Lisp will be interpreted relative to this package. The package inherits all the symbols of Common Lisp, but does *not* contain the various implementation-specific utilities found in the package *user* (or *common-lisp-user*). To refer to these other symbols, either change packages using `(in-package "USER")`, or explicitly qualify the symbols with a package prefix. If you change packages, you can change back to the Nuprl package using `(in-package "NUPRL")`.

If you plan to do significant amounts of programming in Lisp, you might want to look into using Lisp sub-shell packages such as ILISP rather than vanilla sub-shells.

1.4 Customization

1.4.1 Window System Options

The Lisp function `change-options` can be used to set various parameters affecting Nuprl's window system. The `change-options` function takes an argument list consisting of keywords and associated values. For example, to set the options `:host` and `:font-name` to `moose` and `nuprl-20` respectively, put the form

```
(change-options :host "moose" :font-name "nuprl-20")
```

in your init file. The options, together with their default values (in parentheses) are given below.

`:host` (`NIL`). The host where Nuprl windows should appear.

`:title-bars?` (`NIL`). If `T` then Nuprl will draw its own title bars.

`:host-name-in-title-bars?` (`T`). If `T`, then the title of each window will include a substring indicating what host the Lisp process is running on.

`:no-warp?` (`T`). If `T` then Nuprl will never warp the mouse. (Mouse warps apparently annoy some users.) In environments where the position of the mouse determines input focus, setting this option to `NIL` will guarantee that Nuprl retains input focus when windows are closed.

`:frame-left` (30), `:frame-right` (98), `:frame-top` (30), `:frame-bottom` (98). Each of these should be a number between 0 and 100. They give the boundaries, in terms of percentage of screen width or height, of an imaginary frame within which Nuprl will attempt to place most new windows.

`:font-name` (`"nuprl-13"`). The name (a string) of a font to use for the characters in Nuprl windows. Nuprl uses a special 8-bit font. Currently two are available: `nuprl-13` and `nuprl-20`.

`:cursor-font-name` (`"cursor"`), `:cursor-font-index` (22). The name of the font to use for the mouse cursor when it is over a Nuprl window, and an index into that font. The default font should always be available.

`:background-color` (`"white"`). The color for the background in Nuprl windows. The value must be a string argument naming a color. Any color in the X-server's default colormap may be used. Nuprl will get a Lisp error (entering the Lisp debugger) if the color does not exist.

`:foreground-color` (`"black"`). The color for characters etc. in Nuprl windows.

1.4.2 Editor Options

The key bindings for the term and proof editors can be altered by creating your own key macro files and loading them instead of the standard ones in a Lisp initialization file.

1.5 Directory Structure

Nuprl is currently maintained with the help of the CVS version control system. All the Nuprl code resides in a single CVS module called `nuprl4`. The main parts of the directory structure as of January 29th 1994 are as follows:

Directory	Contents
`lib/`	
`lib/ml/`	All ML code
`lib/ml/standard/`	Standard ML source functions and tactics
`lib/theories/`	All Nuprl Theories
`lib/theories/standard/`	Basic Theories
`lib/theories/algebra/`	Abstract Algebra Theories
`lib/theories/reals/`	Real Analysis Theories
`sys/`	All Lisp code
`sys/ml/`	Lisp for ML compiler and interpreter
`sys/prl/`	Lisp for Editors and Refiner
`sys/utils/`	Utilities for loading system
`macro/standard/`	Editor customization
`doc/`	Documentation
`doc/man/`	This Reference Manual
`doc/ml/`	Nuprl ML Manual
`doc/tutorials/`	Introductory Tutorials
`doc/sys/`	System documentation

All directories should eventually contain a `=README` file that describe their contents.

1.6 Learning to use the System

1.6.1 Tips

A few tips are as follows:

- We recommend that you run through the Nuprl term and proof editing tutorials before trying to do anything else with the system.
- The Nuprl ML manual contains a tutorial in the use of ML. Use this as an introduction to ML.
- In learning the proof and term editors, check out all the mouse commands. Many editing operations can be done most easily with the mouse.
- Familiarize yourself with where Nuprl theories are kept and how they are organized. (See Section 1.5 and Section 10.1.) Existing theories are an excellent resource for learning about how to do proofs. In particular, you can use the Unix `grep` command to search theory listings to find examples of uses of tactics you are curious about.

We recommend that fairly early on, you at least browse through this manual, familiarizing yourself with the general contents of each chapter. This will help you know where to look if you have questions.

1.6.2 The Nuprl Book

The Nuprl book *Implementing Mathematics with the Nuprl Proof Development System*, Constable *et al*, published in 1986, is still a good background reference. However, the system has changed sufficiently that none of the tutorials given in the book will work in the current system. The "reference" portion, excluding the parts of Chapter 8 on the type system and its semantics, is superseded by this reference manual. Chapter 9 in the reference portion also contains some useful examples and discussions of tactic writing that are not reproduced here. The "advanced" portion of the book deals with application methodology, gives some extended examples of mathematics formalized in Nuprl, and also describes some extensions to the type theory which have not been implemented.

Substantial changes have been made to Nuprl since the book was written. The most major ones are:

- An X-windows interface has been added.
- All Nuprl terms now have a uniform term structure.
- Rules are now alterable library objects, rather than being hard-wired.
- New display form and abstraction facilities replace the old definition facility.
- A substantial tactic collection has been added.
- ML utilities have been added to format library and proof listings for Latex.

Chapter 2

ML Top Loop

2.1 Introduction

The ML Top Loop provides an interactive interface to ML. You can use it to evaluate ML expressions and declarations. Specific Nuprl-related uses for the ML Top Loop include:

1. controlling the library window,
2. loading and dumping theories,
3. editing library objects,
4. exploring the Nuprl state,
5. experimenting with Nuprl functions.
6. loading ML files,

The ML-Top-Loop runs in its own Nuprl window which is created when Nuprl is started up. This window is a *term editor* window, so most of the commands described in Chapter 4 work in it.

The rest of this section is divided in two. The first part introduces you to the ML Top Loop, and tells you enough about it to get started with the Nuprl system. This part does not assume familiarity with Chapter 4. It should be sufficient for you to work through the ML examples in the tutorial section of the Nuprl ML Manual. The second part describes in more detail the functionality of the top loop, and does assume you have some familiarity with the contents of Chapter 4.

2.2 Basic Top-Loop Operation

Initially the top loop window looks like:

```
ML Top Loop
M> | ;;
```

The `M>` is the ML prompt. The `;;`'s are the usual termination characters for ML expressions and declarations. The top loop always supplies these; you never have to type them yourself. The | is the cursor. To differentiate it from other kinds of cursors we call it a *text* cursor. You can type text whenever you have a text cursor. Other kinds of cursors are *screen* cursors and *term* cursors. These have redily distinguishable appearances; a screen cursor outlines a single character, and a term cursor highlights a region of the screen.

The basic top loop commands are summarized in Table 2.1.

x	INSERT-CHAR-x	insert character x
RETURN	ML-EVALUATE	call ML evaluator
⟨S- RETURN ⟩	INSERT-NEWLINE	add line-break
MOUSE-LEFT	SET-POINT-TO-MOUSE	move cursor to mouse position
⟨C-F⟩	SCREEN-RIGHT	move cursor right 1 character
⟨C-B⟩	SCREEN-LEFT	move cursor left 1 character
⟨C-P⟩	SCREEN-UP	move cursor up 1 character
⟨C-N⟩	SCREEN-DOWN	move cursor down 1 character
⟨C-D⟩	DELETE-CHAR-RIGHT	delete char to right of cursor
DELETE	DELETE-CHAR-LEFT	delete char to left of cursor
⟨C-R⟩	ML-HISTORY-PREV	scroll back through history
⟨M-R⟩	ML-HISTORY-NEXT	scroll forward through history
⟨C-Z⟩	EXIT-TOP-LOOP	return to Lisp Listener

Table 2.1: Basic Top Loop Commands

For convenience, many of the key bindings for the basic commands have been made similar to those used in emacs. Some of these bindings are context senstive; specifically the INSERT-CHAR-x, ML-EVALUATE, INSERT-NEWLINE,DELETE-CHAR-RIGHT and DELETE-CHAR-LEFT all rely on there being a text cursor.

Output from evaluation is usually printed out to the shell from which Nuprl has been invoked. For this reason you will want to keep the shell window visible and perhaps immediately above the ML-Top-Loop window.

To evaluate an ML expression, type in the expression at a text cursor, just after the `M>` prompt, and then use ML-EVALUATE. For example, if you type:

2+2 RETURN

Nuprl responds by evaluating the expression, and printing to the prl-shell the value of the expression (`4`) and its type (`int`):

```
M> 2+2 ;;
4 :  int
```

To correct input you type, use the DELETE-CHAR-BEFORE and DELETE-CHAR-AFTER commands. To move the cursor around, you can use SCREEN-UP,SCREEN-DOWN,SCREEN-LEFT and SCREEN-RIGHT. Alternatively you can use the mouse: To get a text cursor between two given character positions, click MOUSE-LEFT with the mouse pointing at the character position to the right. Using the cursor motion commands you will doubtless encounter the other kinds of cursors. Nuprl uses these other cursors when a text cursor is inappropriate. These cursors don't destructively modify the display. If you get one, continue to use the SCREEN-* commands or MOUSE-LEFT to get back to a text cursor.

To get a continuation line for a command, key ⟨S-RETURN⟩. The continuation prompt is `>`. For example, if you entered:

`1+2+`⟨S-RETURN⟩`3+4`

you would get:

```
ML Top Loop
M> 1+2+
 > 3+4| ;;
```

The continuation prompt behaves much like a character, in that you can use the DELETE-CHAR-* commands to delete it.

The ML Top Loop maintains a command history going back to the start of a session. Use the ML-HISTORY-* commands to scroll back and forth through the history.

To exit the ML Top Loop and return to Lisp, use the EXIT-TOP-LOOP command.

Occasionally you can get the ML Top Loop into an unexpected state. In this case, you can re-initialize the ML-Top-Loop window by deleting the existing term in the window, and then using the INITIALIZE command. The keystroke sequence for doing this from any position in the window is ⟨M-<⟩⟨CM-K⟩⟨CM-I⟩. This will not disrupt your command history.

Nuprl error messages are both output to the shell and for convenience displayed in highlighted text in the ML-Top-Loop window. These messages don't change the contents of the window in any way, and any keystroke or mouse-click directed at the ML-Top-Loop window causes the error display to go away.

Errors can come from various sources. For example, a message is generated if you type an expression for evaluation into ML Top Loopand the expression doesn't parse or type-check properly. In this event, the most appropriate keystroke is ML-HISTORY-PREV to recall the incorrectly entered expression. Error messages are also created when, during the course of evaluation of an ML expression, an exception is generated and uncaught.

Similar error messages appear in rulebox windows when something goes wrong with the entry or evaluation of a tactic. In this case, harmless keystrokes to use to make the message vanish include the screen motion commands and MOUSE-LEFT.

2.3 More Advanced Top-Loop Operation

The ML-Top-Loop is configured to support in general a sequence of ML prompts. Prompts can be inserted and deleted using the usual sequence commands described in Section 4.4.6.

With more than one prompt, there are several alternatives for evaluating an ML expression. The relevant commands are summarized in Table 2.2.

RETURN	ML-EVAL-SCROLL-UPDATE
⟨C-RETURN⟩	ML-EVAL-SCROLL
⟨M-RETURN⟩	ML-EVAL-UPDATE
⟨CM-RETURN⟩	ML-EVAL

Table 2.2: Additional Top Loop Commands

the effect of RETURN (now ML-EVAL-SCROLL-UPDATE) is to evaluate the ML text in the prompt containing the cursor, echo the result of the valuation in a field just beyond the prompt, and then scroll the sequence up one prompt so that the topmost prompt is deleted and a fresh prompt is added at the bottom.

ML-EVAL-SCROLL is like ML-EVAL-SCROLL-UPDATE except that the result of evaluation is not reported. This is useful when for instance you don't care about the value of the evaluation, or the display of the result would be rather large and would undesirably alter the formatting of the ML-Top-Loop window.

ML-EVAL-UPDATE is like ML-EVAL-SCROLL-UPDATE except that the prompt sequence isn't scrolled.

With ML-EVAL the ML expression at the prompt is evaluated and left in place. The output isn't inserted after the prompt and the prompt sequence is not scrolled.

The above commands can be used to evaluate the ML expression at any ML prompt in a prompt sequence, not just the last one. New output text generated by evaluating a command overwrites old output text.

The result of every ML expression evaluation is always also echoed to the shell window from which Nuprl is started up. Also, every evaluated expression is still always inserted onto the history list and still always can be retrieved using the history scrolling commands.

2.4 Alternative Top Loops

If the lisp file `ml-cmd` is loaded at a lisp prompt in the shell window before starting up Nuprl's window system (or after resetting it by evaluating the Lisp expression `(reset)`), then starting up Nuprl does not cause the creation of an ML-top-loop window, and instead, an ML top loop to be established in the shell window.

The ML prompt in the shell is `ML>` . ML expressions typed at this prompt should be terminated with `;;` RETURN to have them evaluated.

Input focus must be manually transferred between the shell window and Nuprl's own windows: to transfer from the shell to Nuprl, use `x` RETURN. To transfer back, use TAB. To move from Lisp to the ML prompt, evaluate `(nuprl)` at the Lisp prompt as before, and to move back to Lisp, evaluate `exit()` RETURN at the ML prompt.

Of course, none of the term-editing facilities are available at this ML prompt.

Though rather rudimentary, this top loop is sometimes useful. For example, when developing ML code, ML expressions and declarations for evaluation can be cut and pasted into this top loop.

To work with both kinds of top loop, view the comment object `cmd`. This contains an ML prompt term, and functions virtually exactly as the term-editor top loop described above.

In general, ML prompt terms can be copied into other term editor objects, and they still function as top loops. It is straightforward to set up objects that contain sequences of prompts already initialized with frequently evaluated ML expressions.

If you want to switch back to having a dedicated top loop window and no top loop in the shell window, then reset Nuprl, load the file `ml-edit-cmd` and restart Nuprl.

Chapter 3

The Library

3.1 Introduction

Nuprl's *library* is a mathematical and logical database. The library is composed of *objects*. There are objects for theorems and definitions, and also for example objects which control the visual appearance of the mathematical notation. See Section 3.2 for a list of object types.

Library objects are grouped into *theories*. Every object belongs to exactly one theory. As yet, there is no nesting of theories. The dependencies of theories on one-another forms a partial order. Within each theory, objects are ordered linearly. The dependencies of library objects on one another is discussed more in Section 3.5. Theories are kept in files. In a Nuprl session, one usually loads into the library only those theories that one needs to reference. These theories would include the theories of immediate interest together with the all the ancestors of those theories.

The library window shows information on a segment of the library. The format of the window is discussed in Section 3.3. Commands for controlling the library window, editing the library and loading and dumping theories are discussed in Section 3.4.

Note that proofs are stored in a compressed format in files, and expansion of proofs loaded from files is only on demand. Expansion can often be quite slow. See Section7.7 for details.

3.2 Objects

There are seven kinds of objects:

rule
: A rule object defines a primitive rule of the object logic.

theorem
: A theorem object contains a proposition and a proof. If the proof is complete, the proposition is a theorem. If incomplete, a conjecture. A proof maybe *compressed* or *expanded*. Theorems are sometimes referred to as *lemmas*. A theorem object for a complete theorem also contains the extract term of the theorem.

abstraction
: An abstraction object introduces the definition of a new term.

ml

An ml object contains ML code.

display

A display object defines display forms for primitive terms and abstractions.

precedence

A precedence object assigns precedences for terms. Precedences control the automatic parenthesization of terms.

comment

A comment object contains a comment. Comments have no logical significance.

Theorem objects are discussed more in Chapter **??** and the rest of the kinds of object are discussed more in Chapter 4.

Every object has associated with it a status, either *raw*, *bad*, *incomplete* or *complete*, indicating the current state of the object. A *raw* status means an object has been changed but not yet checked. A *bad* status means an object has been checked and found to contain errors. An *incomplete* status is meaningful only for theorem objects and signifies that its proof contains no errors but has not been finished. A *complete* status indicates that the object is correct and complete.

3.3 Library Window

The library window displays a linear segment of the library, one object per line. When theories are loaded into the library, they are always placed in a linear order.

An example library display is shown in Figure 3.1. From left to right each line contains:

status

One character: `?` for raw, `-` for bad, `#` for incomplete and `*` for complete.

kind

One character: `R` for *rule*, `t` and `T` for *theorem*, `A` for *abstraction*, `M` for *ML*, `D` for *display*, `L` for *lattice* (the old name we used for the precedence object) and `C` for *comment*. The lower case "`t`" is used for compressed theorems and the upper case "`T`"for expanded theorems.

name

The name of the object.

summary

The first few characters of the object's contents.

3.4 Library ML Functions

These functions are all most commonly typed in at the ML Top-Loop. One is free to define abbreviations or alternative ML functions in terms of these primitives [1] The functions take the following kinds of arguments:

[1]The user should consult the ML file `commands.ml` for details on writing his or her own library functions.

```
Library @ leo
*C num_thy_1_begin  ************ NUM_THY_1 ************
*D divides_df        <b:int:*> | <a:int:*>== divides{}(<b>;
*A divides           b | a == ∃c:ℤ.  a = b * c
*t divides_wf        ∀a:ℤ.  ∀b:ℤ.  (a | b ∈ ℙ{1})
*t comb_for_divides_wf    (λa,b,z.a | b) ∈ (a:ℤ → b:ℤ → ↓{True}  → ℙ{1})
*t zero_divs_only_zero    ∀a:ℤ.  0 | a ⇒ a = 0
*t one_divs_any      ∀a:ℤ.  1 | a
*t any_divs_zero     ∀b:ℤ.  b | 0
*t divides_invar_1   ∀a:ℤ.  ∀b:ℤ.  a | b⇐⇒ -a | b
*t divides_invar_2   ∀a:ℤ.  ∀b:ℤ.  a | b⇐⇒ a | -b
*t divisors_bound    ∀a:ℕ.  ∀b:ℕ+.  a | b ⇒ a ≤ b
*t divides_of_absvals  ∀a:ℤ.  ∀b:ℤ.  |a| | |b|⇐⇒ a | b
*t divides_reflexivity    ∀a:ℤ.  a | a
*t divides_transitivity   ∀a:ℤ.  ∀b:ℤ.  ∀c:ℤ.  a | b ⇒ b | c ⇒ a | c
*t divides_anti_sym_n  ∀a:ℕ.  ∀b:ℕ.  a | b ⇒ b | a ⇒ a = b
*t divides_anti_sym    ∀a:ℤ.  ∀b:ℤ.  a | b ⇒ b | a ⇒ a = ± b
*t divisor_of_sum    ∀a:ℤ.  ∀b1:ℤ.  ∀b2:ℤ.  a | b1 ⇒ a | b2 ⇒ a | b1 + b2
*t divisor_of_mul    ∀a:ℤ.  ∀b:ℤ.  ∀c:ℤ.  a | b ⇒ a | b * c
*t divides_mul       ∀a:ℤ.  ∀b:ℤ.  a | b ⇒ (∀n:ℤ-0.  n * a | n * b)
*t divisor_bound     ∀a:ℕ.  ∀b:ℕ+.  a | b ⇒ a ≤ b
*t divides_iff_rem_zero   ∀a:ℤ.  ∀b:ℤ-0.  b | a⇐⇒ a rem b = 0
```

Figure 3.1: The Library Display Window

obname

An ML string. The name of an object. Acceptable names are composed from the alphabet ⌜`a–zA–Z0–9_`⌝. The first character should be a letter.

place

An ML string. The name of an object. The library position understood is immediately before the object named. `"last"` may be used to refer to a fictitious object after the last object in library.

n

A non-negative number.

`()`

This is the unique inhabitant of the ML type `unit`.

Remember that ML strings are always enclosed in ⌜`"`⌝ characters, and that ML functions are always terminated by ⌜`;;`⌝. Some commands take lists as arguments; ML Lists are delimited by ⌜`[]'s`⌝ and use ⌜`;`⌝ to separate items. Further utility functions related to the library are described in Appendix ???.

3.4.1 Library Window Motion

`jump` *obname*

Position object *obname* at the top of window.

`up` *n*

Scroll window up n lines.

`down` *n*

Scroll window down n lines.

`top ()`

Position window at top of library

`bottom ()`

Position window at bottom of library

3.4.2 Library Editing

`view` *obname*

The object *obname* is displayed in a new window. If the object is not already being viewed the new view will be fully editable; otherwise, it and all other views of the object will be made read-only. The header line of the view will say `SHOW` for a read-only view and `EDIT` for an editable view.

The editor used depends on the kind of object. The *proof editor* is used on theorem objects, while the *term editor* is used for all other objects. For more information on the proof editor see Section 7.6 and on the term editor, see Section 4.4

If `view` is used on a theorem object with an compressed proof, expansion of the proof is forced. This may take some time, especially if the proof is large.

`create_rule` *obname place*
`create_thm` *obname place*
`create_abs` *obname place*
`create_ml` *obname place*
`create_disp` *obname place*
`create_com` *obname place*
`create_lat` *obname place*

Create new objects of the appropriate kind with name *obname*, and position it before object *place*.

`rename` *old-obname new-obname*

Rename object *old-obname* to *new-obname*.

`delete` *obname*

Delete object *obname* from the library.

`delete_objects` *from-obname to-obname*

Delete the range of objects from *from-obname* to object *to-obname* inclusive.

`check` *obname*

Check object *obname*. If necessary, the library window is redisplayed to show the object's new status.

Checking a rule, abstraction, precedence or display form object causes the object's contents to be *verified*. See Section ??? for a description of what verification involves. If the object is well-formed, it is incorporated into the Nuprl environment.

When is checking necessary??? After all one always checks on loading and exiting an object...

Checking an ML object invokes the ML reader on the object's contents.

Checking a theorem object, forces expansion of its proof, if the proof was initially compressed. Note that this might take a while. See Section7.7 for details.

What is effect on extraction???

Checking a comment object has no effect, other than to change the status of a *raw* comment object to *complete*.

`check_objects` *from-obname to-obname*
: Check from object *from-obname* to object *to-obname* inclusive. Stop if the status of any object changes on checking. This prevents a cascade of further status changes which might be caused by this status change.

`move` *obname place*
: Move object *obname* before object *place*.

`move_objects` *from-obname to-obname place*
: Move the range of objects from *from-obname* to object *to-obname* inclusive to immediately before object *place*.

3.4.3 Theory Commands

Each theory has an identifier and is associated with a file. Each theory also has a set of immediate ancestors which it is dependent on. Commands are provided to set up new theories, load theories, dump theories, and print theories.

Theories are named by ML strings. The conventions for naming theories are the same as for naming library objects explained at the beginning of this section. Each theory is associated with a file. The value of the ML reference variable `theory_filenames : (string # string) list` is a list of pairs of theory names and names of associated files. The filenames include a full pathname, but do not have any extension. For example, a valid filename string is ⌜ `nuprl/lib/standard/core_1`⌝. The actual file associated with a theory is this name with a ⌜`.thy`⌝ extension. Examine the `theory_filenames` variable to find out the theories that Nuprl knows about at a given time.

`set_theory_filename` *theory-name file-name*
: Add entry in `theory_filenames` list for *theory-name*. If an entry already exists for *theory-name*, update that entry.

`show_theory_filename` *theory-name*
: Show entry in `theory_filenames` list for *theory-name*.

Theory dependencies are recorded by the value of the `theory_ancestors : (string # string list) list` reference variable. Each entry in this list is a pair of a theory's name and a list of names of other theories that the theory is immediately dependent on. There are ML functions to compute the closure of this graph as and when necessary.

`set_theory_ancestors` *theory-name theory-ancestor-list*
: Add entry in `theory_ancestors` list for *theory-name*. If an entry already exists for *theory-name*, update that entry.

`show_theory_ancestors` *theory-name*
: Show entry in `theory_ancestors` list for *theory-name*.

Typically one adds a few dummy theories to `theory_ancestors` to simplify the description of the ordering of theories.

The theories loaded in a Nuprl session are generally a subset of the theories that Nuprl knows about. Between sessions, modified theories should always be explicitly saved. The functions for loading and dumping theories are:

`load_theory` *theory-name*
: Load *theory-name* at the end of the library from the associated file. If *theory-name* has already been loaded, this function has no effect. Every non-theorem object is checked as it is loaded.

`load_theories_with_ancestors` *theory-name-list*
: Load the theories named in *theory-name-list* at the end of the library, together with any ancestor theories that haven't yet been loaded. Theories are loaded in an order consistent with their dependencies. Every non-theorem object is checked as it is loaded.

`dump_theory` *theory-name*
: dump *theory-name* to the associated file. This leaves the theory *theory-name* in place in the library.

Other utility functions are:

`list_theories ()`
: List all the currently loaded theories. This is very useful if you are not sure which theories are loaded and which not.

`delete_theory` *theory-name*
: Delete *theory-name* from the current library. Does not affect the file associated with the theory.

`short_print_theory` *theory-name*
: Create print-files for *theory-name*. *theory-name* must be loaded for this to work. If the associated file-name is *file-name*, then two files are created; *file-name*`.prl` and *file-name*`.tex`. *file-name*`.prl` is a file viewable by an editor running with Nuprl's special 8-bit font. *file-name*`.tex` is a self-contained LaTeXversion of the theory listing.

`long_print_theory` *theory-name*
: This is similar to `short_print_theory` except that proofs and extracts of all theorems are also included. The files created have names *file-name*`_long.prl` and *file-name*`_long.tex`.

The theory *theory-name*always begins with comment object *theory-name*`_begin` and ends with comment object *theory-name*`_end`. The names of these objects are important. Most of the theory functions rely on these delimiter objects being named the way they are. However, the user is free to alter the contents of these objects to his or her liking. A useful function is:

`add_theory_delimiters` *theory-name*
: Add delimiter objects for the new theory *theory-name* to the end of the library.

There are variants on the load functions which check (and therefore expand) theorem objects at load time. They are:

`load_fully_theory` *theory-name*
: Load *theory-name* at the end of the library from the associated file. If *theory-name* has already been loaded, this function has no effect. Every object is checked as it is loaded.

`load_fully_theories_with_ancestors` *theory-name-list*
Load the theories named in *theory-name-list* at the end of the library, together with any ancestor theories that haven't yet been loaded. Theories are loaded in an order consistent with their dependencies. Every object is checked as it is loaded.

3.5 Object Dependencies and Ordering

A correct library in Nuprl is one where every definition and theorem refers only to objects occurring previously in the library. Unfortunately, Nuprl does not guarantee that this property is maintained when functions are used that modify the library. For example, it is possible to create a circular chain of lemma references.

There is only one way to guarantee that a theory (or collection of theories) is correct. This is to load it (them, sequentially) using one of the *load-fully* functions described at the end of the last section. This will force a theorem's proof to be expanded before the theorem is loaded into the library, and so guarantee that proofs only reference theorems that occur previously in the library.

Loading without using these *load-fully* functions and then using `check_objects` or `check_theory` will *not* guarantee that the library is correct, since during the checking of a theorem, all later theorems will be present in the library and will retain the statuses they had when they were dumped. However it is recommended that one always preceed a *load-fully* check, by loading the relevant theories without expanding theorems, and then using `check_objects` or `check_theory`. There are two reasons for this. Firstly, just to check that all proofs do indeed expand properly. Secondly, the current *load-fully* functions will blithely continue loading a library after an error has occurred, often creating a cascade of further errors. This bad behaviour will be corrected in the near future.

Nuprl does do some dependency checking with definitions. For example, if a definition is deleted then the status of any entry depending on these objects is set to *bad*.

Because of the general lack of dependency checking, a user must be careful to keep library objects correctly ordered or reloading may fail. The `move` function can be used to repair incorrect orderings and ensure that objects occur before their uses.

Here is a list of some of the main object dependencies one should be aware of:

- *Theorems on other theorems.* Each theorem should only depend on theorems occurring earlier in the library. Note that several kinds of theorems are referenced automatically by Nuprl tactics. For example, well-formedness theorems (theorems whose names end in ⌜`_wf`⌝) and various support lemmas used by the rewrite package.
- *Theorems on abstractions.* A theorem shouldn't refer to an abstraction before it is defined.
- *Abstractions on abstractions.* The right-hand-side of an abstraction should only refer to abstactions defined earlier in the library. Note that abstractions should *not* be recursive.
- *ML objects on theorems, abstractions and other ML objects* ML objects frequently assume the existence of certain theorems and abstractions. For example, one might include in an ML object a rewrite tactic which references a set of lemmas. One should always consider the introduction of such dependencies carefully. Continuing the example, if one were to change one of the lemmas would one want the rewrite tactic to automatically use this change? Many functions which access objects in the library can be written in a *lazy* fashion, such that they

look up the object, only when they are called. Such functions however might be considerably less efficient than ones which do need to do a fair amount of preprocessing. Ideally, one wants to just do this preprocessing once. Below we discuss the use of *caches* to resolve this *partial evaluation* problem.

- *Theorems on ML objects* Theorems can be proved using tactics and other ML functions defined in ML objects, so needless to say, those theorems should occur later in the library than the ML objects they are dependent on.

In addition, there are other dependencies one should be aware of:

- *ML files on theories.* It is desirable to be able to compile all ML files without having to have all theories a-priori loaded, so in general any dependencies should be of the lazy variety as explained earlier. It is a very bad idea to have compiled ML files dependent on functions defined in ML objects; whenever an ML function is compiled, it is timestamped, and references between ML functions keep track of these timestamps. All functions in ML objects are compiled afresh every time the objects are loaded, so if one were to load ML files compiled in an earlier session, one could have stale function references which would result in ML crashing.

- *Theories on ML files* This is fine. We will soon be extending the theory mechanism so that one can specify optional ML files to only be loaded when certain theories are loaded.

- *Cache Dependencies.* The ML tactic system maintains a fair amount of state, much in the form of preprocessed lemmas. We have an incremental update strategy for many of these caches to ensure that they track changes in the library state, so for most purposes these caches are invisible to the user. However, to date, not all the code for caches has been updated to use this incremental strategy, so for example, one might run into situations where the system refuses to acknowledge that one has added a missing lemma. In these situations, executing the function `reset_caches : unit -> unit` might help. Caches are discussed in Chapter 9.

3.6 Future Developments

The theory mechanism was only added to Nuprl fairly recently and there are some obvious enhancements which need to be made. For example,

- namespace management

- automatic dependency checking

- support for maintaining sets of conjectured theorems, so one can develop theories in other orders than foundations first in a sytematic way.

Chapter 4

Terms

4.1 Introduction

In Nuprl, a *term* is a tree data-structure. The structure of terms is explained in detail in Section 4.2. Terms have a variety of uses.

- All propositions in Nuprl's logic are represented as terms, as are all expressions and types in its type theory. We refer to them sometimes as *object-language* terms.
- Nearly all library objects are represented as terms. We refer to these terms as *system-language* terms.

Terms are Nuprl's equivalent of Lisp's S-expressions; they are used as a general-purpose uniform data-structure.

Terms are either *primitive* or *abstract*. Primitive terms have fixed pre-defined meanings. Abstract terms or *abstractions* are defined in abstraction library objects as being equal to other terms. An abstraction is *unfolded* when it is replaced by the right-hand side of its definition. Abstractions are discussed in Chapter 5.

The visual appearance of a term is governed by its *display forms*. These are defined in display-form library objects. Display forms are described in detail in Chapter 6.

Terms are interactively edited and viewed using a structured editor. This editor is described in Section 4.4.

4.2 Term Structure

4.2.1 Overview

Here we give an abstract view of the term data-structure. Details follow in Section 4.2.2.

