The Nuprl Proof Development System
Version 4.2

Paul Jackson

September 19, 1994
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3 Editor Cursors</td>
<td>28</td>
</tr>
<tr>
<td>4.3.4 Sequences</td>
<td>29</td>
</tr>
<tr>
<td>4.3.4.1 Term Sequences</td>
<td>30</td>
</tr>
<tr>
<td>4.3.4.2 Text Sequences</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Term Editor</td>
<td>30</td>
</tr>
<tr>
<td>4.4.1 Introduction</td>
<td>30</td>
</tr>
<tr>
<td>4.4.2 Cursor/Window Motion</td>
<td>31</td>
</tr>
<tr>
<td>4.4.2.1 Screen Oriented</td>
<td>31</td>
</tr>
<tr>
<td>4.4.2.2 Tree Oriented</td>
<td>31</td>
</tr>
<tr>
<td>4.4.3 Adding New Text</td>
<td>32</td>
</tr>
<tr>
<td>4.4.4 Adding New Terms</td>
<td>32</td>
</tr>
<tr>
<td>4.4.5 Cutting and Pasting</td>
<td>33</td>
</tr>
<tr>
<td>4.4.5.1 Basic</td>
<td>34</td>
</tr>
<tr>
<td>4.4.5.2 Region</td>
<td>34</td>
</tr>
<tr>
<td>4.4.6 Adding and Removing Slots in Sequences</td>
<td>35</td>
</tr>
<tr>
<td>4.4.7 Opening, Closing, and Changing Windows</td>
<td>36</td>
</tr>
<tr>
<td>4.4.8 Utilities</td>
<td>36</td>
</tr>
<tr>
<td>4.4.9 Mouse Commands</td>
<td>37</td>
</tr>
<tr>
<td>4.5 Exploded Terms</td>
<td>37</td>
</tr>
<tr>
<td>5 Abstractions</td>
<td>41</td>
</tr>
<tr>
<td>6 Display</td>
<td>45</td>
</tr>
<tr>
<td>6.1 Display Form Definitions</td>
<td>45</td>
</tr>
<tr>
<td>6.1.1 Top Level Structure</td>
<td>45</td>
</tr>
<tr>
<td>6.1.2 Formats</td>
<td>45</td>
</tr>
<tr>
<td>6.1.2.1 Slots</td>
<td>45</td>
</tr>
<tr>
<td>6.1.3 Attributes</td>
<td>46</td>
</tr>
<tr>
<td>6.1.4 Right-hand-side terms</td>
<td>46</td>
</tr>
<tr>
<td>6.2 Whitespace</td>
<td>47</td>
</tr>
<tr>
<td>6.2.1 Margin Control</td>
<td>47</td>
</tr>
<tr>
<td>6.2.2 Line Breaking</td>
<td>47</td>
</tr>
<tr>
<td>6.2.3 Optional Spaces</td>
<td>48</td>
</tr>
<tr>
<td>6.3 Parenthesization</td>
<td>48</td>
</tr>
<tr>
<td>6.3.1 Precedence Objects</td>
<td>48</td>
</tr>
<tr>
<td>6.3.2 Parenthesis Selection</td>
<td>48</td>
</tr>
<tr>
<td>6.4 Iteration</td>
<td>49</td>
</tr>
<tr>
<td>6.5 Examples</td>
<td>49</td>
</tr>
<tr>
<td>6.6 The Layout Algorithm</td>
<td>50</td>
</tr>
<tr>
<td>7 Sequents and Proofs</td>
<td>51</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>51</td>
</tr>
<tr>
<td>7.2 Sequent Structure</td>
<td>51</td>
</tr>
<tr>
<td>7.3 Proof Structure</td>
<td>52</td>
</tr>
<tr>
<td>7.4 Refinement Rules</td>
<td>52</td>
</tr>
<tr>
<td>7.4.1 Primitive Refinement Rules</td>
<td>52</td>
</tr>
<tr>
<td>7.4.2 Tactic Rules</td>
<td>52</td>
</tr>
<tr>
<td>7.4.3 Reflection Rules</td>
<td>53</td>
</tr>
<tr>
<td>7.5 Transformation Tactics</td>
<td>53</td>
</tr>
<tr>
<td>7.6 Proof Editor</td>
<td>53</td>
</tr>
<tr>
<td>7.6.1 Proof Window Format</td>
<td>53</td>
</tr>
</tbody>
</table>
CONTENTS

7.6.3.1 Opening a Proof Window .................................... 55
7.6.3.2 Closing a Proof Window ..................................... 56
7.6.3.3 Changing Windows .............................................. 56
7.6.3.4 Editing The Main Goal ......................................... 56
7.6.3.5 Editing a Refinement Rule .................................... 56
7.6.3.6 Viewing Subgoal Sequents .................................... 57
7.6.3.7 Editing a Transformation Tactic ............................. 57
7.6.4 Mouse Commands ................................................. 57

7.7 Proof Compression and Expansion ................................. 57

8 Tactics ........................................................................ 59

8.1 Introduction .................................................................. 59
  8.1.1 Conventions ............................................................ 59
  8.1.2 Universes and Level Expressions .............................. 59
  8.1.3 Formula Structure .................................................. 60
  8.1.4 Soft Abstractions .................................................... 60
  8.1.5 The Sequent .......................................................... 61
  8.1.6 Proof Annotations .................................................. 61
    8.1.6.1 Goal Labels ......................................................... 62
    8.1.6.2 Tactic Arguments ............................................... 62
  8.1.7 Matching and Substitution ....................................... 63

8.2 Basic .......................................................................... 65
  8.2.1 Simple ................................................................. 65
  8.2.2 Decomposition ....................................................... 66

8.3 Tacticals ...................................................................... 67
  8.3.1 Basic Tacticals ....................................................... 67
  8.3.2 Label Sensitive Tacticals ......................................... 68
  8.3.3 Multiple Clause Tacticals ....................................... 68

8.4 Case Splits and Induction ............................................. 69

8.5 Forward and Backward Chaining ................................... 69

8.6 Equational reasoning .................................................. 71
  8.6.1 Direct computation ................................................. 71
  8.6.2 Substitution .......................................................... 71

8.7 Decision Procedures ................................................... 71
  8.7.1 Eq ..................................................................... 71
  8.7.2 Arith ................................................................. 72
  8.7.3 SupInf .............................................................. 73

8.8 Rewriting .................................................................... 73
  8.8.1 Introduction .......................................................... 73
  8.8.2 Environments ....................................................... 73
  8.8.3 Relations ............................................................ 74
  8.8.4 Justifications ....................................................... 75
  8.8.5 Conversion Descriptions ........................................ 75
    8.8.5.1 Trivial Conversions ............................................ 75
    8.8.5.2 Conversionals ................................................... 76
    8.8.5.3 Atomic Direct-Computation Conversions .............. 76
    8.8.5.4 Composite Direct Computation Conversions ........ 77
    8.8.5.5 Macro Conversions ........................................... 77
    8.8.5.6 Lemma and Hypothesis Conversions ................... 77
  8.8.6 Applying Conversions ............................................ 78

8.8.7 Applying Lemmas ................................................... 79
8.8.7.3 Weakening Lemmas .............................................. 80
8.8.7.4 Inversion Lemmas .............................................. 80
8.9 Type Inclusion .......................................................... 81
8.10 Miscellaneous ......................................................... 81
8.11 Autotactic ............................................................... 82
8.12 Transformation Tactics .............................................. 82
8.13 Constructive and Classical Reasoning .......................... 83
  8.13.1 Constructive Reasoning ......................................... 83
  8.13.2 Decidability ....................................................... 83
  8.13.3 Squash Stability and Hidden Hypotheses .................... 84
  8.13.4 Classical Reasoning ............................................. 85
8.14 Further Information .................................................. 85

9 Theories ........................................................................... 87
  9.1 Theory Structure ....................................................... 87
  9.2 Adding Definitions .................................................... 87
    9.2.1 Adding Non-Recursive Definitions .............................. 88
    9.2.2 Adding Recursive Definitions .................................... 88
Chapter 1

Introduction

1.1 Purpose

This manual is a reference manual for version 4.1 of the Nuprl system scheduled for release in the summer of 1994. 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:

- \( \text{\langle C-x \rangle} \) read as “control \( x \)”. Hold down a control key and simultaneously press key \( x \).
- \( \text{\langle M-x \rangle} \) read as “meta \( x \)”. Hold down 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- \texttt{MOUSE\text{-}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, \texttt{RETURN}).

When we say “click \texttt{MOUSE\text{-}LEFT}” on some part of a window, we mean that the mouse cursor should be pointed at that part, and then the \texttt{MOUSE\text{-}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 \texttt{pq} quotes. For example \texttt{pqthis is example outputq}.

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

\begin{center}
\begin{tabular}{|c|}
\hline
ML top loop \\
\hline
M> `[int] * [int]’ ;;
\hline
\end{tabular}
\end{center}

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

\section{1.3 Practical Details}

\subsection{1.3.1 Getting Set Up}

The Nuprl system is written in a combination of Common Lisp and and 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 \texttt{nuprl} has been set up for this script.

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

To set-up, do the following:

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

\begin{verbatim}
xset fp+ \texttt{FONT\text{-}DIR}
xset fp rehash
\end{verbatim}
1.3. PRACTICAL DETAILS

The Nuprl fonts are named 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:

\texttt{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

\[
\texttt{(M-x)shell} \ \texttt{RETURN}
\]

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

\[
\texttt{nuprl} \ \texttt{RETURN}
\]

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

<table>
<thead>
<tr>
<th>ML Top Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{M&gt;</td>
</tr>
</tbody>
</table>

The Lisp code in Nuprl is running in the window in which you typed \texttt{nuprl}. Since output from the Lisp code is immediately displayed in this window, it is a good idea to keep this window visible.
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 Section 1.3.5 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)

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 The Lisp Debugger and Crashes

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:
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
```

Lisp prints a backtrace. For example:

```
-> :b
-> □
```

Enter:

