The Nuprl Proof Development System, Version 3.2 Reference Manual and User's Guide

The Nuprl Group

September 20, 1991

Contents

Chapter 1

Introduction

This document is a reference manual for the version of Nuprl that runs on Unix/X-window platforms. Nuprl will also run on Symbolics Lisp machines and on Sun workstations running the SunView window system, although these versions of Nuprl are now considered obsolete; see the files `doc/symbolics` or `doc/sunview` for further information on these versions.

In addition to the system proper, this manual also documents the tactic collection which is loaded into Nuprl by default. This documentation is somewhat sketchy. Also, many of the mathematics libraries built at Cornell using Nuprl have used tactic collections different from the one described here. The chapter on the tactic collection also gives some examples of their use.

It is assumed that the reader is familiar with Nuprl at the level of the "tutorial" chapters of the book *Implementing Mathematics with the Nuprl Proof Development System*, Constable *et al* — henceforth referred to as "the book". The tutorial part of the book introduces the system and its type theory and provides a number of examples. The "reference" portion, excluding the parts of Chapter 8 on the type system and its semantics, is superseded by this reference manual. Chapter 9 in the reference portion also contains some useful examples and discussions of tactic writing that are not reproduced here. The "advanced" portion of the book deals with application methodology, gives some extended examples of mathematics formalized in Nuprl, and also describes some extensions to the type theory which have not been implemented.

This manual assumes that Nuprl will be running on a machine that has a three-button mouse. However, except for a few inessential user-interface features, anything that can be done with the mouse can be done by other (although possibly less convenient) means.

1.1 System Overview

One interacts with Nuprl using a collection of windows. These windows are at the "top-level" in the X environment, so they can be managed (positioned, sized, etc.) in the same way as other top-level applications such as X-terminals. Consult your local documentation

for details on general window management. 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 which appears as a small solid black rectangle. The specific location of the cursor determines the semantics of keyboard strokes and mouse clicks, and is independent of the location of the mouse.

When Nuprl is first invoked it will automatically create two windows, one called the *command window* and the other the *library* window. These two windows remain throughout the session and new ones cannot be created. The command window, when active, accepts lines of text terminated by a carriage return. The effect of entering a line of text depends on the *mode* of the command window. The mode is indicated by the prompt printed in the window at the beginning of the current line. There are five modes (details are given below): *command mode* (prompt `P>`) for entering basic commands, *ML mode* for entering ML expressions to be evaluated, *eval mode* for entering expressions to be evaluated as type theory programs or as bindings of programs to identifiers, *Lisp mode* for entering Lisp expressions to be evaluated, and *instantiation mode* for entering names of definitions to be instantiated.

The library window displays the current contents of Nuprl's library. The library is a list of named objects. An object is either a definition, a theorem together with its proof, or a text object containing either ML code or expressions appropriate for eval mode.

There are two other kinds of windows. Both are for editing objects in the library. Such windows can be created using commands entered in the command window. One kind of editing window is for text editing. These windows are used for editing definitions, goals of theorems, and rules applied in proofs. The other kind of window is a *refinement-editor* window, which is used to view and manipulate proofs as tree-structured objects.

Proofs are trees whose nodes consist of sequents and justifications. A justification is either the name of a rule, *empty*, or *bad* (ill-formed or inapplicable). Rules are either *primitive* rules, which are rules directly related to the type theory, or the *tactic* rule, which consists of the text of an ML program called a *tactic*. Proofs are built and modified by textually editing justifications and by applying *transformation tactics*.

1.2 Summary of Changes

With the exception of the addition of a tactic collection, the changes that have been made to Nuprl since the book was written have been relatively minor. The main change was the addition of an interface to X-windows. There were also changes and additions to the rules, addition of several user interface features, and implementation of a substantial set of new ML "utilities". The changes to the set of rules are as follows.

- Modification of the *direct-computation*, *extensionality*, *set* and *arith* rules.
- Addition of "reduce" rules for decision terms (`atom_eq` etc).

- Addition of the *monotonicity*, *division*, *thinning*, *eval* and *reverse-direct-computation* rules.
- "Experimental" addition of "simple" recursive types and the "object" type (see Appendix B).

Some of the changes to the user interface are:

- additional bindings of keys to special functions,
- ability to use "special" characters such as $\forall$ in definitions,
- new features involving the mouse, and
- some new commands.

The most significant new ML utilities deal with printing out proofs and libraries. These can be printed out in a reasonably readable textual format, or as a source file for Latex.

1.3 Learning to Use Nuprl

One way to get started learning how to use the Nuprl system is to try some of the examples in Chapters 3 and 4 of the book. These examples give a tutorial-like introduction to basic commands for editing text and proofs. Before doing this, though, Section 1.6 below on special keys and mouse-clicks should be read.

Many of the examples in Chapters 3 and 4 have been worked out and stored in the library `lib/basics/lib`. In particular, the library contains basic definitions for logical notions and proofs of a few simple logical facts and of the existence of integer square roots. The library makes use of one of the special fonts distributed with Nuprl. If these fonts have not been installed at your site, you can use the library "ulib" in place of "lib" above (the "u" is for "ugly", since the library replaces, for example, the symbol for the universal quantifier by the string "All").

To load this library, one can type

```
load fully top from lib/basics/lib
```

(followed by a carriage return) in the command window. In English, this command means: *load the library objects from the file lib/basics/lib into the current library starting at the top, fully reconstructing all proofs.*

1.4 Interactions with Lisp

Nuprl is a Lisp program, and a few Nuprl-related operations, in particular starting Nuprl, must be done in Lisp. This section describes all such operations.

The following assumes that Nuprl has been correctly installed and also that a Lisp image containing Nuprl has been created. If a special Lisp image has not been created, then you will need to follow the instructions on loading Nuprl into Lisp in the document `doc/installing-nuprl`. Once the Lisp image containing Nuprl is running, subsequent input will be interpreted as Lisp expressions to be evaluated. To start Nuprl, evaluate

`(nuprl "`*host*`")`

where *host* is the name of the machine where Nuprl's windows are to appear. The evaluation of this function terminates when the Nuprl session is exited.

If there is a file named `nuprl-init.lisp`[1] in your home directory and if Nuprl has not been started since it was loaded or since the last time it was reset (see below), then starting it will cause the file to be immediately loaded. This allows various window-system customizations to be done. The file `sys/sample-nuprl-init-for-X`" gives an example of some useful customizations for X-windows. Details on possible customizations are given below.

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

Most Lisps allow computations to be interrupted. This is usually done by entering c-C (that is, control-C) in the Lisp window. This will cause Lisp to enter its debugger, from which the computation can be resumed or aborted (in Lucid, `:a` aborts and `:c` resumes). 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. To restart Nuprl it is sufficient to evaluate `(nuprl)`.

It is possible (though not too likely) for Nuprl to get an error which will cause the Lisp debugger to be entered. This will likely be due to a user improperly setting an X-window parameter using the customization facility described below. If this happens, the mistake must be fixed and Nuprl reinvoked.