Let *variables* be some infinite class of atomic individuals. The class of terms as the least set of expressions such that:

- if v is a variable, then v is a term,

- if for $1 \leq i \leq n$ we have that $x^i_1, \ldots, x^i_{a_i}$ are variables, t_i is a term, and we define

  $$s_i \quad = \quad x^i_1, \ldots, x^i_{a_i}.t_i$$

  then

  $$opid\{p_1{:}k_1, \ldots, p_m{:}k_m\}(s_1; \ldots; s_n)$$

  is a term.

We name the parts of a term as follows:

- $opid\{p_1{:}k_1, \ldots, p_m{:}k_m\}$ is the *operator*.
  The parts of the operator are:
  - *opid* is the *operator identifier*.
  - $p_j{:}k_j$ is the jth *parameter*. p_j is its *value*, and k_j is its *type*.
- The tuple $\langle a_1, \ldots, a_n \rangle$ where $a_j \geq 0$ is the *arity* of the term.
- $s_i = x^i_1, \ldots, x^i_{a_i}.t_i$ is the ith *bound-term* of the term. This bound-term binds free occurrences of the *variables* $x^i_1, \ldots, x^i_{a_i}$ in t_i.

When writing terms, we sometimes omit the brackets around the parameter list if it is empty.

4.2.2 Details

Terms are implemented in the current Nuprl system in Lisp. You should rarely have to work with terms at the Lisp level. Rather you either use the term editor to view and edit terms, or a set of term-related functions in ML.

We describe here the the current parameter types, the acceptable strings for opids, parameters, and variables, and the implementation of these strings from the ML point of view.

The current parameter types and associated values are:

`natural`
: natural numbers (including 0). Implemented using ML type `int`. Acceptable strings are generated by the regular expression $[\mathtt{0} - \mathtt{9}]^+$.

`token`
: character strings. Implemented using ML type `tok`. Acceptable strings can draw from any non-control characters in Nuprl's font.

`string`
: character strings. Implemented using ML type `string`. Acceptable strings can draw from any non-control characters in Nuprl's font.

`variable`

Names of variables. Implemented using ML type `var`. Acceptable strings draw on the alphabet ⌜`a-zA-Z0-9_-%`⌝. The % character

has a special use. See Section 7.2. The empty string is not an acceptable name for a variable parameter.

`level-expression`

Universe level expressions. These are used to index universe levels in Nuprl's type theory. Implemented using ML type `level_exp`. The syntax of level expressions is described in Section 9.1.2.

The names of parameter types are usually abbreviated to their first letters.

Opids are character strings drawn from the alphabet ⌜`a-zA-Z0-9_-!`⌝. (Here ⌜`-`⌝ is the ASCII character, ⌜$x - y$⌝ indicates the characters from x to y inclusive.) An ⌜`!`⌝ at the start of a character string indicates that the term does not belong to Nuprl's object language. Opids are implemented using ML type `tok`.

Binding variables are character strings drawn from the same alphabet as variable parameters. In addition, the empty string can be used. We call the binding variable with the empty string as its name, the *null* variable. Null variables can never bind. Binding variables are implemented using ML type `var`.

Earlier, when we described the term type, we said that variables were considered to be terms. This was a slight simplification of the actual state of affairs; In Nuprl, we consider variables and terms to be distinct. We have a special term kind, `variable{v}` for injecting variables into the term type. So when we talk of the variable `foo` *as a term*, we are really thinking of the term `variable{foo:v}`. When we write terms, this injection if often implicit. So for example, we write `bar(x;y)` instead of `bar(variable{x:v}; variable{y:v})`.

We often assume a similar implicit injection for natural numbers. So for example, the term `bar(10;11)` when written out in full is the term `bar(natural_number{10:n}; natural_number{11:n})`.

Some examples of terms in both pretty and plain notation are shown in Table 4.1.

Pretty Notation	Plain Notation
$\mathbb{Z}$	`int()`
x + y	`add(`x`;`y`)`
"abc"	`token{abc:t}()`
$\boldsymbol{\lambda} x.x$	`lambda(`$x.x$`)`
$\forall x,y{:}T.\quad A$	`all(`T`; `$x,y.A$`)`

Table 4.1: Examples of term notation

4.3 Term Display

4.3.1 Notation and Logical Structure

This section introduces the approach we use for entering and displaying terms.

We distinguish between *logical structure* of terms, and the *notation* in which terms are presented. When we talk of the *logical structure* of a term, we are thinking of some abstract object of mathematics. We are not just thinking of the term in uniform syntax, though the regularity of the uniform syntax for a term does reflect the regularity of the underlying abstract object. When we talk of *notation*, we are thinking of the visual presentation of abstract objects on the printed page, or on the computer screen. When you read mathematical or logical expressions in familiar notation, you often mentally construct the abstract object in your mind so readily that you forget the distinction between abstract structure and notation.

Notation understandable by machines became a focus of study when people started to design programming languages. The languages had to not only be easily understandable by humans, but also easily parseable by machine. The study of notation became the study of regular expressions and grammars. People devised sophisticated techniques for designing parsers.

In the programming language world, source texts in ASCII files correspond to our idea of notation, and abstract-syntax-trees get close to our notion of logical structure.

In mathematics notation is crucial issue. Many mathematical developments have heavily depended on the adoption of some clear notation, and mathematics is made much easier to read by judicious choice of notation. However mathematical notation can be rather complex, and as one might want an interactive theorem prover to support more and more notation, so one might attempt to construct cleverer and cleverer parsers. This approach is inherently problematic. One quickly runs into issues of ambiguity. Often to read mathematical notation one has to be aware of the immediate context it is presented in. A simple example is that juxtaposition of symbols can mean function application in one place and multiplication in another. A notion introduced in programming languages to address ambiguity has been that of *overloading* operators; one assumes that the type-checker can sort out what is meant, even if the parser cannot. Closely related to this notion, is the notion of *implicit coercions*. There is also the question of what notation is supported by editors; mathematics presented in ASCII characters is not anywhere as easy to read as mathematics in books and papers.

A half-way solution that is sometimes taken (for example with Mathematica), is to do what one can with a parseable syntax in ASCII characters for input, and then use pretty-printing routines for formatted output (say in Display PostScript).

The approach which we have taken [1] is to design an editor that presents terms in pretty notation, and groups the notation in chunks that correspond to parts of the underlying tree structures. One edits the tree structure directly, so there is no need for a parser. Such editors are often called *structured editors*.

The advantages of a structured editor are:

- We don't have to worry about making the notation be umambiguous to a machine. It just has to unambiguous to a human, who is aware of the full context the notation is used in.
- We have the opportunity to break away from the presentation of mathematics in ASCII characters. Nuprl currently uses a single 8-bit font of up to 256 characters, but the possibilities exist for using LaTeX and Display PostScript -like technology to generate almost text-book quality displays.
- Notation can be freely changed without altering the underlying logical structure of terms.

[1] and which others have taken too, for example in the Cornell Synthesizer-Generator project

- The possibility is opened up for context dependent notation. We could have presentations of theorems, definitions and proofs decorated with information on local abbreviations.

- If you find notation confusing, you need only point and click the mouse on the notation in question for an explanation.

Structured editors do have their disadvantages. The most major one is that they are quite different from conventional text editors such as `vi` or `emacs`, and so it can take a while to learn how to use them. We have tried to design the Nuprl editor to reduce this startup time. We welcome suggestions from users for further improvements. Another disadvantage is that you have less control over formatting, since all display formatting is done automatically. Again we have been working to enhance the pretty-printing algorithm that Nuprl uses so that the formatting is acceptable.

4.3.2 Display Forms

We describe here our notion of a *display form*.

A *display form definition* associates a chunk of notation with a term. For example consider the term `add(`x`;`y`)` for binary addition. The usual notation for this is to use an infix $\ulcorner$`+`$\urcorner$. We could write the notation chunk as:

$$\boxed{\square\ \texttt{+}\ \square}$$

where the $\square$'s are holes for the two subterms, and the outer box shows the extent of the chunk. We call these holes *term slots* because in they can be filled by terms. Later on we shall encounter *text slots* which can only be filled with text strings. Usually we need to indicate how term slots correspond to the logical subterms of a term so we label term slots. For example, the definition of the notation chunk for `add(`x`;`y`)` can be written:

$$\boxed{\boxed{x}\ \texttt{+}\ \boxed{y}} =_{\text{dform}} \texttt{add(}x\texttt{;}\ y\texttt{)}$$

Here, we read $a\ =_{\text{dform}} b$ as saying that a is defined as the *atomic* notation chunk or *display form* for b. Throughout this section, we use rectangular boxes to delimit terms and term slots.

Term slots stretch to accomodate the terms inserted in them. For example say we have the term `mul(1;2)` which is displayed as $\ulcorner$`1 * 2`$\urcorner$. Then the term `add(mul(1;2);3)` will be displayed as:

$$\boxed{\boxed{\boxed{\texttt{1}}\ \texttt{*}\ \boxed{\texttt{2}}}\ \texttt{+}\ \boxed{\texttt{3}}}$$

Nuprl automatically adds parentheses according to display form *precedences*. When a display form of lower precedence is inserted into the slot of display form with higher precedence, parentheses are automatically inserted to delimit the slot. For example, we assign the display form for `mul(`x`;`y a higher precedence than the display form for `add(`x`;`y`)`. The term `add(mul(1;2);3)` is displayed as

```
1 * 2 + 3
```

but the term `mul(1;add(2;3))` is displayed as

```
1 * (2 + 3)
```

Let us move on to a more complicated display form; that for universal quantification. The term `all(`T`;`x`.`P`)` means that "for all x of type T, the proposition P is true". Note that the term `all(`T`;`x`.`P`)` binds free occurrences of x in P. We would normally write `all(`T`;`x`.`P`)` as

$$\forall x{:}T.\quad P$$

The display form definition for the `all(`T`;`x`.`P`)` could be written as:

$$\boxed{\forall\ x\ {:}\boxed{T}.\quad \boxed{P}} =_{\text{dform}} \texttt{all}(T;\ x.P)$$

Here, $\ulcorner \bigcirc \urcorner$ is used to indicate a *text* slot. A text slot is filled with a text string rather than a term. Text slots are used for term parameter values, and binding variables.

A few more display form definitions are:

1. $\boxed{\exists\ x\ {:}\boxed{T}.\quad \boxed{P}} =_{\text{dform}} \texttt{exists}(T;\ x.P)$
2. $\boxed{\boxed{x}\ \texttt{=}\ \boxed{y}} =_{\text{dform}} \texttt{equal\_int}(x;\ y)$
3. $\boxed{\ x\ } =_{\text{dform}} \texttt{variable}\{x\texttt{:v}\}$
4. $\boxed{\ i\ } =_{\text{dform}} \texttt{natural\_number}\{i\texttt{:n}\}$

The last two display form definitions are rather special; 3 is the display form definition for variable terms, and 4 is the display form definition for natural numbers. Both the display forms defined for these terms have only a single text slot, and no other printing or whitespace characters.

Using these display forms the term

```
all(int(); i.exists(int(); j.equal(int(); j; add(i;1))))
```

has the notation:

$$\boxed{\forall\ \mathtt{i}\ {:}\boxed{\mathbb{Z}}.\quad \boxed{\exists\ \mathtt{j}\ {:}\boxed{\mathbb{Z}}.\quad \boxed{\boxed{\mathtt{j}}\ \texttt{=}\ \boxed{\boxed{\mathtt{i}}\ \texttt{+}\ \boxed{\mathtt{1}}}}}}$$

or leaving out the bounding boxes and circles for the slots and the term as a whole:

$$\forall\mathtt{i}{:}\mathbb{Z}.\ \ \exists\mathtt{j}{:}\mathbb{Z}.\ \ \mathtt{j}\ \texttt{=}\ \mathtt{i}\ \texttt{+}\ \mathtt{1}$$

In general, a display form for a term is made up of 0 or more text and term slots, interspersed with printing and space characters. We can annotate display forms with *whitespace formatting* commands which specify where linebreaks can be inserted, and how to control indentation. Chapter 6 describes in detail the structure and appearance of the display form definitions which are contained in *display* objects in Nuprl theories. Chapter 6 also contains information on how to set precedences, and how to control how precedence affects parenthesization.

The notation for some term tree is built up from the display forms associated with each node of the tree — so the structure of the notation mirrors the structure of the term, and it makes sense to talk about the *display form tree* of a term.

The display form tree is the tree structure that you edit with Nuprl's term editor. Nuprl takes care of translating back-and-forth between the two kinds of trees. In a display form tree, each display form and each slot is considered a node of the tree If a text (term) slot is not empty, it is identified with text string (display form) filling it. All the slots of a display form are considered to be the immediate children of the display form. The editor considers slots ordered in the order they appear, left to right, in display form definitions, not in the order in which they occur in the uniform syntax.

In most of this manual, we refer to terms by their display notation rather than their uniform syntax, unless we want to emphasize their logical structure. Also, in talking the term editor, we talk informally about nodes of terms, when we are referring to nodes of the corresponding display-form trees.

When one enters a new terms using Nuprl's structured editor, one most often enters the term in a top-down fashion, starting with the root of the term tree and working on down to the leaves. This means that one has to work with *incomplete* terms. For example, at an intermediate stage of entering the term

$$\forall \mathtt{i}{:}\mathbb{Z}.\ \ \exists \mathtt{j}{:}\mathbb{Z}.\ \ \mathtt{j\ =\ i\ +\ 1}$$

you might be presented with term:

$$\forall \mathtt{i}{:}\mathbb{Z}.\ \ \exists \mathtt{[var]{:}[type]}.\ \ \mathtt{[prop]}.$$

Here `[var]`, `[type]` and `[prop]` are *place-holders* for slots. `[var]` is a place-holder for a text slot, and `[type]` and `prop` are place-holders for term slots. If a slot has a place-holder, we say that the slot is *empty*, or *uninstantiated*. The labels which appear in the place-holders for a display form (the `var`, `type` or `prop` in the example above) are controlled by the display form's definition. If a text (term) slot contains a a text string (term) we say that slot is *filled* or *instantiated*. If a display form has no uninstantiated slots, then it is considered *complete*. Placeholders re-appear when the contents of slots are removed.

4.3.3 Editor Cursors

One navigates around a term by moving a cursor, sometimes called the *point* by analogy with emacs. The cursor can be in one of three modes:

term mode

A term mode cursor is always positioned at some term node of the term tree. The term node it indicated, by highlighting its notation and the notation for all its subtrees. The highlighting is usually achieved by using reverse video; swapping foreground and background colors. In this document we indicate a highlighted region of a term by drawing an outline around it. For example,

$$\forall \mathtt{i}:\mathbb{Z}.\ \ \exists \mathtt{j}:\mathbb{Z}.\ \ \boxed{\mathtt{j = i + 1}}$$

indicates that a term cursor is at the subterm `j = i + 1`. Occasionally a term has no width, and a term cursor on such a term is displayed as a thin vertical line. In this document, we indicate such a cursor by ‖.

text mode

The text cursor is used for editing text in text slots. The cursor is represented as ⌜|⌝. It is positioned either between two adjacent characters of a text slot, or before the first character, or after the last. For example, consider a text slot containing the text string ⌜`abcdef`⌝. Valid text cursors for this string include

`|abcdef`   `abc|def`   `abcdef|`

The text cursor is the insertion point for new characters.

There is a potential ambiguity as to *which* text slot a text cursor is at: consider two adjacent text slots containing the strings ⌜`aaa`⌝ and ⌜`zzz`⌝ and the following text cursor:

`aaa|zzz.`

Display forms are designed so this kind of situation should never occur.

The text cursor is significantly thinner than the term cursor on a no-width term, so it should be easy to distinguish the two.

screen mode

Certain cursor motion commands are designed for moving around a term's display character-by-character in much the same way as with a conventional text editor. When moving with these commands the cursor always occupies a single character position on the screen. If possible, the editor uses a text cursor. Otherwise it uses a *screen cursor*. A screen cursor on a character is displayed by outlining the character.

For example, if we had the following text cursor in a term:

$$\forall|\mathtt{i}:\mathbb{Z}.\ \ \exists \mathtt{j}:\mathbb{Z}.\ \ \mathtt{j = i + 1}$$

then a 'move-left-one-character' command would leave a screen cursor (indicated by a box) over the ⌜∀⌝.

$$\boxed{\forall}\mathtt{i}:\mathbb{Z}.\ \ \exists \mathtt{j}:\mathbb{Z}.\ \ \mathtt{j = i + 1}$$

In the rest of this document we'll never have to explicitly represent a screen cursor, so all outlined terms should be interpreted as term cursors.

4.3.4 Sequences

The term editor has special features for handling certain kinds of *sequences* of terms. It makes sequences appear much like terms with variable numbers of subterms. The kinds of sequences supported are described below.

Sequences are constructed by the right-associated use of pairing terms. Each kind of sequence has its own pairing term, and also a special term to represent the empty sequence. Eventually, we'll relax the right-association restriction. Often there is no need to distinguish between a term and a one element sequence containing that term. So, in specific contexts, the editor treats a term as a one element sequence. These contexts are pointed out at relevant points in this document.

The term tree cursor motion commands skip over this internal structure, and for nearly all purposes the internal structure of sequences can safely be ignored.

4.3.4.1 Term Sequences

A *term sequence* has a linear sequence of term slots. For example, one kind of sequence which happens to have 4 empty slots might be displayed as:

```
([elmnt],[elmnt],[elmnt],[elmnt])
```

. All the term slots of the sequence are considered siblings in the display form tree, and the whole sequence is their immediate parent.

The editor has special commands for inserting and deleting both elements and segments of term lists.

Different kinds of term sequences have different left and right delimiters, (the ⌜`(`⌝ and ⌜`)`⌝ respectively in the example) and different element separators (the ⌜`,`⌝ in the example). Delimiters and separators in term sequences always consist of at least one character.

4.3.4.2 Text Sequences

A *text sequence* is a text string in which zero or more characters are replaced with terms. Text sequences are primarily used for ML code, for comments, and for the left-hand sides of display forms.

The editor presents a text sequence as a display form with alternating text and term slots. A text sequence normally has no left or right delimiters or element separators, in contrast to term sequences. Text sequences are however easily identified because they usually occur in well-defined contexts.

An example of a text sequence is the ML expression:

```
With 'n + 1' (D 0)~
THENW TypeCheck~
```

This text sequence consists of 3 term slots filled with the terms ⌜`'n + 1'`⌝, ⌜`~`⌝, and ⌜`~`⌝, and 4 text slots filled with the text strings ⌜`With `⌝, ⌜` (D 0)`⌝, ⌜`THENW TypeCheck`⌝, and ⌜⌝ (the null or empty text string). The ⌜`~`⌝'s are new-line terms. Keeping new-line characters out of text strings simplifies the display formatting algorithm. Usually we make new-line terms invisible, but here we show them with a printing character for clarity.

The editor supports special operations on text sequences. For example, you can cut out subsequences delimited by any text cursor positions, and paste in at any text cursor position.

4.4 Term Editor

4.4.1 Introduction

Term editor windows are used for viewing and editing terms. The ML Top Loop window is a term editor window, as are the windows opened when you view most kinds of Nuprl library objects. Each window displays a single display form tree representing a single term. The editor accepts input from both the keyboard, keypad and the mouse. All editing operations can be carried out from the keyboard alone, though frequently the mouse and keypad commands are far simpler and easier to remember. Mouse commands are described in Section 4.4.9.

With a text cursor, keystrokes corresponding to printing characters cause those characters to be inserted. With a term or screen cursor, printing characters can form part or all of editor commands.

Input characters typed at the keyboard in multi-character commands are echoed as highlighted text near the position of the cursor, and can be corrected by using $\boxed{\text{DELETE}}$.

The default key bindings are intended to be reminiscent of emacs's key bindings. You may wish to use alternative key bindings. See the editor customization section for details (*not yet written*).

The editor adjusts the display of an object in a window to the size of the window. If the window is too small, not all the object can be displayed at once. In this event, one can resize the window, or scroll the window up and down. Sometimes, if the window is to narrow, some subterms are *elided*. The display form tree for an elided subterm is replaced by ⌜...⌝. Eventually, you will be able to examine elided subterms by moving the root display form of an editor window to some term tree position other than the term root. Currently, the only way to encourage the system to un-elide a subterm is to widen the window as much as possible.

4.4.2 Cursor/Window Motion

Also see Section 4.4.9 for how to use the mouse to move around.

4.4.2.1 Screen Oriented

The screen motion commands are described in Table 4.2.

These cursor motion commands ignore the structure of the term in the window. They allow one to quickly navigate to parts of the screen. After a screen cursor command the cursor is always either in text mode or screen mode. A useful command to use when ending up with the cursor over the printing character of a display form is the SWITCH-TO-TERM command. If one tries to moves the cursor over the top or bottom of the display, the window scrolls appropriately. There are also explicit window scrolling commands.

4.4.2.2 Tree Oriented

The tree walking commands are summarized in Table 4.3.

⟨C-P⟩	SCREEN-UP	move cursor up 1 character
⟨C-N⟩	SCREEN-DOWN	move cursor down 1 character
⟨C-B⟩	SCREEN-LEFT	move cursor left 1 character
⟨C-F⟩	SCREEN-RIGHT	move cursor right 1 character
⟨C-A⟩	SCREEN-START	move to left side of screen
⟨C-E⟩	SCREEN-END	move to right side of screen
⟨C-L⟩	SCROLL-UP	scroll window up 1 line
⟨M-L⟩	SCROLL-DOWN	scroll window down 1 line
⟨C-V⟩	PAGE-DOWN	move window down 1 page
⟨M-V⟩	PAGE-UP	move window up 1 page
⟨C-T⟩	SWITCH-TO-TERM	switch to term mode

Table 4.2: Screen Motion Commands

⟨M-P⟩	UP	move up to parent
⟨M-B⟩	LEFT	structured move left
⟨M-F⟩	RIGHT	structured move right
⟨M-N⟩	DOWN-LEFT	move to leftmost child
⟨M-M⟩	DOWN-RIGHT	move to rightmost child
⟨M-A⟩	LEFTMOST-SIBLING	move to left-most sibling
⟨M-E⟩	RIGHTMOST-SIBLING	move to right-most sibling
⟨M-<⟩	UP-TO-TOP	move up top of term
⟨C- LINEFEED ⟩	RIGHT-LEAF	next leaf to right
⟨M- LINEFEED ⟩	LEFT-LEAF	next leaf to left
RETURN	RIGHT-EMPTY-SLOT	next empty slot to right
⟨C- RETURN ⟩	RIGHT-EMPTY-SLOT	next empty slot to right
⟨M- RETURN ⟩	LEFT-EMPTY-SLOT	next empty slot to left

Table 4.3: Tree Motion Commands

UP, LEFT, RIGHT, DOWN-LEFT, DOWN-RIGHT are the basic walking commands. Within text slots, LEFT and RIGHT stop at each word. (Use screen motion commands to move by character.) These commands recognize text and term sequences, and skip over their internal structure.

RIGHT-LEAF, LEFT-LEAF, RIGHT-EMPTY-SLOT, LEFT-EMPTY-SLOT are particularly good for rapidly moving around terms, since you can often get where you want to go by just repeatedly using one of them. Note that the binding of RETURN to RIGHT-EMPTY-SLOT doesn't work in text sequences. In that case, you need to use ⟨C- RETURN ⟩.

4.4.3 Adding New Text

These commands are for inserting text whenever you have a text cursor. The commands are summarized in Table 4.4.

Standard ASCII printing characters (including space) self insert whenever one has a text cursor. Non-standard characters can be inserted using INSERT-SPEC-CHAR-*num*. *num* is the decimal code for the character. (See Appendix ?? for a table of special character codes, or execute in a unix shell window

x	INSERT-CHAR-x	insert char x
⟨C-#⟩*num*	INSERT-SPEC-CHAR-*num*	insert special char x
RETURN	INSERT-NEWLINE	insert newline

Table 4.4: Text Insertion

```
xfd -font nuprl-13 &
```

to bring up a display of the font. Clicking MOUSE-LEFT on a character results in its decimal code being displayed.) Alternatively, special characters can be copied from the object `FontTest` in the `core-1` theory.

The INSERT-NEWLINE is only appropriate in text sequences, since the newline 'character' is actually a term. This restriction simplifies the display layout algorithm and should not prove to be an inconvenience.

4.4.4 Adding New Terms

The insertion commands for terms are shown in Table 4.5. These commands are only appropriate with a term cursor.

name	INSERT-TERM-*name*	insert *name*
⟨C-I⟩*name*	INSERT-TERM-LEFT-*name*	insert *name*
⟨M-I⟩*name*	INSERT-TERM-RIGHT-*name*	insert *name*
⟨C-S⟩*name*	SUBSTITUTE-TERM-*name*	replace with *name*
⟨CM-I⟩	INITIALIZE-TERM	initialize term slot
⟨CM-S⟩	SELECT-DFORM-OPTION	selects dform variations

Table 4.5: Term Insertion

name in these commands is a string of characters, naming a new term to be inserted. The interpreter for *name* strings checks each of the following conditions until it finds one which applies.

1. *name* is an editor command enabled in a particular context. See sections for examples.

2. *name* is an *alias* for some display form, defined in in the library object for that display form.

3. *name* is the name of a display form object. It refers to the first display form defined in that object.

4. *name* is of the form *ni* where *n* is the name of a display form object and *i* is a natural number. *ni* refers to the *i*th display form definition in the object named *n*. Definitions in objects are numbered starting from 1.

5. *name* is the name of an abstraction object, then *name* refers to the earliest display form in the library for that abstraction.

6. *name* is all numerals, then the term referred to is the `natural-number{`*name*`:n}()` term of Nuprl's object language.

7. *name* refers to the variable term `variable{`*name*`:v}()`.

Names always have acceptable extensions as variable names, so the editor doesn't interpret *name* until some explicit terminator is typed. For example, this can either be NO-OP ($\boxed{\text{SPACE}}$) or a cursor motion command. NEXT-EMPTY-SLOT ($\langle$C-$\boxed{\text{RETURN}}\rangle$) is a particularly useful terminator.

INSERT-TERM-*name* is only applicable at empty term slots. It results in the display form referred to by *name* being inserted into the slot. If *name* is terminated by a NO-OP, then a term cursor is left at the new term. If *name* is teminated by some cursor motion command, then that command is obeyed.

INSERT-TERM-LEFT*name* is intended for use at a filled term slot. Its behavior is to:

1. save the existing term in the slot, leaving the slot empty,
2. insert the new display form referred to by *name* into the slot,
3. paste the saved term into the left-most term slot of the new display form. If the new display form has no term slots, then the saved term is lost.

INSERT-TERM-RIGHT*name* behaves in a similar way to INSERT-TERM-LEFT except that in step 3, the saved term is pasted into the right-most term slot of the new display form.

SUBSTITUTE-TERM-*name* allows you to replace one display form with another which has the same sequence of child text and term slots. The children of the old display form become the children of the new display form. In the event that the new display form has a different sequence of children SUBSTITUTE-TERM-*name* tries something vaguely sensible. In general, in these cases, it is safer to explicitly cut and paste the children.

INITIALIZE-TERM initializes a term slot to some default term if one is appropriate. The term slot must initially be empty. INITIALIZE-TERM is automatically invoked by Nuprlto initialize new windows. If you want to re-initialize a window, place a term cursor at the root of the term in the window, delete the term, and then give the INITIALIZE-TERM command. The default terms for particular contexts are described in various sections of this document. If no default has been designated, INITIALIZE-TERM does nothing.

SELECT-DFORM-OPTION when the term cursor is at certain terms, selects an alternative display form for that term. For example, if term cursor is positioned at an independent function type, it selects the more general dependent-function display form.

4.4.5 Cutting and Pasting

The cut-and-paste commands work on terms, segments of text slots, and segments of text and term sequences. In this section we refer to these collectively as *items*. Cut items can be saved on a *save stack*. All items on the save stack are represented as terms, and it is often possible to cut one kind of item and then paste into another kind of context. For example, one can cut a term, and paste into text sequence, or cut a segment of text from a text slot, and paste into a term sequence.

We define the following kinds of commands:

- SAVE: does not remove an item, but does push a copy onto the save stack. Same idea as *copy-as-kill* in emacs.
- DELETE: remove an item from a buffer, *not* saving it anywhere.

- CUT: (= SAVE + DELETE) removes an item from a buffer and pushes it onto the top of the *save stack*. Same idea as *kill* in emacs, although Nuprl does *not* append together items cut immediately one after the other.

- PASTE: inserts the item on top of the stack back into a buffer, removing it from the stack.

- PASTE-COPY: inserts the item on top of the stack back into a buffer, not removing it from the stack. Same idea as *yank* in emacs.

- PASTE-NEXT: Only used immediately after a PASTE. Removes the item just pasted from the buffer, and then does a PASTE. Same idea as *yank-next* in emacs.

4.4.5.1 Basic

The basic cut and paste commands are shown in Table 4.6.

DELETE	DELETE-CHAR-TO-LEFT	delete char to left of text cursor
⟨C-D⟩	DELETE-CHAR-TO-RIGHT	delete char to right of text cursor
⟨M-D⟩	CUT-WORD-TO-RIGHT	cut word to right of text cursor
⟨C-K⟩	CUT	cut term
⟨M-K⟩	SAVE	save term
⟨CM-K⟩	DELETE	delete term
⟨C-Y⟩	PASTE	paste item
⟨M-Y⟩	PASTE-NEXT	delete item then paste next item
⟨CM-Y⟩	PASTE-COPY	paste copy of item

Table 4.6: Basic Cutting and Pasting

DELETE-CHAR-TO-LEFT and DELETE-CHAR-TO-RIGHT are conventional character deletion commands. They can be used in any text slot of a term or in a text sequence. They will also work on newline terms in text sequences. They do not save the character on the save stack.

CUT-WORD-TO-RIGHT cuts the word to the right of a text cursor. For convenience if a term is to the immediate right of a text cursor in a text sequence, then that term is cut.

CUT, SAVE, AND DELETE all work on a term underneath a term cursor. SAVE pushes a copy of the term onto the save stack leaving the term itself in place, DELETE deletes the term, leaving an empty term slot, and CUT is the same as a SAVE followed by a DELETE. These commands work fine on terms in text and term sequences.

When a term cursor is at an empty term slot, the PASTE and PASTE-COPY commands paste the term on top of the stack into the slot. PASTE always removes the term from the top of the save stack, so successive pastes retrieve successively-earlier cut terms. PASTE-COPY is like PASTE, except the item pasted is also left on top of the save stack. This is useful if you want to make several copies of an item.

PASTE-NEXT is only intended to be used immediately after a PASTE or a previous PASTE-NEXT. It deletes the last term pasted, and replaces it with the term before on the save stack. By repeating PASTE-NEXT, you can search back through the save stack for some desired term.

4.4.5.2 Region

A *region* is a segment of any text slot, or a segment of a text or term sequence. The region cut and paste commands are shown in Table 4.7.

⟨C-$\boxed{\text{SPACE}}$⟩	SET-MARK	set mark at point
⟨C-X⟩⟨C-X⟩	SWAP-POINT-MARK	swap point and mark
⟨C-W⟩	CUT-REGION	cut region
⟨M-W⟩	SAVE-REGION	save of region
⟨CM-W⟩	DELETE-REGION	delete region
⟨C-(Y⟩	PASTE	paste region
⟨M-Y⟩	PASTE-NEXT	replace last paste with new paste
⟨CM-Y⟩	PASTE-COPY	paste copy of region on save-stack top

Table 4.7: Region Cutting and Pasting

A region is delimited by the editor's term or text cursor and an auxiliary text or term cursor position. Following emacs's terminology, we call the cursor's position the *point* and the auxiliary cursor position the *mark*.