```
:n
```

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
Try restarting by entering

\texttt{(nuprl) RETURN}

If another crash follows immediately, the problem might be linked with the window system interface. Enter \texttt{"\(a\)\textasciitilde\:a\textasciitilde"} to get back to the top level, and then enter

\texttt{(reset) RETURN (nuprl) RETURN}

On executing the \texttt{(reset)} function, all the Nuprl windows will close and \texttt{(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 \texttt{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 \textit{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 \texttt{(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.

### 1.3.6 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 term and proof different.
- 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 \texttt{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 \texttt{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 \texttt{user} (or \texttt{common-lisp-user}). To refer to these other symbols, either change packages using \texttt{(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 \texttt{(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 \texttt{change-options} can be used to set various parameters affecting Nuprl’s window system.

The \texttt{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

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

All directories should eventually contain a `README` file that describes 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 9.1.) 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, case studies, and the organization of the system and its editors. It is available through the [Nuprl Documentation](https://nuprl.org/documentation).
• 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 \texttt{M>} is the ML prompt. The \texttt{;;}'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 \texttt{I} is the cursor. To differentiate it from other kinds of cursors we call it a \textit{text} cursor. You can type text whenever you have a text cursor. Other kinds of cursors we call \textit{screen} cursors and \textit{term} cursors. They have readily distinguishable appearances; a screen cursor outlines a single character, and a term cursor highlights a region of the screen.
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 sensitive; specifically the insert-char-, 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:

\[
2+2 \text{ return}
\]

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

\[
\text{M> 2+2 ;;}
\]
\[
4 : \text{ int}
\]

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 hs-return. The continuation prompt is >. For example, if you entered:

\[
1+2+(s- \text{ return})3+4
\]
you would get:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>insert character x</td>
</tr>
<tr>
<td>RETURN</td>
<td>call ML evaluator</td>
</tr>
<tr>
<td>(S- RETURN)</td>
<td>add line-break</td>
</tr>
<tr>
<td>MOUSE-LEFT</td>
<td>move cursor to mouse position</td>
</tr>
<tr>
<td>(C-F)</td>
<td>move cursor right 1 character</td>
</tr>
<tr>
<td>(C-B)</td>
<td>move cursor left 1 character</td>
</tr>
<tr>
<td>(C-P)</td>
<td>move cursor up 1 character</td>
</tr>
<tr>
<td>(C-N)</td>
<td>move cursor down 1 character</td>
</tr>
<tr>
<td>(C-D)</td>
<td>delete char to right of cursor</td>
</tr>
<tr>
<td>DELETE</td>
<td>delete char to left of cursor</td>
</tr>
<tr>
<td>(C-R)</td>
<td>scroll back through history</td>
</tr>
<tr>
<td>(M-R)</td>
<td>scroll forward through history</td>
</tr>
<tr>
<td>(C-Z)</td>
<td>return to Lisp Listener</td>
</tr>
</tbody>
</table>

Table 2.1: Basic Top Loop Commands
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.

2.3 More Advanced Top-Loop Operation

TO BE WRITTEN

2.4 Alternative Top Loops

Describe shell top loop: advantages and disadvantages.
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 Section 7.7 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.
CHAPTER 3. THE LIBRARY

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.
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. The functions take the following kinds of arguments:

- **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 `[` `]` 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**
CHAPTER 3. THE LIBRARY

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 \texttt{\textasciitilde nuprl\textasciitilde lib\textasciitilde standard\textasciitilde core.1}. The actual file associated with a theory is this name with a \texttt{.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.

set_theory_filenames theory-names file-names
Add entries in theory_filenames list for theory-names. If an entry already exists for a 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.

```ml
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.

```ml
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:

```ml
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.

```ml
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.

```ml
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:

```ml
list_theories ()
```

List all the currently loaded theories. This is very useful if you are not sure which theories are loaded and which not.

```ml
delete_theory theory-name
```

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

```ml
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 LATEX version of the theory listing.

```ml
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:

```ml
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:

```ml
load_fully 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 `p_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 abstractions 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 defined in ML objects, and so this...
3.6. **FUTURE DEVELOPMENTS**

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 8.

### 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 systematic 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 $x_1^i, \ldots, x_{a_i}^i$ are variables, $t_i$ is a term, and we define

$$ s_i = x_1^i, \ldots, x_{a_i}^i, t_i $$

then
We name the parts of a term as follows:

- $\text{opid}\{p_1:k_1, \ldots, p_m:k_m\}$ is the operator.
  
  The parts of the operator are:
  - $\text{opid}$ is the operator identifier.
  - $p_j:k_j$ is the $j$th 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 $i$th 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 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 $\text{int}$. Acceptable strings are generated by the regular expression $[1-9][0-9]^*$.

- **token**
  - character strings. Implemented using ML type $\text{tok}$. Acceptable strings can draw from any non-control characters in Nuprl’s font.

- **string**
  - character strings. Implemented using ML type $\text{string}$. Acceptable strings can draw from any non-control characters in Nuprl’s font.

- **variable**
  - Names of variables. Implemented using ML type $\text{var}$. Acceptable strings draw on the alphabet $\mathcal{a-zA-Z0-9_-%}$.

- **level-expression**
  - Universe level expressions. These are used to index universe levels in Nuprl's type theory. Implemented using ML type $\text{level_exp}$. The syntax of level expressions is described in Section 8.1.2.

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

Opids are character strings drawn from the alphabet $\mathcal{a-zA-Z0-9_-%}$ (Here $\mathcal{\sim}$ is the ASCII character, $\mathcal{x-y}$ indicates the characters from $x$ to $y$ inclusive.) The $\mathcal{!}$ 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 $\text{tok}$.

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 $\text{var}$. 

---

CHAPTER 4. TERMS
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, \texttt{variable}\{v\} for injecting variables into the term type. So when we talk of the variable \texttt{foo as a term}, we are really thinking of the term \texttt{variable}\{foo:v\}. When we write terms, this injection is often implicit. So for example, we write \texttt{bar(x;y)} instead of \texttt{bar\{variable\{x:v\}; variable\{y:v\}\}}.

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

<table>
<thead>
<tr>
<th>Pretty Notation</th>
<th>Plain Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathbb{Z})</td>
<td>\texttt{int()}</td>
</tr>
<tr>
<td>(x + y)</td>
<td>\texttt{add(x;y)}</td>
</tr>
<tr>
<td>\texttt{&quot;abc&quot;}</td>
<td>\texttt{token{abc:t}()}</td>
</tr>
<tr>
<td>(\lambda x.x)</td>
<td>\texttt{lambda(x_x)}</td>
</tr>
<tr>
<td>(\forall x,y:T.\ A)</td>
<td>\texttt{all(T; x,y.A)}</td>
</tr>
</tbody>
</table>

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 \texttt{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 \texttt{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 is developing a system where (for example with Mathematica) it is relatively easy to read and type in mathematics, and then use pretty-printing routines for formatted output (say in Display PostScript).
The approach which we have taken 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 unambiguous to a machine. It just has to be 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 textbook-quality displays.

- Notation can be freely changed without altering the underlying logical structure of terms.

- 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 \texttt{add}(x;y) for binary addition. The usual notation for this is to use an infix \texttt{p}+\texttt{q}. We could write the notation chunk as:

\begin{align*}
\texttt{□} + \texttt{□}
\end{align*}

where the □’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 \texttt{add}(x;y) can be written:

\begin{align*}
\texttt{x} + \texttt{y} & =_{\text{dform}} \texttt{add}(x; y)
\end{align*}

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 accommodate the terms inserted in them. For example say we have the term \texttt{mul}(1;2) which is displayed as \texttt{\texttt{p} \texttt{l} \texttt{q} \texttt{v} \texttt{2}}. Then the term \texttt{add(mul(1;2);3)} will be displayed as:

\begin{align*}
\texttt{x} + \texttt{□} + \texttt{□}
\end{align*}
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 \texttt{mul}(x;y) a higher precedence than the display form for \texttt{add}(x;y). The term \texttt{add(mul(1;2);3)} is displayed as

$$1 * 2 + 3$$

but the term \texttt{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 \texttt{all}(T;x.P) means that “for all \(x\) of type \(T\), the proposition \(P\) is true”. Note that the term \texttt{all}(T;x.P) binds free occurrences of \(x\) in \(P\). We would normally write \texttt{all}(T;x.P) as

$$\forall x:T. \ P$$

The display form definition for the \texttt{all}(T;x.P) could be written as:

$$\forall (x: T. \ [P]) =_{\text{dform}} \text{all}(T; x.P)$$

Here, \(\Box\) 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. \[
\exists (x: T. \ [P]) =_{\text{dform}} \text{exists}(T; x.P)
\]

2. \[
[x = y] =_{\text{dform}} \text{equal}_\text{int}(x; y)
\]

3. \[
[x] =_{\text{dform}} \text{variable}\{x:v\}
\]

4. \[
[i] =_{\text{dform}} \text{natural}_\text{number}\{i:n\}
\]

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

\[
\text{all(int(); i.exists(int(); j.equal(int(); j; add(i;1))))}
\]

has the notation:

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

\[ \forall i:Z. \exists j:Z. \ j = i + 1 \]

you might be presented with term:

\[ \forall i:Z. \ \exists [\text{var}]:[\text{type}]. \ [\text{prop}]. \]

Here [\text{var}], [\text{type}] and [\text{prop}] are place-holders for slots. [\text{var}] is a place-holder for a text slot, and [\text{type}] and \text{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 \text{var}, \text{type} or \text{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,

\[ \forall i:Z. \ \exists j:Z. \ [j = i + 1] \]
4.3. TERM DISPLAY

**text mode**

The text cursor is used for editing text in text slots. The cursor is represented as \( \text{cursor} \). 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 \( \text{abcdef} \). Valid text cursors for this string include

\[
\text{abcdef} \quad \text{abc|def} \quad \text{abcdef|}
\]

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 \( \text{aaa} \) and \( \text{zzz} \) and the following text cursor:

\[
\text{aaa|zzz}.
\]

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:

\[
\forall \text{i:Z}. \exists \text{j:Z}. \quad j = i + 1
\]

then a ‘move-left-one-character’ command would leave a screen cursor (indicated by a box) over the \( \forall \).

\[
\Box \forall \text{i:Z}. \exists \text{j:Z}. \quad j = i + 1
\]

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. The term editor is supposed to support these kinds of terms:
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:

\([[\text{elmnt}], [\text{elmnt}], [\text{elmnt}], [\text{elmnt}]]\)

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 \((\p, \p)\) respectively in the example) and different element separators (the \(\p, \p\) 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 \(\p'n + 1'\) (D 0)\(~
THENW TypeCheck~

This text sequence consists of 3 term slots filled with the terms \(\p'n + 1'\), \(\p'\), and \(\p'\), and 4 text slots filled with the text strings \(\p'\text{With}\), \(\p'(D 0)\), \(\p'\text{THENW TypeCheck}\), and \(\p'\) (the null or empty text string). The \(\p'\)'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 sub-sequences 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 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 to display an object in its entirety, only a portion of the object is displayed, depending on the size of the window.

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.