On rare occasions Nuprl's window system may get an error from which it is impossible to recover by just aborting and reinvoking. In this case, abort out of Nuprl, and enter the Lisp form `(reset)`. This should fix the problem. It will cause Nuprl's window system to be reinitialized, but otherwise the state will be left intact.

For those unfamiliar with Lisp-based systems, if the system appears to be inexplicably stuck, check the window running Lisp to see if Lisp is garbage-collecting.

[1] The filename extension for lisp source files, `lisp` in this case, depends on the Common Lisp and kind of machine being used.

1.5 Customizing Nuprl

The Lisp function `change-options` can be used to set various parameters affecting Nuprl's window system. An appropriate place to call this function is from your init file. The default values are reasonable, so you need not have an init file.

The `change-options` function takes an argument list consisting of keywords and associated values. For example, to set the options :host and :font-name to "moose" and "fg-13" respectively, put the form

```
(change-options :host "moose" :font-name "fg-13")
```

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` (`"fixed"`). The name (a string) of a font to use for the characters in Nuprl windows. The font must be fixed-width. The default is reasonable and should always be available, but it does not contain the special characters used in many of the libraries built at Cornell. These libraries will have appearance problems with wimpy fonts, although they should be correct. Some of the better fonts are as follows. fg-25, fg-30, fgb-25, fg-40: these have all the special characters except centre-dot. fg-13, fg-20, fgb-13, fr-25: all except integral, gamma, delta, plus-minus, and circle-plus. These fonts are not supplied in recent releases of the X-window system, but they are included in the Nuprl distribution. To use these fonts they must have been installed correctly and the X-server must be made aware of them — see the file `doc/installation` for details on how to do this.

`: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.

1.6 Mouse Buttons and Special Keys

Some of the keys on the keyboard are bound to special functions. Many of these keys will be referred to symbolically. This section describes the correspondence between the symbolic names and actual keys. In order to accommodate a variety of preferences, many of the names correspond to several keys.

Following are some of the key bindings, together with a brief description (see below for details).

COPY = c-C. Copy text from buffer.

INS = c-I. Instantiate a definition.

KILL = c-K. Kill a character, definition or selected region.

EXIT = c-D. Exit a window or ML or Eval mode.

DELETE = Rubout. Delete the preceding character.

ERASE = c-U. Erase the current input line (command window only).

MODE = c-M. Toggle "bracket mode".

TRANSFORM = c-T. Run a transformation tactic.

OUTPUT = c-O. Output the contents of the active window to a file.

COMMAND = Tab. Make the command window active.

RETURN = Return. Carriage return.

The rest of the special keys are organized as a keypad with the following layout

DIAG	↑	JUMP
←	LONG	→
SEL	↓	

These keys are used to move Nuprl's cursor around and to select text. Almost everything that can be done with these keys can also be done using the mouse. On machines with "meta" keys, symbolic keys are mapped to two simulated keypads: when the "meta" key is held down, the leftmost block of 9 keys (q-w-e-a-s-d-z-x-c) and the rightmost block (u-i-o-j-k-l-m-,-.) are both interpreted as keypads.

As an alternative to the keypad there is a set of Emacs-like bindings that do not involve the "meta" key. These are given below, along with a short description of their main function.

SEL = c-S. Select (a template, an endpoint of a piece of text, ...).

$\downarrow$ = c-N. Move the cursor down ("next").

$\leftarrow$ = c-B. Move the cursor left ("back").

LONG = c-L. Amplify the next cursor movement.

$\rightarrow$ = c-F. Move the cursor right ("forward").

DIAG = c-Q. Move "diagonally" up a proof tree.

$\uparrow$ = c-P. Move the cursor up ("previous").

JUMP = c-J. "Jump" to another window.

By holding down both the "control" and "meta" keys one can obtain some special printing characters (such as some of the Greek letters), if they are available in the current font. These characters should only be used on the left hand sides of the templates in Nuprl definitions. To see a list of these key bindings, right-click the mouse, and a list of them will appear in the command window. If your machine does not have a meta key then you cannot use special characters.

Additional key bindings can be defined in your init file by using the function define-key-binding. This requires a bit of knowledge about how Nuprl deals with characters internally. See the file doc/implementation-notes for more details on this.

Slight variants of the functions of the keys SEL and JUMP are bound to the left and middle mouse buttons, respectively. Henceforth, these buttons will be referred to as mouse-SEL and mouse-JUMP. See below for more on the use of these buttons.

Chapter 2

The Command Language

With the *command language* one can create, delete and manipulate library objects, evaluate terms, call up the editors, and "check" (i.e., parse and generate code for) objects. When a "`P>`" prompt appears in the command/status window, the command processor is waiting for a command.[1] Commands may be abbreviated to their shortest unique prefixes; for example, `check` can be abbreviated to `ch`, and `create` can be abbreviated to `cr`. Names of library objects given as arguments to commands may also be abbreviated: the abbreviation refers to the object with exactly the abbreviated name if such an object exists, or else to the first library entry whose name extends the abbreviation. Typing errors are corrected with the DELETE key (to delete a character) and the ERASE key to delete the whole line.

The commands are divided into four groups: a group consisting of commands that control the library window, a group consisting of commands that manipulate objects, a group consisting of commands that allow work to be saved between sessions, and a miscellaneous group.

The following placeholders are used in the descriptions below of command syntax.

object is the name of an object in the library or one of the reserved names `first` and `last`

objects is *object* or *object*-*object*. This provides a convenient way to refer to a range of entries.

place is `top`, `bot`, `before` *object* or `after` *object*. (`Top` refers to the slot immediately before the first entry, and `bottom` refers to the slot immediately after the last entry.)

kind is `thm`, `def`, `eval` or `ml`.

filename is any legal file name.

Optional arguments and keywords are indicated by curly braces.

[1] If the cursor is within an edit window COMMAND must be pressed to get the "`P>`" prompt.

2.1 Library Display

The library is a list of the objects defined by the user and may contain the theorems, definitions, evaluation bindings and ML code. Information about the library objects is displayed in the *library window*. The `jump` and `scroll` commands change what is being displayed in the library window, while the `move` command changes the order of library objects.

`jump` *object*
: The library is redisplayed so that the specified entry is visible.

`move` *objects place*
: The specified library objects are moved to the position just after the specified place.

`scroll` {`up`} {*number*}
: The library window is moved the specified number of entries, in the specified direction. The default direction is down, and the default number of entries is one. For example, `scroll 5` shifts the window down five entries, while `scroll up` shifts the window up one entry.

`up` {*number*}
: Scroll the library up *number* pages (default 1).

`down` {*number*}
: Scroll the library down *number* pages (default 1).

`top`
: Scroll the library to the top. This is the same as `jump first` (when the library is non-empty).

`bottom`
: Scroll the library to the bottom. This is similar to `jump last` (when the library is non-empty).