The SET-MARK command sets the mark to the current cursor position. and the SWAP-POINT-MARK command can be used to check the mark's position. It doesn't matter whether mark is to the left or the right of point when selecting a region. In what follows, we call the left-most of point and mark the left delimiter, and the right-most, the right delimiter. If a term is used a region delimiter, the term is understood to be included in the region.

Various regions are acceptable: for selecting a text string in a text slot, both delimiters must be text cursor positions. For selecting a segment of a term sequence, both delimiters must be term cursor positions. For selecting a segment of a text sequence, you can use either a text cursor or a term cursor position for each delimiter.

SAVE-REGION saves a region on the save stack. DELETE-REGION deletes the region. The kind of cursor it leaves depends on the kind of region selected. If the region is of a text slot, or a text sequence, DELETE leaves a text cursor at the old position of the region. If the region is of a term sequence, an empty term slot is left in place of the region. CUT-REGION has the same effect as a SAVE-REGION followed by a DELETE-REGION.

The paste commands for regions are the same as the basic paste commands. You can paste with a text cursor in a text slot or text sequence, and a term cursor at any empty term slot. If you paste a sequence into another sequence of the same kind, paste merges the pasted sequence into the sequence being pasted into. In this event, the point is set to be the left-delimiter for the just pasted sequence, and the mark is set to be the right-delimiter. This ensures proper functionality for the PASTE-NEXT operation. Otherwise, if you are pasting into a sequence, the pasted item always is incorporated as a single sequence element, and both the mark and point are set to that element. Note that it doesn't make sense to try to paste a term or a text sequence containing a term into a text slot that is not in a text sequence.

4.4.6 Adding and Removing Slots in Sequences

The commands are summarized in Table 4.8.

⟨C-U⟩	OPEN-SEQ-TO-LEFT	open slot to left of cursor
⟨M-U⟩	OPEN-SEQ-TO-RIGHT	open slot to right of cursor
⟨C-O⟩	OPEN-SEQ-LEFT-AND-INIT	open slot to left and init
⟨M-O⟩	OPEN-SEQ-RIGHT-AND-INIT	open slot to right and init
⟨C-C⟩	CLOSE-SEQ-TO-LEFT	close slot and move left
⟨M-C⟩	CLOSE-SEQ-TO-RIGHT	close slot and move right

Table 4.8: Sequence Term Slot Editing

If a term cursor is at an element of either a term or a text sequence, then OPEN-SEQ-TO-LEFT and OPEN-SEQ-TO-RIGHT add a new empty slot to the left and right respectively of the cursor. The cursor is left at the new empty slot. On an empty term sequence, the two commands have the same effect; they simply delete the nil sequence term. If a text cursor is in a text sequence, both commands open up an empty term slot at the text cursor, and leave the cursor at the new slot.

With text or term sequences represented by a single term, these commands infer the kind of sequence to create from context. Occasionally with term sequences, more than one kind of sequence is permitted in a given context (for example, in precedence objects) and in such cases you can use explicit term insertion commands to create the sequence. Such ambiguity shouldn't arise with text sequences.

OPEN-SEQ-LEFT-AND-INIT and OPEN-SEQ-RIGHT-AND-INIT are similar, but if there is some obvious term to insert in the opened up slot, then that term is automatically inserted and the cursor is left at an appropriate position in the new term.

If a term cursor is at an empty term slot in a term sequence, the CLOSE-SEQ-TO-LEFT and CLOSE-SEQ-TO-RIGHT commands delete the slot, and then (if possible) move the cursor to the element to the left or right respectively of the slot just deleted. If the term slot is filled with a term, that term is first deleted. If the term slot is in a text sequence, these commands leave a text cursor at the position of the deleted slot.

4.4.7 Opening, Closing, and Changing Windows

The relevant editor commands are shown in Table 4.9

⟨C-Q⟩	QUIT	close window without saving
⟨C-Z⟩	EXIT	save, check, and close window
⟨C-J⟩	JUMP-NEXT-WINDOW	jump to next window
TAB	JUMP-ML	jump to ML top loop

Table 4.9: Commands For Changing and Closing Windows

Term editor window are opened by using the ML `view` command on a library object. They are also opened by the proof editor, when then proof editor SELECT command is issued on sequents and ruleboxes, and when the proof editor TRANSFORM command is given.

EXIT first saves a copy of the object. It then checks the object before closing the window. This *checking* has the same effect on library objects as using the ML `check` command. If the check fails, then the window is left open. If you still want to close the window, use QUIT. Seperate save and check commands are described in Section 4.4.8.

QUIT is an abort command. It closes the window, abandoning any changes made to the window since it was last checked by attempting EXIT.

JUMP-NEXT-WINDOW allows one to cycle around all the currently open windows, including any proof editor windows.

JUMP-ML moves the cursor over to the ML top-loop window.

4.4.8 Utilities

⟨C-X⟩id	IDENTIFY-TERM	gives info on term at cursor
⟨C-X⟩su	SUPPRESS-DFORM	suppress display form at cursor
⟨C-X⟩un	UNSUPPRESS-DFORM	unsuppress display form at cursor
⟨C-X⟩ex	EXPLODE-TERM	explode term at cursor
⟨C-X⟩im	IMPLODE-TERM	implode term at cursor
⟨C-X⟩ch	CHECK-OBJECT	check object
⟨C-X⟩sa	SAVE-OBJECT	save object
⟨C-X⟩ab	VIEW-ABSTRACTION	view abstraction def of term
⟨C-X⟩df	VIEW-DFORM	view display form def for term
⟨C-X⟩ns	INSERT-EMPTY-STRING	insert empty string in text slot

Table 4.10: Utility Commands

Various utility commands are shown in Table 4.10 The IDENTIFY-TERM, SUPPRESS-DFORM and UNSUPPRESS-DFORM commands assist one in interpreting unfamiliar or ambiguous display forms.

IDENTIFY-TERM will print out in the ML Top-Loop window information on the term and display form at the current cursor position. If one likes, one can then go and view the appropriate display form and abstraction objects.

SUPPRESS-DFORM suppresses use of the display form the cursor is sitting at for the whole object one is viewing. If multiple display forms are defined for a term, a single SUPPRESS-DFORM might result in some other more general display form being selected. In this case one can repeat SUPPRESS-DFORM. When all appropriate display forms for a term are suppressed, the term is displayed in uniform syntax.

UNSUPPRESS-DFORM restores a suppressed display form, when the editor cursor is at a term to which that suppressed display form belongs. Display forms remain suppressed until one explicitly unsuppresses, or until one closes the editor window.

EXPLODE-TERM replaces the term the cursor is at with a cluster of terms which display the term in uniform syntax, and allow one to change the operator structure. For example one can change the opid name,the number and types of the parameters, or the term's arity. See Section 4.5.2 for details on how to edit an exploded term's structure.

IMPLODE-TERM replaces an exploded term at the cursor by the term which the exploded term represents.

INSERT-EMPTY-STRING is useful for inserting empty text strings into text slots. Normally, when all the characters in a text slot that is outside of a text sequence are deleted, a text slot placeholder is left rather than an empty string. This is because usually such slots are used for things like variable names, and using the empty string for such entities can be confusing. Use this command when an empty text string is what is really wanted.

4.4.9 Mouse Commands

The mouse commands are shown in Table 4.11

MOUSE-LEFT	MOUSE-SET-POINT	set mark then point
⟨C- MOUSE-LEFT⟩	MOUSE-SET-TERM-POINT	set mark then point to term
MOUSE-MIDDLE	MOUSE-VIEW-DISP	view display form of term
⟨C- MOUSE-MIDDLE⟩	MOUSE-PASTE	as PASTE
⟨M- MOUSE-MIDDLE⟩	MOUSE-PASTE-NEXT	as PASTE-NEXT
⟨CM- MOUSE-MIDDLE⟩	MOUSE-PASTE-COPY	as PASTE-COPY
MOUSE-RIGHT	MOUSE-VIEW-AB	view abstraction definition of term
⟨C- MOUSE-RIGHT⟩	MOUSE-CUT	cut term or region
⟨M- MOUSE-RIGHT⟩	MOUSE-SAVE	save term or region
⟨CM- MOUSE-RIGHT⟩	MOUSE-DELETE	delete term or region

Table 4.11: Mouse Commands

The mouse commands are designed to allow easy jumping around terms, cut-and-pasting, and viewing of information on terms.

MOUSE-SET-POINT first sets the mark at the current editor cursor position, (not the mouse position) and *then* sets the point, the editor's cursor, to where the mouse is pointing. MOUSE-SET-POINT sets point to either a term cursor or text cursor. It chooses a text cursor if one is valid between the character pointed to by the cursor and the character to the immediate left. If there is a null width term to the immediate left of the mouse, the cursor is set to that term. Otherwise, the cursor is set to the most immediate surrounding term which contains the character being pointed to by the mouse. This command is set up so that one can select a region by using MOUSE-SET-POINT at one end of the region and then MOUSE-SET-POINT at the other; after the second MOUSE-SET-POINT the mark will be at one end of the region and point will be at the other.

MOUSE-SET-TERM-POINT is like MOUSE-SET-POINT except that point is always set to the term immediately surrounding the character being pointed to.

MOUSE-CUT is the same as CUT-REGION in text sequences. and text slots. Otherwise it behaves the same as CUT. Likewise with MOUSE-SAVE. MOUSE-PASTE is the same as PASTE, and MOUSE-PASTE-COPY is the same as PASTE-COPY.

4.5 Editing Term Structure

4.5.1 New Term Entry

The term editor recognizes certain input sequences as indicating that a new term should be created. A new term structure can be created as follows:

- Position a term cursor and an empty slot.
- Enter the letters of the new term's opid
- Enter a (possibly empty) list of single letters, designating the new term's parameter types. The list should be delimited by ⌜{⌝ and ⌜}⌝ characters, and elements should be seperated by ⌜,⌝ characters. Empty lists of parameter types are optional.

- Enter a (possibly empty) list of numbers, designating the number of binding variables for each subterm. The list should be delimited by ⌜(⌝ and ⌜)⌝ characters, and elements should be seperated by ⌜;⌝ characters. This list specifying the arity of the term must be entered, even when it is empty.

For example, if you enter

```
myid{n,t}(0;1)
```

the term `myid{[natural]:n, [token]:t}([term]; [binding].[term])` is created.

4.5.2 Exploded Terms

A term constructor is *exploded* when it is replaced by a special collection of terms, so arranged so that you can edit the structure of the term constructor; change its opid, change the number and kind of its parameters, or change its arity. Note that in practice, the only time you usually edit exploded terms is when you add or change the definition of an abstraction.

The commands for editing exploded terms are summarized in Table 4.12.

⟨C-X⟩ex	EXPLODE-TERM	explode term at cursor
⟨C-X⟩im	IMPLODE-TERM	implode term at cursor
exterm	INSERT-TERM-exterm	insert new exploded term
lparm	INSERT-TERM-lparm	insert level exp parm
vparm	INSERT-TERM-vparm	insert variable parm
tparm	INSERT-TERM-tparm	insert token parm
sparm	INSERT-TERM-sparm	insert string parm
nparm	INSERT-TERM-nparm	insert natural number parm
⟨C-O⟩	OPEN-SEQ-TO-LEFT	open bterm / parm / bvar slot to left
⟨M-O⟩	OPEN-SEQ-TO-RIGHT	open bterm / parm / bvar slot to right

Table 4.12: Exploded Term Editing

To show how they are used, we walk through the entry of the term `foo{bar:s}`(A;x.B). Position a term cursor at an empty term slot and enter:

exterm SPACE

.

The highlighted term should look like:

```
EXPLODED<<[opid]{}()>>
```

Enter the opid:

⟨C-RETURN⟩foo

to get:

```
EXPLODED<<foo|{}()>>
```

Click MOUSE-LEFT on the }, and you should get a null width term cursor sitting on an empty term sequence for parameters.

```
EXPLODED<<foo{||}()>>
```

Enter ⟨C-0⟩ to add a new slot to the parameter sequence:

```
EXPLODED<<foo{ [parm] }()>>
```

Insert the string parameter with text bar:

```
sparm RETURN bar
```

to get:

```
EXPLODED<<foo{bar|:s}()>>
```

Click MOUSE-LEFT on the) to get a null width term cursor sitting on an empty term sequence for bound terms:

```
EXPLODED<<foo{bar:s}(||)>>
```

Enter ⟨C-0⟩⟨C-0⟩ to make a two element sequence for bound terms, leaving the cursor on the left-most element.

```
EXPLODED<<foo{bar:s}( .[term] ;.[term])>>
```

Click MOUSE-LEFT on the second ⌜.⌝ to get a null width term cursor sitting on an empty term sequence for binding variables:

```
EXPLODED<<foo{bar:s}( .[term];||.[term])>>
```

Enter ⟨C-0⟩ RETURN to open up a slot in the sequence, and enter a binding variable term:

```
EXPLODED<<foo{bar:s}( [term];|[bvar].[term])>>
```

Finally, click MOUSE-LEFT on any part of EXPLODED and then enter

```
⟨C-X⟩im
```

to implode the exploded terms. You should now have the term:

```
foo{bar:s}([term];[var].[term])
```

You could now go ahead and fill in the binding variable, and subterm slots. In general, when imploding and exploding terms the parameter values, binding variable names, and subterms stay the same, so entering and/or editing them when a term is exploded has the same effect as when the term is imploded.

Chapter 5

Abstractions

Abstractions are terms which are definitionally equal to other terms. They are introduced by abstraction objects in Nuprl theories. An abstraction can be defined in terms of other abstractions, but the dependency graph for abstractions should be acyclic. In particular, an abstraction may not depend on itself. Recursive definitions can be introduced as described in Section 10.2.4.

Abstraction definitions have form:

$$lhs \;\texttt{==}\; rhs$$

The terms *lhs* and *rhs* are *pattern* terms, and there is implicit universal quantification over the the free variables in *lhs* and *rhs*. When Nuprl unfolds some instance *lhs-inst* of *lhs*, it first matches *lhs-inst* against *lhs*, generating bindings for the free variables of *lhs* such that if the bindings were applied as a substitution to *lhs*, one would get back *lhs-inst*. It then applies the substitution to *rhs* to calculate the term *rhs-inst* which *lhs-inst* unfolds to.

For an example of a abstraction, see Figure 5.1. Here we define a type of segments of integers.

EDIT ABS int_seg
$\texttt{\{i..j}^{-}\texttt{\}== \{k:}\mathbb{Z}\texttt{\|i} \leq \texttt{k < j\}}$

Figure 5.1: Definition of the `int_seg` abstraction

The structure of the left-hand side is more redily apparent if we write it in uniform syntax: $\texttt{\{i..j}^{-}\texttt{\}}$ is `int_seg(i;j)`, a term with opid `int_seg`, no parameters, and 2 subterms. An instance of the left-hand side is $\texttt{\{0..10}^{-}\texttt{\}}$ and this would unfold to $\texttt{\{k:}\mathbb{Z}\texttt{|0} \leq \texttt{k < 10\}}$.

Abstractions can have binding structure; for example, consider the *exists unique* abstraction in Figure 5.2.

To handle abstractions with binding variables in a systematic way, we define the procedure for unfolding an abstraction using *second-order* matching and substitution functions.

If you are familiar with second-order matching and substitution, you can skip this paragraph. First-order matching and substitution functions are inadequate for handling terms with binding structure.

```
EDIT ABS exists_uni

∃!u:T. P[u]==∃u:T. P[u] ∧ ∀ v:T. P[v] ⇒ v = u ∈ T
```

Figure 5.2: Definition of the `exists_uni` abstractions

For example, there is no way of applying a first order substitution to the pattern term "$\lambda x.\ y$" to get the instance "$\lambda x.\ x+1$"; if we attempt to apply the substitution $[y \mapsto x+1]$ to "$\lambda x.\ y$", we force renaming of the bound variable x to x', and we get "$\lambda x'.\ x+1$". One could somehow suppress renaming but then substitution becomes ill-behaved; on substitution, free variables can become bound - a process known as *capture*. For more on this, consult some introductory book on predicate logic.

A *second-order* binding is a binding of a *second-order variable* to a *second-order term*. A second-order variable is essentially an identifier as with normal variables, but it also has an associated *arity*; some $n \geq 0$. Second-order terms are a generalization of terms, and can be represented by bound-terms such as $x_1, \ldots, x_{a_n}.t$. They can be thought of as 'terms with holes', terms with zero or more subtrees missing. The binding variables are place-holders for the missing subtrees. In any second-order binding $v \mapsto x_1, \ldots, x_{a_n}.t$, the arity of v must be equal to n.

An *instance* of a second-order variable v with arity n, is a term we write as $v[a_1; \ldots; a_n]$, where $a_1, \ldots, a_n$ are terms. We call $a_1, \ldots, a_n$ the *arguments* of v.

A second-order substitution is a list of second-order bindings. The result of applying the binding $[v \mapsto w_1, \ldots, w_n.t_{w_1,\ldots,w_n}]$ to the variable instance $v[a_1; \ldots; a_n]$, is the term $t_{a_1,\ldots,a_n}$ – the second-order variable's arguments filling the holes of the second-order term.

Going back to the example, the variable `P` is a second order variable with arity 1, and the terms `P[u]` and `P[v]` are second-order-variable instances. Consider unfolding an instance of the left-hand side, say the term

$$\exists!\mathtt{i}:\mathbb{Z}.\ \ \mathtt{i} = 0 \in \mathbb{Z}$$

. Here, "$\sqcup = \sqcup \in \sqcup$" is a 3 place typed equality relation. $a = b \in T$ means that a and b are equal, and are both members of type T. The substitution generated by matching this against

$$\exists!\mathtt{u}:\mathtt{T}.\ \mathtt{P[u]}$$

would be

$$[\mathtt{P} \mapsto \mathtt{i.i} = 0 \in \mathbb{Z}\ ;\ \mathtt{T} \mapsto \mathbb{Z}],$$

and the result of applying this to

$$\exists\mathtt{u}:\mathtt{T}.\ \mathtt{P[u]}\ \wedge \forall \mathtt{v}:\mathtt{T}.\ \mathtt{P[v]}\ \Rightarrow \mathtt{v} = \mathtt{u} \in \mathtt{T}$$

would be

$$\exists!\mathtt{u}:\mathbb{Z}.\ \ \mathtt{u} = 0\ \wedge \forall \mathtt{v}:\mathbb{Z}.\ \ \mathtt{v} = 0 \in \mathbb{Z} \Rightarrow \mathtt{v} = \mathtt{u} \in \mathbb{Z}.$$

The matching and substitution functions used by Nuprl are a little smarter than shown above; they try to maintain names of binding variables. So the result one would get using Nuprl would be:

$$\exists ! \mathtt{i}:\mathbb{Z}.\ \ \mathtt{i} = 0 \wedge \forall \mathtt{v}:\mathbb{Z}.\ \ \mathtt{v} = 0 \in \mathbb{Z} \Rightarrow \mathtt{v} = \mathtt{i} \in \mathbb{Z}.$$

Just as abstractions can be unfolded by applying their definition left-to-right, so instances of their right-hand sides can be folded up to be instances of their left-hand sides. Folding doesn't always work. For example, information can be lost in the unfolding process; Definitions can have variables, parameters and terms that occur on the left-hand side but that don't occur on the right-hand side [1].

Note that only variables and second-order variables with all first-order variable arguments are allowed as subterms of the left-hand side of abstraction definitions.

Abstractions can also contain *meta-parameters*, which the matching and substitution functions treat as variables. We usually indicate that a parameter is meta, be prefixing it with a $ sign. For example, we might define an abstraction `label{`x`:t,`i`:n}`, as shown in Figure 5.3.

```
  EDIT ABS label

label{$tok:t,$nat:n}==pair(token{$tok:t};natural{$nat:n})
```

Figure 5.3: An abstraction with meta-parameters

However, note that all level-expression variables occurring in level-expression parameters in abstraction definitions are always treated as meta-parameters, so there is no need to make them explicitly meta.

In general, the term on the left-hand side of an abstraction can have a mixture of normal and meta parameters. You can define a family of abstractions which differ only in the constant value of some parameter. However it is an error to make two abstraction definitions with left-hand sides which have some common instance.

A recently added feature of abstraction definitions is an optional list of *conditions*. A condition is simply an alpha-numeric label associated with the abstraction. We intend abstraction conditions to be used to hold information about abstractions which would be useful to tactics and other parts of the Nuprl system. For example, abstraction conditions could be used to group abstactions into categories, and when doing a proof, one could ask for all abstractions in a given category to be treated in a particular way.

The general form of an abstraction with conditions $c_1, \ldots, c_n$ is:

$$(c_1, \ldots, c_n)\texttt{::}\,lhs\ \texttt{==}\ rhs$$

In this section, we describe the editor support for entering abstraction definitions. Abstraction objects are created and viewed as described in Chapter??. You can also view the abstraction for some term by using the VIEW-ABSTRACTION command. See Section 4.4.8.

[1]For example, it can be useful to define an abstraction that has some typing information associated with it, but that unfolds to a term without that information

⟨CM-`I`⟩	INITIALIZE	initialize object / condition
⟨CM-`S`⟩	SELECT-TERM-OPTION	open condition seq
⟨C-`O`⟩	OPEN-SEQ-TO-LEFT	open slot in cond seq to left
⟨M-`O`⟩	OPEN-SEQ-TO-RIGHT	open slot in cond seq to right
⟨C-`M`⟩	CYCLE-META-STATUS	make parameter meta / normal
`so_var`n	INSERT-TERM`so_var`n	insert second order var with n args

Table 5.1: Editor commands for Abstraction Objects

The editor support commands are summarized in Table 5.1.

When an abstraction object is first visited, it is initialized with an uninstantiated abstraction definition term. This looks like:

```
[lhs] == [rhs]
```

If you delete the whole term in an abstraction object and then give the INITIALIZE command the object is re-initialized to this state.

The default abstraction definition term has an empty condition sequence as a subterm. You cannot position a cursor at this sequence because a display form hides it. Use the SELECT-TERM-OPTION command with a term cursor over the whole abstraction definition to get an abstraction definition term with an empty term slot for a condition term.

Use the INITIALIZE command with a term cursor at an empty condition sequence slot to initialize the slot with a condition term. The condition term is much like the term for variables; it has a single text slot, and otherwise no other display characters. Use OPEN-SEQ-TO-LEFT or OPEN-SEQ-TO-RIGHT to add additional slots for conditions terms.

To make a parameter into a meta-parameter, position a text cursor in the parameter's text slot and use the CYCLE-META-STATUS. If the parameter is already meta, using this twice will cycle its status back to being a normal parameter. Note that this is not necessary with level-expression parameters. All level-expression variables are treated as meta.

Second order variable instances are entered on the left and right hand sides of the definition using the `variable{`x`:v}(`a_1`;...;`a_n`)` term where x is the variable's name, and $n > 0$. The library display form object defining the display form for `variable{`x`:v}(`a_1`;...;`a_n`)` is named `so_var`n so this family of names can be used to reference them. Note that abstraction objects are the *only* places where these second-order variable instances are used. When writing propositions, second-order variable instances are simulated using the `so_apply(`n`)` abstraction.

Chapter 6

Display

6.1 Display Form Definitions

6.1.1 Top Level Structure

$$
\begin{array}{rcl}
\textit{def-seq} & ::= & \textit{definition}\ \texttt{;;} \\
 & | & \textit{definition}\ \texttt{;;}\ \textit{def-seq} \\
\textit{definition} & ::= & \textit{format-seq}\ \texttt{==}\ \textit{term} \\
 & | & \textit{attr-seq}\ \texttt{::}\ \textit{format-seq}\texttt{==}\ \textit{term} \\
\textit{format-seq} & ::= & \textit{format} \\
 & | & \textit{format}\ \textit{format-seq} \\
\textit{attr-seq} & ::= & \textit{attribute} \\
 & | & \textit{attribute}\ \texttt{::}\ \textit{attr-seq}
\end{array}
$$

Figure 6.1: Display Object Structure

The top level structure of a display form object is summarized by the grammar shown in Figure 6.1. An object contains one or more display form *definitions*. Each definition has a *term* which the display form applies to, and a sequence of *formats* that specify how to display the *term*. A definition also has an optional sequence of *attributes* that specify extra information about the *definition*. Usually, all the definitions in one object refer to a closely related set of terms. When choosing a display form to use for a term, the layout algorithm tries definitions in a backward order, so definitions are usually ordered more general to more specific.

6.1.2 Formats

The various kinds of formats are summarized in Table 6.1. The 'Name' column gives the names by which you can refer to the formats when entering them. The format sequence is always a text sequence so every alternate format is a text string. Since the text strings are always present, there is no need to have to enter them explicitly and consequently we don't give them a name. The slot formats are for children of the display form. The `L`,`E` and `*` options on the term slot formats control parenthesization of the slot, and are discussed in Section 6.3. All the formats enclosed in `{}`'s control insertion of optional spaces, linebreaking, and indentation. They are discussed in Section 6.2.

Display	Name	Description
string		text string
<*id*:*ph*>	slot	text slot
<*id*:*ph*:L>	lslot	term slot
<*id*:*ph*:E>	eslot	term slot
<*id*:*ph*:*>	sslot	term slot
{space}	space	optional space
{→*i*}	pushm	push margin
{←}	popm	pop margin
{*a*}	break	break
{\\?*a*}	sbreak	soft break
{[HARD}	hzone	start hard break zone
{[SOFT}	szone	start soft break zone
{[LIN}	lzone	start linear break zone
{]}	ezone	end break zone

Table 6.1: Formats

6.1.2.1 Slots

The *id* in a slot format is the name of the slot. The slot corresponds to the parameter, variable or subterm of the *term* on the right-hand side of a definition that has the same name. *ph* is place-holder text. This text enclosed within []'s appears in the slot whenever the slot is uninstantiated in some instance of the definition.

6.1.3 Attributes

Definition attributes are summarized in Table 6.2.

Display	Name	Description
$(c_1,\ldots,c_n)$	conds	conditions
EdAlias(*a*)	alias	alias for definition input
#Hd(*a*)	ithd	head of iteration family
#Tl(*a*)	ittl	tail of iteration family
Parens	parens	parenthesis control
Prec(*a*)	prec	precedence

Table 6.2: Attributes

As with the format table, the 'Name' column gives the names by which you can refer to the attributes when entering them.

Conditions provide extra information about a definition to the editor. The argument of the `conds` term is a sequence of *conditions*. Each *condition* is a term with a single text slot holding the name of the condition. Use the INITIALIZE command (⟨CM-I⟩) with a term cursor over a condition sequence slot to insert a condition term.

The alias attribute provides an alternate name which the input editor recognises as refering to the definition. Alternate names are often convenient abbreviations for the full names of definitions.

The iteration attributes control selection of a definition by the display layout algorithm. They are used to come up with convenient notations for iterated structures. They are discussed in Section 6.4.

The `parens` and `prec` attributes both affect parenthesization. See Section 6.3.

The display form that you get for a display form definition when you first open up a display object assumes there are no attributes, and hides the attribute slot. To open up the attribute slot of a display form definition that hides the slot, position a term cursor over the whole definition and use the SELECT-DFORM-OPTION (⟨CM-`S`⟩) command.

6.1.4 Right-hand-side terms

The right-hand-side term is a pattern. A definition applies to some term t if t is an instance of the rhs term. The display definition matcher has a notion of *meta-variable* different from that of Nuprl's usual matching routines; it has 3 kinds of meta-variable: meta-parameters meta-bound-variables and meta-terms[1]. Meta-parameters and meta-bound-variables correspond to text slots on the left-hand side of a definition, and meta-terms correspond to term slots.

The rhs term is restricted to being a term whose subterms are either constant terms (terms with no meta-variables) or meta-terms. To enter a meta-term use the name **`mterm`**. To make meta-parameters or meta-bound-variables, position a text cursor in the appropriate parameter or bound variable slot and give the CYCLE-META-STATUS (⟨C-`M`⟩) command. Display-meta-variables are redily recognized because they have `<>` as delimiters.

The rhs term can contain normal parameters, bound variables and variable terms. These must match exactly for a definition to be applicable.

6.2 Whitespace

6.2.1 Margin Control

The margin control format `{`$\rightarrow i$`}`(`pushm`) where $i \geq 0$ pushes a new left margin i characters to the right of the format position onto the *margin stack*. The layout algorithm uses the top of the margin stack to decide the column to start laying out at after a line break.

The margin control format `{`$\leftarrow$`}` (`popm`) pops the current margin off the top of the margin stack and restores the left margin to a previous margin.

Usually display forms should have matching `pushm`'s and `popm`'s.

6.2.2 Line Breaking

Line-breaking formats divide the display into nested *break zones*. There are 3 kinds of break zone: *hard*, *linear*, and *soft*. The effect of `{\\`a`}` (`break`) formats depends on the break zone kind:

- In a *hard* zone, `{\\`a`}` always causes a line break.
- In a *soft* zone, either none or all of the `{\\`a`}` are taken.

[1]The meta-parameters are different from those used in abstraction definitions. To be clear, we sometimes call those ones *abstraction-meta-variables* and the ones in display definitions, *display-meta-variables*.

- In a *linear* zone, {\\a} never causes a line break. Instead, its position is filled by the text string a.

The zones are started and ended by zone delimiters. There is one end delimiter {]} (ezone) for all kinds of zones. Each kind of zone has its own start delimiter:

- {[HARD} (hzone) starts a hard zone.
- {[SOFT} (szone) starts a soft zone.
- {[LIN} (lzone) starts a linear zone.

A linear zone is special in that all zones nested inside are also forced to be linear. Therefore a linear zone contains no line-breaks and always is laid out on one line. If a linear zone doesn't fit on a single line, the layout algorithm chooses subterms to elide to try and make it fit.

When laying out a soft zone, the layout algorithm first tries treating it as a linear zone. If that results in any elision, then it treats the zone as a hard zone.

The soft break format {\\?a} sbreak is similar to the break format but is not as sensitive to the zone kind. Soft breaks in linear zones are never taken, but otherwise, the layout algorithm uses a separate procedure to choose which soft breaks to take and which not. This procedure uses various heuristics to try and layout a term sensibly in a given size window with at little elision of subterms as possible.

Display form format sequences should usually include matching start and end zone formats.

6.2.3 Optional Spaces

The {space} (space) format inserts a single blank character if the character before it isn't already a space. Otherwise it has no effect.

6.3 Parenthesization

Automatic parenthesization is controlled by certain display definition attributes, term slot options, and by definition *precedences*. A *precedence* is an element in the *precedence order*. The order is determined by the precedence objects in the Nuprl library. A definition is assigned a precedence by giving it a prec attribute which names some precedence element.

6.3.1 Precedence Objects

Precedence objects collectively introduce a set of precedence elements, and define a partial order on them.

Table 6.3 shows the components of a *precedence* object, and the names used to enter them by. The par, ser, and eq terms are sequence constructors so the standard sequence commands work on the sequences built with these terms.

Each display form not explicitly associated with any precedence element is implicitly associated with a unique precedence element unrelated to all other precedence elements. The uniqueness implies that two such display forms have unrelated precedence.

The core_1 theory should be consulted to see how a base set of precedences has been set up for the current Nuprl theories.