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C-P)</td>
<td>SCREEN-UP</td>
<td>move cursor up 1 character</td>
</tr>
<tr>
<td>(C-N)</td>
<td>SCREEN-DOWN</td>
<td>move cursor down 1 character</td>
</tr>
<tr>
<td>(C-F)</td>
<td>SCREEN-LEFT</td>
<td>move cursor left 1 character</td>
</tr>
<tr>
<td>(C-B)</td>
<td>SCREEN-RIGHT</td>
<td>move cursor right 1 character</td>
</tr>
<tr>
<td>(C-A)</td>
<td>SCREEN-START</td>
<td>move to left side of screen</td>
</tr>
<tr>
<td>(C-E)</td>
<td>SCREEN-END</td>
<td>move to right side of screen</td>
</tr>
<tr>
<td>(C-L)</td>
<td>SCROLL-UP</td>
<td>scroll window up 1 line</td>
</tr>
<tr>
<td>(M-L)</td>
<td>SCROLL-DOWN</td>
<td>scroll window down 1 line</td>
</tr>
<tr>
<td>(C-V)</td>
<td>PAGE-DOWN</td>
<td>move window down 1 page</td>
</tr>
<tr>
<td>(M-V)</td>
<td>PAGE-UP</td>
<td>move window up 1 page</td>
</tr>
<tr>
<td>(C-T)</td>
<td>SWITCH-TO-TERM</td>
<td>switch to term mode</td>
</tr>
</tbody>
</table>

Table 4.2: Screen Motion Commands

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 move 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.

<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M-P)</td>
<td>UP</td>
<td>move up to parent</td>
</tr>
<tr>
<td>(M-B)</td>
<td>LEFT</td>
<td>structured move left</td>
</tr>
<tr>
<td>(M-F)</td>
<td>RIGHT</td>
<td>structured move right</td>
</tr>
<tr>
<td>(M-N)</td>
<td>DOWN-LEFT</td>
<td>move to leftmost child</td>
</tr>
<tr>
<td>(M-M)</td>
<td>DOWN-RIGHT</td>
<td>move to rightmost child</td>
</tr>
<tr>
<td>(M-A)</td>
<td>LEFTMOST-SIBLING</td>
<td>move to leftmost sibling</td>
</tr>
<tr>
<td>(M-E)</td>
<td>RIGHTMOST-SIBLING</td>
<td>move to rightmost sibling</td>
</tr>
<tr>
<td>(M-&lt;)</td>
<td>UP-TO-TOP</td>
<td>move up top of term</td>
</tr>
<tr>
<td>(C- RETURN)</td>
<td>RIGHT-LEAF</td>
<td>next leaf to right</td>
</tr>
<tr>
<td>(M- RETURN)</td>
<td>LEFT-LEAF</td>
<td>next leaf to left</td>
</tr>
<tr>
<td></td>
<td>RIGHT-EMPTY-SLOT</td>
<td>next empty slot to right</td>
</tr>
<tr>
<td></td>
<td>RIGHT-EMPTY-SLOT</td>
<td>next empty slot to right</td>
</tr>
<tr>
<td></td>
<td>LEFT-EMPTY-SLOT</td>
<td>next empty slot to left</td>
</tr>
</tbody>
</table>

Table 4.3: Tree Motion Commands
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 \texttt{RETURN} to RIGHT-EMPTY-SLOT doesn't work in text sequences. In that case, you need to use (C- \texttt{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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x}</td>
<td>\texttt{INSERT-CHAR-x}</td>
</tr>
<tr>
<td>(C-#)\texttt{num}</td>
<td>\texttt{INSERT-SPEC-CHAR-num}</td>
</tr>
<tr>
<td>\texttt{RETURN}</td>
<td>\texttt{INSERT-NEWLINE}</td>
</tr>
</tbody>
</table>

Table 4.4: Text Insertion

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

\texttt{xfd -font nuprl-13 &}

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

The \texttt{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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{name}</td>
<td>\texttt{INSERT-TERM-name}</td>
</tr>
<tr>
<td>(C-I)\texttt{name}</td>
<td>\texttt{INSERT-TERM-LEFT-name}</td>
</tr>
<tr>
<td>(M-I)\texttt{name}</td>
<td>\texttt{INSERT-TERM-RIGHT-name}</td>
</tr>
<tr>
<td>(C-S)\texttt{name}</td>
<td>\texttt{SUBSTITUTE-TERM-name}</td>
</tr>
<tr>
<td>\texttt{CM-I}</td>
<td>\texttt{INITIALIZE-TERM}</td>
</tr>
<tr>
<td>\texttt{CM-S}</td>
<td>\texttt{SELECT-DFORM-OPTION}</td>
</tr>
</tbody>
</table>

Table 4.5: Term Insertion

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

1. \textit{name} is an editor command enabled in a particular context. See sections for examples.
2. \textit{name} is an \textit{alias} for some display form, defined in in the library object for that display form.
3. \textit{name} is the name of a display form object. It refers to the first display form defined in that object.
4. \textit{name} is of the form \textit{ni} where \textit{n} is the name of a display form object and \textit{i} is a natural number. \textit{ni} refers to the \textit{i}th display form definition in the object named \textit{n}. Definitions in objects are numbered starting from 1.
6. *name* is all numerals, then the term referred to is the \texttt{natural-number\{name:n\}()} term of Nuprl’s object language.

7. *name* refers to the variable term \texttt{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 \texttt{NO-OP} (SPACE) or a cursor motion command. \texttt{NEXT-EMPTY-SLOT} ((C-RETURN)) is a particularly useful terminator.

\texttt{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 terminated by some cursor motion command, then that command is obeyed.

\texttt{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.

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

\texttt{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 \texttt{SUBSTITUTE-TERM-name} tries something vaguely sensible. In general in these cases, it is safer to explicitly cut and paste the children.

\texttt{INITIALIZE-TERM} initializes a term slot to some default term appropriate to the cursor’s context. The term slot must initially be empty. \texttt{INITIALIZE-TERM} is automatically invoked by Nuprl to 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 \texttt{INITIALIZE-TERM} command. The default terms for particular contexts are described in various sections of this document. If no default has been designated, \texttt{INITIALIZE-TERM} does nothing.

\texttt{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 \textit{items}. Cut items can be saved on a \textit{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 \texttt{copy-as-kill} in emacs.
- **DELETE**: remove an item from a buffer, \textit{not} saving it anywhere.
- **CUT**: (= \texttt{SAVE} + \texttt{DELETE}) removes an item from a buffer and pushes it onto the top of the \textit{save stack}. Same idea as \texttt{kill} in emacs, although Nuprl does \textit{not} append together items cut immediately after each other.
• **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 |
|        | DELETE-CHAR-TO-RIGHT    | delete char to right of text cursor |
|        | CUT-WORD-TO-RIGHT      | cut word to right of text cursor |
|        | CUT                     | cut term |
|        | SAVE                    | save term |
|        | DELETE                  | delete term |
|        | PASTE                   | paste item |
|        | PASTE-NEXT              | delete item then paste next item |
|        | 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-(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 |
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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hC-i</td>
<td>open-seq-to-left</td>
</tr>
<tr>
<td>hM-i</td>
<td>open-seq-to-right</td>
</tr>
<tr>
<td>hC-O</td>
<td>open-seq-left-and-init</td>
</tr>
<tr>
<td>hM-O</td>
<td>open-seq-right-and-init</td>
</tr>
<tr>
<td>hC-C</td>
<td>close-seq-to-left</td>
</tr>
<tr>
<td>hM-C</td>
<td>close-seq-to-right</td>
</tr>
<tr>
<td>open slot to left of cursor</td>
<td></td>
</tr>
<tr>
<td>open slot to right of cursor</td>
<td></td>
</tr>
<tr>
<td>open slot to left and init</td>
<td></td>
</tr>
<tr>
<td>close slot and move left</td>
<td></td>
</tr>
<tr>
<td>close slot and move right</td>
<td></td>
</tr>
</tbody>
</table>

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 event that the term is already present in the slot, the term is simply copied to the new slot and the cursor is left at the new slot.
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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c-Q)</td>
<td>QUIT close window without saving</td>
</tr>
<tr>
<td>(c-Z)</td>
<td>EXIT save, check, and close window</td>
</tr>
<tr>
<td>(c-J)</td>
<td>JUMP-NEXT-WINDOW jump to next window</td>
</tr>
<tr>
<td>(tab)</td>
<td>JUMP-ML jump to ML top loop</td>
</tr>
</tbody>
</table>

Table 4.9: Commands For Changing and Closing Windows

Term editor windows 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`. Separate 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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c-X)id</td>
<td>IDENTIFY-TERM gives info on term at cursor</td>
</tr>
<tr>
<td>(c-X)su</td>
<td>SUPPRESS-DFORM suppress display form at cursor</td>
</tr>
<tr>
<td>(c-X)un</td>
<td>UNSUPPRESS-DFORM unsuppress display form at cursor</td>
</tr>
<tr>
<td>(c-X)ex</td>
<td>EXPLODE-TERM explode term at cursor</td>
</tr>
<tr>
<td>(c-X)im</td>
<td>IMPLODE-TERM implode term at cursor</td>
</tr>
<tr>
<td>(c-X)ch</td>
<td>CHECK-OBJECT check object</td>
</tr>
<tr>
<td>(c-X)sa</td>
<td>SAVE-OBJECT save object</td>
</tr>
<tr>
<td>(c-X)ab</td>
<td>VIEW-ABSTRACTION view abstraction def of term</td>
</tr>
<tr>
<td>(c-X)df</td>
<td>VIEW-DFORM view display form def for term</td>
</tr>
</tbody>
</table>

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.

The utility commands `EXPLODE-TERM`, `IMPLODE-TERM`, and `CHECK-OBJECT` can be helpful for debugging display forms, which are generated automatically.
EXPLODED TERMS

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 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.

4.4.9 Mouse Commands

The mouse commands are shown in Table 4.11

<table>
<thead>
<tr>
<th>MOUSE-LEFT</th>
<th>MOUSE-MIDDLE</th>
<th>MOUSE-RIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C-)</td>
<td>MOUSE-SET-POINT</td>
<td>MOUSE-SET-TERM-POINT</td>
</tr>
<tr>
<td>MOUSE-SET-TERM-POINT</td>
<td>MOUSE-VIEW-DISP</td>
<td>MOUSE-SET-TERM-POINT</td>
</tr>
<tr>
<td>MOUSE-MIDDLE</td>
<td>MOUSE-MIDDLE</td>
<td>MOUSE-SET-TERM-POINT</td>
</tr>
<tr>
<td>(C-)</td>
<td>MOUSE-PASTE</td>
<td>MOUSE-PASTE</td>
</tr>
<tr>
<td>MOUSE-PASTE</td>
<td>MOUSE-PASTE-NEXT</td>
<td>MOUSE-PASTE</td>
</tr>
<tr>
<td>MOUSE-MIDDLE</td>
<td>MOUSE-PASTE-NEXT</td>
<td>MOUSE-PASTE</td>
</tr>
<tr>
<td>(C-)</td>
<td>MOUSE-PASTE-COPY</td>
<td>MOUSE-PASTE-COPY</td>
</tr>
<tr>
<td>MOUSE-PASTE-COPY</td>
<td>MOUSE-VIEW-AB</td>
<td>MOUSE-PASTE-COPY</td>
</tr>
<tr>
<td>MOUSE-PASTE-COPY</td>
<td>MOUSE-CUT</td>
<td>MOUSE-PASTE-COPY</td>
</tr>
<tr>
<td>MOUSE-PASTE-COPY</td>
<td>MOUSE-SAVE</td>
<td>MOUSE-PASTE-COPY</td>
</tr>
<tr>
<td>MOUSE-PASTE-COPY</td>
<td>MOUSE-DELETE</td>
<td>MOUSE-PASTE-COPY</td>
</tr>
</tbody>
</table>

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 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.