2.2 Manipulating Objects

These are the commands that directly manipulate objects. The most used of these commands is `view`, since most interaction with the system takes place within the editors started up by the `view` command.

`check` *objects*
: Each of the indicated objects is checked. If necessary the library window is redisplayed to show the new statuses.

Checking a definition object causes the text in the object to be parsed. If it is well-formed, then the definition becomes available for use in other objects.

Checking an ML object simply calls ML on the text contained in the object.

Checking an eval object causes the contained text to be processed as if it were entered in eval mode (see below).

Checking a large theorem can take a while. If the body of the theorem was dumped to a file and has not been fully reloaded (see the `dump` and `load` commands below) then checking the theorem reconstructs the body of the proof. If it has already been reconstructed, then the proof is not changed by checking.

Checking a theorem causes extraction to be done on the proof (unless the proof has not been touched since the last time extraction was performed).

`create` *name kind place*

A new library entry of the specified name and kind is created at the specified place. The library is redisplayed so that the newly created object is the first object in the library window.

`delete` *objects*

The specified objects are removed from the library.

`view` *object*

The object 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 refinement editor (*red*) is used on theorems, while the text editor (*ted*) is used for other objects. For more information on ted and red see sections 7.4 and 7.5, respectively. Typing EXIT in an edit window will make Nuprl delete the window and end its editing of the object. If there are other edit views on the screen the cursor will be placed in one of them; otherwise, the cursor will go back to the command window.

When `view` is done on a large theorem, it may take some time to activate red. If the body of the theorem has been dumped but not fully reloaded (see the `dump` and `load` commands below) then Nuprl reads the body in and reconstructs it before starting the editor.

2.3 Storing Results

The `load` and `dump` commands let one save results between Nuprl sessions. When a library is dumped to a file, proofs are given a more compact representation. The usual representation of a proof is a tree of sequents and rules. The dumped representation is a "rule-body

tree", which is obtained, roughly, by removing all the sequents from the usual representation, and by converting all the rules to their printed representations. Thus the dumped representation contains essentially just the text that a user would type to reconstruct the proof. One of the savings that this provides has to do with tactics. A field of the tactic rule contains the (hidden) subproof that was built by the tactic; this subproof is discarded in a dump, and only the text of the tactic is retained as a record of the inference.

When libraries are loaded, the loaded proofs retain their compact representation. 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. This has several drawbacks. First, it can be time consuming: all the tactics used to construct the proof must be re-executed. Secondly, if the tactics that were used to construct the proofs have since been modified, the reconstruction may fail. Proofs are rather rigid objects, and seemingly insignificant changes to tactics can cause quite a bit of damage to previously constructed proofs.

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

`dump` *objects* {`to` *filename*}

Nuprl writes a representation of the selected objects from the current library to the file. If no filename is given the name of the file most recently loaded is used. The dumped objects are not removed from the library.

`load` {`fully`} *place* `from` *filename*

This is the command used to read in a file produced by `dump`. The decoded library entries are added one-by-one to the library starting at location *place*. If an object in the dumped file has the same name as an object already in the library, the object in the file is ignored. When the load is finished a message is displayed giving the number of objects loaded and the number of duplicates discarded. The name of the loaded file is displayed in the header of the library window.

Theorem objects are not fully reconstructed unless `fully` is specified. The top of the proof (its goal and status) and the term extracted from the proof are loaded, but the actual body of the proof (the rule applications and subgoals) is not. This body is loaded on demand when the theorem is checked or viewed or when the extractor needs the body of the proof. When the body is actually loaded, the theorem's status is recalculated. If the new status is different from the status the theorem had when it was dumped, a warning is given.

`save` *name*

This allows temporary storing of Nuprl sessions in the Lisp environment. The command will save the current Nuprl state, attaching the name *name*. By "state" is meant all aspects of the state of the Nuprl session except the current window configuration; this includes the library, the values and types associated with ML identifiers,

and the eval environment. There is a special state name "initial" which is the state of Nuprl right after it is first loaded — in particular it includes the default collection of tactics that is loaded with the system (unless the loading of the collection was overridden). This initial state is restored automatically after the *save* command is invoked. Note that since states are stored in the Lisp environment, they disappear when Lisp is exited.

`restore` *name*
: Restore a copy of the state named *name*, replacing the current state.

`copystate` *name new-name*
: Create a copy of state named *name*, giving it the name *new-state*.

`states`
: List the names of states that have been created (and not killed).

`kill` *name*
: Make the state named *name* go away.

2.4 Miscellaneous Commands

This section contains those commands which do not fit in any specific category.

`eval` Puts the command window into *eval* mode. The prompt changes from "`P>`" to "`EVAL>`" to indicate the change. Two kinds of inputs can be entered in eval mode: terms and bindings. Each input must be terminated by a double semicolon, "`;;`". If a term is entered Nuprl evaluates it and prints its value. Bindings have the form `let` *id* = *term*. If a binding is entered Nuprl evaluates the term, prints its value and binds the value to the identifier. The evaluator's binding environment persists from one eval session to the next. It also includes bindings made when EVAL objects are checked. Eval mode is terminated by EXIT.

`ml` Puts the command window into *ML* mode. It operates as eval mode, except that the input (terminated by "`;;`") is evaluated by ML and the result is printed in the command window.

`exit` Terminates the current Nuprl session, returning control to Lisp.

There is one final command, the OUTPUT command. This "snapshot" window operation appends a printout of the visible contents of the current window to the current "snapshot file". The default for the current snapshot file is "snapshot.text" in the current user's home directory; to change the snapshot file, use the ML function "set_snapshot_file".

Chapter 3

The Library

The library is a list of all the objects defined by the user and includes theorems, definitions, evaluation bindings and ML code. The library window displays information about the various objects— their names, types, statuses and summary descriptions. In this section we give a short description of the different kinds of objects and their statuses.

3.1 Object Types

The library contains four kinds of objects: DEF, THM, EVAL and ML. DEF objects define new notations that can be used whenever text is being entered. THM objects contain proofs in the form of proof trees. Each node of the proof tree contains a number of assumptions, a conclusion, a refinement rule and a list of children produced by applying the refinement rule to the assumptions and conclusion. THM objects are checked only when a check command is issued or they are viewed or are used as objects from which code is extracted.

EVAL objects contain lists of bindings, where a binding has the form `let` *id* = *term* and is terminated by a double semicolon, "`;;`". Checking an EVAL object adds its bindings to the evaluator environment so that they available to the `eval` command. All EVAL objects are checked when they are loaded into the library. ML objects contain ML programs, including tactics, which provide a general way of combining the primitive refinement rules to form more powerful refinement rules. Checking an ML object enriches the ML environment. All ML objects are checked when they are loaded into the library.

3.2 Library Entries

Every object has associated with it a status, either *raw*, *bad*, *incomplete* or *complete*, indicating the current state of the object. A *raw* status means an object has been changed but not yet checked. A *bad* status means an object has been checked and found to contain errors. An *incomplete* status is meaningful only for proofs and signifies that the proof

contains no errors but has not been finished. A *complete* status indicates that the object is correct and complete.