Display	Name	Description
`[`p_1`|...|`p_n`]`	`prpar`	parallel prec term
`(`p_1`>...>`p_n`)`	`prser`	serial prec term
`{`p_1`=...=`p_n`}`	`preq`	equal prec term
obname	`prel`	element of precedence order
`*`*obname*`*`	`prptr`	precedence object pointer

Table 6.3: Precedence Object Elements

6.3.2 Parenthesis Selection

The parenthesization of a term slot of a display form is controlled by the parenthesis slot-option of the term slot in the display form definition (the `L`, `E`, or `*` in the 3rd field), by the `parens` attribute of the display form filling the term slot, and the relative precedences of the term slot itself and the filling term. The precedence of a term slot is usually that of the display form containing it, although it is possible to assign precedences to individual slots. The parenthesis control works as follows:

- It is only possible to parenthesize the term slot if the filling display form has a `parens` attribute. If this attribute is absent, the slot is never parenthesized. Therefore the `parens` attribute must be explicitly added to a display form definition for that definition to ever be parenthesized.
- The parenthesis slot-option controls how precedence affects parenthesization. The parenthesis slot-options have the following meanings:

  L Suppress parentheses if display-form precedence is *less than* child display-form precedence.

  E Suppress parentheses if display-form precedence is *less than or equal to* child display-form precedence.

  * Always suppress parentheses.

Note that the `L` and `E` options give the behavior you might expect; if they are used in the definitions of infix display forms for the arithmetic terms `plus(`a`;`b`)`, and `times(`a`;`b`)`, then `plus(`a`;times(`b`;`c`))` is displayed as $a + b * c$, but `times(`a`;plus(`b`;`c`))` is displayed as $a * (b + c)$. With the `L` and the `E` options you can set up an infix term as being either right or left associative.

The `L`, `E` and `*` characters in the display of term slot formats are display forms for parenthesization control terms. These terms can be entered using the names shown in Table 6.4. The

Display	Name	Description
`L`	`lparens`	L option
`E`	`eparens`	E option
`*`	`sparens`	* option

Table 6.4: Slot Options for Parenthesization Control

parenthesization control terms also allow the specification of the delimiter characters used for parenthesization, and a precedence for the individual slot. No specific editor support has yet been provided for these features.

6.4 Iteration

The iteration attributes control choice of display form definition based on immediately-nested occurrences of the same term. The idea is to group occurrences into *iteration families*. An iteration family has a *head* display form definition and one or more tail definitions. A tail definition can only be used as an immediate subterm of a head in the same family or another tail in the same family. Choice of display form is also affected by the use of the *iterate* variable **#** as the id of a term slot format. If **#** is used in some term slot of a definition, then the definition is only usable if the same term occurs in the subterm slot that uses the **#**.

An example should make this clearer. Say we want a set of display forms for λ abstraction terms such that the λ character is suppressed on nested occurrences. The following definitions would work:

```
λ<x:var>.<t:term:E>== lambda(<x>.<t>)
;; #Hd A ::λ<x:var>,<#:term:E>== lambda(<x>.<#>) ;;
#Tl A ::<x:var>.<t:term:E>== lambda(<x>.<t>) ;;
#Tl A ::<x:var>,<#:term:E>== lambda(<x>.<#>) ;;
```

Using these the term `lambda(x.lambda(y.lambda(z.x)))` would be displayed as:

λx,y,z.x

6.5 Examples

We walk through entry of a display form for the term $\exists ! x{:}T.P_x$.

Start by creating a new display form object and viewing it. Enter in the ML top loop:

`create_disp "test_df" "+scratch"`⟨S-RETURN⟩
`view "test_df"`⟨S-RETURN⟩

where `+scratch` is some suitable position in your library. The window initially looks like:

```
EDIT DISP test_df
== [rhs]
```

Click MOUSE-LEFT on the first `=`, to get a text cursor in the empty format sequence on the left-hand side of the definition. Enter the initial text and a slot for the variable:

⟨C-`#`⟩`163!`⟨C-`0`⟩`slot` RETURN `x` RETURN `var`⟨C-`F`⟩

The definition should now look like:

```
∃!<x:var>|== [rhs]
```

Enter the type slot and the second term slot:

`:`⟨C-`0`⟩`sslot` RETURN `T` RETURN `type`⟨C-`F`⟩⟨C-`F`⟩⟨C-`F`⟩
`.` ⟨C-`0`⟩`eslot` RETURN `P` RETURN `prop`

The definition should now look like:

```
∃!<x:var>:<T:type:*>.  <P:prop:E>== [rhs]
```

Now enter the right-hand side of the display form. Click MOUSE-LEFT on the `[rhs]` placeholder, and enter `exists_unique(`T`.`x`.`P`)` as an exploded term. See Section 4.5.2 for details on how to do this. Do not fill in the variable slot or either of the subterm slots. The definition should now look like:

```
∃!<x:var>:<T:type:*>.  <P:prop:E>== exists_unique{}([term];[binding].[term])
```

Click MOUSE-LEFT on the left-most term slot and to enter the meta terms and meta variable, key:

> `mterm` RETURN `T` RETURN `x` ⟨C-M⟩ RETURN
> `mterm` RETURN `P`

The definition is now complete. It should look like:

```
∃!<x:var>:<T:type:*>.  <P:prop:E>== exists_unique{}(<T>;<x>.<P>)
```

This definition includes no linebreaking or parenthesization information. The display form has an *open* right-hand side, in that there is nothing delimiting the end of the `prop` slot. We therefore want the layout algorithm to automatically parenthesize the display form. To add parenthesizing attributes, click MOUSE-LEFT on the second `=` character, to get a term cursor over the whole definition, and then enter:

> ⟨CM-S⟩ RETURN ⟨C-O⟩

to get two empty attribute slots, with a term cursor over the first:

```
[attr]::[attr]::∃!<x:var>:<T:type:*>.  <P:prop:E>== exists_unique{}(<T>;<x>.<P>)
```

To instantiate the attribute slots enter:

> `parens` RETURN `prec` RETURN `exists`

To get:

```
Parens::Prec(exists)::∃!<x:var>:<T:type:*>.  <P:prop:E>== exists_unique{}(<T>;<x>.<P>)
```

Here, we assign the term the same precedence to $\exists!x{:}T.P_x$ as is assigned in the standard libraries to the $\exists x{:}T.P_x$ term.

We illustrate adding extra formats, by adding a soft-break format such that the ⌜`.`⌝ separating the `type` slot from the `prop` slot is only included if the break is not taken. Click MOUSE-LEFT on the ⌜`.`⌝ character and delete it using ⟨C-D⟩. Enter:

> ⟨C-O⟩`sbreak` SPACE

click MOUSE-LEFT on the `}`after the `?` character in the soft break display form, and enter ⌜`.`⌝.

6.6 The Layout Algorithm

Describe layout algorithm. and selection of dfs.

- *How term matching options affect selection.*
- *How whitespace considerations affect selection (if at all...).*
- *Display form iteration*

Chapter 7

Sequents and Proofs

7.1 Introduction

Nuprl's type theory is formulated in a sequent calculus. The structure of sequents is described in Section 7.2 and of proofs in Section 7.3.

Both structures are defined in Lisp and are accessible from ML. For convenientce, we use term-like notation to describe them, although they are not implemented or edited as terms. Perhaps they will be at some stage in the future.

7.2 Sequent Structure

We write a sequent as

$$H_1, \ldots, H_n \vdash C$$

where C is the conclusion of the sequent, and , the ith hypothesis H_i is either an assumption A_i or a type declaration $x_i : T_i$, and $n \geq 0$. A type declaration $x_i : T_i$ is considered to bind free occurrences of x in terms to the right; that is in $H_{i+1}, \ldots, H_n$ and C. Sometimes we refer collectively to the hypotheses and the conclusion of the sequent as *sequent clauses* or just *clauses*. In the older Nuprl literature, `>>` instead of the turnstile symbol $\vdash$ is used to seperate the hypothesis list from the conclusion. The word *goal* is sometimes used either to refer to a whole sequent or to just the conclusion. Which should be clear from context.

Usually Nuprl displays sequents vertically and explicitly numbers the hypotheses, so the sequent $H_1, \ldots, H_n \vdash C$ is displayed as:

$$\begin{array}{l} 1.\ H_1 \\ \vdots \\ n.\ H_n \\ \vdash C \end{array}$$

A sequent can be considered as either a *conjecture* or a *proved truth*. As a *conjecture* one understands the sequent as expressing the as yet unproved conjecture that the conclusion of the sequent is deducible from the assumptions and declarations of the sequent. As a *proved truth*, one understands the sequent as expressing that that conclusion of the sequent has been proved true, given the assumptions and declarations of the sequent.

There are a few details left out of the above account that we now describe.

1. Logic is encoded into Nuprl's type theory using the propositions-as-types analogy, so all clauses of sequents are really types. Clauses are made to appear like propositions by using abstractions. All hypotheses declare variables, but the system currently hides the display of any variable whose name starts with a `%` character. We sometimes refer to such variables *invisible* variables. When sketching sequents in this document, we suppress variables that would normally be invisible.

2. Hypotheses can be *hidden*. Hidden hypotheses are displayed with `[]`'s around the hypothesis's type or assumption term. For a discussion of hypothesis hiding, see Section 9.12.

3. Hypothesis variable names have to be distinct.

7.3 Proof Structure

Proofs are tree structures. Using term notation, the class of *proofs* is the least set of terms such that

- `unrefined`(g) is a proof.
- if $p_1, \ldots, p_n$ are proofs, $n \geq 0$, then `refined`($g; r; p_1; \ldots; p_n$) is a proof,

where:

- g is a sequent.
- r is a refinement rule. See Section 7.4

The sequent g in the proof `refined`($g; r; p_1; \ldots; p_n$) or `unrefined`(g) is referred to as the *root goal*, or simply the *goal* of the proof. Similarly, the goals of the proofs $p_1; \ldots; p_n$ are referred to as the *subgoals* of the proof `refined`($g; r; p_1; \ldots; p_n$).

A proof is *good* when is satisfies various conditions, including

1. every sequent in the proof is closed; every free variable of a sequent clause is bound by some declaration of the sequent,

2. at every refined node of the proof tree, the rule proves the goal sequent, assuming the provability of the subgoal sequents.

A proof is *complete* exactly when it is good and contains no unrefined nodes. A proof is *incomplete* if it is good but does contain unrefined nodes.

Each theorem object in Nuprl's library contains one proof. The root goal of this proof is sometimes referred to as the *main goal* of the theorem. It always has no hypotheses.

7.4 Refinement Rules

7.4.1 Primitive Refinement Rules

The primitive refinement rules are all introduced by rule objects. The current system has primitive rules for a constructive type-theory, closely related to Martin-Löf type-theory. All proofs in the Nuprl system are eventually justified by these primitive rules. More precisely, the correctness of every Nuprl proof depends only on the correctness of these rules, and of Nuprl's *refiner*. The *refiner* is a fixed piece of Lisp code which applies primitive rules to unrefined leaves of proofs. Users rarely invoke primitive rules directly; they are at too low a level, and one has to understand how logic is coded within type-theory. Almost always, tactics are used instead.

7.4.2 Tactic Rules

As explained in detail in Chapter 9, tactics are ML functions which enable one to automate application of primitive rules. A simplified but conceptually useful idea of a tactic, is as a function mapping proofs to proofs. If one applies a tactic in ML to an unrefined proof and the tactic doesn't fail, then the tactic returns a proof built (usually) from primitive rules with 0 or more unrefined leaves.

We give a description of what a *tactic rule* is, and what happens when a tactic is executed as a refinement rule. Assume that a proof editor window is viewing some proof node `unrefined`(g) and that one enters *TacticText*, the text of some suitable tactic as the refinement rule.

1. *TacticText* is parsed by the ML parser into a tactic, and is applied to the proof node `unrefined`(g). Let the resulting proof term be p. Note that the root goal of p is always the same as g.

2. p is *not* simply inserted back into the proof tree, replacing `unrefined`(g). Rather it is stored in a *tactic rule* along with the ML text of the tactic. Let us represent the tactic rule by the term `tactic_rule`(*TacticText* ; p).

3. What *is* inserted back into the proof tree to replace `unrefined`(g) is

   `refined`(g ; `tactic_rule`(*TacticText* ; p) ; p_1 ; $\ldots$; p_n).

   Here, $n \geq 0$, and $p_1 ; \ldots ; p_n$ are all the unrefined leaf nodes of the proof p in the same left-right order as they occur in p.

The tactic rule hides the proof tree p. When one views a proof term, or a tactic rule refinement, one only ever sees the text of the tactic. From a logical point of view, it is not strictly necessary to keep p around at all, after the tactic has executed. However, it is necessary for *extraction* purposes.

In the event that a tactic is applied as a refinement rule to an already refined proof term, the proof term is first changed to an unrefined proof, discarding the existing refinement rule and all the sub-proofs, before it is passed to the tactic.

Running a tactic as a refinement rule makes it appear in a proof as a high level rule of inference, and consequently greatly increases the readability of proofs.

The *TacticText* is represented as an `!ml_text` alternating sequence and has structure identical to that of ML library objects.

7.4.3 Reflection Rules

7.5 Transformation Tactics

Transformation tactics have the same type as normal tactics. However, they can be run on any node of a proof, not just leaf nodes. Examples of transformation tactics can be found in Section 9.11.

When a transformation tactic is run on a proof p, the proof editor replaces p with the proof resulting from the tactic; it doesn't create a special proof node that just has the unproven subgoals of the resulting proof as its immediate subgoals. Nor is the text of the transformation tactic saved anywhere.

7.6 Proof Editor

The proof editor is designed principally to support the 'top-down' *refinement* style generation of proofs. The refinement style entails repeatedly choosing an unrefined leaf node of a proof and a rule (usually a tactic) to try on that node. If the rule applies, the Nuprl system changes the node to a refined node, and automatically generates appropriate children nodes.

The editor also supports the application of *transformation tactics* to proofs. These are usually applied to already refined nodes of a proof tree and either change the structure of the proof they are applied to or have some side effect. Transformation tactics are described in Section 7.5.

The proof editor generates windows onto sections of proofs. One can have windows open on different proofs at the same time, and even multiple windows onto the same proof. In the latter event, the windows become 'read-only'.

Proofs associated with theorem objects are not first copied when they are viewed with the proof editor, so all changes made to proofs take effect immediately. This is in contrast to the situation with the term editor where changes are only committed when you exit an object or ask for changes to be explicitly saved. If you want to make tentative changes to a section of a proof, you can use the `Mark` transformation tactic to first make a good copy of that section, or you can make a copy of the whole theorem object.

7.6.1 Proof Window Format

Each proof window is associated with a node of a proof. It shows the goal sequent at that node, the refinement rule if any at that node and any immediate subgoals.
Figure 7.1 shows an example of a window onto a refined node of a proof, and Figure 7.2 shows an example of a window onto an unrefined node of a proof.

The numbered parts of these windows are as follows:

① The `EDIT` indicates that the proof is being viewed in *edit* mode. In this mode the proof can be changed. This is replaced by `SHOW` if the proof is viewed in the read-only mode. The `THM` indicates that a theorem object is being viewed, and `cantor` is the name of the theorem.

② The `#` indicates that this proof node is considered incomplete. Other symbols used here, are `*` for complete, and `-` for bad. the `top 1 1` and the `top 1 1 2` are tree addresses of the nodes being viewed. Figure 7.2 shows the 1st child of the 1st child of the root of the proof, and Figure 7.1 shows the 2nd child of the proof node in Figure 7.2.

```
①EDIT THM cantor
②# top 1 1
③1.  f:  ℕ → ℕ → ℕ
 2.  ∀g:ℕ → ℕ.  ∃i:ℕ.  f i = g ∈ ℕ → ℕ
 ⊢ False

④BY With 'λn.f n n + 1' (D 2) THENW Auto

⑥1* 2.  n:  ℕ
    ⊢ 0 ≤ f n n + 1

⑥2# 2.  ∃i:ℕ.  f i = λn.f n n + 1 ∈ ℕ → ℕ
 ⊢ False
```

Figure 7.1: Proof Window on Refined Proof Node

```
①EDIT THM cantor
②# top 1 1 2
③1.  f:  ℕ → ℕ → ℕ
 2.  ∃i:ℕ.  f i = λn.f n n + 1 ∈ ℕ → ℕ
 ⊢ False

⑤BY <refinement rule>
```

Figure 7.2: Proof Window on Unrefined Proof Node

③ This is the goal sequent of the proof node.

④ This is a tactic which was executed on the goal ③ above in order to generate the subgoals ⑥below. The BY is part of the proof node display, and is not part of the tactic.

⑤ This is the refinement rule placeholder.

⑥ These are the subgoals of the proof node. Each one is numbered. The * or # by the subgoal number shows the status of the subproof. The symbols are the same as those for the goal. Note that for brevity, only hypotheses which have changed or been added are displayed in the subgoal sequents.

Sometimes the proof window is too short to display all the goal, rule, and subgoals. In this case the cursor motion commands described in section ??? will automatically scroll the window. One can of course also resize the window.

7.6.2 Proof Motion Commands

⟨M-B⟩	BACK-PART	move back part.sibling to immediate left
⟨M-F⟩	FORWARD-PART	move to sibling to immediate right
⟨M-A⟩	FIRST-PART	move to left-most sibling
⟨M-E⟩	LAST-PART	move to right-most sibling
⟨M-P⟩	UP-TO-PARENT	move up to previous level of proof
⟨M-<⟩	UP-TO-TOP	move up to top of proof
⟨M-N⟩	DOWN-TO-CHILD	move down to next level of proof
RETURN	NEXT-UNREFINED-LEAF	jump to next unrefined node

Table 7.1: Proof Motion Commands

The keyboard commands for moving about proofs are summarized in Table 7.1. The commands closely match a subset of the term editor motion commands (described in Section 4.4.2). A *part* of the window is a either a goal sequent, a refinement rule, a rule placeholder or a subgoal.

The FORWARD-PART and BACK-PART commands move the cursor within a proof window from part to part, if necessary scrolling the window. FIRST-PART moves the cursor to the goal and LAST-PART moves the cursor to the last subgoal if there are any subgoals; otherwise it moves the cursor to the refinement rule part.

The UP-TO-PARENT command, executed with a cursor in any part of the window, shifts the window one level up the proof tree. DOWN-TO-CHILD, executed with the cursor over a subgoal part, shifts the window down the proof tree to that subgoal. The NEXT-UNREFINED-LEAF command shifts the window to the next unrefined proof node in a preorder traversal of the proof tree. If there are none, NEXT-UNREFINED-LEAF shifts the window to the root of the proof.

The mouse can also be used to move about a proof. See Section 7.6.4 for details. Most users find these easier to use than the key bindings.

7.6.3 Opening, Closing, and Changing Windows

⟨C-Z⟩	EXIT-PROOF	close proof window
⟨C-J⟩	JUMP-NEXT-WINDOW	jump to next window
TAB	JUMP-ML	jump to ML top-loop
⟨C-S⟩	SELECT	open term window onto rule or sequent
⟨C-T⟩	TRANSFORM	open term window for transformation tactic

Table 7.2: Commands For Opening and Closing and Changing Windows

The relevant proof editor commands are shown in Table 7.2, and described below. the *select* and TRANSFORM commands open up term editor windows. You can edit ML in these windows in the same way you would edit ML in an ML object or in the ML Top Loop.

7.6.3.1 Opening a Proof Window

A proof editor window is opened onto a proof in a theorem object whenever the ML `view` function is applied to the object's name in the ML top-loop. If `view` is used on an theorem object with an compressed proof, expansion of the proof is forced. This may take some time, especially if the proof is large. Section 7.7 describes proof compression and expansion.

7.6.3.2 Closing a Proof Window

To close a proof window, use EXIT PROOF.

7.6.3.3 Changing Windows

JUMP-NEXT-WINDOW cycles the cursor through all the open proof and term windows, except the ML-top-loop window. JUMP-ML moves the Nuprl cursor to the ML top-loop.

7.6.3.4 Editing The Main Goal

```
EDIT THM cantor
? top   <main proof goal>
```

Figure 7.3: A Proof Window on a New Proof

When a new proof window is opened, the window appears as in Figure 7.3. By using SELECT with the cursor over `<main proof goal>` a term window is opened up to allow one to enter the main goal of the proof.

One can also use SELECT on the main goals of incomplete or complete proofs. For example, one might want to copy the main goal of one theorem and use it as the basis for the main goal of another, or one might want to correct a mis-stated main goal. Note however that if a main goal is destructively modified *and* checked, then any existing proof is lost. As explained in Section 4.4.7, the window is checked whenever the EXIT term window command is used. If the window is not modified but checked, or modified and then quitted, then any existing proof will not be changed. Warning: a window counts as being modified even if changes have been made and then undone, so it *looks* the same as it originally was.

7.6.3.5 Editing a Refinement Rule

A refinement rule window is opened whenever the SELECT COMMAND is used with the proof cursor over the rule place-holder. (For example see Figure 7.2) The window always has the title `EDIT rule of` *theorem* where *theorem* is the name of the library object holding the proof. A new refinement rule window is always initialized to allow one to type in an ML tactic. The structure and special editing commands for this term window are the same as for ML objects. Soon, it will be possible to initialize the refinement rule window to hold other kinds of rules (For example a primitive rule). We will describe special term editor support for these options here. Most users will only use tactics in refinement rule windows. After a rule has been keyed in, and the term window EXIT command has been given, Nuprl parses the rule and tries to apply it to the goal of the current proof. If the rule succeeds the proof window is redrawn with new statuses and subgoals as necessary. If it fails then one of two things may happen. If the error is severe, the status of the node (and the proof) will be set to *bad*, an error message will appear in the command/status window, and the rule will be set to `??bad refinement rule??`. If the error is mild and due to a missing input, Nuprl will display some diagnostic message and leave the rule window on the screen so that it can be fixed.

One can also use SELECT on existing refinement rules. For example, one might want to copy one rule in order to use it as the basis of another or one might want to change the rule. If a refinement rule is destructively modified *and* checked, then any existing subproofs below the rule are lost. As explained in Section 4.4.7, the window is checked whenever the EXIT term window command is used. If the window is not modified but checked, or modified and then quitted, then any existing proof will not be changed.

7.6.3.6 Viewing Subgoal Sequents

If SELECT is invoked on any sequent of a proof but the main goal, a read-only term window onto that subgoal is generated. This is useful, for example, if one wants to use a term from a sequent as an argument to a tactic, and one doesn't want to have to retype in the term.

Should describe here special editor support for subgoal sequents. e.g. what are represented as alternating lists???

7.6.3.7 Editing a Transformation Tactic

To invoke a transformation tactic at some node of a proof, position a proof editor window at that node and use the TRANSFORM command. This opens up a transformation-tactic window and initializes it to take tactic text. Type the name of the tactic and any arguments into the window and then use the EXIT term editor command. Nuprl will apply the tactic and redisplay the proof window to show any effects. If the expression entered doesn't parse or typecheck, a diagnostic message is printed and the window is left as is. If the tranformation tactic fails, the proof is left unchanged.

$\boxed{\text{MOUSE-RIGHT}}$	MOUSE-SELECT	select main goal, rule or sequent
$\boxed{\text{MOUSE-LEFT}}$	MOUSE-JUMP	jump to window / parent / child

Table 7.3: Mouse Commands for Proof Windows

7.6.4 Mouse Commands

The mouse commands are shown in Table 7.3 The mouse can be used for shifting a proof window about a proof, jumping between different windows, and selecting the main goal, rules, and sequents displayed in a proof window.

7.7 Proof Compression and Expansion

7.7.1 Introduction

When a theory is dumped to a file, proofs are stored in a compressed format. This format retains only the main goal, the text of tactics in tactic rules, and the text of primitive rules not buried inside tactic rules. All other sequents, and all subproofs associated with tactic rules, are discarded. Thus the dumped representation contains essentially just the text that a user would type to reconstruct the proof.

This format contains just enough information to regenerate the full proof data-structure. When a theory is loaded, the loaded proofs are retained in their compressed format. When a proof is needed, as when it is to be viewed, checked, or extracted from, then the system will reconstruct the usual proof tree.

The reconstruction can be time consuming since all the tactics used to construct the proof must be re-executed. Also, if the tactics that were used to construct the proofs have since been modified, the reconstruction may fail.

When a theorem object is extracted from, the extraction is stored with the theorem in the library. When a theorem object is dumped to a file, if it has an extraction, then the extraction is also dumped. This feature can sometimes reduce the need for proof expansion.

The proof-script of a theorem is updated whenever a complete proof of the theorem is built. Note however that it is not updated if expansion of a previously complete theorem results in an incomplete theorem. This is to give the user a chance to fix the proof script. The proof-script of a theorem is also updated when a theorem is dumped to a file, and when the proof is compressed.

7.7.2 Editing Proof Scripts

Editing facilities are provided for proof-scripts, as are tactics for explicitly executing a proof-script on a node of a proof. These are particularly of use when fixing proof-scripts that have broken due to changes in tactics or the library context.

Proof scripts are Lisp data structures. They are made editable by translating to and from 'proof-script terms' or 'ps-terms' for short. An example display of a ps-term is:

```
TACTIC  top :
  (Unfold ‘not‘ 0
   THENM D 0
   THENM InstConcl [⌜λn.f n n + 1⌝] ...’)
SUBTREES
  TACTIC  top 1 :
    (With ⌜i⌝ (EqHD 3) THENM Reduce 3 ...a)
  SUBTREES
    TACTIC  top 1 1 :
      Auto
    SUBTREES
      <no subtrees>
    END
  END
END
```

Figure 7.4 summarizes the structure of ps-terms.

node name		alternatives / structure	*alias*
pscript-node	`::=`	`TACTIC \\` *tactic* `\\ SUBTREES` *subtree** `END`	`psnode`
	`|`	`TACTIC` *addr* `: \\` *tactic* `\\ SUBTREES` *subtree** `END`	
*subtree**	`::=`	*subtree* `\\` ... `\\` *subtree* `| <no subtrees>`	
subtree	`::=`	*pscript-node*	

Figure 7.4: Proof-Script Terms

The ‘alias’ column gives the name by which proof-script nodes can be entered. Proof-script subtrees are considered to be a term sequence, and the usual term sequence editing commands work with them. The *addr* fields in ps-terms are for tree addresses in the same format as used in proofs. They serve solely as a guide the user; they have no logical significance. Users should never need to explicitly fill these addresses in. Instead, the addresses are generated by ML functions that build proof script terms.

Useful ML functions include:

`psterm_of_thm_object` *nam*`:tok =` *psterm*`:term`

Takes the name *nam* of a theorem in the library and returns its proof-script term *psterm*. In doing so, it appropriately fills in the address fields of the proof-script term.

`psterm_anno` *psterm*`:term =` *psterm'*`:term`

Adds address annotations to a proof-script term, ignoring any previous annotations.

`RunPSTerm` *psterm*`:term =` *T*`:tactic`

A transformation tactic for executing proof-script terms.

Chapter 8

Rule Interpreter

8.1 Term Structure of Rules

Rule definitions are expressed as 'terms' in the sense described in Chapter 4. They are normally stored in library objects of kind *rule*.

Figure 8.1 shows the basic tree structure of rule terms. I have included kinds of 'virtual tree nodes' in this description. These virtual nodes do not correspond to term constructors used in building up rules, but they do help in explaining the structure of rule terms.

Italic type is used in Figure 8.1 for kinds of tree nodes that correspond to terms and term slots. Roman type is used for tree nodes that correspond to text strings and text slots. `\\` indicates a linebreak in a display form, `\null` indicates a display form with 'zero width', and ␣ is the usual invisible space. The suffix * character on a node-kind names indicate a sequence of nodes. The *prl-term* node kind is for terms in Nuprl's object language.

Names for the term constructors that are suitable for entering the constructors are shown in the 'alias' column. The term constructors for sequences do not need to be entered explicitly by name. The editor recognizes the context of each sequence, and sequence items can be added and deleted using the ⟨C-O⟩ and ⟨C-C⟩ sequence commands.

For rule terms to be well-formed, there are several extra constraints on their structure. These include:

- *substitution* terms can only occur in
  - the *concl* or *hyp-item* term of a *subgoal*,
  - the right-hand subterm of a *matching-constraint*,
  - *extract* term of a *goal*.
- there should be a hyp-index term as a rule *arg* for each *hyp-list hyp-item* that is followed by a *hyp hyp-item* in the *goal* of the rule definition.
- Adjacent *hyp-items* should not be both *variables*.
- The *extract* of a *subgoal* (if it exists) should always be a *variable*.

8.2 Semantics of Rule Interpreter

<table>
<tr><th>node name</th><th></th><th>alternatives / structure</th><th>physical node name</th><th>alias</th></tr>
<tr><td>rule-def</td><td>::=</td><td>goal \\BY rule \\ subgoal*</td><td>simple-rule</td><td>rldef</td></tr>
<tr><td></td><td>|</td><td>goal \\ BY rule \\ constraint \\ subgoal*</td><td>constrained-rule</td><td>crldef</td></tr>
<tr><td>goal</td><td>::=</td><td>sequent</td><td></td><td></td></tr>
<tr><td>subgoal*</td><td>::=</td><td>subgoal \\ ... \\ subgoal | No Subgoals</td><td></td><td></td></tr>
<tr><td>subgoal</td><td>::=</td><td>sequent</td><td></td><td></td></tr>
<tr><td>sequent</td><td>::=</td><td>hyp-item* ⊢ concl</td><td>no-ext-sequent</td><td>seq</td></tr>
<tr><td></td><td>|</td><td>hyp-item* ⊢ concl ext extract</td><td>ext-sequent</td><td>eseq</td></tr>
<tr><td>hyp-item*</td><td>::=</td><td>hyp-item , ... , hyp-item | \null</td><td></td><td></td></tr>
<tr><td>hyp-item</td><td>::=</td><td>hyp | hyp-list</td><td></td><td></td></tr>
<tr><td>hyp</td><td>::=</td><td>variable : subst-term</td><td>normal-declaration</td><td>decl</td></tr>
<tr><td></td><td>|</td><td>[variable : subst-term]</td><td>hidden-declaration</td><td>hdecl</td></tr>
<tr><td>hyp-list</td><td>::=</td><td>variable | substitution</td><td></td><td></td></tr>
<tr><td>concl</td><td>::=</td><td>subst-term</td><td></td><td></td></tr>
<tr><td>extract</td><td>::=</td><td>subst-term</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>constraint</td><td>::=</td><td>Let term = subst-term</td><td>matching-let</td><td>mlet</td></tr>
<tr><td></td><td>|</td><td>Let arg* = Call{lisp-fun-name}</td><td>lisp-let</td><td>llet</td></tr>
<tr><td>rule</td><td>::=</td><td>rule-name arg*</td><td></td><td>rule</td></tr>
<tr><td>arg*</td><td>::=</td><td>arg ␣ ... ␣ arg | ()</td><td></td><td></td></tr>
<tr><td>arg</td><td>::=</td><td>variable</td><td></td><td></td></tr>
<tr><td></td><td>|</td><td>nat-number</td><td></td><td></td></tr>
<tr><td></td><td>|</td><td>#</td><td>hyp-index</td><td>hypi</td></tr>
<tr><td></td><td>|</td><td>#hyp-num</td><td>hyp-index-n</td><td>hypind</td></tr>
<tr><td></td><td>|</td><td>level{level}</td><td>level-exp</td><td>level</td></tr>
<tr><td></td><td>|</td><td>parm-sub{parm-sub}</td><td>parm-substitution</td><td>parmsub</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>subst-term</td><td>::=</td><td>prl-term | substitution</td><td></td><td></td></tr>
<tr><td>substitution</td><td>::=</td><td>variable[prl-term/variable]</td><td>subst-1</td><td>subst</td></tr>
<tr><td></td><td>|</td><td>variable[prltm,prltm/var,var]</td><td>subst-2</td><td>subst2</td></tr>
<tr><td></td><td>|</td><td>variable[prltm,prltm,prltm/var,var,var]</td><td>subst-3</td><td>subst3</td></tr>
<tr><td></td><td>|</td><td>variable[parm-sub]</td><td>subst-parms</td><td>substp</td></tr>
<tr><td>variable</td><td>::=</td><td>variable</td><td></td><td>var-name</td></tr>
<tr><td>nat-number</td><td>::=</td><td>natural</td><td></td><td>nat-digits</td></tr>
</table>

Figure 8.1: Structure of Rules

Chapter 9

Tactics

9.1 Introduction

9.1.1 Conventions

For brevity, we assume unless otherwise stated that arguments to tactics have the following types and uses:

`T*`	:	`tactic`	
`c*`	:	`int`	*clause index.*
`i*`	:	`int`	*hypothesis index.*
`t*`	:	`term`	*term of Nuprl's type theory*
`n*`	:	`tok`	*name of lemma object in library*
`a*`	:	`tok`	*name of abstraction object in library*
`v*`	:	`var`	*variables in terms of Nuprl's object language*
`l*`	:	`tok`	*subgoal label*
`p*`	:	`proof`	*current goal*

An `s` suffix on the name of an argument indicates that it is a list. For example `vs` is considered to have type `var list`.