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

\[
\text{EXPLODED}<<[\text{opid}]\{}()>>
\]

Enter the opid:

\[
\langle c-\text{RETURN}\rangle\text{foo}
\]
to get:

\[
\text{EXPLODED}<<\text{foo}\{}()>>
\]

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

\[
\text{EXPLODED}<<\text{foo}\{}()>>
\]

Enter \langle c-0\rangle to add a new slot to the parameter sequence:

\[
\text{EXPLODED}<<\text{foo}\{[\text{parm}]\}()>>
\]

Insert the string parameter with text \text{bar}:

\[
\text{sparm}\text{RETURN}\text{bar}
\]
to get:

\[
\text{EXPLODED}<<\text{foo}\{\text{bar}:s\}\{}()>>
\]

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

\[
\text{EXPLODED}<<\text{foo}\{\text{bar}:s\}\{}()>>
\]

Enter \langle c-0\rangle\langle c-0\rangle to make a two element sequence for bound terms, leaving the cursor on the left-most element.

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

Click \text{MOUSE-LEFT} on the second \text{\} \text{\}} to get a null width term cursor sitting on an empty term sequence...
Enter `<c-0>RETURN` to open up a slot in the sequence, and enter a binding variable term:

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

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 ??.

Abstraction definitions have form:

\[ lhs \equiv rhs \]

The terms \( lhs \) and \( rhs \) are pattern terms, and there is implicit universal quantification over the free variables in \( lhs \) and \( rhs \). When Nuprl unfolds some instance \( lhs\text{-inst} \) of \( lhs \), it first matches \( lhs\text{-inst} \) 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 \( lhs\text{-inst} \). It then applies the substitution to \( rhs \) to calculate the term \( rhs\text{-inst} \) which \( lhs\text{-inst} \) unfolds to.

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

<table>
<thead>
<tr>
<th>EDIT ABS</th>
<th>int_seg</th>
</tr>
</thead>
<tbody>
<tr>
<td>{i..j} == {k:Z</td>
<td>i ≤ k &lt; j}</td>
</tr>
</tbody>
</table>

Figure 5.1: Definition of the \texttt{int_seg} abstraction

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

Abstractions can have binding structure; for example, consider the \texttt{exists unique} abstraction in Figure 5.2.

<table>
<thead>
<tr>
<th>EDIT ABS</th>
<th>exists_uni</th>
</tr>
</thead>
<tbody>
<tr>
<td>\exists!u:T. \ P[u]==\exists u:T. \ P[u] &amp; \forall v:T. \ P[v] \Rightarrow v = u \in T</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2: Definition of the \texttt{exists\_uni} abstractions

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.
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 \(x_1, \ldots, x_a, t\). 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 \(v \mapsto x_1, \ldots, x_n, t\), 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

\[\exists!i:Z. \ i = 0 \in Z\]

Here, \(\subseteq \subseteq \subseteq \subseteq\) 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

\[\exists!u:T. \ P[u]\]

would be

\[\exists!u:T. \ P[u]\]

and the result of applying this to

\[\exists!u:T. \ P[u] \land \forall v:T. \ P[v] \Rightarrow v = u \in T\]

would be

\[\exists!u:Z. \ u = 0 \land \forall v:Z. \ v = 0 \in Z \Rightarrow v = u \in Z\]

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:

\[\exists!i:Z. \ i = 0 \land \forall v:Z. \ v = 0 \in Z \Rightarrow v = i \in Z\]

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.

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 \(\text{label}{$x$ : $t$ ; $i$ : $p$}\), as shown in Figure 5.3.
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 abstractions 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:

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

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.

| cm-I | INITIALIZE | initialize object / condition |
| cm-S | SELECT-TERM-OPTION | open condition seq |
| c-0  | OPEN-SEQ-TO-LEFT | open slot in cond seq to left |
| M-0  | OPEN-SEQ-TO-RIGHT | open slot in cond seq to right |
| C-M  | TOGGLE-META-STATUS | make parameter meta / normal |
| so-varn | INSERT-TERM so-varn | 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 condition slots.
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
\texttt{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 \texttt{variable(x:v)(a_1;...;a_n)} is named \texttt{so\_varn} so this family of names
can be used to reference them. Note that abstraction objects are the \textit{only} places where these second-order
variable instances are used. When writing propositions, second-order variable instances are simulated
using the \texttt{so\_apply(n)} abstraction.
Chapter 6

Display

6.1 Display Form Definitions

6.1.1 Top Level 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 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.

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 that matched the definition. Since the slot names are not always visible in the term, The
### 6.1.3 Attributes

Definition attributes are summarized in Table 6.2.

<table>
<thead>
<tr>
<th>Display</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c₁,…,cₙ)</td>
<td>conds</td>
<td>conditions</td>
</tr>
<tr>
<td>EdAlias(a)</td>
<td>alias</td>
<td>alias for definition input</td>
</tr>
<tr>
<td>#Hd(a)</td>
<td>ithd</td>
<td>head of iteration family</td>
</tr>
<tr>
<td>#Tl(a)</td>
<td>ittl</td>
<td>tail of iteration family</td>
</tr>
<tr>
<td>Parens</td>
<td>parens</td>
<td>parenthesis control</td>
</tr>
<tr>
<td>Prec(a)</td>
<td>prec</td>
<td>precedence</td>
</tr>
</tbody>
</table>

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 referring 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-$>$) 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. 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 TOGGLE-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 \{→\[i\]}(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 \{←\} (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.
- 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 \texttt{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 \textit{precedences}. A \textit{precedence} is an element in the \textit{precedence order}. The order is determined by the precedence objects in the Nuprl library. A definition is assigned a precedence by giving it a \texttt{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>
<thead>
<tr>
<th>Display</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{p_1 \ldots \ldots p_n}</td>
<td>prpar</td>
<td>parallel prec term</td>
</tr>
<tr>
<td>\langle p_1 \ldots \ldots p_n\rangle</td>
<td>prser</td>
<td>serial prec term</td>
</tr>
<tr>
<td>{p_1 \ldots \ldots = p_n}</td>
<td>preq</td>
<td>equal prec term</td>
</tr>
<tr>
<td>obname</td>
<td>prel</td>
<td>element of precedence order</td>
</tr>
<tr>
<td>*obname*</td>
<td>prptr</td>
<td>precedence object pointer</td>
</tr>
</tbody>
</table>

Table 6.3: Precedence Object

Table 6.3 shows the components of a \textit{precedence} object, and the names used to enter them by. The \texttt{par}, \texttt{ser}, and \texttt{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.

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 \texttt{L}, \texttt{E}, or \texttt{*} in the 3rd field), by the \texttt{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 \texttt{parens} attribute. If this attribute is absent, the slot is never parenthesized. Therefore the \texttt{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:
  
  \begin{itemize}
  \item \texttt{L} Suppress parentheses if display-form precedence is \textit{less than} child display-form precedence.
  \item \texttt{E} Suppress parentheses if display-form precedence is \textit{less than or equal to} child display-form precedence.
  \item \texttt{*} Always suppress parentheses.
  \end{itemize}
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 \(\text{plus}(a;b)\), and \(\text{times}(a;b)\), then \(\text{plus}(a;\text{times}(b;c))\) is displayed as \(a + b * c\), but \(\text{times}(a;\text{plus}(b;c))\) 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.

### 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 \(\lambda\) abstraction terms such that the \(\lambda\) character is suppressed on nested occurrences. The following definitions would work:

```ml
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:

\[ \lambda x, y, z. x \]

### 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:

```ml
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 f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```

The definition should now look like:

```
∃!<x:var>= [rhs]
```

Enter the type slot and the second term slot:

```
:(c-0)sslot RETURN f RETURN type(c-F)(c-F)(c-F)
```
Now enter the right-hand side of the display form. Click `MOUSE-LEFT` on the [rhs] placeholder, and enter \( \text{exists}_{\text{unique}}(T.x.P) \) as an exploded term. See Section 4.5 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:

\[
\exists!x:x:T. P:E == \text{exists}_{\text{unique}}\{([\text{term}];[\text{binding}].[\text{term}])\}
\]

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

\[
\begin{align*}
\text{mterm} & \quad [\text{return}] \quad x\langle\text{c-M}\rangle \quad \text{return} \\
\text{mterm} & \quad [\text{return}] \quad P
\end{align*}
\]

The definition is now complete. It should look like:

\[
\exists!x:x:T. P:E == \text{exists}_{\text{unique}}\{<T>;<x>.<P>\}
\]

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:

\[
\langle\text{c-M-S}\rangle \quad \text{return} \quad \langle\text{c-0}\rangle
\]

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

\[
\begin{align*}
\text{[attr]} & \quad :\quad [\text{attr}]:\quad \exists!x:x:T. P:E == \text{exists}_{\text{unique}}\{<T>;<x>.<P>\}
\end{align*}
\]

To instantiate the attribute slots enter:

\[
\text{parens} \quad \text{return} \quad \text{prec} \quad \text{return} \quad \text{exists}
\]

To get:

\[
\text{Paren}:\text{Prec}(\text{exists})\quad :\quad \exists!x:x:T. P:E == \text{exists}_{\text{unique}}\{<T>;<x>.<P>\}
\]

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 \( \text{\textbackslash .}\text{\textbackslash }\text{\textbackslash }\text{\textbackslash }\text{\textbackslash }\) separating the \text{type} slot from the \text{prop} slot is only included if the break is not taken. Click `MOUSE-LEFT` on the \( \text{\textbackslash .}\text{\textbackslash }\) character and delete it using (c-D). Enter:

\[
\langle\text{c-0}\rangle \quad \text{sbreak} \quad \text{SPACE}
\]

click `MOUSE-LEFT` on the } after the ? character in the soft break display form, and enter \( \text{\textbackslash .}\text{\textbackslash }\).