An entry for an object in the library is displayed in the library window as a single line contains its kind, status, type, name and a summary of the contents. The kind is `D` for *definition*, `t` and `T` for *theorem*, `M` for *ML*, `E` for *evaluation*. The lower case "`t`" is used for theorems that have proofs in the compact representation used for storage in files; viewing or checking such an object will cause expansion of the proof, and this can be time consuming. The status is encoded as a single character: `?` for raw, `-` for bad, `#` for incomplete and `*` for complete. A typical entry might be the following.

```
Library
*T simple   >> int in U1
```

The library window provides the mechanism for viewing these entries. The entries are kept in a linear order, and at any one time a section of the entries is visible in the library window. The library placement commands described above can be used to change the order of the entries and to move the window around within the list of entries.

If mouse-SEL is pressed when pointing at a library entry, the name of the entry is "typed in" by the system. This works in the command window and in ted (the text editor). So, for example, you can instantiate a definition by INS, mouse-SEL on library object, RETURN.

If mouse-JUMP is pressed with the mouse above the first line in the library window, then the library scrolls up one page. If it is pressed with the mouse just after the last line of the library window (but still in the window or on the bottom border) then the library scrolls down one page. Thus, you can scroll through the library using the mouse, instead of the scroll commands.

If mouse-JUMP is pressed with the mouse in the library scroll bar, then the library is scrolled to the specified point. After the scroll, the scroll bar will indicate that the library is being shown from the point that the mouse selected.

If mouse-JUMP is pressed with the mouse on a library window line that displays a theorem object, then the name, goal, and extracted term (if available) will be displayed in the command window. This allows you to see a theorem more fully, especially if it is too long to fit on one line of the library window.

3.3 Object Dependencies and Ordering

A correct library in Nuprl is one where every definition and theorem refer only to objects occurring previously in the library. Unfortunately, Nuprl does not guarantee that this property is maintained when commands 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 library (or sequence of libraries) is correct. This is to load it (them, sequentially) into an empty library using the "load fully" command. 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 in the library into an empty library without using the "fully" option, and then executing "check first-last", 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.

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` command is often used to make sure that objects occur before their uses.

Chapter 4

The Proof Editor

Nuprl proofs are sequent proofs organized in a refinement style. In other words they are trees where each node contains a *goal* and a *rule application*. The children of a node are the *subgoals* of their parent; in order to prove the parent it is sufficient to prove the children. (Nuprl automatically generates the children, given their parent and the rule to be applied; it is never necessary to type them in.) Not all goals have subgoals; for example, a goal proved by `hyp` does not. Also, an *incomplete* goal (one whose rule is bad or missing) has no subgoals.

In what follows the *main goal* (*the* goal for short) is the goal of the current node; *a* goal is either the main goal or one of its subgoals.

4.1 Basics of the Proof Editor

The proof editor, *red* (for *refinement editor*), is invoked when the `view` command is given on a theorem object. The proof editor window always displays the *current node* of the proof. When the editor is first entered the current node is the root of the tree; in general, the current node can be changed by going into one of its subgoals or by going to its parent. For convenience there are also ways to jump to the next unproved leaf and to the root of the theorem.

Figure 4.1 shows a sample proof editor window. The numbers below refer to the lines of text in the window, starting with the line `EDIT THM t`.

1. The header line names the theorem being edited, `t` in this case.

2. The *map*, `top 1 2`, gives the path from the root to the current node. (If the path is too long to fit on one line it is truncated on the left.) The displayed node is the second child of the first child of the root.

3. The goal's first (and only) hypothesis, `a:int`, appears under the map.

4. The conclusion of the goal is the formula to be proved, given the hypotheses. The conclusion is shown to the right of the the sequent arrow, `>>`. In the example it is

```
EDIT THM t
* top 1 2
1. a:int
>> b:int # ((a=b in int) | ((a=b in int) -> void)) in U1

BY intro

1* >> int in U1

2* 2. b:int
   >> ((a=b in int) | ((a=b in int) -> void))  in U1
```

Figure 4.1: A Sample Proof Editor Window

```
b:int # ((a=b in int)|((a=b in int) -> void)) in U1.
```

5. The *rule* specifies how to refine the goal. In the example the `intro` rule is used to specify *product introduction*, since the main connective of the conclusion is the product operator, `#`.

6. The first subgoal has one hypothesis, `a:int`, which is not displayed since it already occurs as a hypothesis of the main goal, and the conclusion `int in U1`.

7. The second subgoal has two hypotheses, `a:int` and `b:int`, the first of which is not displayed, and the conclusion

```
((a=b in int) | ((a=b in int) -> void)) in U1.
```

Notice that lines (2), (6) and (7) have asterisks, which serve as *status indicators*. The asterisk in line (2) means that this entire subgoal has the status *complete*; similarly, the asterisks in lines (6) and (7) mean that the two subgoals also are *complete*. Other status indicators are `#` (incomplete), `-` (bad), and `?` (raw).

4.2 Motion within a Proof Node

We first make the following definitions. The proof cursor is *in* a goal if it points to a hypothesis or the conclusion; it is *at* a goal if it points to the first part of the goal. If the cursor is not in a goal, it is *at* the rule of the current node or *in* or *at* a subgoal.

The cursor can be moved around within the current proof node in the following ways:

↑, ↓
: Moves the cursor up or down to the next or previous minor stopping point. A minor stopping point is an assumption, a conclusion or the rule.

LONG ↑, LONG ↓
: Moves the cursor up or down to the next or previous major stopping point. A major stopping point is a subgoal, a rule or the goal.

LONG LONG ↑
: Moves the cursor to the goal.

LONG LONG ↓
: Moves the cursor to the conclusion of the last subgoal.

4.3 Motion within the Proof Tree

The following commands allow one to move around within the proof. They change the current node by moving into or out of subproofs. It is helpful, in memorizing these commands, to think of the proof tree as lying on its side, having children of a node layed out vertically with the first child level with its parent.

→
: Only has an effect when the cursor is in a subgoal; the node for that subgoal becomes the current node.

←
: Sets the current node to be the parent of the current node. The cursor will still point to the current goal, but the current goal will be displayed as one of the subgoals of the parent.

DIAG
: If the cursor is not in the goal DIAG moves it to point to the goal. If it *is* in the goal, the parent of the current node will become the new current node, (as in ←) and the cursor will point to its goal (unlike ←).

LONG DIAG
: Has the same effect as four DIAGs.

LONG LONG DIAG
: Sets the current node to be the root of the proof. The cursor will point to the environment of its goal.

LONG →
: Sets the current node to be the next unproved node of the proof tree. The search is done in preorder and wraps around the tree if necessary. If the proof has no incomplete nodes the current node is set to be the root.

Any other combination of LONG and the arrow keys is ignored. Extra LONGs are ignored.