9.1.2 Universes and Level Expressions

In Nuprl's type theory, types are grouped together into universes. Types built from the base types such as $\mathbb{Z}$ or `Atom` using the various type constructors are in universe $\mathbb{U}_1$. The subscript 1 is the *level* of the universe. Types built from universe terms with level at most i, are in universe $\mathbb{U}_{i+1}$. Universe membership is cumulative; each universe also includes all the types in lower universes.

Since propositions are encoded as types, propositions reside in universes too. In keeping with the propositions-as-types encoding, we define a family of propositional universe abstractions $\mathbb{P}_1 \ldots \mathbb{P}_i \ldots$, which unfold to the corresponding primitive type universe terms $\mathbb{U}_1 \ldots \mathbb{U}_i \ldots$.

If one is only allowed to use constant levels for universes, one often has to choose arbitrarily levels for theorems. One would then find that one needed theorems which were stated at a higher level, and would have to reprove those theorems. This was the case in Nuprl V3.

Nuprl V4 allows one to prove theorems which are implicitly quantified over universe levels. Quantification is achieved by parameterizing universe terms by *level expressions* rather than natural number constants. The syntax of level expressions is given by the grammar:

$$\begin{array}{lll} L & ::= & v \\ & | & k \\ & | & L\ i \\ & | & L' \\ & | & [L|\cdots|L] \end{array}$$

The v are level-expression variables. v can be any alphanumeric string. These variables are implicitly quantified over all positive integer levels. the k are level expression constants. k can be any positive integer. The i are level expression increments. i can be any non-negative integer. The expression $L\ i$ is interpreted as standing for levels $L + i$. L' is an abbreviation for L `1`. The expression $[L_1|\cdots|L_n]$ is interpreted as being the maximum of expressions $L_1 \cdots L_n$.

Usually when stating theorems, only level expressions of the form v and v' need be used. Other expressions get automatically created by tactics. Further, it is normally sufficient to use a single level-expression variable throughout a theorem statement. For example, we normally prove the theorem:

$$\forall A{:}\mathbb{P}_i.\forall B{:}\mathbb{P}_i.A \Rightarrow (B \Rightarrow A)$$

rather than

$$\forall A{:}\mathbb{P}_i.\forall B{:}\mathbb{P}_j.A \Rightarrow (B \Rightarrow A)$$

9.1.3 Formula Structure

Many of Nuprl's tactics work on formulae generated by the grammar

$$\begin{array}{lll} P & ::= & \forall x{:}A.\ P \mid Q \Rightarrow P \mid P \Leftarrow Q \\ & | & P \wedge P \mid P \iff P \\ & | & R \end{array}$$

where A is a type, and R is a propositional term not of the above form. We call these *general universal* formulae or just *universal* formulae. They are sometimes called positive definite formulae or horn clauses. We call the formulae generated by this grammar without the $\wedge$ and $\iff$ connectives, *simple universal* formulae. We call the proposition R, a *consequent* and each Q, an *antecedent.* Occasionally we refer to the types A as *type antecedents.*

We view a general universal formula as being composed of several simple formulae, one for each consequent. The simple components are numbered from 1 up, starting with the leftmost consequent.

Such formulae are the standard way of summarizing derived rules of inference, and are used as such by the forward and backward chaining tactics. Often, a consequent R of a formula will be an equivalence relation, in which case the formula can be used as a rewrite rule by the rewrite package.

Occasionally, one has a universal formulae, where the outermost constructor of R is also one of the constructors which makes up the universal formulae. In this case, one can surround R by a *guard* abstraction. A guard abstraction takes a single subterm as argument and unfolds to this subterm. The tactics which take apart universal formulae recognise and automatically remove guard abstractions, so the user rarely has to explicitly unfold them.

9.1.4 Soft Abstractions

Certain abstractions can be designated as *soft*. Some tactics treat soft abstractions as being transparent — those tactics behave as if all soft abstractions had first been unfolded. In practice, those tactics only unfold soft abstractions when they need to and for the most part are careful not to leave unfolded soft abstractions in the subgoals that they generate.

Specific tactics and functions which unfold soft abstractions are:

- The `MemCD` and `EqCD` tactics. For example, if `MemCD` is run on a sequent with conclusion $\vdash\ t \in T$ where t is soft and no well formedness lemmas exist for t, then it unfolds t.
- The `NthHyp`, `NthDecl`, `Eq` and `Inclusion` tactics unfold soft abstractions in the relevant clauses.
- Most tactics using matching routines treat soft abstractions as transparent. For example the forward and backward chaining tactics, and the atomic rewrite conversions based on lemmas and hyps.

In the basic libraries, the soft abstractions are

`member`	$t \in T$	$=_{def}$	$t = t \in T$
`nequal`	$x \neq y \in T$	$=_{def}$	$\neg(x = y \in T)$
`prop`	$\mathbb{P}_i$	$=_{def}$	$\mathbb{U}_i$
`and`	$A \wedge B$	$=_{def}$	$A \times B$
`or`	$A \vee B$	$=_{def}$	$A + B$
`implies`	$A \Rightarrow B$	$=_{def}$	$A \rightarrow B$
`rev_implies`	$A \Leftarrow B$	$=_{def}$	$B \Rightarrow A$
`iff`	$A \Leftrightarrow B$	$=_{def}$	$(A \Rightarrow B) \wedge (A \Leftarrow B)$
`exists`	$\exists x{:}A.\ B_x$	$=_{def}$	$x{:}A \times B_x$
`all`	$\forall x{:}A.\ B_x$	$=_{def}$	$x{:}A \rightarrow B_x$
`ge`	$i \geq j$	$=_{def}$	$j \leq i$
`gt`	$i > j$	$=_{def}$	$j < i$
`lelt`	$i \leq j < k$	$=_{def}$	$(i \leq j) \wedge (j < k)$
`lele`	$i \leq j \leq k$	$=_{def}$	$(i \leq j) \wedge (j \leq k)$

The logic abstractions (`and`, `or`, `implies`, `exists`, `all`) are made soft because the well formedness rule for the underlying primitive term is simpler and more efficient than the well formedness lemma would be. The softness is also useful when one wishes to blurr the distinction between propositions and types, for example when reasoning explicitly about the inhabitants of propositions. `member`, `nequal`, `rev_implies`, `ge` and `gt` are soft principally because it can simplify matching.

Abstractions are not soft by default. They are declared soft by supplying their opids to the function `add_soft_abs : tok list -> unit`. Instances of this function are usually kept in ML objects in close proximity to the abstraction definitions that they are declaring soft. For an example use of `add_soft_abs`, see the object `soft_ab_decls` in the `core_2` theory.

9.1.5 The Sequent

Sequents are introduced in Section 7.2. We describe here some specific tactic-related details.

Hypotheses are conventionally numbered from left to right, starting from 1. These hypothesis numbers are displayed by the proof editor, and tactics usually refer to hyps by these numbers. Sometimes, it is convenient to consider the hyps numbered from right to left, and for this reason tactics consider a hyp list $H_1, \ldots, H_n$ to also be numbered $H_{-n}, \ldots, H_{-1}$. Occasionally, the index $n+1$ or 0 is used to refer to the hyp position to the right of the last hyp.

There are tactics which work in similar ways on both hyps and the concl. In this case, we call the hyps and concl collectively *clauses*, refer to the concl as *clause 0*, and hyp i, $i \neq 0$ as *clause i*. So far, we have not encountered tactics where we would want to both refer to the position after the last hyp as clause 0 and refer to the conclusion, so this numbering scheme has not caused problems.

When we want to indicate explicitly the number of a hyp in a schematic sequent, we prefix the hyp with the number followed by a period. So for example, if hyp i is proposition P, we write the hyp as $i.\ P$.

Tactics currently use the visibility of the variable as an indication of whether it is ever used in subsequent hyps or the concl. Some tactics working on hyps are more efficient when they work on hyps whose variables are unused. The variables declared in a hypothesis list must all be distinct. Tactics are careful to use invisible variables for new hypotheses that are to be considered assumptions rather than declarations.

9.1.6 Proof Annotations

Nuprl *proof* terms (ML terms of type `proof`) can be annotated with extra information which isn't relevant to the logical correctness of a proof. Nuprl currently supports two kinds of annotations; *goal labels* and *tactic arguments*.

9.1.6.1 Goal Labels

A Nuprl tactic generates various kinds of subgoals, and often subsequent tactics want to discriminate on subgoal kind. Sometimes a subgoal's kind can be deduced directly from its structure, but this can be a error-prone process and so tactics attach explicit labels to subgoals indicating their kind. Labels take the form of an ML token, and an optional number. Examples of labels are `main`, `upcase` and `wf`. Most descriptions of tactics include information on subgoal labelling. It is also a simple matter to find out what labels are generated by experimentation.

The tacticals which discriminate on labels are described in the tacticals section below. For convenience, labels are divided into the the classes *main* and *aux*. The discriminating tacticals allow one to select either subgoals with a particular label, or subgoals of one of the two classes. One selects a class by using one of the class names `main` or `aux`.[1]

Sometimes tactics generate a set of subgoals which are all the same kind, but where the order of the subgoals is important. The number labels are used to discriminate between these subgoals.

Labels used not to be visible when editing proofs with the proof editor and are therefore sometimes known as *hidden labels*.

Label related tactics are:

`AddHiddenLabel lab`

Add hidden label `lab` to the current goal.

[1] `main` is currently used as both a class name, and a particular label name, so there is currently no way to select only subgoals in class `main` with label `main`.

`AddHiddenLabelAndNumber lab i`
: Add hidden label `lab` to the current goal along with the integer label `i`.

`UnhideLabel`
: Make the hidden label on a goal visible. This wraps a special abstraction around the conclusion term of the goal which makes the label visible. Since the 'hidden' labels are usually visible, this tactic is no longer that necessary.

`RemoveLabel`
: Remove a visible label.

See Section 9.3.2 for how to discriminate on labels.

9.1.6.2 Tactic Arguments

Unlike Lisp functions, ML functions cannot take optional arguments, although it is natural to want to write tactics which do take optional arguments. One approach is to provide a set of variants of each tactic for the most common combinations of arguments. This can be confusing, and places an extra burden on the user who has to keep track of these variants. Nuprl V4 allows optional arguments to be passed to tactics by attaching these arguments to the proof argument which all tactics operate on. Currently argument types of `int`, `tactic`, `term`, `tok`, `var` and `(var # term) list` are supported. Each argument is given a token label, and arguments are looked up by these labels. Sets of arguments are maintained on a stack, so nesting of tactics which use optional arguments is possible.

Note that some tactics do useful preprocessing on some of their arguments, and in these cases there would be a performance penalty if such arguments were supplied, annotated to the proof.

Tactic arguments are also used for the *analogy* tactics. See the relevant section below.

Tacticals for manipulating these arguments are:

`With (t:term) T`
: Runs `T` with `t` as a `` `t1` `` argument.

`New ([v1;...;vn] :  var list) T`
: Runs `T` with `v1` to `vn` as arguments `` `v1` `` to `` `vn` ``.

`At (U:term) T`
: Runs `T` with `U` as a `` `universe` `` argument. Term `U` should either be either a type universe or a propositional universe term.

`Using (sub:(var # term) list) T`
: Runs `T` with the substitution `sub` as a `sub` argument.

`Sel (n:int) T`
: Runs `T` with the integer `n` as an `n` argument. Used for selecting a simple component of a universal formula or a subterm of a term.

These tactics are all special cases of:

`WithArgs (args:  (tok # arg) list) T`
: Run `T` with the arguments in `args` on the top of the stack. `arg` is an ML abstract data type, defined as the disjoint union of the types listed above. There exist injection and projection functions for each of the types listed above.

Each tactic description includes information on the optional arguments (if any) that it takes.

9.1.7 Matching and Substitution

Nuprl has complex matching routines, which allow for automatic instantiation of universal formulae in a variety of cases. Given a pattern term P and instance term I, we say that I matches P, if there exists a substitution θ such that $I \; R \; \theta P$. For many purposes, R is $\alpha - \beta$ equality, but on some occasions it is useful to use a slightly weaker R. The weaker R allows level expressions in I and θP to be related by an order relation rather than an equivalence relation. The weaker R also can allow I and θP to differ by the folding or unfolding of soft abstractions.

Since matching is all about guessing substitutions, we describe first the possible kinds of substitution.

1. **First Order Term**
   We replace a variable term free in P by some other term. For example, if $P = x + y$ and $\theta = [x \mapsto 3]$, then $I = 3 + y$. We call such substitution *first order* because in θ, each variable is bound to a first-order term rather than a higher-order term. (See next item).

2. **Second Order Term**
   Second-order terms are a generalization of terms. They can be thought of as 'terms with holes', terms with zero or more subtrees missing. A second-order term can be represented as a pair of a variable list and a first-order term, the first-order term being generated from the second-order term by filling the holes with variables from the variable list. Naturally the hole-filling variables need to be distinct from any other variables in the term to avoid confusion. We will write a second-order term as $w_1, \ldots, w_n.t_{w_1,\ldots,w_n}$.

   A second-order variable instance has form $v[a_1; \ldots; a_n]$, where v is the variable itself, and $a_1, \ldots, a_n$ are its arguments. A second-order substitution is a list of second-order bindings, pairs of second-order variables and the second-order terms they are bound to. The result of applying the binding $[v \mapsto w_1, \ldots, w_n.t_{w_1,\ldots,w_n}]$ to the variable instance $v[a_1; \ldots; a_n]$, is the term $t_{a_1,\ldots,a_n}$ – the second-order variable's arguments filling the holes of the second-order term.

   Second order substitution is useful for instantiating pattern terms involving binding structure. For example the second-order substitution $[P \mapsto i.i \geq 0]$ applied to the pattern $\forall x{:}\mathbb{N}.P[x]$ yields the instance $\forall x{:}\mathbb{N}.x \geq 0$.

3. **Parameter**
   Nuprl terms can be parameterized by families of objects such as natural numbers and tokens. When defining abstractions using such parameters, one replaces an instance of a parameter by a parameter variable. Such parameter variables are replaced by parameter constants using parameter substitution.

4. **Level Expression**
   Level expression substitution involves replacing level expression variables within level expression parameters by other level expressions.

5. **Bound Variable**
   Such substitutions are useful for alpha-conversion of terms.

Matching involves recursively comparing the structure of an instance term against that of a pattern term. For a match to possibly succeed, the structures must only disagree at positions where there is some kind of variable in the pattern. Each disagreement must generate or confirm a binding for that variable.

The kinds of matches of instance parts to some sort of variable are:

1. **Term to First-Order Variable Term**
   For example, the instance $2+2$ matches the pattern $x+x$ giving the first-order term binding $[x \mapsto 2]$. In general, since instance terms also contain variable terms, the match routine distinguishes between variable terms in the pattern which can and cannot take part in matching. The variable terms which do take part are called *meta-variables*. In the example above, the variable x is considered a meta-variable. If a meta-variable occurs more than once, then all matches for it must be alpha-equal.

2. **Term to Second-Order Variable Term**
   In addition to making distinction between meta and non-meta variable terms, The match routines distinguishes between *active* second-order variables and *passive* second-order variables. Active second-order variables generate bindings. Passive second-order variables are used to confirm matches generated by other active second-order variables.

   For example, with P a second-order meta-variable, the instance $\forall i{:}\mathbb{N}.\ i \geq 0$ matches the pattern $\forall x{:}\mathbb{N}.\ P[x]$, giving the second-order term binding $[P \mapsto i.i \geq 0]$.

3. **Parameter Constant to Parameter Variable**
   For example, the instance term `apple{cox:tok}` matches the pattern term `apple{$x:tok}` giving the parameter binding `$x` $\mapsto$ `cox`.

4. **Bound Variable to Bound Variable**
   For example, the instance term $\lambda x.\ x$ matches the pattern term $\lambda y.\ y$, giving the bound variable binding $y \mapsto x$.

5. **Level Expression to Level Expression**
   This kind of matching is rather complex, since we sometimes desire that the pattern, when instantiated with the result of the match, be related to the instance by an order relation rather than an equality. For example, the instance $\mathbb{U}_i \times \mathbb{U}_j$ might match the the pattern $\mathbb{U}_k \times \mathbb{U}_{[k\ n]}$ giving a level expression substitution of $[k \mapsto [i\ j],\ n \mapsto j]$. The directions of the inequalities between level expressions in the instance and instantiated pattern are dependent on the position of the level expressions in the terms. They are usually chosen such that there is a certain inclusion relation between the instance and the instantiated pattern when each is considered as a type.

Term to first-order variable, term to second-order variable, and bound variable matching is always used in tactics which do matching. Parameter matching is only used when folding or unfolding abstractions. Level expression matching is only used by matching tactics which refer to lemmas. Unless otherwise stated, all tactics do *soft* matching - if necessary they will try to unfold soft abstractions to make a match go through.

Second-order variable instances cannot appear in Nuprl sequents, nor can second-order terms. Instead, we simulate them using respectively application, and lambda abstraction. Specifically, we define families of abstract terms with `so_apply` and `so_lambda`. (We need families to cope with the different possible arities.)

Often, when we match against the consequent of a lemma, we cannot obtain all the bindings to instantiate the lemma directly from the match. In these cases, we try to extend the match by inferring types of right-hand sides of existing bindings, and matching those inferred types against the type declarations in the lemma of the left-hand-sides of the bindings. For example, the typing lemma for the `length` function is:

$$\forall A{:}\mathbb{U}_i.\forall l{:}A\ \texttt{List}.\ \texttt{length}(l) \in \mathbb{N}$$

Consider the goal $\vdash$ $\texttt{length}(3{::}2{::}1{::}[]) \in \mathbb{N}$. A match of the concl of this goal against the consequent of the pattern gives the binding $l \mapsto 3{::}2{::}1{::}[]$, but doesn't give a binding for A. However, we can get the binding $A \mapsto \mathbb{Z}$ by matching the inferred type of $3{::}2{::}1{::}[]$ which is $\mathbb{Z}$List, against the declaration type of l, which is AList. Similarly, by inferring the universe which $\mathbb{Z}$ inhabits, we can get a binding for the universe level i.

For convenience, we inject the different kinds of bindings into the single type `var # term`. A pair of this type is interpreted according to the first entry in the following table which it matches. (Equating ML objects of type `var` and the objects of type `tok` they are isomorphic to.)

pattern	**interpretation**
$< v, \texttt{so\_lambda}(xs.t') >$	The higher-order binding $v \mapsto xs.t'$
$< v, \texttt{parameter}\{le{:}\texttt{l}\} >$	The level exp binding $v \mapsto le$
$< v, \texttt{parameter}\{w{:}\texttt{v}\} >$	The bound variable binding $v \mapsto w$
$< v, \texttt{parameter}\{p{:}*\} >$	The parameter variable binding $v \mapsto p$
$< v, t >$	The first-order binding $v \mapsto t$

Most tactics which do matching, take an optional `sub` argument which can be used to provide bindings which the match fails to find, or to override matches which are found.

9.2 Basic

9.2.1 Structural

`Id`
  The identity tactic.

`Fail`
  A tactic which always fails. Usually used inside other tactics.

`NthHyp i`
  Proves goals of form $\ldots A \ldots \vdash A$ where A is the `i`th hypothesis.

`NthDecl i`
  Proves goals of form $\ldots x{:}T \ldots \vdash x \in T$ or $\ldots x{:}Tldots \vdash x = x \in T$ or where $x{:}T$ is the `i`th declaration.

`AssertAt j t`
  Assert term `t` before hypothesis `j`. Generates `main` subgoal with `t` asserted, and `assertion` subgoal to prove `t`.

`Assert t`
  $=_{def}$ `AssertAt 0 t`. Assert `t` as last hypothesis.

`MoveToHyp j i`
  Move hyp `j` to before hyp `i`.

`MoveToConcl j`
  If hyp `j` is a proposition,

$$\ldots j.A \ldots \vdash C$$

`BY MoveToConcl`j

`main` $\ldots\ldots \vdash A \Rightarrow C$

  If hyp `j` is a declaration,

$$\ldots j.x{:}T \ldots \vdash C$$

`BY MoveToConcl`j

`main` $\ldots\ldots \vdash \forall x{:}T.\ C$

  `MoveToConcl` first invokes itself recursively on any hyp which might depend on hyp `j`.

`MoveDepHypsToConcl j`
  Use `MoveToConcl` to move hyps which use variable declared by hyp `j`.

`Thin i`
  Delete hypothesis `i`.

`RenameVar v i`
  Rename the variable declared in hypothesis `i` to `v`.

`RenameBVars (vsub : (var # var) list) c`
  Rename occurrences of bound variables in clause `c`.

9.2.2 Single-Step Decomposition

The *decomposition* tactics invoke the primitive so-called *introduction* and *elimination* rules of Nuprl's logic. We prefer the use of the word *decomposition* because it suggests in most cases the effect of the rules when they are applied.

`D c`
  Decompose the outermost connective of clause `c`. Usually `D` unfolds all top level abstractions and applies the appropriate primitive decomposition rule. `D` can take several optional arguments:

- A 'universe' argument, usually applied using the `At` tactical.
- A 't1' argument for a term. This argument is for instance necessary when decomposing a hypothesis with a universal quantifier outermost, or decomposing the conclusion with an existential quantifier outermost. For example: `With 'a' (D 0)`.
- 'v1' and 'v2' arguments for new variable names. These are useful for some hypothesis decompositions if one is not satisfied with the system supplied variable names. For example, `New ['x';'y'] (D 3)`.
- An 'n' argument to select a subterm. This is necessary when applying `D` to a disjunct in the conclusion. For example, `Sel 1 (D 0)`.

`D` is somewhat intelligent with instances of *set* and *squash* terms.

`ID c`

Intuitionistically decompose clause `c`. This behaves as `D` does, except that when decomposing a function, a universal quantifier, or an implication, in a hypothesis, the original hypothesis is left intact rather than thinned.

`MemCD`

Decomposes terms which are the immediate subterms of an membership term in the conclusion. Labels subgoal corresponding to subterm n with label `subterm` and number n. Other subgoals are labelled `wf`. For primitive terms `MemCD` uses the appropriate primitive rule. For abstractions, `MemCD` tries to use an appropriate well-formedness lemma. The lemma for term a t with opid *opid* should have name *opid*`_wf` and should be a simple universal formula with consequent $t \in T$. The subterms of t usually should be all variables. Constants are acceptable as subterms too. If more than one lemma is needed, the lemmas should be distinguished by suffices to the *opid*`_wf` root. `MemCD` attempts to use lemmas in the reverse of the order in which they occur in the library. If the concl is $a \in A$ where a is an instance of t and A doesn't match any of the T of the lemmas, then `MemCD` tries matching a against t of the last lemma. If this succeeds and generates some substitution Θ, `MemCD` produces a subgoal $\theta T \subseteq A$ and tries to use the `Inclusion` tactic to prove it.

An example application of the `MemCD` tactic is

$\ldots \;\vdash\; < a, b > \in x{:}A \times B_x$

`BY MemCD`

`subterm1`:	$\ldots \;\vdash\; a \in A$
`subterm2`:	$\ldots \;\vdash\; b \in B_a$
`wf`:	$\ldots x{:}A \;\vdash\; B_x \in \mathbb{U}_*$

`EqCD`

`EqCD` is like `MemCD` except that it works on the immediate subterms of equality terms rather than membership terms and the subgoals generated are equality terms rather than membership terms. Equalities don't have to be reflexive. `EqCD` is good for congruence reasoning and is used extensively by the rewrite package.

`EqHD i`

Decompose terms which are immediate subterms of equality hypotheses. Works when the type is a `product` or `function` type.

`MemHD i`

Like `EqHD` but works on the immediate subterms of membership terms.

`EqTypeCD`

Decompose just the type subterm of a conclusion equality term. Only works when the type is a `set` type or is an abstraction that eventually unfolds to a set type.

`MemTypeCD`

As `EqTypeCD` but works on membership terms.

`EqTypeHD i`

Decompose just the type subterm of a hypothesis equality term. Only works when the type is a `set` type or is an abstraction that eventually unfolds to a set type.

`MemTypeHD i`

As `EqTypeHD` but works on membership terms.

9.2.3 Iterated Decomposition

Several tactics do iterated decomposition of clauses. Here are a few that are commonly used, along with the connectives that they work on.

Tactic	In Hyps	In Concl
`UnivCD`		$\forall \Longrightarrow$
`GenUnivCD`		$\forall \Longrightarrow \wedge \Longleftrightarrow$
`RepD`	$\wedge$	$\forall \Longrightarrow$
`GenRepD`	$\wedge$	$\forall \Longrightarrow \wedge \Longleftrightarrow$
`ExRepD`	$\exists\wedge$	$\forall \Longrightarrow$
`GenExRepD`	$\exists \wedge \vee$	$\forall \Longrightarrow \wedge \Longleftrightarrow$

If a guard term is encountered in the process of decomposing the conclusion, the guard term is removed and decomposition of the conclusion stops.

9.3 Tacticals

Tacticals are functions for composing tactics. Infix tacticals are distinguished by having the first part of their name in all capitals. Infix tacticals always associate to the left.

9.3.1 Basic Tacticals

`T1 ORELSE T2`
: Try running `T1`. If it fails, run `T2` instead.

`T1 THEN T2`
: Run `T1`, and then on all subgoals generated by `T1`, run `T2`.

`T THENL [T1;...;Tn]`
: Run T1, generating exactly `n` subgoals. Then run `Ti` on the ith subgoal (numbering subgoals from left to right.)

`Try T`
: $=_{def}$ `T ORELSE Id`

`Complete T`
: Run `T`. Fail if `T` generates one or more subgoals.

`Progress T`
: Run `T`. Fail if `T` makes no progress. (For example, if `T` is `Id`.)

`Repeat T`
: Repeat application of `T` on subgoals generated by previous tries, until no further progress made

`RepeatFor i T`
: Repeat application of `T` exactly `n` times.

`If (e:proof -> bool) T1 T2`
: If `e p` evaluates to `TRUE` then run `T1`. Otherwise, run `T2`.

9.3.2 Label Sensitive Tacticals

`IfLab lab T1 T2`
If `lab` matches label of `p`, run `T1`. Otherwise, run `T2`.

`T1 THENM T2` $=_{def}$ `T1 THEN IfLab 'main' T2 Id`

`T1 THENA T2` $=_{def}$ `T1 THEN IfLab 'aux' T2 Id`

`T1 THENW T2` $=_{def}$ `T1 THEN IfLab 'wf' T2 Id`

`IfLabL [l1,T1; l2,T2; ...  ;ln,Tn]`
Run the first `T`i for which `l`i matches label of `p`

`T THENLL [l1,Ts1; l2,Ts2; ...  ;ln,Tsn]`
Run tactic `T` then do the following on each subgoal, scanning the subgoals from left to right. Match the subgoal's label against each `li` until a match succeeds. Then if the subgoal also has number label j, retrieve the jth tactic from `Tsi` and run that tactic on the subgoal. If the subgoal has no number label, then pop the first tactic off the `Tsi` list and run that tactic. If there are not sufficient tactics in the appropriate lists, `THENLL` fails. If there are too many, then the excess are ignored. If a subgoals label doesn't match any of the `li` then run the `Id` tactic.

`SeqOnM [T1;...;Tn]`
Run the tactics `T1` to `Tn` on successive `main` subgoals.

`RepeatM T`
Repeat the tactic `T` on `main` subgoals.

`RepeatMFor i T`
Repeat the tactic `T` on `main` subgoals exactly `i` times.

9.3.3 Multiple Clause Tacticals

`AllHyps (T : int -> tactic)`
Try running `T` on each hypothesis starting with the end of the hypothesis list and working backwards. If `T` succeeds on some hypothesis, then `AllHyps` only continues on subgoals created by `T` that are labelled `main`.

`All (T : int -> tactic)`
Similar to `AllHyps`, except that also tries applying `T` to conclusion after trying to apply it to hypotheses.

`On [c1;...;cn] (T : int -> tactic)` $=_{def}$ `T c1 THENM ...  THENM T cn`

9.4 Case Splits and Induction

There are two ways of doing case splits and induction. The more general way is to backchain through an appropriate lemma. For example, look at the lemmas `int_upper_ind` and `int_seg_ind` at the end of the `int_2` theory. To use these lemmas, you must ensure that the type the induction is being done over is in an outermost universal quantifier in the conclusion. For example, to use `int_seg_ind`, the conclusion must be of form $\forall i{:}\{j \ldots k^{-}\}.P_i$. The following case-split and induction tactics are good for a few common cases. With them, the variable the induction / case-split is being done over should be declared in some hypothesis.

`BoolCases i`

Do case split on whether variable declared to be of type $\mathbb{B}$ (the booleans) in hyp `i` is `tt` or `ff`. Generates `truecase` and `falsecase` subgoals.

`ListInd i`
: Do list induction on hypothesis `i`. Generates `upcase` and `basecase` subgoals. First moves any depending hyps to concl.

`IntInd i`
: Do integer induction on hypothesis `i`. Generates `upcase`, `basecase` and `downcase` subgoals. First moves any depending hyps to concl. This is a little smarter than the primitive rule, in that it maintains the name of the induction variable.

`NatInd i`
: Do natural-number induction on hypothesis `i`. Hypothesis must be a `nat` abstraction. Generates `upcase` and `basecase` subgoals. First moves any depending hyps to concl. This is a little smarter than the primitive rule, in that it maintains the name of the induction variable.

`NSubsetInd i`
: Do induction on subrange of the natural numbers. Hyp `i` should be a `nat`, `nat_plus`, `int_upper` or `int_seg` abstraction. Generates two main subgoals - `basecase` and `upcase`- and approximately 15 `aux` subgoals which should always be easily solvable by `Auto`.

`CompNatInd i`
: Do complete natural number induction on hyp `i`. Hyp `i` must be a `nat` abstraction.

Sometimes using a lemma results in unprovable well-formedness goals. This occurs in particular when proving well-formedness lemmas. In these cases, you should try to use one of the tactics above.

The theory `well_fnd` has some definitions for well-founded induction. In particular it defines the tactic `Rankind`. This is useful when you know how to do induction over some type A and you want to do induction over a type B using some *rank* function which maps elements of B to elements of A. The tactic is described in the objects `inv_image_ind_tac` and `rank_ind`.

The theory `bool_1` defines various tactics for case splitting on the value of boolean expressions in the conclusion. Tactics include `BoolCasesOnCExp` and `SplitOnConclITE`. View the theory for details.