### 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 convenience, 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 \( \vdash \) the \( i \)th 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, \( \gg \) instead of the turnstile symbol \( \vdash \) is used to separate 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:

1. \( H_1 \)

\vdots

\( n. \ H_n \)

\( \vdash C \)

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 bound variables are assumed to have been properly declared; that is, declaration for each variable.

```plaintext
H_1, \ldots, H_n \vdash C
```

```plaintext
1. H_1

\vdots

n. H_n

\vdash C
```
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 8.13.

3. Hypothesis variables 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 it 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 8, 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.

Users can declaratively describe what they want to prove, and the refiner takes care of the unrefined parts of the proof.
1. TacticText is parsed by the ML parser into a tactic, and is applied to the proof node \texttt{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 \texttt{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 \texttt{tactic\_rule}(TacticText; \( p \)).

3. What is inserted back into the proof tree to replace \texttt{unrefined}(g) is

\[
\texttt{refined}(g; \texttt{tactic\_rule}(\texttt{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 \texttt{!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 8.12.

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 unproved 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 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 not visible until the user asks for changes to be explicitly saved.
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.

```
1 EDIT THM cantor
2 # top 1 1
3 1. f: N → N → N
   2. ∀g:N → N. ∃i:N. f i = g ∈ N → N
      ⊢ False
4 BY With 'λn.f n n + 1' (D 2) THENW Auto
5 1* 2. n: N
     ⊢ 0 ≤ f n n + 1
6 2# 2. ∃i:N. f i = λn.f n n + 1 ∈ N → N
     ⊢ False
```

Figure 7.1: Proof Window on Refined Proof Node

```
1 EDIT THM cantor
2 # top 1 1 2
3 1. f: N → N → N
   2. ∃i:N. f i = λn.f n n + 1 ∈ N → N
      ⊢ False
4 BY <refinement rule>
```

Figure 7.2: Proof Window on Unrefined Proof Node

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.

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 4 below. The BY is part of the proof node display, and is not part of the tactic.

5. This is the refinement rule placeholder.

Further examples that should be viewed in edit mode. Figure 7.1 shows the first child of the root of the proof, and Figure 7.2 shows the second child of the proof node in Figure 7.1.
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

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M-B)</td>
<td>\textsc{back-part}</td>
</tr>
<tr>
<td>(M-F)</td>
<td>\textsc{forward-part}</td>
</tr>
<tr>
<td>(M-A)</td>
<td>\textsc{first-part}</td>
</tr>
<tr>
<td>(M-E)</td>
<td>\textsc{last-part}</td>
</tr>
<tr>
<td>(M-P)</td>
<td>\textsc{up-to-parent}</td>
</tr>
<tr>
<td>(M-)</td>
<td>\textsc{up-to-top}</td>
</tr>
<tr>
<td>(M-N)</td>
<td>\textsc{down-to-child}</td>
</tr>
<tr>
<td>RETURN</td>
<td>\textsc{next-unrefined-leaf}</td>
</tr>
</tbody>
</table>

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 \textit{part} of the window is either a goal sequent, a refinement rule, a rule placeholder or a subgoal.

The \textsc{forward-part} and \textsc{back-part} commands move the cursor within a proof window from part to part, if necessary scrolling the window. \textsc{first-part} moves the cursor to the goal and \textsc{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 \textsc{up-to-parent} command, executed with a cursor in any part of the window, shifts the window one level up the proof tree. \textsc{down-to-child}, executed with the cursor over a subgoal part, shifts the window down the proof tree to that subgoal. The \textsc{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, \textsc{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

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C-Z)</td>
<td>\textsc{exit-proof}</td>
</tr>
<tr>
<td>(C-J)</td>
<td>\textsc{jump-next-window}</td>
</tr>
<tr>
<td>[TAB]</td>
<td>\textsc{jump-ml}</td>
</tr>
<tr>
<td>(C-S)</td>
<td>\textsc{select}</td>
</tr>
<tr>
<td>(C-T)</td>
<td>\textsc{transform}</td>
</tr>
</tbody>
</table>

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 \textit{select} and \textit{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 \texttt{view} function is used on the object name in the ML top-loop. However, even when the object name is not expanded, it may still be useful to use the \texttt{view} function on theorem objects with compressed proofs. If the proof is large, expansion of the proof may take some time. 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.

7.6.4 Mouse Commands

| MOUSE-RIGHT | MOUSE-SELECT | select main goal, rule or sequent |
| MOUSE-LEFT  | MOUSE-JUMP   | jump to window / parent / child |

Table 7.3: Mouse Commands for Proof Windows

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

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.
Chapter 8

Tactics

8.1 Introduction

8.1.1 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 \( \text{var list} \).

8.1.2 Universes and Level Expressions

In Nuprl’s type theory, types are grouped together into universes. Types built from the base types such as \( Z \) or \( \text{Atom} \) 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 \( P_1 \ldots P_i \ldots \), 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:

\[
L ::= v \\
| k \\
| L \ i \\
| L' \\
| [L] \ldots [L]
\]
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:

$$\forall A : P_i . \forall B : P_i . A \Rightarrow (B \Rightarrow A)$$

rather than

$$\forall A : P_i . \forall B : P_j . A \Rightarrow (B \Rightarrow A)$$

### 8.1.3 Formula Structure

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

\[
P ::= \forall x : A . P \mid Q \Rightarrow P \mid P \Leftrightarrow Q \\
| P \land P \mid P \Leftarrow P \\
| R
\]

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

### 8.1.4 Soft Abstractions

Certain abstractions can be designated as \textit{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.
In the basic libraries, the soft abstractions are

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>member</code></td>
<td>( t \in T ) =def ( t = t \in T )</td>
</tr>
<tr>
<td><code>nequal</code></td>
<td>( x \neq y \in T ) =def ( \lnot(x = y \in T) )</td>
</tr>
<tr>
<td><code>prop</code></td>
<td>( \mathbb{P}_i ) =def ( \bigcup_i )</td>
</tr>
<tr>
<td><code>and</code></td>
<td>( A \land B ) =def ( A \times B )</td>
</tr>
<tr>
<td><code>or</code></td>
<td>( A \lor B ) =def ( A + B )</td>
</tr>
<tr>
<td><code>implies</code></td>
<td>( A \Rightarrow B ) =def ( A \rightarrow B )</td>
</tr>
<tr>
<td><code>rev_implies</code></td>
<td>( A \Leftarrow B ) =def ( B \Rightarrow A )</td>
</tr>
<tr>
<td><code>iff</code></td>
<td>( A \Leftrightarrow B ) =def ( (A \Rightarrow B) \land (A \Leftarrow B) )</td>
</tr>
<tr>
<td><code>exists</code></td>
<td>( \exists x:A. B_x ) =def ( x:A \rightarrow B_x )</td>
</tr>
<tr>
<td><code>all</code></td>
<td>( \forall x:A. B_x ) =def ( x:A \rightarrow B_x )</td>
</tr>
<tr>
<td><code>ge</code></td>
<td>( i \geq j ) =def ( j \leq i )</td>
</tr>
<tr>
<td><code>gt</code></td>
<td>( i &gt; j ) =def ( j &lt; i )</td>
</tr>
<tr>
<td><code>lelt</code></td>
<td>( i \leq j &lt; k ) =def ( (i \leq j) \land (j &lt; k) )</td>
</tr>
<tr>
<td><code>lele</code></td>
<td>( i \leq j \leq k ) =def ( (i \leq j) \land (j \leq k) )</td>
</tr>
</tbody>
</table>

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 blur 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.

### 8.1.5 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.

### 8.1.6 Proof Annotations

Nuprl proofs (ML terms of type `proof`) can be marked with extra information which is then kept with the ML code. Annotations are currently only used for proof editing and are not part of the formal system. They are useful for providing additional context to human proof readers. Annotations are not part of the proof itself.

Nuprl supports two kinds of annotations:

- **Goal labels**: These are used to label goals, which are statements that need to be proven. Goal labels can be used to organize the proof, make it easier to follow, and provide context for the reader.
- **Tactic arguments**: These are used to pass additional information to tactics, which are proof procedures that can be applied to goals. Tactic arguments can be used to customize the behavior of tactics based on the specific goals they are applied to.

Annotations are not part of the formal system and are not required to be preserved in the formal proof. They are intended to be read and understood by humans and can be ignored by automated proof checking systems.
8.1.6.1 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.

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 8.3.2 for how to discriminate on labels.

8.1.6.2 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.

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.
8.1. INTRODUCTION

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.

8.1.7 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 and instance term I, we say that I matches P, if there exists a substitution \( \theta \) 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 \( \theta P \) to be related by an order relation rather than an equivalence relation. The weaker \( R \) also can allow I and \( \theta P \) 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 \( \theta \), 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.
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 $[x \mapsto 2]$. Since in general 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 route 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 $\forall i : \mathbb{N}. \quad i \geq 0$ matches the pattern $\forall x : \mathbb{N}. \quad P[x]$, giving the second-order term binding $[P \mapsto i.i \geq 0]$.

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

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 $U_i \times U_j$ might match the the pattern $U_k \times U_{[k \ n]}$ 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 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:

$$\forall A : U_i. \forall l : A \quad \text{List. length}(l) \in \mathbb{N}$$
binding $A \rightarrow Z$ by matching the inferred type of $3::2::1::[]$ which is $Z\text{List}$, against the declaration type of $l$, which is $A\text{List}$. Similarly, by inferring the universe which $Z$ inhabits, we can get a binding for the universe level $i$.

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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; v, \text{so}_\text{lambda}(xs.t')&gt;$</td>
<td>The higher-order binding $v \mapsto xs.t'$</td>
</tr>
<tr>
<td>$&lt; v, \text{parameter}{le:1}&gt;$</td>
<td>The level exp binding $v \mapsto le$</td>
</tr>
<tr>
<td>$&lt; v, \text{parameter}{w:v}&gt;$</td>
<td>The bound variable binding $v \mapsto w$</td>
</tr>
<tr>
<td>$&lt; v, \text{parameter}{p:*}&gt;$</td>
<td>The parameter variable binding $v \mapsto p$</td>
</tr>
<tr>
<td>$&lt; v, t&gt;$</td>
<td>The first-order binding $v \mapsto t$</td>
</tr>
</tbody>
</table>

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

## 8.2 Basic

Nearly all tactics in this section correspond to one or two primitive rules.

### 8.2.1 Simple

**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 $\ldots x:T\ldots \vdash x = x \in T$ 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** $t$

$=_{def} \text{AssertAt} 0 \ t$. 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$j$

$$\text{main} \ldots \vdash A \Rightarrow C$$

If hyp $j$ is a declaration,

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

BY MoveToConcl$j$
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.