The cursor may also be moved within a single window by pointing at the desired location and clicking mouse-JUMP. However, note that scrolling the contents of the window cannot be done just with the mouse. Pointing at a non-active window and clicking mouse-JUMP will make the window active (but leave the cursor where it was when the window was last active).

4.4 Initiating a Refinement

To refine a goal, a refinement rule must be supplied. To do this, first move the cursor to point to the rule of the current node and press SEL; this will bring up an instance of the text editor in a window labeled `EDIT rule of` *thmname*. Type the rule into this window and press EXIT. Nuprl will parse the rule and try to apply it to the goal of the current proof tree node. If it succeeds the proof window will be 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 a `HELP` message in the command/status window and leave the rule window on the screen so that it can be fixed.

If an existing rule is selected for editing Nuprl will copy it to the edit window. If the text of the rule is changed, then when EXIT is pressed Nuprl will again parse and refine the current goal. When it does it will throw away any proofs for the children of the current node, even if they could have been applied to the new children of the current node. On the other hand, if the text is has not been touched, then when EXIT is pressed Nuprl will not reparse the rule.

Warning. If the text has been touched at all, then, even if the text is *identical* to when the window was opened, the text will be reparsed and the entire subproof will be lost.

4.5 Showing Goals, Subgoals and Rules

In some circumstances it is convenient to be able to copy goals or subgoals into other proofs. This cannot be done directly, but there is a roundabout way. If SEL is pressed while the cursor is in a goal Nuprl creates a read-only text window containing the text of the goal. The window will be labeled `SHOW goal of` *thmname*. The copy mechanism of the text editor can then be used to copy portions of the goal to other windows.

4.6 Invoking Transformation Tactics

Transformation tactics are tactics which act as proof transformers rather than as generalized refinement rules. To invoke a transformation tactic, press TRANSFORM. This will

bring up a text edit window labelled `Transformation Tactic`. Type the name of the tactic and any arguments into the window and press EXIT; Nuprl will apply the tactic and redisplay the proof window to show any effects. If the tactic returns an error message it will be displayed in the command/status window.

To apply a transformation tactic, the system proceed as with the tactic rule (see Section 7.19), with the following two differences. First, the argument given to the tactic, instead of being a degenerate proof, is the entire proof tree p rooted at the node to which the tactic is applied. Second, the editor replaces p with the proof resulting from the application of the tactic.

4.7 The Autotactic

Nuprl provides the facility for a distinguished transformation tactic known as the *autotactic*. This tactic is invoked automatically on the results of each primitive refinement step. After each primitive refinement performed in the refinement editor the autotactic is run on the subgoals as a transformation tactic. There is a default autotactic, but any refinement or transformation tactic may be used. For example, to set the autotactic to a transformation tactic called `transform_tac`, the following ML code is used:

```
set_auto_tactic ‘transform_tac‘;;
```

This code could be used either in ML mode or placed in an ML object (and checked). To see what the current autotactic is, type

```
show_auto_tactic ();;
```

in ML mode.

The autotactic is applied to subgoals of primitive refinements only when a proof is being edited. When performing a library load or when using a tactic while editing a proof, the autotactic is inhibited.

Chapter 5

Definitions and Definition Objects

In Nuprl text is used to represent proof rules, theorem goals and and ML tactic bodies. This text is made up of characters and calls to text macros. Since internally pieces of text are stored as trees of characters and macro references, they are also called *text trees* or *T-trees*.

Text macros are used mainly to improve the readability of terms and formulas. For example, since there is no $\geq$ relation built into Nuprl, a formula like $x \geq y$ might be represented by `((`x`<`y`) -> void)`. Instead of always doing this translation one may define

```
<x>>=<y>==((<x><<y>) -> void)
```

and then use `x>=y` instead of `((x<y) -> void)`. (Note that for the definition to be used it must be *instantiated* (see section 4 above) — one cannot just type "`x>=y`".) Since most macros are used to extend the notation available for terms and formulas, they are also known as *definitions* and the library objects that contain them are called *definition objects*.

The left-hand side of a definition specifies its formal parameters and also shows how to display a call to it. The right-hand side shows what the definition stands for. In the example above the two formal parameters are `<x>` and `<y>`. (Formal parameters are always specified between angle brackets.)

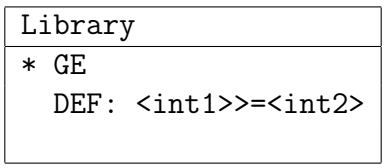
The left-hand side of definitions should contain zero or more formal parameters plus whatever other characters one wants displayed. For instance,

```
<x> greater than or equal to <y>
(ge <x> <y>)
<x>>=<y>
```

are all perfectly valid left-hand sides. A formal parameter is an *identifier* enclosed in angle brackets or an *identifier*:*description* enclosed in angle brackets, where a *description* is a piece of text that does not include an unescaped `>`. (The escape mechanism is explained below.) After the definition object has been checked the description is used to help display the object in the library window, as <*description*> is inserted in place of the formal parameter. For example, given the following definition of `GE`,

```
<x:int1>>=<y:int2>==((<x><<y>) -> void)
```

the library window would show the following:



If the description is omitted or void, the empty string is used, so that formal parameter is displayed as "<>".

Sometimes one wishes to use < and > as something other than angle brackets on the left-hand side of definitions. Backslash is used to suppress the normal meanings of < and >. To keep a < from being part of a formal parameter or to make a > part of a description, precede it with a backslash, as in

```
<x>\<=<y>==((<y><<x>) -> void)
```

and

```
square root(<x: any int\>0 >)== ... .
```

To get a backslash on the left-hand side, escape it with another backslash:

```
<x>\\<y>== ... .
```

The right-hand side of a definition shows what the definition stands for. Besides arbitrary characters it can contain formal references to the parameters (identifiers enclosed in angle brackets) and uses of checked definitions. (Note: definitions may not refer to each other recursively.) Every formal reference must be to a parameter defined on the left-hand side; no nonlocal references are allowed.

Here is an example of a definition that uses another definition:

```
<x>>=<y>>=<z>==<x>>=<y> & <y>>=<z>.
```

An example of a use of this definition is `3>=2>=1`, which expands ultimately into `((3<2) -> void) & ((2<1) -> void)`. Blanks are *very* significant on the right-hand sides of definitions; the three characters `<x>` are considered to be a formal reference, but the four characters `<x >` are not.

Extra backslashes on the right-hand side do not affect parsing, although they do affect the readability of the generated text. For example, the (unnecessary) backslash in

```
<x>>=<y>==((<x>\<<y>) -> void)
```

is displayed whenever the generated text is displayed. For better readability it is best to avoid backslashes on the right-hand side whenever possible.

The routine that parses the right-hand side of definitions does backtracking to reduce the number of backslashes needed on right-hand sides. The only time a backslash is absolutely required is when there is a < followed by what looks like an identifier followed by a >, all with no intervening spaces. For example, the right-hand side below requires its backslash, since <x> looks like a parameter reference:

```
... == <f1>, 0\<x>> <f2>.
```

If the definition were rewritten as

```
... == <f1>, 0<x >> <f2>
```

then it would no longer need a backslash.