9.5 Forward and Backward Chaining

Forward and backward chaining involves treating a component of a universal formula (see Section 9.1.3) as a derived rule of inference. Backward chaining involves matching the conclusion of a sequent against the consequent of a universal formula. The antecedents of the universal formula, instantiated using the substitution resulting from the match, then become new subgoals. Forward chaining involves matching hypotheses of a sequent against antecedents of a universal formula. The consequent of the universal formula, instantiated using the substitution resulting from the match, then becomes a new hypothesis.

`BackThruLemma name`
`BackThruHyp i`
: The `name` (`i`) argument selects the lemma (hypothesis) to back-chain through. Subgoals corresponding to antecents of the lemma (hyp) are labelled with `antecedent`. The rest are labelled `wf`. Aliases are `BLemma` and `BHyp`.

`FwdThruLemma name is`
`FwdThruHyp i is`

The `name` (`i`) argument selects the lemma (hypothesis) to forward chain through. `is` selects the hypotheses which are to be matched against antecedents of the chaining formula. The order of the `is` is immaterial; the tactics try all possible pairings of hypotheses with antecedents. If there are more antecedents that hyps listed in the `is`, the antecedents not matched will manifest themselves as new subgoals to be proved. The main subgoal with the consequent of the lemma (hyp) asserted is labelled `main`. Unmatched antecedents are labelled `antecedent` and the rest are labelled `wf`. Aliases are `FLemma` and `FHyp`.

Chaining tactics take a number of optional arguments.

- An explicit list of variable bindings as a `sub` argument. This argument is necessary when all variable bindings cannot be inferred from matching. The `sub` argument is supplied using the `Using` tactical. For example:

  `Using ['n'.'3'] (BackThruLemma 'int_upper_induction')`

  would bind the variable `n` in the lemma `int_upper_induction` to the value `3`.

- A specific simple component of a general formula can be selected using an `'n'` argument, supplied by using the `Sel` tactical. For example

  `Sel 2 (FwdThruLemma 'add_mono_wrt_eq')`

  An 'n' argument of `-1` forces the tactic to treat the formula as a simple formula.

`Backchain bc_names`
`CompleteBackchain bc_names`

Repeatedly try `BackThruLemma` using lemmas named in `bc_names` in order given. `Backchain` leaves alone any subgoals which don't match the consequent of any of the lemmas. `CompleteBackchain` backtracks in the event of any such subgoal coming up.

In addition to lemma names, a few special names are recognized:

- An integer i. Use hypothesis i. (i must be a positive integer. Negative integers can't be used to refer to hypotheses here.)
- `hyps`: hyps $1 \cdots n$ where n is the number of hyps. Skips hyps which declare variables.
- `rev_hyps`: As `hyps` but in order $n \cdots 1$.
- `new_hyps`: new hyps introduced by backchaining, least recent first.
- `rev_new_hyps`: As `new_hyps` but in opposite order.

`HypBackchain` $=_{def}$ `Backchain ''rev_new_hyps rev_hyps''`
`CompleteHypBackchain` $=_{def}$ `CompleteBackchain ''rev_new_hyps rev_hyps''`

`InstLemma name [t1;...;tn]`

Instantiate lemma `name` with terms `t1` through `tn`. If the lemma has m distinct level expressions, the first m terms should be level expressions to substitute for the lemma's level expressions. (Inject level expression le into the term type using the special term `parameter{`le`:l}`. In the term editor you select this term by the name `parameter`.)

`InstHyp [t1;...;tn] i`
: Instantiate universal formula in hyp `i` with terms `t1` through `tn`.

`InstConcl [t1;...;tn] i`
: Instantiate existential quantifiers in conclusion with terms `t1` through `tn`.

9.6 Decision Procedures

9.6.1 ProveProp

The `ProveProp` family of tactics are useful for partially or completely proving goals that involve simple propositional reasoning. The strategy is basically to that for classical tableau: Propositions in hypotheses and the conclusion are exhaustively decomposed and applications of the `Hypothesis` tactic are sought. A slight tweak is the tactic has to do 'or' branching and backtracking when it tackles an $\vee$ conclusion or an $\Longrightarrow$ or $\neg$ hypothesis, because the Nuprl sequents only allow one conclusion rather than many as is commonly the case in classical sequent calculi.

`ProveProp`
: The basic tactic. Fails if doesn't succeed in solving all main goals.

`ProveProp1`
: Like `ProveProp`, but leaves main subgoals at 'or' branching points of the search for a solution when the search down every branch fails.

`ProvePropWith (T : tactic)`
: Like `ProveProp1`, but tries running tactic `T` before completely abandoning a search path. If `T` creates any `main` subgoals, search continues on these subgoals.

`ProveProp` is not complete for intuitionistic propositional logic, because it always thins $\Longrightarrow$ and $\neg$ hypotheses that are decomposed.

9.6.2 Eq

`Eq` does trivial equality reasoning. It proves goals of form $H \ \vdash \ t = t' \in T$ using hypotheses that are equalities over T and the laws of reflexivity, commutativity and transitivity. It also uses hypotheses that are equalities over T' when $T = T'$ is deducible from other hypotheses using reflexivity, commutativity and transitivity. The `Eq` rule is implemented as a procedure coded in Lisp.

9.6.3 RelRST

`RelRST` is a tactic that tries to solve goals by exploiting common properties of binary relations, including reflexivity, symmetry, transitivity, irreflexivity, antisymmetry, and linearity.

The heart of this tactic is a routine that builds a directed graph based on the binary relations in a sequent and finds shortest paths in the graph. Extensions to this routine to allow it to handle strict order relations and relations with differing strengths.

`RelRST` uses the the same database on relations and some of the same lemmas as the rewrite package. In addition, it relies on lemmas in the library of the following forms.

- *Irreflexivity* lemmas. These should have form

$$\forall x_1{:}T_1 \ldots x_m{:}T_m.\ \forall y{:}S.\ A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow y\ R\ y \Rightarrow \mathsf{False} Allxs{:}As, Ally{:}TBs => y < y => False$$

and be named *opid-of-*$<$`_irreflexivity`.

- *Antisymmetry* lemmas. These should have form

$$\forall x_1{:}T_1 \ldots x_m{:}T_m.\ \forall y, y'{:}S.\ A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow y \leq y' \Rightarrow y' \leq y \Rightarrow y = y'$$

  and be named *opid-of-*$\leq$`_antisymmetry`.

- *Complementing* lemmas. These should have form

$$\forall x_1{:}T_1 \ldots x_m{:}T_m.\ \forall y, y'{:}S.\ A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \neg(y \leq y') \Rightarrow y' < y$$

  and be named *opid-of-*$\leq$`_complement`, or

$$\forall x_1{:}T_1 \ldots x_m{:}T_m.\ \forall y, y'{:}S.\ A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \neg(y < y') \Rightarrow y' \leq y$$

  and be named *opid-of-*$<$`_complement`.

$$All xs{:}As, All y, y'{:}T Bs => not(y LTR y') => y' LER y$$

`RelRST` generalizes the `Eq` tactic in previous versions of Nuprl that only handled such reasoning with the equality relation of Nuprl's type theory.

Here are a couple of examples of `RelRST`'s use from a theory of divisibility over the integers:

```
1. a: ℤ
2. a': ℤ
3. b: ℤ
4. b': ℤ
5. ...
6. a' | a
7. b | b'
8. ...
9. a | b
⊢ a' | b'
|
BY (RelRST ...)
```

and

```
1. a: ℤ
2. b: ℤ
3. y1: ℤ
4. ...
5. gcd(a;b) = y1
6. y2: ℤ
7. ...
8. gcd(b;a) = y2
9. ...
10. y1 ~ y2
⊢ gcd(a;b) ~ gcd(b;a)
|
BY (RelRST ...)
```

Here, I have elided hypotheses that were not required by `RelRST` to solve the goals. The $\sim$ relation is the *associated* relation and `gcd(a;b)` is a function that computes the greatest common divisor of `a` and `b`. The second example illustrates how `RelRST` is able to cope with relations of differing strengths.

Unlike `Eq`, `RelRST` doesn't involve any extensions being made to the primitive rule refiner.

9.6.4 Arith

The `Arith` tactic is used to justify conclusions which follow from hypotheses by a restricted form of arithmetic reasoning. Roughly speaking, `Arith` knows about the ring axioms for integer multiplication and addition, the total order axioms of $<$, the reflexivity, symmetry and transitivity of equality, and a limited form of substitutivity of equality. We will describe the class of problems decidable by `Arith` by giving an informal account of the procedure which `Arith` uses to decide whether or not C follows from H.

`Arith` understands standard arithmetic relations over the integers; namely terms of the form $s < t$, $s \leq t$, $s > t$, $s \geq t$, $s = t \in \mathbb{Z}$ and $s \neq t \in \mathbb{Z}$. It also recognizes negations of these terms. As the only equalities `Arith` concerns itself with are those of the form $s = t \in \mathbb{Z}$, we will drop the $\in \mathbb{Z}$ and write only $s = t$ in the rest of this description. The `arith` rule may be used to justify goals of the form

$$H \ \vdash \ C_1 \vee \ldots \vee C_m$$

where each C_i is a term denoting an arithmetic relation. If `Arith` can justify the goal it will produce subgoals requiring the user to show that the left- and right-hand sides of each C_i denote integer terms. As a convenience `Arith` will attempt to prove goals in which not all of the C_i are arithmetic relations; it simply ignores such disjuncts. If it is successful on such a goal, it will produce subgoals requiring the user to prove that each such disjunct is a well-formed proposition.

`Arith` analyzes the hypotheses of the goal to find relevant assumptions. In particular, it will maximally decompose each hypothesis into a term of the form $A_1 \wedge \ldots \wedge A_n$ ($n \geq 1$), and will use as an assumption any of the A_i which are arithmetic relations of the form describe above. It also extracts assumptions from declarations of variables in types that are subsets of $\mathbb{Z}$. For example from the declaration $i{:}\mathbb{N}$ it extracts the assumption that $i \geq 0$.

`Arith` begins by normalizing the relevant formulas of the goal according to the following conventions:

1. Rewrite each relation of the form $s \neq t$ as the equivalent $s < t \vee t < s$. A conclusion C follows from such an assumption if it follows from either $s < t$ or $t < s$; hence `arith` tries both cases. Henceforth, we assume that all negations of equalities have been eliminated from consideration.

2. Replace all occurrences of terms which are not addition, subtraction or multiplication by a new variable. Multiple occurrences of the same term are replaced by the same variable. `Arith` uses only facts about addition, subtraction and multiplication, so it treats all other terms as atomic. At this point all terms are built from integer constants and integer variables using $+$, $-$ and $*$.

3. Rewrite all terms as polynomials in canonical form. The exact nature of the canonical form is irrelevant (the reader may think of it as the form used in high school algebra texts) but has the important property that any two equal terms are identical. Each term now has the form $p + c\theta p' + c'$, where p and p' are nonconstant polynomials in canonical form, c and c' are constants, and θ is one of $<$, $=$ or $\geq$ ($s \geq t$ is equivalent to $\neg t < s$).

4. Replace each nonconstant polynomial p by a new variable, with each occurrence of p being replaced by the same variable. This amounts to treating each nonconstant polynomial as an atom. Now each formula is of the form $z + c\theta z' + c'$. `Arith` now decides whether or not the conclusion follows from the hypotheses by using the total order axioms of $<$, the reflexivity, symmetry, transitivity and substitutivity of $=$, and the following so-called *trivial monotonicity* axioms (c and d are constants).

   - $x \geq y, c \geq d \Rightarrow x + c \geq y + d$
   - $x \geq y, c \leq d \Rightarrow x - c \geq y - d$

   These rules capture all of the acceptable forms of reasoning which may be applied to formulas in canonical form.

9.6.5 SupInf

9.6.5.1 Description

The algorithm used in the `Arith` tactic cannot solve general sets of linear inequalities over the integers, though such problems are abundant (for example when doing array bounds checking). Solving linear inequalities over the integers is a strictly harder problem than over the rationals: polynomial time algorithms are known for the solving linear inequalities over the rationals, but integer linear programming is NP complete.

The `SupInf` tactic is uses the *Sup-Inf* method of Bledsoe [?] for solving integer inequalities. While method is only complete for the rationals, not the integers, it does work well in practice for the integers.

The basic algorithm considers a conjunction of inequalities $0 \leq e_1 \wedge \ldots \wedge 0 \leq e_p$ where the e_i are linear expressions over the rationals in variables $x_1 \ldots x_n$ and determines whether or not there exists an assignment of values to the x_j that satisfies the conjunction. The algorithm works by determining upper and lower bounds for each of the variables in turn — hence the name 'sup-inf'. The bound calculations are always conservative, so that if some upper bound is strictly below some lower bound, then the conjunction is unsatisfiable.

Shostak [?] showed that the calculated bounds are the best possible, and hence that the algorithm is complete for the rationals. He proposed a simple modification that made the algorithm return an explicit satisfying assignment when the conjunction is satisfiable.

When used over the integers, the Sup-Inf algorithm is sound, but not complete; if there is no satisfying assignment over the rationals, then there is also none over the integers. However, there are cases when the algorithm finds a rational-valued satisfying assignment even though none exists that is integer valued. There are standard techniques for restoring completeness, but it has been both Shostak's and our experiences to date that examples for which the algorithm is incomplete are rare in practice.

The procedure implemented currently does the following:

1. Takes a goal g and extracts a logical expression P built from the logical connectives $\wedge, \vee, \neg$, the order relations on the integers $\leq$ and $<$, and the equality relation $=$ on the integers, such that if $\neg P$ is not satisfiable, then the goal g is true. If the goal has the form

$$x_1, \ldots x_n{:}\mathbb{Z},\ r_1, \ldots r_k \vdash r_0$$

where the r_i are all instances of the $\leq, <, =$ relations over the integers involving expressions over the integer variables $x_1, \ldots, x_n$, then $\neg P$ has form

$$r_1 \wedge \ldots \wedge r_k \wedge \neg r_0.$$

2. The expression $\neg P$ is put into disjunctive normal form. Occurrences of $=$ and $<$ relations are eliminated in favour of $\leq$. Where possible, $=$'s are eliminated by substitution rather than splitting into inequalities.

3. The left-hand argument of each $\leq$ is moved to right-hand side and the integer expressions are put into a sum of products normal form. Each product has any constant coefficient brought out to the left of the product.

4. Each distinct non-linear expression is generalized to a new rational variable. These non-linear expressions might involve $*$ and $\div$, as well as integer-valued functions (for example, the list length function). The arithmetic expressions are now all linear.

5. Each disjunct is augmented with extra arithmetic information suitably normalized that comes from various sources including:

   (a) typing of variables and generalized non-linear expressions. If variable i has type $\{j \ldots\}$, then $j \leq i$ can be added.

   (b) arithmetic property lemmas. An example is a lemma stating that the length of two lists appended is the sum of the lengths of each list.

   This augmentation is in general a recursive procedure; the inferred arithmetic propositions can themselves contain variables and non-linear expressions about which further information can be inferred.

6. The Sup-Inf algorithm is run on each disjunct. If none is satisfiable, then the original goal is true. If a satisfying assignment is found, then it is returned to the user as a counter-example.

7. When no disjunct is satisfiable, the procedure creates several well-formedness subgoals. Some of these check the well-formedness of the arithmetic expressions in the conclusion of the original goal g. Others check that the arithmetic property lemmas can be instantiated as the procedure assumed they could be.

The inference of arithmetic properties from typing and from property lemmas greatly increases the procedure's usefulness.

Unlike most other tactics, but like the `arith` rule which `SupInf` largely supercedes, `SupInf`'s inferences are not refined down to primitive rule level, so Nuprl's soundness now depends on the soundness of a core part of `SupInf`'s implementation.

9.6.5.2 Details

First a few definitions.

- An *arithmetic type* is either the type $\mathbb{Z}$ (`int`) or one of the standard subtypes of $\mathbb{Z}$ defined in the `int_1` theory. Specifically: $\mathbb{N}$ (`nat`), $\mathbb{N}^+$ (`nat_plus`), $\mathbb{Z}^{-o}$ (`int_nzero`), `{`i`...}` (`int_upper`), `{`$i...j^-$`}` (`int_seg`), or `{`$i...j$`}` (`int_iseg`).

- An *arithmetic literal* is one of

  - the relations a = b $\in T$ and $a \neq b$ $\in T$ where T is an arithmetic type,
  - the relation a = b $\in T$ where T is an arithmetic type,
  - Any of the inequalities $<$, $>$, $\leq$, or $\geq$,
  - Either of the above inside one or more negations ($\neg$).

`SupInf` recognizes the arithmetic literals that occur in a sequent either at the top level of a clause or buried under `mwedge` and `mvee` connectives.

There are two variants on the `SupInf` tactic: `SupInf` and `SupInf'`. Only `SupInf'` tries inferring additional arithmetic information about the non-linear terms in arithmetic expressions.

The information on a non-linear term is gathered in two ways:

1. The standard type-inference function `get_type` is run on the term, and if it returns one of the standard arithmetic subtypes of Z, then the predicate information from the subtype is used.

2. Arithmetic property lemmas are examined.

An *arithmetic property lemma* should have form

$$\forall x_1{:}T_1 \ldots x_m{:}T_m.\ A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow C$$

where C is constructed from $\wedge$'s, $\vee$'s and arithmetic literals. *match handles* are selected from C. A match handle is a non-linear arithmetic term that occurs as an argument to an arithmetic literal in C such that all the free variables contained in C and the A_i are also contained in the non-linear term. An arithmetic lemma is considered as providing information about each of its match handles.

Currently, an arithmetic property lemma is only used if after matching the match handle in C with the non-linear term that information is desired on, all the instantiated A_i are equal to hypotheses of the sequent. This is perhaps an overly strict condition.

Arithmetic property lemmas are identified by invoking the ML function

```
add_arith_lemma
  lemma-name : tok
=
 ():unit
```

`SupInf'` is currently under development and should be used with caution. Hopefully all changes to it will only be enhancements, so if it is successful now, it will be successful in the future. One problem with it is that the process of type-checking subgoals created by instantiating arithmetic property lemmas can cause the creation of subgoals that require arithmetic reasoning to solve. When `SupInf'` is used as part of a type-checking tactic (as it is in `Auto'`), there are sometimes cases when SupInf' unendingly gets invoked on subgoals derived from ones that it itself created. Further work is needed on SupInf' to avoid this happening.

SupInf identifies counter-examples if it fails. These can be viewed by looking at value of the ML variable supinf_info. The value gives a list of bindings of variables in the goal for the counter-example. There are two kinds of counter-examples; integer and rational. If SupInf finds an integer counterexample, then you know that the goal is definitely unprovable. If a rational counter-example, then SupInf is unsure whether the goal is true or not. These latter cases should be rare in practice.

A couple of examples of uses of the SupInf tactic are as follows. It is able to prove the goal

```
1. x: ℤ
2. y: ℤ
3. z: ℤ
4. 2 * y + 3 ≤ 5 * z
5. z ≤ x - y
6. 3 * x ≤ 5
⊢ 2 * y ≤ 3
```

but on

```
1. x: ℤ
2. y: ℤ
3. 3 * x ≥ y
4. y ≥ 2
⊢ x + y > 3
```

finds the counterexample x= 1 and y= 2. Examples of arithmetic property lemmas are:

```
⊢ ∀ i:ℤ. ∀ j:ℤ. i ≥ 0 ⇒ j > 0 ⇒ 0 ≤ i rem j < j
```

where rem is the remainder function and:

```
⊢ ∀A:𝕌. ∀as:A list. ∀n:ℕ|as|. (|nth_tl(n;as)| = |as| - n)
```

where nth_tl(n;as) takes the nth tail of list as, |·| is the list length function and ℕ|as|. is an abbreviation for the integer segment {0...|as|-1}. The latter lemma is invoked when SupInf proves the goal:

```
1. T: 𝕌
2. as: T List
3. m: ℕ
4. n: ℕ
5. |as| ≤ m + n
⊢ |nth_tl(n;as)| ≤ m
```

9.7 Rewriting

9.7.1 Introduction

Rewriting is basically the process of using equations as transformational rules.

In Nuprl, rewrite rules are derived from

- abstractions,

- primitive reduction rules, and
- formulas of form

  $$\forall x_1{:}T_1 \ldots x_i{:}T_i.\ a = b$$

  that occur as lemmas or hypotheses.

Nuprl's rewrite package is a collection of ML functions for creating rewrite rules and applying them in various fashions to clauses of a sequent.

The package supports rewrite rules involving various equivalence relations. Examples include the 3-place equality-in-a-type relation, the iff relation, and the permutation relation on lists. Nuprl's logic doesn't guarantee that all equivalence relations are respected. In cases when there is no guarantee, the package takes care of automating proofs that the relations are respected.

The notion of rewriting is extended to including rules involving any transitive relation. Here, the package takes care of checking that relevant terms are appropriately monotonic.

The package is based around ML objects of type `convn` called *conversions*, similar to those found in other tactic based theorem provers such as LCF, HOL and Isabelle. Conversions provide a language for systematically building up rewrite rules in a fashion similar to the way tactics are assembled using tacticals.

For convenience, a few concise rewriting tactics are provided that completely hide the conversion language (see Section 9.7.2). These are sufficient in many situations, though most Nuprl users will need eventually to familiarize themselves with many of the details of the package.

Note that Section 9.7.11 describes some older substitution tactics that can be used for certain very simple kinds of rewriting.

9.7.2 Concise Rewriting Tactics

`Unfolds as c`

Unfold all visible occurrences of abstractions listed in the token list `as` in clause `c`

`Unfold a c` $=_{def}$ `Unfolds [a] c`

`RepUnfolds as c`

Repeatedly try unfolding any occurrences of abstractions listed in the token list `as` in clause `c`

`Folds as c`

Fold all visible occurrences of abstractions listed in the token list `as` in clause `c`

`Fold a c`$=_{def}$ `Folds [a] c`

`Reduce c`

Repeatedly contract all both primitive and abstract redices in clause `c`.

The `Reduce` tactic can take an optional *force* argument. The tactic

`With` *force* `(Reduce` *c*`)`

only reduces those redices with strength less than or equal to *force*. For more on defining abstract redices and setting the strength of redices, see Section **??** and Section 9.7.5.6.

A couple of rewrite tactics provide access to all kinds of rewrite rules. These tactics take a control string to specify the rewrite rules to use. Control string should be a whitespace-seperated list of tokens as specified in Table 9.1. The tactics are:

`RWW (`*ctl-str* `:  string)` *i*

Token	Rule
i	Use hyp i as an l-to-r rule
i`<`	Use hyp i as an r-to-l rule
name	Use lemma *name* as an l-to-r rule
name`<`	Use lemma *name* as an r-to-l rule
`r:`*id*	Reduce redex with opid *id*
`r*`	Reduce any redex
`r*`*force*	Reduce any redex with force *force*
`u:`*id*	Unfold abstraction with opid *id*
`f:`*id*	Fold abstraction with opid *id*

Table 9.1: Format of Tokens in Rewrite Control Strings

Repeatedly apply rewrite rules specified by *ctl-str* to all nodes of clause i until no further progress is made.

`RW0` (*ctl-str* : `string`) i

Apply rewrite rules specified by *ctl-str* in one top-down pass over clause i. If one of the rewrite rules succeeds on some subterm, then don't try rewriting the subterms of the rewritten subterm.

9.7.3 Introduction to Conversions

This section presents a simplified implementation of conversions and conversionals to convey the general ideas. Later sections describe the actual conversions that are implemented. Note however that the conversionals introduced here have the same names and same behaviours as those in the actual system.

Let `convn` be an ML concrete type alias for the type of conversions. In this section, assume that `convn` is an alias for the type `term -> term`, where `term` is a type of terms that we want to rewrite. Later on, I describe the type that `convn` is actually an alias for. If c is of type `convn`, then we can use c to rewrite t of type `term` by simply running the ML evaluator on the application $c\ t$.

For the purposes of the section only, let me introduce a basic conversion called `RuleC : term -> convn`. The conversion `RuleC` expects its term argument to be of form $a = b$ where the free variables of b are a subset of those in a. If the conversion `RuleC '`$a = b$`'` is applied to a term t, `RuleC` tries to find a substitution θ such that $\theta a = t$. If it succeeds, it returns the term θb. If a substitution cannot be found, `RuleC` raises an exception. The conversion `RuleC '`$a = b$`'` therefore rewrites instances of a to corresponding instances of b. For example:

`RuleC '`$x + 0 = x$`'`

when applied to the term $(2 \times 3) + 0$ yields the term 2×3.

`RuleC` cannot by itself rewrite subterms of a term; if

`RuleC '`$x + 0 = x$`'`

is applied to the term $(1 + 0) \times 3$, it fails. There are a variety of higher-order conversions that map a conversion such as `RuleC` over all subterms of a term. An example of a conversional is `SweepUpC : convn -> convn`. If c is a conversion, then `SweepUpC` c is also a conversion. if `SweepUpC` c is applied to some term t, an attempt is made to apply c once to each subterm of t working from the leaves of term t up to its root. `SweepUpC` c only fails every every application of c fails. So if

```
SweepUpC (RuleC 'x + 0 = x')
```

is applied to term $(1 + 0) \times 3$, it succeeds and returns the term 1×3.

The basic conversion for sequencing conversions is `ANDTHENC : convn -> convn -> convn`. In Nuprl, we reserve all-capital names for infix functions so a normal application of `ANDTHENC` to conversions c_1 and c_2 has form c_1 `ANDTHENC` c_2. When applied to a term t, c_1 `ANDTHENC` c_2 first applies c_1 to t. If c_1 succeeds, returning a term t', then c_2 is applied to t' and the result is returned. If either c_1 or c_2 fails, then c_1 `ANDTHENC` c_2 also fails. By analogy with *tacticals* being higher-order tactics, `ANDTHENC` is called a *conversional.*

The `ORELSEC : convn -> convn -> convn` conversional is for combining alternative conversions. When c_1 `ORELSEC` c_2 is applied to a term t, it first tries applying c_1 to t, and if this succeeds returns the result. If the application of c_1 fails, then it tries applying c_2 to t, failing if c_2 fails.

The definition for `SweepUpC` is:

```
letrec SweepUpC c t = (SubC (SweepUpC c) ORTHENC c) t
```

`SubC : convn -> convn` when applied to a conversion c and a term t, applies c to each of the immediate subterms of t. It fails only when c fails on every immediate subterm. Hence, it always fails when t is a leaf node and has no immediate subterms. c_1 `ORTHENC` c_2 is similar to c_1 `ANDTHENC` c_2 in that it first tries c_1 and then c_2. However `ORTHENC` only fails if both c_1 and c_2 fail. So, a call of `SweepUpC` c on argument t first tries to apply `SweepUpC` c to the immediate subterms of t and then then tries to apply c to t itself. Note that without the `t` argument on the left and right sides of the definition, `SweepUpC` in ML's call-by-value evaluation scheme would recurse indefinitely.

The definition for `ORTHENC` is:

```
let c1 ORTHENC c2 = (c1 ANDTHENC TryC c2) ORELSEC c2
```

where `TryC` c is defined as c `ORELSE IdC` and `IdC: convn`, when applied to any t, always returns t.

Other conversionals that are commonly used in the work described in this thesis are:

- `FirstC : convn list -> convn` which is an n-ary version of `ORELSEC`,
- `RepeatC : convn -> convn` which repeatedly tries applying a conversion till no further progress is made,
- `HigherC : convn -> convn` which applies a conversion to only nodes higher in a term tree. What I mean by 'higher' is probably best understood by studying the definition of HigherC:

  ```
  letrec HigherC c t = (c ORELSEC SubC (HigherC c)) t
  ```

- `SweepDnC : convn -> convn` which sweeps a conversion down over a term tree from the root towards the leaves. Its definition is:

  ```
  letrec SweepDnC c t = (c ORTHENC SubC (SweepDnC c)) t
  ```

- `NthC int -> convn -> convn`. `NthC` i c t tries c on each node in t in pre-order order, but only on the ith success of c does it go through with the rewrite that c suggests. This is very useful during interactive proof when for example you want to unfold one instance of a definition but not any others.

9.7.4 Nuprl Conversions

In Nuprl, `convn` is an ML concrete type abbreviation for the type

`env -> term -> (term # reln # just).`

`env`, `reln` and `just` are abstract types for *environments*, *relations* and *justifications*. A conversion is a function that takes as arguments an environment e and a term to be rewritten t, and returns a triple $\langle t', r, j\rangle$. The environment e specifies amongst other things the types of all the variables which might be free in the term t. The term t' is the rewritten version of the term t. The relation r specifies the relationship between t and t'. Writing r using infix notation, we can say that $t\ r\ t'$ is true. The relation r is usually some equivalence relation but it also can be an order relation. The justification j is an object that tell Nuprl how to prove that $t\ r\ t'$. Conversions fail if they are not appropriate for the term they are applied to. More information on environments, rewrite relations, and justifications can be found in Section 9.7.8,Section 9.7.9 and Section 9.7.10 respectively.

A tactic `Rewrite : convn -> int -> tactic` is used for applying a conversion to some clause of a sequent. It takes care of executing the justifications generated by conversions. Section 9.7.6 lists common variations on this tactic.

Atomic conversions are either based on direct computation rules or can be created from lemmas and hypotheses. Conversions are composed using higher order functions called *conversionals* in a way similar to the way in which tactics are composed using tacticals. One set of conversionals are for controlling the sequence in which atomic conversions are applied to the subterms of a term.

Conversionals rely for their correct functioning on a variety of different kinds of lemmas being proven about the relations r and the terms making up any clause being rewritten; lemmas are required that state reflexivity, transitivity and symmetry properties of the relations r and congruence properties of the terms making up the clauses. These lemmas are described in Section 9.7.7.

9.7.5 Conversion Descriptions

The descriptions assume the conversion has been applied to an environment `e:env` and a term `t:term`. Types of arguments to conversions are:

c*	: convn	*type of conversions*
cs*	: convn list	
e*	: env	*environment*
i j	: int	*hypothesis or clause indices*
addr	: int list	*subterm address*
a	: tok	*name of abstraction*
name	: tok	*name of lemma or cached conversion*
names	: tok list	*list of names*
t t1 t2	: term	

9.7.5.1 Trivial Conversions

`FailC`	always fail
`IdC`	identity conversion

9.7.5.2 Lemma and Hypothesis Conversions

The conversions here derive rewrite rules from lemmas and hypotheses. Formulae that are lemma goals or hypotheses are treated as having the structure of either simple or general universal formulae (see Section 9.1.3 for definitions of these) . The consequents of these formulae must each be of form $a\ r\ b$ where r usually (but not always) an equivalence relation. Any relation r that is going to be recognized by the rewrite package must be initially declared to it. See Section 9.7.9 for details.

The conversions described here all rewrite in a left-to-right direction: they replace instances of a's by instances of b's. Each has a twin conversion that works right-to-left. These twins have the prefix `Rev` to their names. For example, `RevLemmaC` is the twin to `LemmaC`.

`LemmaC name`

Considering `name` as a simple universal formula with consequent $a\ r\ b$, `LemmaC` creates a conversion to rewrite instances of a to instances of b.

`HypC i`

Considering hyp `i` as a simple universal formula with consequent $a\ r\ b$, `HypC` creates a conversion to rewrite instances of a to instances of b.

`LemmaC` is an instance of

```
GenLemmaWithThenLC
 (n:int)
 (hints: (var # term) list)
 (Tacs:tactic list)
 (name: tok)
```

and `HypC` is an instance of

```
GenHypWithThenLC
 (n:int)
 (hints: (var # term) list)
 (Tacs:tactic list)
 (i: int)
```

The arguments to these conversions are as follows:

`n` indicates that the `n`th consequent of a general universal formula. If `-1` is used for `n` then the formula is always treated as simple. In particular, a $\Leftrightarrow$ relation is treated as the relation in the consequent rather than part of the structure of a general universal formula.