8.2.2 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 an initial axiom and a, t \in T, then there are two cases: one where a is an instance of the axioms of the library, and one where a is an instance of one of the a\_\_x variables or any of the a\_\_x\_x variables. These are exceptional cases, and MemCD will report an error message in both cases.
An example application of the MemCD tactic is
\[ ... \vdash <a, b> \in x:A \times B_x \]

BY MemCD

\[
\begin{align*}
\text{subterm1: } & \vdash a \in A \\
\text{subterm2: } & \vdash b \in B_a \\
\text{wf: } & \vdash x:A \vdash B_x \in \mathbb{U}_x
\end{align*}
\]

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

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

MemHD

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

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

As EqTypeHD but works on membership terms.

UnivCD

Repeatedly decompose $\forall$ and $\Rightarrow$ terms in conclusion. If UnivCD reaches a guard term, it removes the term and stops. In other words, UnivCD decomposes simple universal formulae. (Section 8.1.3 describes the structure of universal formulae.)

GenUnivCD

As UnivCD except works on general universal formulae in the concl.

ExistHD

Repeatedly decompose $\exists$ and $\land$ terms in hypothesis $i$.

GenExistHD

Repeatedly decompose $\exists$, $\land$ and $\lor$ terms in hypothesis $i$.

## 8.3 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.

### 8.3.1 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]

Repeatedly apply T1, T2, ..., Tn to the current goal.
Run T1, generating exactly $n$ subgoals. Then run Ti on the $i$th subgoal (numbering subgoals from left to right.)

Try T

$$=_{def} T \mathrm{ORELSE} \ Id$$

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.

### 8.3.2 Label Sensitive Tacticals

IfLab lab T1 T2

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

T1 THENM T2 $$=_{def} T1 \mathrm{THENM} T2$$

T1 THENA T2 $$=_{def} T1 \mathrm{THENA} T2$$

T1 THENW T2 $$=_{def} T1 \mathrm{THENW} T2$$

IfLabL [l1,T1; l2,T2; ... ;ln,Tn]

Run the first Ti 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 $l_i$ until a match succeeds. Then if the subgoal also has number label $j$, retrieve the $j$th tactic from $Ts_i$ 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 $l_i$ 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.

### 8.3.3 Multiple Clause Tacticals

OnClause c (T : int -> tactic)

Run T on clause c. Use of this tactical can make tactics a little more readable. For example, you can write OnClause 3 D rather than D 3.

OnConcl (T : int -> tactic) $$=_{def} \mathrm{OnClause} 0 T$$

OnHyp i (T : int -> tactic) $$=_{def} \mathrm{OnClause} i T$$

OnClauses [c1;...;cn] (T : int -> tactic)

Run T on clauses c1 to cn in that order.

OnHyps [i1;...;in] (T : int -> tactic)

Run T on clauses c1 to cn in that order.
OnMHyps \[i1;...;in\] (T : int -> tactic) = def OnClauses \[c1;...;cn\] T

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 \texttt{int\_upper\_ind} and \texttt{int\_seg\_ind} at the end of the \texttt{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 \texttt{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.

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

- \texttt{ListInd i}
  - Do list induction on hypothesis \(i\). Generates \texttt{upcase} and \texttt{basecase} subgoals. First moves any depending hyps to concl.

- \texttt{IntInd i}
  - Do integer induction on hypothesis \(i\). Generates \texttt{upcase}, \texttt{basecase} and \texttt{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.

- \texttt{NSubsetInd i}
  - Do induction on subrange of the natural numbers. Hyp \(i\) should be a \texttt{nat}, \texttt{nat\_plus}, \texttt{int\_upper} or \texttt{int\_seg} abstraction. Generates two main subgoals - \texttt{basecase} and \texttt{upcase}- and approximately 15 aux subgoals which should always be easily solvable by \texttt{Auto}.

- \texttt{NCompInd i}
  - Do complete natural number induction on hyp \(i\). Hyp \(i\) must be a \texttt{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 \texttt{well\_fnd} has some definitions for well-founded induction. In particular it defines the tactic \texttt{InvImageInd}. 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 \texttt{rank} function which maps elements of \(B\) to elements of \(A\). The tactic is described in the object \texttt{inv\_image\_ind\_tac}.

The theory \texttt{bool\_1} defines various tactics for case splitting on the value of boolean expressions in the conclusion. Tactics include \texttt{BoolCasesOnCExp} and \texttt{SplitOnConclITE}. View the theory for details.

### 8.5 Forward and Backward Chaining

Forward and backward chaining involves treating a component of a universal formula (see Section 8.1.3) 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 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.
CHAPTER 8. TACTICS

BackThruLemma name
BackThruHyp i
The name (i) argument selects the lemma (hypothesis) to back-chain through. Subgoals corresponding to antecedents of the lemma (hyp) are labelled with antecedent. The rest are labelled wf.

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.

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 ··· n where n is the number of hyps. Skips hyps which declare variables.
- rev_hyps: As hyps but in order n···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:1}. In the term editor you select this term by the name parameter.)

InstHyp [t1;...;tn] i
Instantiate universal formula in hyp with terms t1 through tn.
8.6 Equational reasoning

We describe here a number of fairly simple tactics. For more general equational and inequational reasoning, see section 8.8 about the rewrite package.

8.6.1 Direct computation

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

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 primitive redices in clause c.

8.6.2 Substitution

Subst (eq:term) c
eq should be a proposition of form \( t_1 = t_2 \in T \). The effect of Subst is to replace all occurrences of \( t_1 \) in clause c by \( t_2 \). 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 \( t_1 = t_2 \in T \). 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 \( H_1 \ldots H_n, z:T \vdash C_z = C_z \in U_\ast \)
equality \( H_1 \ldots H_n \vdash a = b \in T \)

where \( U_\ast \) 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.

8.7 Decision Procedures

8.7.1 Eq

Eq does trivial equality reasoning. It proves goals of form \( H \vdash t = t' \in T \) using hypotheses that are equalities. For example:

\[
H_1 \ldots H_n \vdash C_a
\]

BY Eq 'a = b \in T' 0

main \( H_1 \ldots H_n \vdash C_b \)
wf \( H_1 \ldots H_n, z:T \vdash C_z = C_z \in U_\ast \)
equality \( H_1 \ldots H_n \vdash a = b \in T \)

where \( U_\ast \) is the inferred universe for clause c. Universe inference can be overridden by supplying an optional universe argument with the Using tactical.
8.7.2 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 $s < t$, $s \neq t$, $s \leq t$, $s \geq t$, $s = t \in \mathbb{Z}$ 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

$$H \vdash C_1 \lor \ldots \lor C_m$$

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 $A_1 \land \ldots \land A_n (n \geq 1)$, 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 $s < t \lor t < s$. 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 $\ast$.

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 $\theta$ is one of $<$, $=$ or $\geq$ ($s \geq t$ is equivalent to $-t < s$).

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).

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

SupInf is a Nuprl's implementation of Bledsoe's Sup-Inf procedure for solving linear equalities and inequalities over the integers. SupInf recognizes the standard integer subset types \textit{nat}, \textit{nat}+\textit{plus}, \textit{int}+\textit{upper int}+\textit{iseg} and \textit{int}+\textit{seg}. SupInf currently solves slightly different kinds of arithmetic goals from \textit{Arith}. It is better at dealing with reasoning about inequalities, but it cannot handle arithmetic expressions that have non-linear components or involve functions that return integer values. This is not a limitation of the algorithm; just no-one has yet generalized it to these cases.

SupInf identifies counter-examples if it fails. These can be viewed by looking at value of the ML variable \textit{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.

8.8 Rewriting

8.8.1 Introduction

Rewriting in general involves the repeated application of a set of rewrite rules to a clause of a sequent. A rewrite rule is commonly derived from a lemma of form: \( \forall \tau. \; a_{\tau} = b_{\tau} \). The derived rule might be to replace instances of \( a_{\tau} \) with corresponding instances of \( b_{\tau} \). Sets of rewrite rules are often used for simplifying expressions and putting them in normal form.

Nuprl's rewrite package is a collection of ML functions for creating rewrite rules and applying them in various fashions to a clause of a sequent. It is based around ML objects of type \textit{convn} called conversions, similar to those found in other tactic based theorem provers such as LCF, HOL and Isabelle.

\textit{convn} is an ML concrete type abbreviation for the type

\[
\text{env} \rightarrow \text{term} \rightarrow (\text{term} \# \text{reln} \# \text{just}).
\]

\textit{env}, \textit{reln} and \textit{just} are abstract types for \textit{environments}, \textit{relations} and \textit{justifications}. Their precise structures are not important now and are explained later. 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.

A tactic \textbf{Rewrite} : \textit{convn} \rightarrow \textit{int} \rightarrow \textbf{tactic} is used for applying a conversion to some clause of a sequent. It takes care of executing the justifications generated by conversions.

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 \textit{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 8.8.7.

8.8.2 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 \textit{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 \texttt{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 8.1.7).
- 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

\[ \vdash \forall z: T\ A_z \Rightarrow t_z \triangleq t'_z \]

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 \( \Rightarrow \). The system assumes other terms do not modify the environment, unless otherwise told by the user. see the \texttt{rw-types.ml} file for further details.

### 8.8.3 Relations

User defined equivalence and order relations as well as Nuprl equality are supported.

Each user defined relation can be declared by a call to one of the ML functions:

```
add_equiv_rel_info (name:tok) (rels:tok list) ;;
add_order_rel_info (name:tok) (inverse-name:tok) (is-refl:bool) (rels:tok list) ;;
```

This call should be inserted into an ML library object that is positioned before any uses of the relation. \( name \) should be the opid of the relation. \( rels \) should be a list of the opids of relations which are strictly stronger. \( rels \) should always include the opid \texttt{equal} and the pseudo-opid \texttt{identity} if the relation is reflexive. \( inverse-name \) should be the opid of the inverse relation. \( is-refl \) should be \texttt{TRUE} if the relation is reflexive. Otherwise it should be \texttt{FALSE}.

Built in is knowledge of Nuprl's equality relation, and also \( \Rightarrow, \triangleq \) and \( \iff \).

If a relation is not explicitly named by the designer, then the system assumes that the user is referring to the built-in version of the relation.
An order relation $R$'s inverse should always be defined directly in terms of $R$. For example, the definition of the abstraction \texttt{rev_implies} is

$$P \Leftarrow Q \overset{\text{def}}{=} Q \Rightarrow P$$

Order relation inversion is accomplished by folding and unfolding such definitions.

User defined relations should always have at least two subterms. The last two subterms are taken to be the principle arguments of the relation.

Relations can be a type or a value in the type \texttt{Bool}. In the latter case, uses of the relation in lemmas where a type is expected should be wrapped in the abstraction \texttt{assert}.

### 8.8.4 Justifications

There are two types of justifications. Conversions generating both types of justification can be freely intermixed; the system takes care of converting computational justifications to tactic justifications when necessary.