Notice that the right-hand side does not have to refer to all of the formal parameters and can even be empty, as in the "comment" definition.

```
(* <string:comment> *)== .
```

A use of such a definition might be the following.

```
EDIT main goal of thm
>> x:int # ( ... )      (* This text is ignored *)
```

Chapter 6

The Text Editor

Nuprl text consists of sequences of characters and *definition references*. Since these sequences are stored internally as recursive trees, they are also known as *text trees*, or *T-trees*. The text editor, *ted*, is a structure editor; the cursor on the screen represents a cursor into the text tree being edited, and movements of the tree cursor are mapped into movements of the screen's cursor.

The text editor is used to create and modify DEF, ML and EVAL objects as well as rules and main goals of theorems. Changes to objects are reflected immediately.

The text editor is almost completely modeless. To insert a character, type it; to insert a definition, *instantiate* it; to remove a character, delete it or KILL it; to move the cursor, use the arrow keys; to exit the editor, use EXIT. There is a *bracket* mode, explained in the next section, which changes the way definition references are displayed.

6.1 Tree Cursors, Tree Positions and Bracket Mode

A *tree cursor* is a path from the root of a text tree to a *tree position*, namely a character, a definition reference or the end of a subtree. When text is being edited the tree cursor is mapped to a location on the screen, and this is where the editor places the screen cursor. Unfortunately, more than one tree position can map to a screen position. Consider the following definition of the $\geq$ relation.

```
<x>>=<y>==((<x><<y>) -> void).
```

In the tree represented by `3>=1` there are six tree positions: the entire definition, the character `3`, the end of the subtree containing `3`, the character `1`, the end of the subtree containing `1` and the end of the entire tree. These six tree positions are mapped into four screen positions:

```
3>=1
^^ ^^
```

The first is for both the entire definition and the character 3, the second is for the end of the subtree containing 3, the third is for the character 1, and the fourth is for the end of the subtree containing 1 and also for the end of the entire tree. If the screen cursor were under the 3, pressing the KILL key would either delete the entire definition or just the 3, depending on where the *tree* cursor actually was.

To eliminate this ambiguity the window may be put into *bracket mode*; in this mode all definition references are surrounded by square brackets, and the screen position that corresponds to the entire definition reference is the left square bracket:

```
[3>=1]
^^^ ^^^
```

Notice that the right square bracket corresponds to the end of the subtree containing 1; this gets rid of the other ambiguous screen position.

For another example of an ambiguity removed by bracket mode, consider the two trees

```
[23>=1]   and   2[3>=1].
```

The first is quite reasonable, but the second is probably a mistake, since it expands to `2((3<1) -> void)`. In unbracketed mode both trees would be displayed as `23>=1`.

The bracket mode toggle, MODE, switches the current window into or out of bracket mode.

6.2 Moving the Cursor

The cursor can be cycled through the edit windows with the JUMP key. This is most useful when copying text from one window to another. To move the cursor around within the current window, use the arrow and LONG keys, as follows.

↑, ↓
: Move the screen cursor up or down one line. The tree cursor is moved to the tree position that most closely corresponds to the screen position one line up or down. If the new tree position is not already being shown in the text window, scroll the window.

←, →
: Move the tree cursor left or right one position. That is, move one position backward or forward in a preorder walk of the tree. Keep in mind that, as explained above, this one-position change of the tree cursor does not always correspond to a one-column change of the screen cursor.

LONG ↑, LONG ↓
: Move up or down four lines in the screen. If necessary, scroll the window.

LONG ←, LONG →
: Move the tree cursor left or right four positions.

LONG LONG ↑, LONG LONG ↓
: Move up or down a screen.

LONG LONG ←, LONG LONG →
: Move the screen cursor to the left or right end of the current line. The tree cursor is moved left or right to match.

If LONG is used before any other key besides an arrow key it is ignored. (The LONG DIAG command works only in the proof editor.) Also, any extra LONGs are ignored.

The cursor may also be moved within a single window by pointing at the desired location and clicking mouse-JUMP. However, note that scrolling the contents of the window cannot be done just with the mouse. Pointing at a non-active window and clicking mouse-JUMP will make the window active (but leave the cursor where it was when the window was last active).

6.3 Adding Text

To add a character, simply type it; it will be added to the left of the current position. This also includes the newline character; to start a new line, enter carriage return.

To *instantiate* a definition, use INS. Nuprl will prompt for the name of the definition in the command window with `I>`; type the name of the definition and press RETURN. (The name can be abbreviated in the same way as in command arguments—see above.) Nuprl will insert an instance of the definition to the left of the current position and move the tree cursor to the first parameter.

6.4 Deleting and Killing Text

There are two ways to remove a character. The KILL key removes the one at the current position, while the DELETE key removes the one to the left of the current position, assuming it is in the same subtree as the current position. (If it is not DELETE does nothing.) To remove a definition reference, use KILL. The cursor must point to the beginning of the reference. A *selection* of text may also be killed. This involves three steps.

1. With the SEL key choose the left or right endpoint of the selection. Alternatively, point at the endpoint with the mouse and click mouse-SEL (this will not move the cursor).

2. Select the other endpoint. It must be in the same window as the first. Also, if the first endpoint was within an argument to a definition, the second must be within the same argument.

3. Press KILL; Nuprl will delete the two endpoints and everything between.

The killed text is moved to the *kill buffer* and can be used by the COPY command (see below). Only killed text goes into the kill buffer; text deleted by DELETE does not.

There can be at most two selected endpoints at any time. If another endpoint is selected the oldest one is forgotten. When a KILL or COPY is done *both* endpoints are forgotten. The two selected endpoints must be in the same window; if they are not KILL and COPY will give error messages.

6.5 Copying and Moving Text

To copy text from one location to another, select it, move the cursor to the target location, and press the COPY key. If two endpoints were specified the selected text, including the endpoints, is copied to the left of the current position. If only one endpoint was specified the selected character or definition reference is copied to the left of the current position. In either case all endpoints are forgotten.

To move text, use KILL and COPY together; if COPY is pressed when no selections have been made the kill buffer is copied to the left of the current position. Therefore, to move some text, select it, KILL it, move the cursor to the target position, and press COPY. The kill buffer is not affected by a COPY, so the easiest way to replicate a piece of text in many places is to type it, KILL it and then use COPY many times.

6.6 Miscellaneous

When typing at the command window, a mouse-SEL while it is over a definition instance in another ted (text editing) window will cause the name of the definition to be inserted into the command window's input line. So if you are reading and do not know what a certain notation means, you can enter COMMAND to get a `P>` prompt, type "view", and then point at the notation in question. The name of the definition will be typed in for you and a return will bring it up in a view window.