`hints` is for supplying bindings for variables in the lemma that Nuprl's matching routines can't guess on their own.

`Tacs` is used for *conditional* rewriting. Conditional rewriting is when the antecedents of a formula are checked for validity before the rewrite rule is used.

Tactics in `Tacs` are paired up with subgoals formed from instantiated antecedents and each tactic is run on its corresponding subgoal. The rewrite goes through only if every tactic completely proves its subgoal.

If there are fewer `Ts` than antecedents, `Ts` is padded on the left up to the length of the antecedents with copies of the head of `Ts`. If `Ts` is empty, then the rewrite goes through unconditionally.

`name` is the name of the lemma.

`i` is the number of a hypothesis. As with tactics, hypotheses are considered to be numbered positively from the beginning of the list onwards, and numbered negatively from the end backwards. Occasionally, negative numbers can have unexpected results. See below for an explanation.

Other useful specializations of `GenLemmaWithThenLC` and `GenHypWithThenLC` are:

```
GenLemmaC n name        =  GenLemmaWithThenLC n [] [] name
LemmaWithC hints name   =  GenLemmaWithThenLC (-1) hints [] name
LemmaThenLC Tacs name   =  GenLemmaWithThenLC (-1) [] Tacs name

GenHypC n i             =  GenHypWithThenLC n [] [] i
HypWithC hints name     =  GenHypWithThenLC (-1) hints [] i
HypThenLC Tacs name     =  GenHypWithThenLC (-1) [] Tacs i
```

The hypothesis conversions described here derive their rewrite rules from *local* hypotheses in the environments that they are presented with on their first applications (see Section 9.7.8) for a description of what *local* hypotheses are). If the conversions are applied with conversionals such as `HigherC` or `NthC` that start applying their argument conversion at the top of a term, then the local hypothesis list is always the same as the hypothesis list of the sequent containing the clause being rewritten.

Older versions of the hypothesis conversions required that the environment from which the rewrite rule was to be captured be provided as a seperate argument to the conversion. This turned out in most instances to be rather clumsy.

9.7.5.3 Atomic Direct-Computation Conversions

These low level conversions are not usually invoked directly by the user. However, there are cases when they turn out to be useful.

`TagC tagger`	Do forward computations indicated by tags in `(tagger t)`. Fail if tagger fails. tagger should be simple. e.g. `tag_redex`
`RedexC`	If `t` is a primitive redex, contract it.
`ExtractC names`	If `t` is the extract term of a theorem listed in `names` then expand it.
`AnyExtractC`	If `t` is any extract term, then expand it.
`UnfoldTopAbC`	If `t` is an abstraction, unfold it.
`UnfoldsTopC as`	If `t` is an abstraction with opid listed in `as`, unfold it.
`UnfoldTopC a`	= `UnfoldsTopC [a]`
`FoldsTopC as`	Try to fold an instance of an abstraction whose definition is given in a library object named in `as`.
`FoldTopC a`	= `UnfoldsC [a]`
`RecUnfoldTopC a`	If `a` is a recursively defined term, then unfold the recursive definition.
`RecFoldTopC a`	Try to fold an instance of the recursively defined term `a`.

`RecUnfoldTopC` and `RecFoldTopC` work only with recursive definitions that have been introduced with the `recdef` function. See Section 10.2.4 for more details.

9.7.5.4 Attributed Abstractions

Abstractions can be grouped by adding attributes (sometimes called conditions) to abstraction objects. An atomic conversion for unfolding abstractions that is sensitive to attributes is:
`AUnfoldsTopC` (*attrs* : `tok list`)

If applied to a term t that has any of the attributes *attrs*, then unfold it.

9.7.5.5 Abstract Redices

It is frequently useful to augment the primitive redices that the system recognizes with *abstract redices*, primitive redices that are buried under abstractions. For example, the first and second projection functions for pairs are abstractions:

```
*A pi1                         t.1 ==  let <x,y> = t in x
*A pi2                         t.2 ==  let <x,y> = t in y
```

The term `<"a","b">.1` is an abstract redex. It contracts to the term `"a"`.

The basic conversion for contracting both primitive and abstract redices is `AbRedexC`. Abstract redices are added to a table that `AbRedexC` accesses using the function

```
add_AbReduce_conv
  ( id:tok)
  ( c:conv) ,
```

where *id* is the opid of the outermost term of the redex, and c is a direct-computation conversion for contracting instances of the redex. Note that if the outermost term is an `apply` term, then the *id* is taken from the term at the head of the possibly-iterated application.

Instances of `add_AbReduce_conv` are usually included in ML objects positioned immediately after the definitions of non-canonical abstractions. For example, the declaration of the redex for the `pi1` term could be

```
*M pi1_eval
               let pi1_evalC =
                 SimpleMacroC `pi1_evalC`
                 ⌈<a, b>.1⌉ ⌈a⌉ ``pi1``
               ;;

               add_AbReduce_conv `pi1` pi1_evalC ;;
```

An alternative method for indicating an abstract redex is to associate a 'reducible' attribute with an abstract non-canonical term. The `AbRedexC` conversion on being applied to a term first unfolds all reducible abstractions at the top level of the term tree before further analyzing the term to see if it is a redex. When this method is applicable, it is more concise than using `add_Reduce_conv`. A reducible attribute is associated with a term using the function

```
add_reducible_ab : tok -> unit
```

It takes as its token argument the opid of the term.

9.7.5.6 Reduction Strengths and Forces

Cases come up when it is desirable to have some redices contracted but not others. To this end, an option is provided for specifying the *strength* of a redex, and a variant of the `Reduce` tactic allows for a *force* to be specified for reducing with. Strengths are forces are drawn from a partial order. A redex is only contracted when the reduction force applied to it is equal or greater than its strength.

A strength is associated with a redex in two ways:

- The strength is directly associated with the reduction rule for the redex.
- The strength is associated with the canonical term that is the principal argument of the non-canonical term of the redex.

Strengths and forces are ML tokens. The current supported strengths, in increasing order of strength, are

`‘1‘` beta redices

`‘2‘` other primitive redices

`‘3‘` abstract redices recursive

`‘4‘` abstract redices non-recursive

`‘6‘` module projection funs with coercion arguments. For example, the redex `grp_op(add_grp_of_rng(r))` which contracts to `rng_plus(r)` has this strength.

`‘7‘` functions creating module elements from concrete parts.

`‘1‘` quasi-canonical redices. An example is the `set_car` term from the `sets_1` theory when it has list and product discrete-set constructors as its principal argument. These constructors can also be found in the `sets_1` theory.

`‘9‘` irreducible terms.

Contraction conversions that should be sensitive to the force of reduction can be added to the abstract redex table using the function

```
add_ForceReduce_conv
  ( id:tok)
  ( c:tok -> conv) .
```

The token argument that the conversion c takes is for the force with which reduction is being attempted.

Strengths can be associated with canonical terms using the function

```
note_reduction_strength
   opid : tok
   strength : tok .
```

The basic conversion for reducing with force F is `ForceRedexC (`F`:tok)`. The conversion `AbRedexC` always reduces with maximum force.

9.7.5.7 Composite Direct Computation Conversions

`PrimReduceC`	= `RepeatC (SweepUpC RedexC)`
`UnfoldsC as`	= `SweepUpC (UnfoldTopsC as)`
`UnfoldC a`	= `UnfoldsC [a]`
`FoldsC as`	= `SweepUpC (FoldTopsC as)`
`FoldC a`	= `FoldsC [a]`
`SemiNormC as`	= `SweepDnC (RepeatC (UnfoldsC as)) ANDTHENC PrimReduceC`
`RecUnfoldC a`	= `SweepUpC (RecUnfoldTopC a)`
`RecFoldC a`	= `SweepUpC (RecFoldTopC a)`
`ReduceC`	= `Repeat (SweepDnC AbRedexC)`

9.7.5.8 Macro Conversions

`MacroC name c1 t1 c2 t2`

`MacroC` will rewrite an instance of `t1` to the corresponding instance of `t2` using forward and reverse computation steps. For example, a macro conversion might unfolds an abstraction, unroll a recursive or inductive primitive term, and then fold up an abstraction. Specific examples can be found in the standard libraries - Look at the definitions of non-canonical abstractions. Specifically, look at `ycomb_unroll` or `pi1_eval`.

`c1` and `c2` must be direct computation conversions which rewrite the pattern terms `t1` and `t2` respectively to the same term. `MacroC` uses second-order matching when matching instance terms against `t1`. Also, any parameter variables in `t1` will also be used in the match. `name` is used in the conversion's failure token.

For examples of the use of `MacroC` look at the `length_unroll` object in the `list_1` theory.

`SimpleMacroC name t1 t2 as`	=	`MacroC name (SemiNorm as) t1 (SemiNorm as) t2`
`FwdMacroC name c t`	=	`MacroC c t IdC (apply_conv c t)` `apply_conv` returns the term resulting from applying `c` to `t`

9.7.5.9 Conversionals

`c1 ORELSEC c2`	apply `c1`. If `c1` fails, apply `c2`
`TryC c`	= `c ORELSEC IdC`
`c1 ANDTHENC c2`	apply `c1`. If `c1` succeeds then apply `c2`. Otherwise fail
`c1 ORTHENC c2`	= `(c1 ANDTHENC TryC c2) ORELSEC c2`
`ProgressC c`	apply c, but fail if result same as IdC c
`RepeatC c`	= `TryC (ProgressC c ANDTHENC RepeatC c)`
`Repeat1C c`	= `c ANDTHENC RepeatC c`
`RepeatForC n c`	apply `c` `n` times.
`FirstC cs`	= `c1 ORELSEC ... ORELSEC cn` (Fail if `cs` = `[]`, `c1` if `cs` = `[c1]`)
`SomeC cs`	= `c1 ORTHENC ... ORTHENC cn` (Fail if `cs` = `[]`, `c1` if `cs` = `[c1]`)
`AllC cs`	= `c1 ANDTHENC ... ANDTHENC cn` (`IdC` if `cs` = `[]`, `c1` if `cs` = `[c1]`)
`IfC p c`	= if `p t` then `c` else `FailC`
`SubIfC q c`	apply `c` to selected subterms in left to right order. apply to `i`th subterm if `(q t i)` is true.
`SubC c`	= `SubIfC (\t i.true) c`
`NthSubC n c`	= `SubIfC (\t i. i = n) c`
`AddrC addr c`	apply `c` to addressed subterm
`NthsC ns c`	Walk `t` in preorder order, tentatively applying `c`, but only doing conversions on the successes of `c` numbered in `ns`. Avoid walking into subterms of any converted subterm.
`NthC n c`	= `NthsC [n] c`
`HigherC c`	= `c ORELSEC SubC (HigherC c)`
`LowerC c`	= `SubC (LowerC c) ORELSEC c`
`SweepDnC c`	= `c ORTHENC SubC (SweepDnC c)`
`SweepUpC c`	= `SubC (SweepUpC c) ORTHENC c`
`TopC c`	= `HigherC (Repeat1C c)`
`DepthC c`	= `SweepUpC (Repeat1C c)`

9.7.6 Applying Conversions

`Rewrite (c:convn) (i:int) : tactic`

Apply conversion `c` to clause `i`. The subgoal with the result of the conversion is always labelled `main`. The rest have various labels that all fall into the `aux` subgoal label class.

`RW` $=_{def}$ `Rewrite`
`RWH c` $=_{def}$ `RW (HigherC c)`
`RWU c` $=_{def}$ `RW (SweepUpC c)`
`RWD c` $=_{def}$ `RW (SweepDnC c)`
`RWN n c` $=_{def}$ `RW (NthC n c)`
`RWAddr addr c` $=_{def}$ `RW (AddrC addr c)`

`RewriteType (c:convn) (i:int) : tactic`

Apply conversion `c` to type of member or equality term in clause `i`. The subgoal with the result of the conversion is always labelled `main`. The rest have various labels that all fall into the `aux` subgoal label class. The advantage of this tactic over the usual `Rewrite` is that this generates simpler well-formedness goals. In particular, this tactic generates no well-formedness goals involving the equands of the equality or the element of the member term.

`RWT` $=_{def}$ `RewriteType`

`apply_conv (c:convn) (t:term) : term`

`apply_conv` evaluates a conversion with an empty environment. It is very useful for testing conversions.

9.7.7 Lemma Support

The rewrite package must have access to several kinds of lemmas in order to construct justifications for rewrites. This section describes those lemmas.

Note that for order relations, one only needs lemmas for one direction. For example, one doesn't require both the lemma

$$\vdash \ \forall a, b, c.\ a \leq b \Rightarrow b \leq c \Rightarrow a \leq c$$

and

$$\vdash \ \forall a, b, c.\ a \geq b \Rightarrow b \geq c \Rightarrow a \geq c$$

If Nuprl finds a lemma missing in the course of constructing a rewrite justification it prints out an error message suggesting the kind and structure of the missing lemma. After entering it, you need to evaluate the function

`initialize_rw_lemma_caches : unit -> unit`

on argument `()` in the ML Top Loop; for efficiency reasons, the rewrite code caches information about these lemmas and hasn't been set up yet to automatically update caches after changes to the available lemmas in the library.

9.7.7.1 Functionality Lemmas

Functionality lemmas give congruence and monotonicity properties of terms. They are required by the `SubC` conversional to construct tactic justifications for the rewrite of terms based on the tactic justifications for rewrites of the immediate subterms of those terms.

A functionality lemma for a term with operator *op* should have the form

$$\begin{array}{l}\forall z_1{:}S_1 \ldots z_k{:}S_k.\ \forall x_1, y_1{:}T_1, \ldots, x_n, y_n{:}T_n.\ A_1 \Rightarrow \ldots \Rightarrow A_m \\ \ \Rightarrow x_1\ r_1 y_1 \Rightarrow \ldots \Rightarrow x_n r_n y_n \Rightarrow op(x_1; \ldots; x_n)\ R\ op(y_1; ...; y_n)\end{array}$$

where $k \geq 0$ and $m \geq 0$. The $\forall$'s and A's can be intermixed, but the antecedents containing the r_i must come afterward and be in the same order as the subterms of *op*.

The `SubC` conversional finds functionality lemmas in the library by assuming that a naming convention has been followed. Specifically, the functionality lemmas in the library for operator *op* should be named *opid*_`functionality`[_*index*] where *opid* is the opid of *op* and _*index* is an optional suffix, used to distinguish lemmas when there is more than one for a given op.

Functionality lemmas are not explicitly needed when the r_i and r are all Nuprl's equality. In this case the functionality information can be derived from the well-formedness lemma for *op*.

If *op* binds variables in its subterms, then those same variables should be bound by universal quantifiers wrapped around the appropriate r_i antecedents. For example, the lemma for functionality of $\exists$ with respect to the $\Leftrightarrow$ relation is:

$$\begin{array}{ll} \vdash & \forall A_1, A_2{:}\mathbb{U}_i \\ & \forall P_1{:}A_1 \rightarrow \mathbb{P}_i\ \forall P_2{:}A_2 \rightarrow \mathbb{P}_i \\ & A_1 = A_2 \in \mathbb{U}_i \\ & \Rightarrow (\forall x{:}A_1 P_1[x] \;\Leftrightarrow\; P_2[x]) \\ & \Rightarrow \exists x{:}A_1.\ P_1[x] \;\Leftrightarrow\; \exists x{:}A_2.\ P_2[x] \end{array}$$

When more than one functionality lemma is created for a given operator, they must be ordered with the specific $r_1 \ldots r_n$ first. `SubC` searches for functionality lemmas in the order in which they appear in the library and if this recommended order is not followed then it might pick up the wrong lemma.

9.7.7.2 Transitivity Lemmas

Transitivity lemmas give transitivity information for rewrite relations. They are used to construct the tactic justification in the `ANDTHENC` conversional.

Transitivity lemmas should be of form:

$$\forall z_1{:}S_1 \ldots z_k{:}S_k.\ \forall x_1, x_2, x_3{:}T.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r_a\ x_2 \Rightarrow x_2\ r_b\ x_3 \Rightarrow x_1\ r_c\ x_3$$

where $k \geq 0$ and $m \geq 0$. For now there is a restriction that r_c should be the weaker of r_a and r_b.

`ANDTHENC` finds transitivity lemmas in the library by assuming that a naming convention has been followed. The lemmas must be named *opid-of-*r_c_`transitivity`_*index*, where the _*index* is optional, and is only needed to distinguish lemmas if there is more than one for a given r_c. A transitivity lemma is not needed for equality.

9.7.7.3 Weakening Lemmas

Weakening lemmas should have form

$$\forall z_1{:}S_1 \ldots z_k{:}S_k.\ \forall x_1, x_2{:}T.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r_a\ x_2 \Rightarrow x_1\ r_b\ x_2$$

where $k \geq 0$, $m \geq 0$ and r_b is some weaker relation than r_a.

Lemmas in the library must be named *opid-of-*r_b_`weakening`_*index*, where the _*index* is optional, and is only needed to distinguish lemmas if there is more than one for a given r_b.

Currently weakening lemmas are required for all reflexive relations r_b with r_a being equality. They extend the usefulness of the transitivity and functionality lemmas.

9.7.7.4 Inversion Lemmas

Inversion lemmas should have form

$$\forall z_1{:}S_1 \ldots z_k{:}S_k.\ \forall x_1, x_2{:}T.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r\ x_2 \Rightarrow x_2\ r\ x_1$$

where $k \geq 0$ and $m \geq 0$. Inversion lemmas in the library must be named *opid-of-r*`_inversion`.

Inversion lemmas are required for equivalence relations, but not equality or order relations. They are used by the `Rev*` atomic conversions, and in conjunction with the weakening, transitivity, and functionality lemmas when these lemmas mix order and equivalence relations.

9.7.8 Environments

An *environment* is a list of propositions and declarations of variable types and that are being assumed. The environment of the conclusion of a sequent is the list of hypotheses of the sequent. The environment of a hypothesis is all the hypotheses to the left of it. We can also talk about *local* environments of subterms of sequent clauses. For example, in the sequent

$$x_1{:}H_1,\ \ldots, x_n{:}H_n\ \vdash\ \forall y{:}T.\ B \rightarrow C$$

the local environment for subterm C of the conclusion is

$$x_1{:}H_1,\ \ldots, x_n{:}H_n,\ y{:}T,\ B$$

The rewrite conversionals keep track of the local environment each conversion is being applied in, and every conversion takes as its first argument an `e` of type `env` which supplies this local information.

The environment information is used by conversions in three ways by the atomic lemma and hypothesis conversions.

- Declarations in the environment are used to infer types which help to complete matches. (See Section 9.1.7).
- Environments are used to form the subgoals that have to be discharged for conditional lemma rewrites to go through. For example, if C in

  $$x_1{:}H_1,\ \ldots, x_n{:}H_n\ \vdash\ \forall y{:}T.\ B \rightarrow C$$

  is rewritten by a rewrite rule based on the lemma

  $$\vdash\ \forall z{:}T.\ A_z \Rightarrow t_z = t'_z$$

  and the variable z in the lemma is bound to a term s by the match of C against t, then the subgoal which has to be proven for the rewrite rule to be valid is

  $$x_1{:}H_1,\ \ldots, x_n{:}H_n,\ y{:}T, B,\ \vdash\ A_s$$

- The hypothesis conversions access the hyp list via environment terms.

Currently the system must be told explicitly how the environment is extended when it descends to the subterms of a term. Built in is knowledge of the $\forall$, $\exists$ and $\Rightarrow$ terms. The system assumes other terms do not modify the environment, unless otherwise told by the user. see the `env.ml` file for further details.

9.7.9 Relations

9.7.9.1 Introduction

The rewrite package supports rewriting with respect to both primitive and user-defined equivalence relations. Some examples are:

- $\sim$, the computational equality relation,
- $\cdot = \cdot \in \cdot$, the primitive equality relation of the type theory,
- $\Longleftrightarrow$, if and only if,
- $\cdot = \cdot \mathsf{mod} \cdot$, equality on the integers, modulo a positive natural,
- $=_q$, equality of rationals represented as pairs of integers,
- $\equiv$, the permutation relation on lists.

The package also supports 'rewriting' with respect to any relation that is transitive but not necessarily symmetric or reflexive. This needs a bit of explaining. Proofs involving transitive relations and monotonicity properties of terms can be made very similar in structure to those involving equivalence relations and congruence properties.

For example, consider the following proof step that came up Forester's development of real analysis in Nuprl [**?**].

$i{:}\mathbb{N}^+$
$j{:}\mathbb{N}^+$
$f{:}\mathbb{N}^+ \to \mathbb{N}^+$
$\mathtt{mono}(f)$
$\vdash$
$1/fi +_q 1/fj \;\; \leq_q \;\; 1/i +_q 1/j$

```
BY RWH (RevLemmaC `monotone_le`) 0
```

$\vdash$
$1/i +_q 1/j \;\; \leq_q \;\; 1/i +_q 1/j$

Here, the definition $\mathtt{mono}(f)$ is:

$$\mathtt{mono}(f) =_{def} \forall a, b{:}\mathbb{N}^+.\ a < b \Rightarrow f\ a < f\ b$$

and the theorem `monotone_le` is:

$$\vdash \forall f{:}\mathbb{N}^+ \to \mathbb{N}^+.\ \mathtt{mono}(f) \Rightarrow \forall n{:}\mathbb{N}^+.\ n \leq f\ n$$

The tactic `RWH` c i tries to apply the conversion c once to each subterm of clause i of the sequent and the conversion `RevLemmaC` *name* converts lemma *name* into a right-to-left rewrite rule. Other examples of monotone rewriting can be found in Section **??**.

It is interesting to note that logical implication $\Rightarrow$ can be treated a rewrite relation, since it is transitive. When it is, we have a generalization of forward and backward chaining.

For each user-defined relation, the user provides the rewrite package with lemmas about transitivity, symmetry, reflexivity and strength (a binary relation R over a type T is stronger than a relation R' over T if for all a and b in T, the relation $a\ R\ b$ implies that $a\ R'\ b$). These lemmas are used by the package for the justification of rewrites (see Section 9.7.7).

The user also provides a declaration in an ML object that identifies relation families and extra properties of relations. These declarations are described in the next section.

9.7.9.2 Declaring Relations

The rewrite package treats rewrite relations as first-order terms: the two principal arguments of relations are expected to be supplied as subterms rather than by application. If a relation term also takes additional parameters as subterms, these should always be positioned before the principal argument subterms. For example, the $t = t' \in T$ equality relation has T as a parameter and has logical structure `equal(`T`;`t`;`t'`)`.

Relations are most commonly typed-valued, but boolean-valued relations are also accepted by the rewrite package. When a boolean-valued relation is to be used in a context where a type-valued relation is expected, the relation should be wrapped in the `assert` abstraction which converts boolean-valued expressions to type-valued ones.

Equivalence relations should be declared by an invocation of

```
declare_equiv_rel
   rnam : tok
   stronger-rnam : tok
=
  () : unit
```

This declares the term with opid *rnam* to be an equivalence relation, and the term with opid *stronger-rnam* to be an immediately stronger equivalence relation. Commonly, *stronger-rnam* will be `` `equal` ``. Most often, there will be only one such declaration for each *rnam*. However, multiple declarations for a single *rnam* are sometimes needed and are quite acceptable.

Order relations are grouped into *relation families*. A family is a lattice of order and equivalence relations of form:

$$\begin{array}{ccccccccc} & & \leq & & & & \geq & & \\ & / & & \backslash & & / & & \backslash & \\ < & & & & = & & & & > \end{array}$$

where weaker relations are higher in the lattice, and relations within a family satisfy

$$\begin{aligned} a < b &\iff b > a \\ a \leq b &\iff b \geq a \\ a = b &\iff a \leq b \wedge b \leq a \\ a < b &\iff a \leq b \wedge \neg(b \leq a) \end{aligned}$$

The converse of both strict and non-strict order relations R should always be defined directly in terms of R. For example, the definition of the abstraction `rev_implies` is

$$P \Leftarrow Q =_{def} Q \Rightarrow P$$

The package assumes that order relations can be inverted by folding and unfolding such definitions.

Family should be declared using an invocation of

```
declare_rel_family
  lt : term
  le : term
  eq : term
  ge : term
  gt : term
=
  () : unit
```

Dummy terms (with structure `dummy()`, displaying as `?`, and entered using `dummy`) should be used as placeholders when a member of a family is missing. A notational abbreviation has been defined for such invocations which can be entered by the name `relfam`. It displays as

```
Relation Family
 <: [lt]
 ≤: [le]
 ≡: [eq]
 ≥: [ge]
 >: [gt]
```

Examples include for the standard order relations on the integers

```
Relation Family
 <: i < j
 ≤: i ≤ j
 ≡: i = j
 ≥: i ≥ j
 >: i > j
```

and for the 'divides' and 'associate' relation in a theory of cancellation monoids [**?**, **?**]

```
Relation Family
 <: a p| b in g
 ≤: a | b in g
 ≡: a ∼{g} b
 ≥: a |by b in g
 >: ?
```

If there are no defined order relations associated with an equivalence relation, then there is no need to include the equivalence relation in an order relation family. It is quite permissable (and not infrequent) that several order relation families share the same equivalence relation.

The partial order of strengths of relations is taken to be the reflexive transitive closure of the relation defined by the M-shaped graphs for each family and the equivalence relation declarations. Additional relations between order relations can be noted by using

`declare_order_rel_pair`
    stronger-rtm : `term`
    weaker-rtm : `term`

So far these methods of describing the partial order have been adequate, though the methods might well need reorganisation in the future.

These relation declarations should be inserted in ML objects that are positioned after the referred-to relations have been defined, but before they are used in any lemmas that might be accessed by the rewrite package.

In Nuprl4.1, the relation declarations

```
add_equiv_rel_info ( name:tok) ( rels:tok list) ;;
add_order_rel_info ( name:tok) ( inverse-name:tok) ( is-refl:bool) ( rels:tok list) ;;
```

were used. These are now obselete and should be replaced by the ones described above.

9.7.10 Justifications

There are two types of justifications.

Computational Justifications These indicate precise applications of the forward and reverse direct computation rules. They are comparatively very fast and generate no well formedness subgoals. The rewrite package uses these whenever possible.

Tactic Justifications These are more generally applicable. Extensive use is made of lemmas, and many well formedness subgoals are generated.

Conversions generating both types of justification can be freely intermixed; the system takes care of converting computational justifications to tactic justifications when necessary.

9.7.11 Substitution

Nuprl's logic has a couple of rules for carrying out simple kinds of substitutions. These rules are accessible through the tactics described here. Occasionally these rules are useful; they can generate fewer and easier-to-solve well-formedness goals than the rewrite package.

`Subst (eq:term) c`

`eq` should be a proposition of form $t1 = t2 \in T$. The effect of `Subst` is to replace all occurrences of $t1$ in clause `c` by $t2$. Three subgoals are generated; a `main` subgoal with the substitution carried out, a `wf` subgoal to prove functionality of the clause, and an `equality` subgoal to prove that $t1 = t2 \in T$.For example:

$H_1 \ldots H_n \ \vdash \ C_a$

`BY Subst` $'a = b \in T'$ `0`

`main`	$H_1 \ldots H_n \ \vdash \ C_b$
`wf`	$H_1 \ldots H_n, \ z{:}T \ \vdash \ C_z = C_z \in \mathbb{U}_*$
`equality`	$H_1 \ldots H_n \ \vdash \ a = b \in T$

where $\mathbb{U}_*$ is the inferred universe for clause `c`. Universe inference can be overridden by supplying an optional `universe` argument with the `Using` tactical.

`HypSubst i c`

Runs the `Subst` tactic using the equality proposition in hypothesis `i`. Generates a `wf` subgoal and a `main` subgoal.

`RevHypSubst i c`

As `HypSubst`, except that equality hypothesis is used right-to-left rather than left-to-right.

`SubstClause t c`

Replace clause `c` with term `t`. Generates a `main` subgoal and an `equality` subgoal.

9.8 Type Inclusion

`Inclusion i`

The `Inclusion` tactic solves goals of form

$$\ldots,\ i.x{:}T, \ldots \vdash x \in T'$$

or

$$\ldots,\ i.t \in T, \ldots \vdash t \in T'$$

where either types T and T' are equivalent or T is a proper subtype of T'. The specific kinds of relations between T and T' that the `Inclusion` currently handles are roughly:

- T and T' are the same once all soft abstractions are unfolded
- T and T' are both universe or prop terms and the level of T is always no greater than the level of T' for any instantiation of level variables.
- T and T' are each formed by using one or more subset types, and both have some common superset type. In this case `Inclusion` tries to show that the subset predicates (if any) of T' are implied by the subset predicates (if any) of T together with other hypotheses.
- T and T' have the same outermost type constructor. In this case, the inclusion goal is reduced to one or more inclusion goals involving the immediate subterms of T and T'. Currently works for function, product, union and list types.
- There is a lemma in the library stating that T is a subtype of T'.

`Inclusion` also solves similar goals where one or both of the membership terms are replaced by equality terms.

For the inclusion reasoning involving subset types to work, you need to supply information about abstractions involving subset types using the function `add_set_inclusion_info`. See the theory `int_1` for several examples of the use of this function.

9.9 Miscellaneous

`Cases [t1;...;tn]`

Does n-way case split. For example:

$\ldots \vdash C$

BY Cases $[t_1;\ldots;t_n]$

assertion: $\ldots \vdash t_1 \vee \ldots \vee t_n$
$\ldots t_1 \vdash C$
$\vdots$
$\ldots t_n \vdash C$

`GenConcl '`$t = v \in T$`'`
v should be a variable. Generalizes occurrences of t as subterms of the concl to variable v. Adds new hypotheses declaring v to be of type T and stating that $t = v \in T$.

`ApFunToHypEquands (`x`:var) (`v_x`:term) (`V_x`:term) (`i`:int)`
If hypothesis `i` is of form $a = b \in T$, then this tactic applies the function $\lambda x.v_x$ to the terms a and b to give a new hypothesis in the subgoal labelled `main` of form $v_a = v_b \in V_a$. Also creates a couple of `aux` subgoals. See file `inclusion-tactics.ml` for details.

`Fiat`
If you about to give up hope on a theorem, don't despair. This tactic is guaranteed to provide satisfaction.[2]

9.10 Autotactics

The autotactics are used primarily for typechecking. `Trivial`
Does various steps of trivial reasoning, including.

- `NthHyp`
- `NthDecl`
- `Eq`
- `Contradiction` - Both P and $\neg P$ occur in hypothesis list.
- Concl is the term `True` or one of the hypotheses is either the term `False` or the term `Void`.

`Auto`
Repeatedly tries the following until no further progress is made.

- `Trivial`
- `GenExRepD`
- `MemCD` for member and `EqCD` on reflexive equality conclusions. Only works on non-recursive primitive terms.
- `Arith`

[2]This tactic uses Nuprl's *because* rule. The use of this rule should be regarded as experimental. Despite much hard work, neither Stu nor Doug have yet succeeded in proving it valid according to any of their semantics.

- Arithmetic equality reasoning, in case concl is $a = b \in T$ where a and b arithmetically simplify to same. (By arithmetically simplify, we mean simplify subterms which involve the basic arithmetic operators $+, -, *, /$ and rem[3].)
- If concl is $a \in T$ or $a = b \in T$ where T is subset of the integers, then open up T.

`SIAuto`
: Like `Auto` but also tries using the `SupInf` tactic.

`Auto'`
: Like `Auto` but uses the `SupInf'` tactic instead of the `Arith` tactic.