**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.

### 8.8.5 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 \hspace{1cm} \textit{type of conversions}
- $cs^*$ : convn list
- $e^*$ : env \hspace{1cm} \textit{environment}
- $i j$ : int \hspace{1cm} \textit{hypothesis or clause indices}
- $addr$ : int list \hspace{1cm} \textit{subterm address}
- $a$ : tok \hspace{1cm} \textit{name of abstraction}
- $name$ : tok \hspace{1cm} \textit{name of lemma or cached conversion}
- $names$ : tok list \hspace{1cm} \textit{list of names}
- $t t1 t2$ : term

### 8.8.5.1 Trivial Conversions

- \texttt{FailC} always fail
- \texttt{IdC} identity conversion
8.8.5.2 Conversionals

\( c_1 \) ORELSEC \( c_2 \) apply \( c_1 \). If \( c_1 \) fails, apply \( c_2 \)

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 ith 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)

8.8.5.3 Atomic Direct-Computation Conversions

TagC tagger do forward computations indicated by tags in \((\text{tagger } t)\).

Fail if tagger fails. tagger should be simple. e.g. \( \text{tag } \text{redex} \)

RedexC if \( t \) is a primitive redex, contract it.

AbRedexC if \( t \) is a primitive redex or a redex hidden under abstractions, contract it.

ExtractC names if \( t \) is the extract term of a theorem listed in \( \text{names} \)

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 \( \text{as} \), 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 \( \text{as} \).

FoldTopC a = UnfoldsC [a]

RecUnfoldTopC a If a is a recursively defined term, then

unfold the recursive definition.

RecFoldTopC a If a is an unfolding of a recursively defined term, then

fold the recursive definition.
but tactic-based conversions can be added too.

RecUnfoldTopC and RecFoldTopC work only with recursive definitions that have been introduced with the add_rec_def function. See Section 9.2.2 for more details.

8.8.5.4 Composite Direct Computation Conversions

ReduceC = 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 ReduceC
NormalizeC = SweepDnC (RepeatC UnfoldTopAbC) ANDTHENC ReduceC
LazyReduceC = TryC (SubIfC is_principle_arg LazyReduceC)
               ANDTHENC TryC (RedexC ANDTHENC LazyReduceC)
RecUnfoldC a = SweepUpC (RecUnfoldTopC a)
RecFoldC a = SweepUpC (RecFoldTopC a)
AbReduceC = Repeat (SweepDnC AbRedexC)

8.8.5.5 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 list1 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

8.8.5.6 Lemma and Hypothesis Conversions

The conversions here turn lemmas and hypotheses that have the structure of either simple or general universal formulae (see Section 8.1.3 for definitions of these) into rewrite rules. The consequents of these formulae must each be of form a r b where r is some suitable equivalence or order relation.

The conversion 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

LemmaC derives a rewrite rule from the simple universal lemma name. LemmaC is an instance of

GenLemmaWithThenLC
(n:int)
(hints:(var # term) list)
(Tacs:tactic list)
(name: tok)
The arguments to \texttt{GenLemmaWithThenLC} are as follows:

- \texttt{n} indicates that the \texttt{n}th consequent of a general universal formula. If \(-1\) is used for \texttt{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.

\textbf{hints} is for supplying bindings for variables in the lemma that Nuprl's matching routines can't guess on their own.

\textbf{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 \textbf{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 \textbf{Tacs} than antecedents, \textbf{Tacs} is padded on the left up to the length of the antecedents with copies of the head of \textbf{Tacs}. If \textbf{Tacs} is empty, then the rewrite goes through unconditionally.

\textbf{name} is the name of the lemma.

The specializations of \texttt{GenLemmaWithThenLC} are:

\begin{align*}
\text{LemmaC name} & = \text{GenLemmaWithThenLC} (-1) \emptyset \emptyset \text{name} \\
\text{GenLemmaC n name} & = \text{GenLemmaWithThenLC} n \emptyset \emptyset \text{name} \\
\text{LemmaWithC hints name} & = \text{GenLemmaWithThenLC} (-1) \text{hints} \emptyset \emptyset \text{name} \\
\text{LemmaThenLC Tacs name} & = \text{GenLemmaWithThenLC} (-1) \emptyset \text{Tacs \ name}
\end{align*}

Note that \texttt{GenLemmaWithThenLC} does a fair amount of preprocessing based on the arguments up to and including \textbf{name}.

\textbf{HypC} \ i \ e'

\textbf{HypC} is similar in function to \textbf{LemmaC} except that it uses the simple universal formula in local hyp \(i\) to form the rewrite rule. A local hyp is an assumption in the local environment in which \textbf{HypC} is applied (see Section 8.8.2).

\textbf{HypC} obtains the local hypothesis from the environment \(e'\). In theory, this argument is unnecessary. Remember that whenever a conversion is evaluated, it is applied to an environment \(e\) and a term \(t\). \textbf{HypC} could access the local hypothesis from \(e\). However then it would do so \textit{every} time the conversion were applied to a term which would be rather inefficient. Usually the local hyp being used is one of the hypotheses in the sequent to which the rewrite tactic is applied. In this case it is best to grab a 'top-level' environment for the \(e'\) argument to \textbf{HypC}. For example, you might construct the rewrite tactic:

\texttt{Rewrite (\!\!e. HigherC (HypC 2 e) e) 0}

The rewrite conversions are shortly going to be rewritten such that for most purposes you will not have to worry about supplying this \(e'\) argument.

\textbf{HypC} and other hypothesis conversions are special cases of \texttt{GenHypWithThenLC}:

\begin{align*}
\text{HypC n i e'} & = \text{GenHypWithThenLC} (-1) \emptyset \emptyset \text{i e'} \\
\text{GenHypC n i e'} & = \text{GenHypWithThenLC} n \emptyset \emptyset \text{i e'} \\
\text{HypWithC hints name} & = \text{GenHypWithThenLC} (-1) \text{hints} \emptyset \emptyset \text{i e'} \\
\text{HypThenLC Tacs name} & = \text{GenHypWithThenLC} (-1) \emptyset \text{Tacs \ i e'}
\end{align*}

The first three arguments of \texttt{GenHypWithThenLC} are the same as explained for \texttt{GenLemmaWithThenLC} above.

\section*{8.8.6 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 \texttt{main}.

The rest have various labels that all fall into the \texttt{aux} subgoal label class.
8.8. REWRITING

\[ \text{RW} = \text{def} \ \text{Rewrite} \]
\[ \text{AbReduce} = \text{def} \ \text{RW} \ \text{AbReduceC} \]

To speed rewrites, try executing the ML statement `quick_rw := true` This tells \text{Rewrite} to suspend execution of justifications and use the \text{Fiat} tactic instead. Speedup can be dramatic: 20x or more! When doing this, you should always add \textsc{THEN} \text{Auto} to take care of the subgoals created by execution of the justification when proofs are checked with `quick_rw := false`. Needless to say, you should be confident that \text{Auto} will completely prove all subgoals created by the justification; if it doesn’t on proof replay with `quick_rw := false` then the replay will break.

\text{apply_conv} (c:\text{convn}) (t:\text{term}) : \text{term}

\text{apply_conv} evaluates a conversion with an empty environment. It is very useful for testing conversions.

8.8.7 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

\text{initialize_rw_lemma_caches} : \text{unit} \rightarrow \text{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.

8.8.7.1 Functionality Lemmas

Functionality lemmas give congruence and monotonicity properties of terms. They are required by the \text{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 \( \text{op} \) should have the form

\[ \forall z_1; S_1 \ldots z_k; S_k. \ \forall x_1; y_1; T_1; \ldots; x_n; y_n; T_n. \ A_1 \Rightarrow \ldots \Rightarrow A_m \]
\[ \Rightarrow x_1 \ r_1 \ y_1 \Rightarrow \ldots \Rightarrow x_n \ r_n \ y_n \Rightarrow \text{op}(x_1; \ldots; x_n) \ R \ \text{op}(y_1; \ldots; y_n) \]

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 \( \text{op} \).

The \text{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 \( \text{op} \) should be named \text{opid}\_functionality\_[\_index] where \text{opid} is the opid of \text{op} and \_index is an optional suffix, used to distinguish functionalities with the same opid from one another.
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:

\[
\forall A_1, A_2; U_i \\
\forall P_1: A_1 \rightarrow \Pi_i \forall P_2: A_2 \rightarrow \Pi_i \\
A_1 = A_2 \in U_i \\
\Rightarrow (\forall x: A_1 P_1[x] \leftrightarrow P_2[x]) \\
\Rightarrow \exists x: A_1, P_1[x] \leftrightarrow \exists x: A_2, P_2[x]
\]

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. \texttt{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.

### 8.8.7.2 Transitivity Lemmas

Transitivity lemmas give transitivity information for rewrite relations. They are used to construct the tactic justification in the \texttt{ANDTHENC} conversional.

Transitivity lemmas should be of form:

\[
\forall z_1: S_1 \ldots z_k: S_k, \forall x_1, x_2, x_3: T. A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1 \ x_2 \Rightarrow x_2 \ x_3 \Rightarrow x_1 \ r_c \ x_3
\]

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 \( r_b \).

\texttt{ANDTHENC} finds transitivity lemmas in the library by assuming that a naming convention has been followed. The lemmas must be named \texttt{opid-opid-transitivity-index}, where the \texttt{-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.

### 8.8.7.3 Weakening Lemmas

Weakening lemmas should have form

\[
\forall z_1: S_1 \ldots z_k: S_k, \forall x_1, x_2: T. A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1 \ r_a \ x_2 \Rightarrow x_2 \ r_b \ x_2
\]

where \( k \geq 0, m \geq 0 \) and \( r_b \) is some weaker relation than \( r_a \).

Lemmas in the library must be named \texttt{opid-opid-weakening-index}, where the \texttt{-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.

### 8.8.7.4 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 \texttt{opid-opid-inversion}.

Inversion lemmas are required for equivalence relations, but not equality or order relations. They are used by the \texttt{Rev*} atomic conversions, and in conjunction with the weakening, transitivity, and functionality lemmas when these lemmas mix order and equivalence relations.
8.9 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 \texttt{add_set_inclusion_info}. See the theory \texttt{int_1} for several examples of the use of this function.