6.7 Terms and Evaluation

This documents the syntax and evaluation of the terms of Nuprl's type theory. In giving the procedure for evaluation and specifying the refinement rules of the theory, we will use the following notation for substitution. If $0 \leq n$, $x_1, \ldots, x_n$ are variables and $t_1, \ldots, t_n$ are terms then

$$t[t_1, \ldots, t_n / x_1, \ldots, x_n]$$

is the result of simultaneously substituting t_i for x_i.

Figure 6.7 shows the terms of Nuprl. Variables are terms, although since they are not closed they are not executable. Variables are written as identifiers, with distinct identifiers indicating distinct variables.[1] Nonnegative integers are written in standard decimal notation. There is no way to write a negative integer in Nuprl; the best one can do is to write a noncanonical term, such as `-5`, which evaluates to a negative integer. Atom constants are written as character strings enclosed in double quotes, with distinct strings indicating distinct atom constants.

The free occurrences of a variable x in a term t are the occurrences of x which either *are* t or are free in the immediate subterms of t, excepting those occurrences of x which become *bound* in t. In figure 6.7 the variables written below the terms indicate which variable occurrences become bound; some examples are explained below.

- In x`:`A`#`B the x in front of the colon becomes bound and any free occurrences of x in B become bound. The free occurrences of variables in x`:`A`#`B are all the free occurrences of variables in A and all the free occurrences of variables in B except for x.

- In `<`a`,`b`>` no variable occurrences become bound; hence, the free occurrences of variables in `<`a`,`b`>` are those of a and those of b.

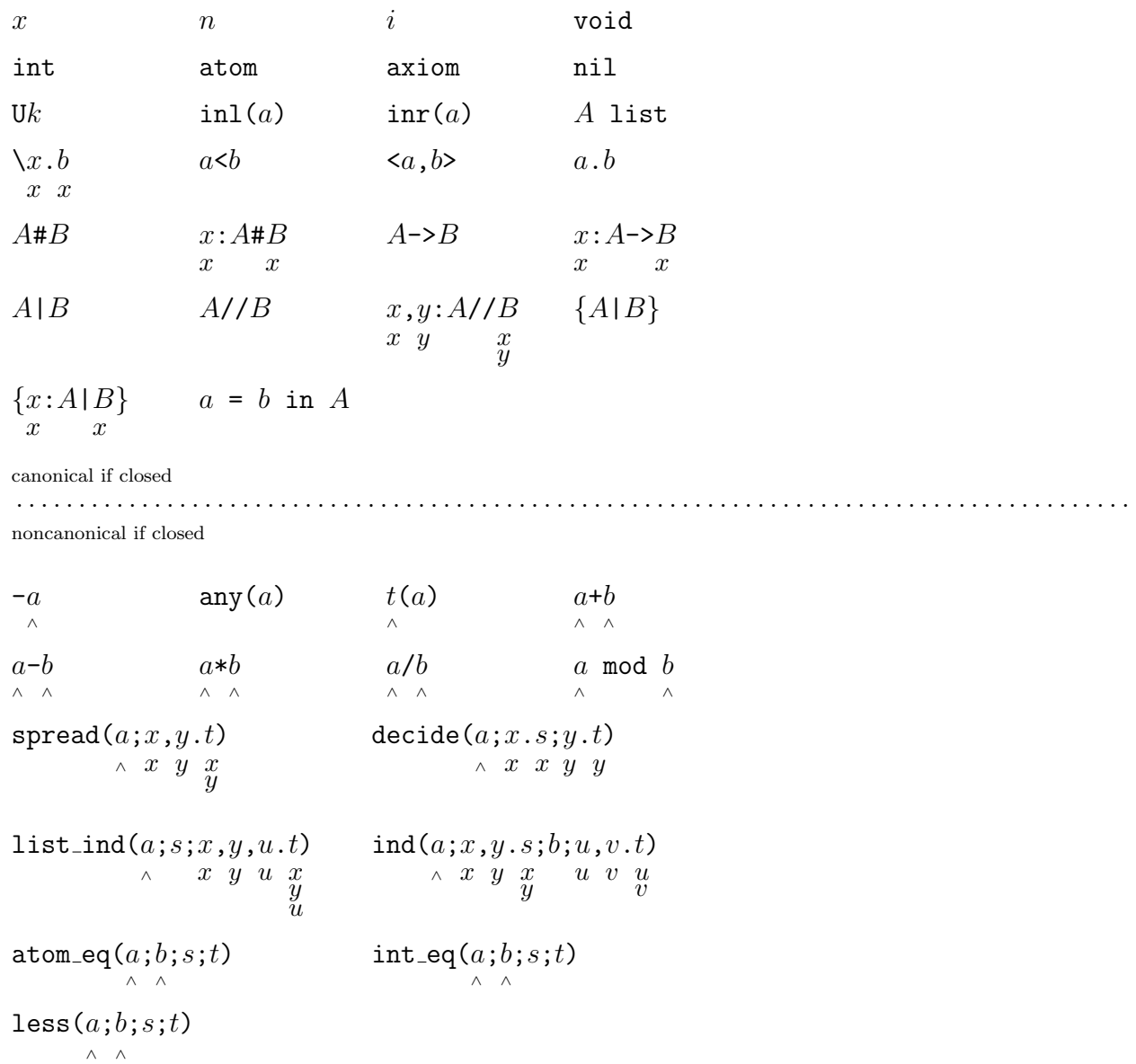
- In `spread(`s`;`x`,`y`.`t`)` the x and y in front of the dot and any free occurrences of x or y in t become bound.

Parentheses may be used freely around terms and often must be used to resolve ambiguous notations correctly. Figure 6.2 gives the relative precedences and associativities of Nuprl operators.

The closed terms above the dotted line in figure 6.7 are the canonical terms, while the closed terms below it are the noncanonical terms. Note that carets appear below most of the noncanonical forms; these indicate the *principal argument places* of those terms. This notion is used in the evaluation procedure below. Certain terms are designated as *redices*, and each redex has a unique *contractum*. Figure 6.3 shows all redices and their contracta.

The evaluation procedure is as follows. Given a (closed) term t,

[1] An identifier is any string of letters, digits, underscores or at–signs that starts with a letter. The only identifiers which cannot be used for variables are `term_of` and those which serve as operator names, such as `int` or `spread`.



x, y, u, v	range over variables.
a, b, s, t, A, B	range over terms.
n	ranges over integers.
k	ranges over positive integers.
i	ranges over atom constants.

Variables written below a term indicate where the variables become bound.
"^" indicates principal arguments.

Figure 6.1: Terms

Lower Precedence

`=`,`in`	left associative
`#`,`->`,`|`,`//`	right associative
`<` (as in a`<`b)	left associative
`+`,`-` (infix)	left associative
`*`,`/`,`mod`	left associative
`inl`,`inr`,`-` (prefix)	—
`.` (as in a`.`b)	right associative
`\x.`	—
`list`	—
`(`a`)` (as in t`(`a`)`)	—

Higher Precedence

Figure 6.2: Operator Precedence

If t is canonical then the procedure terminates with result t.