`Auto` and its variants frequently encounter the same goals over and over again, so

9.11 Transformation Tactics

`PrintTexFile (name:string)`
: `name` should be a filename without extension. Two files are created. `name.prl` is a file which can be viewed by an appropriate version of emacs running with one of Nuprl's 8-bit fonts. `name.tex` is a self-contained file suitable for input to LaTeX.

`Mark (a:tok)`
: `Mark` stores the proof tree at and below the point in the proof where it is invoked in the proof register named `a`.

`Restore (a:tok)`
: Restore the proof stored in proof register `a` by a previous `Mark`.

`Copy (a:tok)`
: Run all the tactics associated with the proof stored in proof register `a`. `Copy` is useful if you want to copy a pattern of reasoning used in one part of a proof to another part. `Copy` can also be used to copy from one proof to another.

`Z`
: Stores the current proof in the ML variable `pf`. This is very useful when debugging tactics.

9.12 Constructive and Classical Reasoning

9.12.1 Constructive Reasoning

The constructivity of Nuprl's logic manifests itself in two main ways with the tactics:

1. For any proposition P, the goal $P \vee \neg P$ is not in general provable.
2. When applying the `D` tactic to a hypothesis that has a set term outermost, the predicate part of the set term becomes a *hidden* hypothesis. For example:

$$\ldots i.x{:}\{y{:}T|P_y\},\ \ldots\ \vdash\ \ldots$$

`BY D`i

$$\ldots i.x{:}T,\ [i+1].P_x,\ \ldots\ \vdash\ \ldots$$

[3]Currently simplification involving / and *rem* isn't working

Here, the [] surrounding the hypothesis number $i+1$ indicate that this hypothesis is hidden. A hidden hypothesis is not immediately usable though there are ways in which it might become usable later in a proof.

Tactics to simplify dealing with these issues are described in the next two sections.

9.12.2 Decidability

Many useful instances of $P \vee \neg P$ are provable constructively and the `ProveDecidable` tactic is set up to construct these proofs in a systematic way. To discuss it, we first introduce the abstraction:

$$\texttt{decidable:}\quad \mathsf{Dec}(P) \quad =_{def} \quad P \vee \neg P$$

which can be found in the `core_2` theory. It turns out that the property $\mathsf{Dec}(P)$ can be inferred for many P from knowing that $\mathsf{Dec}(Q)$ for the immediate subterms Q of P. `ProveDecidable` takes advantage of this fact and attempts to prove goals of the form

$$\ldots \vdash \mathsf{Dec}(P)$$

by backchaining with any lemmas in the Nuprl library that have names with prefix `decidable__`. (Note the *two* underscores.) There are many examples of such lemmas in the `core_2` theory. `ProveDecidable` is usually invoked via the `Decide` tactic:

`Decide (`Q`:term)`
: Used to case-split on whether proposition Q is true or false. Generates two `main` subgoals; one with the new assumption Q and the other with the new assumption $\neg Q$. `Decide` also creates a subgoal $\ldots \vdash \mathsf{Dec}(Q)$ and immediately runs the `ProveDecidable` tactic on this subgoal. `ProveDecidable` generates only subgoals with labels in the `aux` class. If `ProveDecidable` fails then `Decide` fails too. To understand why, use the tactic `Assert` to assert $\mathsf{Dec}(Q)$ and run the `ProveDecidable1` tactic to try and prove this assertion. `ProveDecidable1` will generate subgoals with label `decidable?` to indicate those components of Q that it couldn't prove were decidable. Use `ProveDecidable1` rather than `ProveDecidable`; `ProveDecidable` fails when it sees such subgoals.

9.12.3 Squash Stability and Hidden Hypotheses

A hidden hypothesis P in a sequent σ can be unhidden if one of two conditions are met:

1. The proposition P is squash stable.
2. The conclusion of σ is squash stable.

A proposition is *squash stable* if it is possible to figure out what its computational content is, given that you know that some computational content exists (in the classical sense). The *computational content* of a proposition is some term that inhabits the proposition when it is considered as a type.

The squash stable predicate is defined in the `core_2` theory as follows:

$$\texttt{sq\_stable:}\quad \mathsf{SqStable}(P) \quad =_{def} \quad \downarrow P \Rightarrow P$$

The proposition $\downarrow P$ (read 'squash P') is considered true exactly when P is true. However, P's computational content when true can be arbitrary whereas $\downarrow P$'s computational content when true can only be the trivial constant term that inhabits the unit type. ($\downarrow P$ is defined as $\{x{:}\mathsf{Unit}|P\}$ where x does not occur free in P).

As with decidability, it turns out that the property $\mathsf{SqStable}(P)$ can be inferred for many P from knowing that $\mathsf{SqStable}(Q)$ for the immediate subterms Q of P. It is also true that $\mathsf{Dec}(P) \Rightarrow \mathsf{SqStable}(P)$ for any P. The tactic `ProveSqStable` takes advantage of these facts and attempts to prove goals of the form

$$\ldots \vdash \mathsf{SqStable}(P)$$

by backchaining with any lemmas in the Nuprl library that have names with the prefixes `sq_stable__` or `decidable__` (note the *two* underscores in each case). There are many examples of such lemmas in the `core_2` theory.

Since `ProveSqStable` can be rather slow, it isn't called by the `D` tactic when `D` is applied to a set type hypothesis $\{x{:}T|P_x\}$. However, `D` does check the sequent conclusion for trivial ways in which it might be squash stable.

The `D` tactic does recognise *property lemmas* for abstractions that are wrapped around set types. Property lemmas state that particular set type predicates are squash stable. If `D` is applied to an abstraction wrapped around a set type and there is a property lemma for the abstraction, then the predicate of the set type is always added unhidden to the hypothesis list.

Consider some abstraction $A_{\overline{x}}$ where the $\overline{x}$ are variables that have been slotted in for the immediate subterms of A. Say that $A_{\overline{x}}$ unfolds to $\{y{:}T_{\overline{x}}|P_{\overline{x},y}\}$. Then the property lemma for A should have form

$$\vdash \forall \overline{x}{:}\overline{T}.\ \forall z{:}A_{\overline{x}}.\ P_{\overline{x},z}$$

and should be named *opid*`_properties` where *opid* is the opid of A. Examples of properties lemmas can be found in the theory `int_1`. Property lemmas can often be completely proven using the tactic `ProvePropertiesLemma`.

Tactics related to unhiding are as follows:

`UnhideSqStableHyp i`
: Hypothesis `i` should be a hidden hypothesis. This tactic tries to prove the hidden hypothesis squash stable using `ProveSqStable`.

`UnhideAllHypsSinceSqStableConcl`
: This tactic tries to prove the conclusion squash stable using `ProveSqStable`. If it succeeds in this, all hidden hypotheses are unhidden.

`Unhide`
: Tries to unhide hidden hypotheses, first by checking whether the conclusion is squash stable and then, if this fails, by checking whether each hidden hypothesis is squash stable.

`AddProperties` i
: Hypothesis i should be declaration of form A or a proposition of form $t \in A$ or $t = t' \in A$ where in either case A is an abstraction with a properties lemma. `AddProperties` adds the predicate part of the set type underlying A as a new hypothesis immediately after i. It does not change hypothesis i.

9.12.4 Classical Reasoning

To reason classically, you need to have as an explicit hypothesis

$\forall P{:}\mathbb{P}_i. P \vee \neg P$

It is best to have this hypothesis in the form of the `xmiddle` abstraction:

`xmiddle`: $\mathsf{XM}_i \;\; =_{def} \;\; \forall P{:}\mathbb{P}_i.\mathsf{Dec}(P)$

The `Decide` tactic recognizes whenever the `xmiddle` abstraction occurs as some hypothesis, and in this case is trivially able to justify decidability. Also, the `D` tactic on set types, maybe with abstractions wrapped around them, always yields an unhidden set predicate, whether or not there is an abstraction with a properties lemma.

If you want to prove a non-constructive theorem, it is simplest to add the `xmiddle` proposition as a precondition of the theorem, so that the theorem is of form $\vdash \mathsf{XM}_i \Rightarrow P$.

There are two common cases when in proving a part of a constructive theorem, classical reasoning becomes admissable. These cases, and a recommended method in each case for adding an `xmiddle` hypothesis are as follows:

1. If the conclusion is the *squashed exists* term $\downarrow \exists x{:}T.\ P_x$ and you are about to apply the tactic `With` t `(D 0)`. Squashed exists is defined in `core_1` as:

   `sq_exists`: $\downarrow \exists x{:}T.\ P[x] \;\; =_{def} \;\; \{x{:}T|P[x]\}$

   First use the tactic `AddXM`:

   $\ldots \vdash \downarrow \exists x{:}T.\ P_x$

   `BY AddXM 1 THEN With` t `(D 0)`

   `wf` $\quad \mathsf{XM}i \ldots \vdash \; t \in T$
   `main` $\quad \mathsf{XM}i \ldots \vdash \; P_t$
   `wf` $\quad \mathsf{XM}i \ldots x{:}T \; \vdash \; P_x \in \mathbb{P}_i$

   `AddXM 1` alone adds the hypothesis $\mathsf{XM}i$ as a hidden hypothesis, so there are no soundness problems here. $\mathsf{XM}i$ becomes unhidden in the first and third subgoals since here the conclusion is recognized as being trivially squash stable. $\mathsf{XM}i$ becomes unhidden in the second subgoal since from here on, any computational content in the proof cannot contribute to the computational content of the original goal of the theorem.

2. If the conclusion is squash stable. Again first run the tactic `AddXM 1`. If the conclusion is obviously squash stable, then $\mathsf{XM}i$ is added unhidden. If the conclusion is not obviously squash stable and $\mathsf{XM}i$ gets added hidden, you should then run the `Unhide` tactic.

The `AddXM` tactic assumes that the proposition $\downarrow (\forall P{:}\mathbb{P}_i.P \vee \neg P)$ is true; that is, the corresponding type is inhabited. This is a very reasonable assumption to make, but it is not true according to the semantics given for Nuprl's type theory in S. Allen's thesis.

9.13 Further Information

Consult the ML files. Start with `load-ml.ml` which loads all the other ML files.

Chapter 10

Theories

10.1 Theory Structure

The main directories containing theories are listed in Section 1.5. Each theory directory contains an ML file `theory-init.ml`. This file contains commands that tell Nuprl about the theories in the theory directory, including information about the dependencies of theories on one another. It also should contain comments that summarize the contents of each theory in the directory.

Theory directories should also contain up-to-date listings of each theory. Short listings are named *theory-name*`.prl` and long listings containing printouts of proofs are named *theory-name*`_long.prl`. These listings use characters from Nuprl's 8-bit character set and so are best viewed using an editor running with one of Nuprl's fonts. There are also self-contained LaTeX versions of the listings in files with `.tex` rather than `.prl` file-name extensions.

ML commands for creating, loading, editing, dumping and printing theories are described in Section 3.4.3.

10.2 Definitions

10.2.1 Structure

A definition in a Nuprl theory for a term with opid *opid* usually includes the following objects in the order in which they are listed here:

- A display form object, usually named *opid*`_df`, specifying how instances of the definition should be displayed. The right-hand-side of each clause in the display form definition shows the abstraction without any display forms. Its useful to look at this if you are confused as to the structure of a abstraction.

- An abstraction object, usually named *opid*, that specifies how the definition unfolds. You indicate to the `Unfold` tactic abstractions to unfold by giving the opids of the abstractions. However, the `Fold` tactic takes the names of the objects in which the abstractions are defined. When *opid* is used as the name of the abstraction object, the same name can be used for referring to an abstraction when folding and unfolding.

- A well-formedness lemma, usually named *opid*_wf, that helps Nuprl type-check the definition. Occasionally there is more than one well-formedness lemma, in which case the objects are distinguished by adding suffices to *opid*_wf.

Sometimes there are extra ML objects and lemmas associated with a definition. Definitions are not only used for the Nuprl object language; they are also in nearly all the structures that are edited using the Nuprl editor. No well-formedness lemma is applicable in these cases.

Definitions can be set up, by creating each object in turn and editing its contents from scratch. This is a rather laborious process. The following sections describe various ML functions that can be evaluated in Nuprl's ML Top Loop to more rapidly set up new definitions.

Nuprl abstractions cannot be recursive. However, recursive definitions can be introduced by using the Y combinator. See Section 10.2.4 for details.

10.2.2 Adding Untyped Definitions

The function

```
utdef
   lhs : term
   rhs : term
   place : string
=
  () : unit
```

creates definitions without typing lemmas. *lhs* is the new term constructor and *rhs* is the what it is defined as. *lhs* and *rhs* are used for the left-hand and right-hand sides of the abstraction for the new definition. *place* is a library position as described in Chapter 7. *lhs* is invariably a new term, so you usually will want to use the new term creation feature of the term editor to enter it (see Section 4.5.1). If the opid of the new definition is *id*, then `utdef` adds 2 new objects to the library starting at position *place*:

- a display object named *id*`_df` which defines a default display form for the *lhs* term,

- an abstraction object named *id*. *lhs* and *rhs* are used for the left-hand and right-hand sides of the abstraction.

`utdef` has no effect if an object with name *id* already exists. Often after running `utdef` you will want to customize the display form. For example you might add whitespace related formats to the display form left-hand side.

10.2.3 Adding Typed Non-Recursive Definitions

The function

```
def
   tdef : term
   place : string
=
  () : unit
```

creates definitions with typing lemmas. The term *tdef* should have structure

$$\forall x_1{:}T_1 \ldots x_k{:}T_k.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow \mathit{lhs} = \mathit{rhs} \in V$$

where $k \geq 0$ and $m \geq 0$. The term *lhs* is the new term being defined and the term *rhs* is what it is being defined as. All the free variables in *lhs* should be appropriately typed in the context $x_1{:}T_1 \ldots x_k{:}T_k$. *place* is a library position.

On evaluation of `def`, a display form object *opid*`_df`, an abstraction *opid*, and a typing lemma *opid*`_wf` are created. The typing lemma is constructed from the *tdef* term. It has form:

$$\forall x_1{:}T_1 \ldots x_k{:}T_k.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow \mathit{lhs} \in V$$

An attempt is made to prove it by unfolding the abstraction and running the `Auto` tactic. In most cases, this attempt succeeds in completely proving the lemma.

Note that the `so_apply` abstraction should be used for the second-order variables in the *lhs* and *rhs* terms. `def` takes care of converting these to second-order variables for the abstraction object.

When doing proofs, the basic tactics for folding and unfolding definitions are the `Fold` and `Unfold` tactics. See Chapter 9 for details.

Currently, the only kinds of parameter variables that can be used in typed definitions are level expression variables. This is because Nuprl doesn't currently allow any other kinds of parameter variables in lemma goals and in proofs. Hopefully this should be fixed soon.

An example invocation of `def` is:

```
M> def
 >  ⌜∀A,B:𝕌. ∀p:A × B.  pr_swp{}(p) = <p.2, p.1> ∈ B × A⌝
 >  "-test_1_end" ;;
```

This creates the library objects

```
*D pr_swp_df
                pr_swp(<p:p:*>)== pr_swp{}(<p>)
*A pr_swp                         pr_swp(p) ==  <p.2, p.1>
*T pr_swp_wf
                ∀A,B:𝕌. ∀p:A × B.  pr_swp(p) ∈ B × A
```

10.2.4 Adding Recursive Definitions

Nuprl's object language contains terms for doing recursion over types such as lists and integers. These terms can be awkward to use, and we recommend instead that all recursive definitions are built using the Y-combinator. The ML function `recdef` greatly simplifies constructing general recursive definitions using the Y-combinator. `recdef` also takes care of introducing a well-formedness lemma for the definition. Its usage is:

```
recdef
   tdef : term
   place : string
=
  () : unit
```

The term *tdef* should have structure

$$\forall x_1{:}T_1 \ldots x_j{:}T_j.\ \forall y_1{:}S_1 \ldots y_k{:}S_k.\ A_1 \Rightarrow \ldots \Rightarrow A_l \Rightarrow lhs = rhs \in V$$

where the term *lhs* has form

$$id\{p_1, \ldots, p_i\}(x'_1; \ldots; x'_{j'})\ y_1\ y_2\ \ldots\ y_k\ ,$$

$i, j, j', k, l \geq 0$, the variables x'_n are a subset of the variables variables x_n, and none of the variables y_n are free in any of the antecedent propositions A_n.

`recdef` allows you to create recursive definitions with both curried arguments $y_1\ \ldots\ y_k$ supplied by application, and subterm arguments $x'_1\ \ldots\ x'_{j'}$. The parameters $p_1\ \ldots\ p_i$ are specified as in abstraction definitions.

The *rhs* term should include at least one instance of the head of the application in *lhs*. That is, a term of form

$$id\{p_1, \ldots, p_i\}(t_1; \ldots; t_{j'})$$

where the p_n's are the same as in *lhs*. All the free variables in *lhs* should be appropriately typed in the context $x_1{:}T_1 \ldots x_j{:}T_j$. *place* is a library position.

On evaluation of `recdef`, four objects are added to the library.

- A display object named *id*`_df` which defines a default display form for the `id` term.
- an abstraction object named *id* which defines the abstraction

  $$id\{p_1, \ldots, p_i\}(x'_1; \ldots; x'_{j'})$$
  $$=_{def}$$
  $$\mathtt{Y}\ (\lambda f\ x'_1 \ldots x'_{j'} y_1 \ldots y_k.\ rhs[id\{p_1, \ldots, p_i\}(t_1; \ldots; t_{j'}) \mapsto f\ t_1 \ldots t_{j'}])\ x'_1\ \ldots\ x'_{j'}$$

- An ML object named *id*`_ml`. This contains an invocation of a ML function that updates caches that hold information conversions related to the definition. These conversions are described below. This function invocation is given the display form:

  lhs `==r` *rhs*

  and it serves to document the recursive definition. For this reason, the abstraction objects in recursive definitions are usually made invisible in theory listings.

- A theorem object named *id*`_wf` for a well-formedness theorem. The goal of the theorem is constructed from the *tdef* term. It has form

  $$\forall x_1{:}T_1 \ldots x_j{:}T_j.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow id\{p_1, \ldots, p_v\}(x_1; \ldots; x_j) \in y_1{:}S_1 \rightarrow \ldots y_k{:}S_k \rightarrow V\ .$$

  An initial tactic is executed on the goal of this theorem to set up a subgoal that is usually suitable for induction. You then need to go in and complete the proof of this theorem yourself.

As with def, so_apply abstraction should be used for the second-order variables in *lhs* and *rhs*. recdef takes care of converting these to second-order variables for the abstraction object.

The basic conversions associated with a recursive definition are RecUnfoldTopC and RecFoldTopC. If the definition has arguments provided by application, then additional conversions RecEtaExpC and RecEtaConC are defined to η=-expand and η-contract the definition. More information on these and related conversions can be found in Section 9.7.5.

An example invocation of recdef from the ML Top Loop is:

```
M> recdef
 >   ⌜∀S,T:𝕌. ∀f:S → T. ∀as:S List.
        cmap{}(f) as
        = case as of
            [] => []
            a::as' => (f a)::(cmap{}(f) as')
          esac
        ∈ T List⌝
 >    "-test_1_end"  ;;
```

This creates the following library objects:

```
*D cmap_df                    cmap(<f:f:*>)== cmap{}(<f>)
*A cmap
               cmap(f) ==
                 Y
                 (λcmap,as.
                   case as of
                     [] => []
                     a::as' => (f a)::(cmap as')
                   esac)
*M cmap_ml
               cmap(f) as
               ==r case as of
                     [] => []
                     a::as' => (f a)::(cmap(f) as')
                   esac
#T cmap_wf
               ∀S,T:𝕌. ∀f:S → T.
                 cmap(f) ∈ S List → T List
```

10.2.5 Adding Set Definitions

Often new types are defined as subsets of old types using Nuprl's *set* type. Usually, there are standard extra objects associated with set definitions. The function setdef is a variant on def that automatically creates default versions of these extra objects that are good enough for most purposes. It's usage is identical to that of def.

The extra objects created are:

- an ML object *id*_ml_inc. This tells Nuprl how to prove type-inclusion relationships between the old and new types.

- a properties lemma *id*_properties. This helps in accessing the set predicate information. The structure and use of properties lemmas is described in Section 9.12.3.

The properties lemma is automatically proven providing the set predicate is squash stable. In the event that you want to introduce a set definition with a predicate that isn't squash stable, it is useful to use instead a squashed version of the predicate; squashed predicates are always squash-stable.

In order to add these extra objects for existing definitions the function

```
add_inc_objs : tok -> unit
```

can be used. Its argument should be the *opid* of the definition.

An example invocation of `setdef` is

```
M> setdef
 >   ⌜∀T:𝕌. ∀n:ℕ.
        vec{}(T; n) = {as:T List| ||as|| = n}  ∈ 𝕌⌝
 >   "-test_1_end" ;;
```

and the objects created by this invocation are:

```
*D vec_df
              vec(<T:T:*>;<n:n:*>)== vec{}(<T>; <n>)
*A vec
              vec(T;n) ==  {as:T List| ||as|| = n}
*T vec_wf                   ∀T:𝕌. ∀n:ℕ.  vec(T;n) ∈ 𝕌
*M vec_ml_inc
              add_set_inclusion_info
                `vec` `list`
                AbSetDForInc Auto ;;
*T vec_properties
              ∀T:𝕌. ∀n:ℕ. ∀as:vec(T;n).  ||as|| = n
```

Sometimes, the predicate in a set definition can have a number of parts and it is desirable to have a property lemma for each part. The function

```
add_seperate_property_lemmas : tok -> unit
```

can be used to add such property lemmas. It assumes that *compound* and *basic* attributes have appropriately been added to the definitions of the abstractions used in the predicate.

10.3 Notational Abbreviations for ML

Nuprl abstractions can be used to provide notational abbreviations for ML. A couple of current abstractions for ML add common suffices to tactics.

- `(`T `...)` is an abbreviation for T `THEN Auto`. An editor command ⟨C-.⟩`t`, given when at a text cursor when editing ML, inserts this abbreviation and leaves a text cursor at the internal slot for the argument T.

- Similarly, `(`T `...a)` is an abbreviation for T `THENA Auto`, and the appropriate editor command is ⟨C-.⟩`a`.

10.4 Module Types

Support is provided for defining *module* types. These are essentially record types, where the type of each field of an instance can depend on previous fields of the instance. They are very useful for defining ADT's (abstract data types) and algebraic classes. Module types are allowed to have parameters. For example, an ADT for queues could be created that takes the type of queue elements as a parameter. Module types are currently implemented using Nuprl's Σ type.

An ML function `create_module` helps with setting up new module type definitions, adding projection functions, and updating the `AbReduce` tactic to recognize applications of the projection functions.

The `create_module` function should be used as follows:

1. Pick a name for your module; say *modname*.

2. Create an ML object `create_`*modname* to hold the `create_module` invocation. The invocation needs to be part of the library to ensure that the relevant computation rules are added each time the theory containing the module definition is loaded. The function only creates the definition objects themselves if they are initially absent. The invocation also serves to document the module. With the abstractions explained below, it presents a clearer and more succinct account of the module structure than do the definitions themselves that make up the module.

3. It is most elegant to use a few abstractions to pretty up the invocation. Remember, when you have a text cursor in a text sequence, a term slot can be opened up using ⟨C-u⟩, and when you have a term cursor at an empty term slot, a text sequence can be inserted using ⟨C-;⟩.

   The abstractions for modules currently reside in the `ml_1` theory.

   The abstraction holding the `create_module` invocation has the editor alias `classdecl`. Enter it in the ML object by opening up a term slot in the object's top level text sequence, and typing the name `classdecl`. By keeping the top level of the ML object a text sequence, you are free for example to add ML comments preceding the `create_module` invocation. Without fields filled in, the class declaration template should look like:

   ```
   Class Declaration for: [example]

     Long Name: [string]
     Short Name: [string]

     Parameters:
       [parms]

     Fields:
       [fields]

     Universe: [uni]
   ```

4. Enter *modname* for the `Long Name` and some abbreviation of this for the `Short Name`. The short name is used as a prefix for the names of the projection functions.

5. Initialize the `[parms]` and `[fields]` slots with empty text sequences.

6. Open up a term slot in the `Parameters` list for each parameter. Parameters are pairs of variable names and types. They should be entered using another abstraction with alias name `mlbd`. This initially looks like

```
[var] : [type]
```

where `[var]` is a text slot for the variable name, and `[type]` is a term slot for the corresponding type of the variable. If there are no parameters, just leave the empty text sequence in this slot.

7. Open up a term slot in the `Fields` list for each field. As with parameters, you can use the `mlbd` abstraction for the fieldname and field type. Alternatively, you can use the `mlbde` abstraction:

```
([example]) [var] : [type]
```

This has an extra term slot for an example of the projection function for that field. Initially, leave the term slot empty. Enter in the `[var]` slot the name of the field. The name of the projection function will be the name entered here with a suffix of the short module name. The type of a field can refer to constant types, the inhabitants of earlier fields or parameters.

Since the `mlbd` or `mlbde` terms are embedded in a text sequence, you can insert linebreaks in the sequence, by simply getting a text cursor in the sequence and typing $\boxed{\text{RETURN}}$.

8. Fill in the `[uni]` slot with an appropriate universe term. This should be the lowest universe that the module type inhabits. For example, if the module type itself involves no universes, then this probably will be $\mathbb{U}_1$; if the module type involves universe terms $\mathbb{U}_i$, then this will be $\mathbb{U}_{i+1}$, (commonly abbreviated to $\mathbb{U}'$).

Here is an example of a class definition that has been instantiated according to the procedure given so far:

```
Class Declaration for: [example]

  Long Name: action_set
  Short Name: aset

  Parameters:
     T : 𝕌

  Fields:
     ([example])  car : 𝕌
     ([example])  act : T → car → car

  Universe: 𝕌'
```

9. If you now check this object (use ⟨C-x⟩ch), the definitions for the module are created in the library. For the above example, the following definitions are created:

```
*D action_set_df
              action_set{<i:level>}(<T:T:*>)== action_set{<i>:l}(<T>)
*A action_set
              action_set{i}(T) ==  car:𝕌 × (T → car → car)
*T action_set_wf            ∀T:𝕌. action_set{i}(T) ∈ 𝕌'
*D aset_car_df              <a:aset:E>.car== aset_car{}(<a>)
*A aset_car                 a.car ==  a.1
*T aset_car_wf              ∀T:𝕌. ∀a:action_set{i}(T).  a.car ∈ 𝕌
*D aset_act_df              <a:aset:E>.act== aset_act{}(<a>)
*A aset_act                 a.act ==  a.2
*T aset_act_wf
              ∀T:𝕌. ∀a:action_set{i}(T).  a.act ∈ T → a.car → a.car
```

10. If desired, edit the display forms for the module type and for the projections. For example, the display forms could be changed to:

```
*D action_set_df
              ASet{<i:level>}(<T:T:*>)== action_set{<i>:l}(<T>)
              ASet(<T:T:*>)== action_set{i:l}(<T>)
*D aset_car_df              |<a:aset:*>|== aset_car{}(<a>)
*D aset_act_df
              Parens ::Prec(preop)::  ·<a:aset:E>== aset_act{}(<a>)
              ·== aset_act{}(<a>)
```

Here a couple of abbreviated display forms are defined for the the module type, one of which hides the level expression in the case that it is simply the level variable `i`. Standard notation is chosen for the carrier, and the action is denoted by a '·' character. To avoid clutter, a display form for the action is defined that hides the instance of the `action_set` module. This can nearly always be determined from context.

11. Fill in the `[example]` slots in the module declaration. These serve purely to document the module, and are discarded when the system unfolds the class declaration abstraction into raw ML code. In the `[example]` slot on the first line, you should put an instance of the module type, with the same parameter names as declared in the `Parameters` section. You might also insert a membership term or an `mlbd` term to indicate a prototypical element. For the above example, you could insert the term `s ∈ ASet(T)` indicating that `s` is a prototypical element.

    Fill in the example slots of the `mlbd` terms with examples of the relevant projection functions, operating on the prototypical element declared in the uppermost example slot. In the example, the terms `|s|` and `·s` could be used.

    With the given display form changes, and example terms, the final version of the example class definition looks like:

```
Class Declaration for: s ∈ ASet(T)

  Long Name: action_set
  Short Name: aset

  Parameters:
     T : 𝕌

  Fields:
     (|s|)  car : 𝕌
     (·s)  act : T ⟶ car ⟶ car

  Universe: 𝕌'
```

12. Exit the `create_`*modname* object.

Examples of class definitions can be found in many of the algebra theories.

Appendix A

The Lisp Debugger

You can get thrown into the Lisp debugger in several ways; for example if Lisp is interrupted, if a breakpoint was mistakenly left in the Nuprl code or if you hit a bug. The particular debugger appearance and commands given below are for Lucid Common Lisp. Other Lisps should be similar.

The initial message put out by the debugger should tell you what caused it to be invoked. To resume after an interrupt or breakpoint, enter:

`:c` RETURN

To abort the current computation and restart, enter:

`:a` RETURN `(nuprl)` RETURN

If you get a crash, you can get more information on it as follows. The initial crash message might look something like:

```
>>Error:  The value of S, (TTREE 108 97 100 116 59 59), should be a STRUCTURE

SYSTEM:STRUCTURE-REF:
Required arg 0 (S): (TTREE 108 111 97 100 116 59 59)
Required arg 1 (I): 1
Required arg 2 (TYPE): TERM
:C 0:  Use a new value
:A 1:  Abort to Lisp Top Level

-> ▯
```

Here, the ⌜`-> `⌝ is the debugger prompt. Enter:

`:b` RETURN

Lisp prints a backtrace. For example:

```
-> :b
SYSTEM:STRUCTURE-REF <- OPERATOR-OF-TERM <- IDFORM-TERM-P <- (:INTERNAL
SOURCE-REDUCE VISIT) <- SOURCE-REDUCE <- TERM-TO-ML-ISTRING <-
PRL-SCANNER-INITIALIZE <- PRL-MLLOOP <- ML <- ML-MODE$ <- PROCESS-CMD
<- CMD-WAIT <- PRL-LOOP <- PRL <- EVAL <- SYSTEM:ENTER-TOP-LEVEL
-> ▯
```

Enter:

`:n` RETURN

2 or 3 times. For example:

```
tt -> :n
OPERATOR-OF-TERM:
Required arg 0 (TERM): (TTREE 108 111 97 100 116 59 59)
-> :n
IDFORM-TERM-P:
Required arg 0 (TERM): (TTREE 108 111 97 100 116 59 59)
-> ▯
```

This provides a bit more information on the last function calls. When you send in a bug report, include the error message, backtrace and function calls you have obtained as above. Also, mention briefly what you were doing at the time of the crash.

To recover from a crash, enter

`:a` RETURN

at the debugger prompt. Lisp then prints a top level prompt. For example:

```
   -> :a
> ▯
```

Try restarting by entering

`(nuprl)` RETURN

If another crash follows immediately, the problem might be linked with the window system interface. Enter ⌜`:a` RETURN⌝ to get back to the top level, and then enter

`(reset)` RETURN `(nuprl)` RETURN

On executing the `(reset)` function, all the Nuprl windows will close and `(nuprl)` will open up an initial ML-Top-Loop and library window. Any proofs you were just working on will not have been lost. Use the `view` function to open them up again. However, you might have lost the contents of the last term-editor window you were working in.

If still the system crashes but you *can* execute functions in the ML Top Loop, try dumping any theories you haven't previously saved, quit the Nuprl session, and start a new one (The quit function is `(quit)`. If you can't even get the ML Top Loop running, the bug is very serious. Your only choice is to quit the session, losing any work you've done and haven't saved, and start a new session.