8.10 Miscellaneous

Cases \([t_1;\ldots;t_n]\)

Does n-way case split. For example:

\[ \ldots \vdash C \]

BY Cases \([t_1;\ldots;t_n]\)

assertion: \[ \ldots \vdash t_1 \lor \ldots \lor t_n \]

\[ \ldots t_1 \vdash C \]

\[ \vdots \]

\[ \ldots t_n \vdash C \]

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 \((x:\text{var}) (v_x:\text{term}) (V_x:\text{term}) (i:\text{int})\)

If hypothesis \(\text{h}\) 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 \(v_x = v_x\), where \(v_x\) is the unlabelled term of free occurrences \(V_x\) with respect to rule \(\text{h}\).
If you about to give up hope on a theorem, don’t despair. This tactic is guaranteed to provide satisfaction.²

### 8.11 Autotactic

**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
- D on \( \land \) hyps.
- D on \( \land, \Rightarrow, \Leftarrow, \Leftarrow \), and \( \forall \) concl.
- MemCD for member and EqCD on reflexive equality conclusions. Only works on non-recursive primitive terms.
- Arith
  - 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 \( \text{rem}^3 \).)
  - If concl is \( a \in T \) or \( a = b \in T \) where \( T \) is subset of the integers, then open up \( T \).

**StrongAuto**

Like Auto but also does MemCD and EqCD on recursive primitive terms. This isn’t always a good idea, so the normal Auto doesn’t do this.

### 8.12 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.
8.13 Constructive and Classical Reasoning

8.13.1 Constructive Reasoning

The constructivity of Nuprl's logic manifests itself in two main ways with the tactics:

1. For any proposition \( P \), the goal \( P \lor \neg P \) is not in general provable.

2. When applying the \( \text{D} \) tactic to a hypothesis that has a set term outermost, the predicate part of the set term becomes a *hidden* hypothesis. For example:

\[
\ldots i.x:\{y:T|P_y\}, \ldots \vdash \ldots
\]

\[\text{BY \text{D}_i}\]

\[
\ldots i.x:T, [i+1].P_x, \ldots \vdash \ldots
\]

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.

8.13.2 Decidability

Many useful instances of \( P \lor \neg P \) are provable constructively and the \text{ProveDecidable} tactic is set up to construct these proofs in a systematic way. To discuss it, we first introduce the abstraction:

\[
\text{decidable}: \text{Dec}(P) =_{\text{def}} P \lor \neg P
\]

which can be found in the core.2 theory. It turns out that the property \text{Dec}(P) can be inferred for many \( P \) from knowing that \text{Dec}(Q) for the immediate subterms \( Q \) of \( P \). \text{ProveDecidable} takes advantage of this fact and attempts to prove goals of the form

\[
\ldots \vdash \text{Dec}(P)
\]

by backchaining with any lemmas in the Nuprl library that have names with prefix \text{decidable__}. (Note the \text{two} underscores.) There are many examples of such lemmas in the core.2 theory. \text{ProveDecidable} is usually invoked via the \text{Decide} tactic:

\text{Decide \( (Q:\text{term})\)}

Used to case-split on whether proposition \( Q \) is true or false. Generates two \text{main} subgoals; one with the new assumption \( Q \) and the other with the new assumption \( \neg Q \). \text{Decide} also creates a subgoal \( \ldots \vdash \text{Dec}(Q) \) and immediately runs the \text{ProveDecidable} tactic on this subgoal. \text{ProveDecidable} generates only subgoals with labels in the aux class. If \text{ProveDecidable} fails then \text{Decide} fails too. To understand why, use the tactic \text{Assert} to assert \text{Dec}(Q) and run the \text{ProveDecidable1} tactic to try to prove this assertion. ProveDecidable1 will generate subgoals with label \text{decidable?} to indicate
8.13.3 Squash Stability and Hidden Hypotheses

A hidden hypothesis $P$ in a sequent $\sigma$ can be unhidden if one of two conditions are met:

1. The proposition $P$ is squash stable.
2. The conclusion of $\sigma$ 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:

$$\text{squashstable} : \text{SqStable}(P) =_{\text{def}} \downarrow P \Rightarrow P$$

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 $\{ x: \text{Unit} | P \}$ where $x$ does not occur free in $P$).

As with decidability, it turns out that the property $\text{SqStable}(P)$ can be inferred for many $P$ from knowing that $\text{SqStable}(Q)$ for the immediate subterms $Q$ of $P$. It is also true that $\text{Dec}(P) \Rightarrow \text{SqStable}(P)$ for any $P$. The tactic $\text{ProveSqStable}$ takes advantage of these facts and attempts to prove goals of the form

$$\vdash \text{SqStable}(P)$$

by backchaining with any lemmas in the Nuprl library that have names with the prefixes squashtable___ or decidable__ (note the two underscores in each case). There are many examples of such lemmas in the core_2 theory.

Since $\text{ProveSqStable}$ can be rather slow, it isn’t called by the $\text{D}$ tactic when $\text{D}$ is applied to a set type hypothesis $\{ x:T | P_x \}$. However, $\text{D}$ does check the sequent conclusion for trivial ways in which it might be squash stable.

The $\text{D}$ 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 $\text{D}$ 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 $\text{ProvePropertiesLemma}$.

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 $\text{ProveSqStable}$.

- **UnhideAllHypsSinceSqStableConcl**
  - This tactic tries to prove the conclusion squash stable using $\text{ProveSqStable}$. 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 a declaration of form $A$ or a proposition of form $t \in A$ or $t = t' \in A$ where $A$ is an abstraction wrapped around a set type.
8.13.4 Classical Reasoning

To reason classically, you need to have as an explicit hypothesis
\[ \forall P \forall i. P \lor \neg P \]

It is best to have this hypothesis in the form of the \texttt{xmiddle} abstraction:
\[ \text{xmiddle: } XM_i =_{df} \forall P \forall i. \text{Dec}(P) \]

The \texttt{Decide} tactic recognizes whenever the \texttt{xmiddle} abstraction occurs as some hypothesis, and in this case is trivially able to justify decidability. Also, the \texttt{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 \texttt{xmiddle} proposition as a precondition of the theorem, so that the theorem is of form \( \vdash XM_i \Rightarrow P \).

There are two common cases when in proving a part of a constructive theorem, classical reasoning becomes admissible. These cases, and a recommended method in each case for adding an \texttt{xmiddle} hypothesis are as follows:

1. If the conclusion is the \textit{squashed exists} term \( \vdash \exists x:T. P_x \) and you are about to apply the tactic \texttt{With} \( t \ (D \ 0) \). Squashed exists is defined in \texttt{core-1} as:

\[
\text{sq_exists: } \vdash \exists x:T. P[x] =_{df} \{ x:T | P[x] \}
\]

First use the tactic \texttt{AddXM}:

\[
\vdash \vdash \exists x:T. P_x
\]

BY \texttt{AddXM 1 THEN With} \( t \ (D \ 0) \)

\[
\begin{align*}
\text{wf} & \ XM_i \ldots \vdash t \in T \\
\text{main} & \ XM_i \ldots \vdash P_t \\
\text{wf} & \ XM_i \ldots x:T \vdash P_x \in \mathcal{P}_i
\end{align*}
\]

\texttt{AddXM 1} alone adds the hypothesis \( XM_i \) as a hidden hypothesis, so there are no soundness problems here. \( XM_i \) becomes unhidden in the first and third subgoals since here the conclusion is recognized as being trivially squash stable. \( XM_i \) 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 \texttt{AddXM 1}. If the conclusion is obviously squash stable, then \( XM_i \) is added unhidden. If the conclusion is not obviously squash stable and \( XM_i \) gets added hidden, you should then run the \texttt{Unhide} tactic.

The \texttt{AddXM} tactic assumes that the proposition \( \vdash (\forall P \forall i. P \lor \neg P) \) 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.

8.14 Further Information

Consult the ML files. Start with \texttt{load-ml.ml} which loads all the other ML files.
Chapter 9

Theories

9.1 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.

9.2 Adding Definitions

Each definition in a Nuprl theory usually includes the following objects in the order in which they are listed here: (Note that `opid` is the opid for the abstraction for the definition.)

- 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 can be set up, by creating each object in turn and editing its contents from scratch. This is a rather laborious operation and it is much easier to use one of the ML functions `add` described in Section 9.2.1.
9.2.1 Adding Non-Recursive Definitions

`add_def`'s usage is:

```ml
add_def
lhs : term
rhs : term
place : string
=
() : unit
```

`lhs` and `rhs` should be the left-hand and right-hand sides of the desired 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 exploded term editing feature of the term editor to enter it. If the opid of the new definition is `id`, then `add_def` adds 3 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.
- a theorem object named `id wf` for a well-formedness theorem.

`add_def` can be run from the ML top loop or from within an ML object. It does nothing if an object with name `id` already exists in the library.

Often after running `add_def` you will want to customize the display form. For example you might add whitespace related formats to the display form left-hand side.

When doing proofs, the basic tactics for folding and unfolding abstractions are the `Fold` and `Unfold` tactics. See Chapter 8 for details.

9.2.2 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 `add_rec_def` greatly simplifies constructing general recursive definitions using the Y-combinator. `add_rec_def`'s usage is as follows:

```ml
add_rec_def
lhs : term
rhs : term
=
() : unit
```

The `lhs` term should have form:

```
\( id\{p_1, \ldots, p_i\}(x_1; \ldots; x_j)y_1 y_2 \ldots y_k \)
```

where \( i, j, k \geq 0 \) and the \( x_m \) and the \( y_n \) are all variables. `add_rec_def` allows you to create recursive definitions with both curried arguments supplied by application, and subterm arguments.
The \( rhs \) term should include at least instance of the head of the application on the left-hand side; i.e., a term of form:

\[
id\{p_1, \ldots, p_i\}(t_1; \ldots; t_j)
\]

where the \( p_q \)'s are the same as in \( lhs \).

\texttt{add_rec_def} should always be put in an ML object with name \texttt{id.ml}. It needs to be kept around because it supplies information to the \texttt{RecFoldC} and \texttt{RecUnfoldC} conversions on how to fold an unfold the recursive definitions. These are described in Chapter 8. This ML object also serves to document the recursive definition.

\texttt{add_rec_def} constructs the recursive definition using the \( Y \) combinator. It adds three objects to the library:

- a display object named \texttt{id_df} which defines a default display form for the \texttt{id} term.
- an abstraction object named \texttt{id}, which defines the following abstraction:

\[
\texttt{id}\{p_1, \ldots, p_i\}(x_1; \ldots; x_j) = \texttt{def } Y(\lambda f \ x_1 \ldots x_jy_1 \ldots y_k.\texttt{rhs}\{\texttt{id}\{p_1, \ldots, p_i\}(t_1; \ldots; t_j) \mapsto f \ t_1 \ldots t_j\})x_1 \ldots x_j
\]

- a theorem object named \texttt{opid_wf} for a well-formedness theorem.

The \texttt{id_df} and \texttt{id} objects are added immediately before the \texttt{id.ml} object, and the \texttt{id_wf} object is added immediately after.

Often after running \texttt{add_rec_def} you will want to customize the display form. For example you might add whitespace related formats to the display form left-hand side.

We give an example of the use of \texttt{add_rec_def}.