Otherwise, execute the evaluation procedure on each principal argument of t, and if each has a value, replace the principal arguments of t by their respective values; call this term s.

If s is a redex then the procedure for evaluating t is continued by evaluating the contractum of s.

If s is not a redex then the procedure is terminated without result; t has no value.

Redex	**Contractum**
`(\`$x.b$`)(`a`)`	$b[a/x]$
`spread(<`a`,`b`>;`x`,`$y.t$`)`	$t[a,b/x,y]$
`decide(inl (`a`) ;`$x.s$`;`$y.t$`)`	$s[a/x]$
`decide(inr (`b`) ;`$x.s$`;`$y.t$`)`	$t[b/y]$
`list_ind(nil;`s`;`x`,`y`,`$u.t$`)`	s
`list_ind(`$a.b$`;`s`;`x`,`y`,`$u.t$`)`	$t[a,b,$`list_ind(`b`;`s`;`x`,`y`,`$u.t$`)`$/x,y,u]$
`atom_eq(`i`;`j`;`s`;`t`)`	s if i is j; t otherwise
`int_eq(`m`;`n`;`s`;`t`)`	s if m is n; t otherwise
`less(`m`;`n`;`s`;`t`)`	s if m is less than n; t otherwise
`-`n	the negation of n
m`+`n	the sum of m and n
m`-`n	the difference
m`*`n	the product
m`/`n	`0` if n is `0`; otherwise, the floor of the obvious rational.
m `mod` n	`0` if n is `0`; otherwise, the positive integer nearest `0` that differs from m by a multiple of n.
`ind(`m`;`x`,`$y.s$`;`b`;`u`,`$v.t$`)`	b if m is `0`; $t[m,$`ind(`$m-1$`;`x`,`$y.s$`;`b`;`u`,`$v.t$`)`$/u,v]$ if m is positive; $s[m,$`ind(`$m+1$`;`x`,`$y.s$`;`b`;`u`,`$v.t$`)`$/x,y]$ if m is negative.

a, b, s, t	range over terms.
x, y, u, v	range over variables.
m, n	range over integers.
i, j	range over atom constants.

Figure 6.3: Redices and Contracta

Chapter 7

Refinement Rules

The Nuprl system has been designed to accommodate the top-down construction of proofs by refinement. In this style one proves a judgement (i.e., a *goal*) by applying a *refinement rule*, thereby obtaining a set of judgements called *subgoals*, and then proving each of the subgoals. In this section we will describe the refinement rules themselves. First we give some general comments regarding the rules and then proceed to give a description of each rule.

7.1 The Form of a Rule

To accommodate the top-down style of the Nuprl system the rules of the logic are presented in the following *refinement* style.

$$
\begin{array}{l}
H \texttt{ >> } T \texttt{ ext } t \texttt{ by } \mathit{rule} \\
\quad H_1 \texttt{ >> } T_1 \texttt{ ext } t_1 \\
\qquad \vdots \\
\quad H_k \texttt{ >> } T_k \texttt{ ext } t_k
\end{array}
$$

The goal is shown at the top, and each subgoal is shown indented underneath. The rules are defined so that if every subgoal is true then one can show the truth of the goal (see Section 8.1 of the book for an explanation of truth of judgments and sequents). If there are no subgoals ($k = 0$) then the truth of the goal is axiomatic.

One of the features of the proof editor is that the extraction terms are not displayed and indeed are not immediately available. The idea is that one can judge a term T to be a type and T to be inhabited without explicitly presenting the inhabiting object. When one is viewing T as a proposition this is convenient, as a proposition is true if it is inhabited. If T is being viewed as a specification this allows one to implicitly build a program which is guaranteed to be correct for the specification. The extraction term for a goal is built as a function of the extraction terms of the subgoals and thus in general cannot be built until each of the subgoals have been proved. If one has a specific term, t, in mind as the inhabiting object and wants it displayed, one can use the explicit intro rule and then show

that the type t `in` T is inhabited. The rules have the property that each subgoal can be constructed from the information in the rule and from the goal, exclusive of the extraction term. As a result some of the more complicated rules require certain terms as parameters.

Implicit in showing a judgement to be true is showing that the conclusion of the judgement is in fact a type. We cannot directly judge a term to be a type; rather, we show that it inhabits a universe. An examination of the semantic definition will reveal that this is sufficient for our purposes. Due to the rich type structure of the system it is not possible in general to decide algorithmically if a given term denotes an element of a universe, so this is something which will require proof. The logic has been arranged so the proof that the conclusion of a goal is a type can be conducted simultaneously with the proof that the type is inhabited. In many cases this causes no great overhead, but some rules have subgoals whose only purpose is to establish that the goal is a type, that is, that it is *well-formed*. These subgoals all have the form H `>>` T `in U`i and are referred to as *well-formedness* subgoals.

7.2 Organization of the Rules

The rules for reasoning about each type and objects of the type will be presented in separate sections. Recall from above that for each judgement of the form H `>>` T `ext` t where the inhabiting object t is left implicit, there is a corresponding explicit judgement H `>>` t `in` T `ext axiom`. As the content of these judgements is essentially the same, the rules for reasoning about them will be presented together.

For each type we will have the following categories of rules:

- *Formation*
  These rules give the conditions under which a canonical type may be judged to inhabit a universe, thus verifying that it is indeed a type.

- *Canonical*
  These rules give the conditions under which a canonical object (implicitly or explicitly presented) may be judged to inhabit a canonical type. Note that the formation rules are all actually canonical rules, but it is convenient to separate them.

- *Noncanonical*
  These rules give the conditions under which a noncanonical object may be judged to inhabit a type. The elimination rules all fall in this category, as the extract term for an elimination rule is a noncanonical term.

- *Equality*
  These rules give the conditions under which objects having the same outer form may be judged to be equal. Recall that the rules are being presented in implicit/explicit pairs, H `>>` T `ext` t and H `>>` t `in` T. The explicit judgement H `>>` t `in` T is simply the reflexive instance of the general equality judgement H `>>` t `=` t' `in` T,

and in most cases the rule for the general form is an obvious generalization of the rule for the reflexive form, and thus will be omitted. As the rules for the reflexive judgement are given in one of the other categories, there will be no equality rules presented for some types.

- *Computation*
  These rules allow one to make judgements of equalities resulting from computation.

Rules such as the *sequence*, *hypothesis* and *lemma* rules which are not associated with one particular type are grouped together under the heading "miscellaneous".

7.3 Specifying a Rule

In the context of a particular goal a rule is specified by giving a name and, possibly, certain parameters. As there are a large number of rules it would be unfortunate to have to remember a unique name for each one. Instead, there are small number of generic names, and the proof editor infers the specific rule desired from the form of the goal. In fact, for the rules dealing with specific types or objects of specific types, there are only the names *intro*, *elim* and *reduce*. The *intro* rules are those which break down the conclusion of the goal, and the *elim* rules are those which use a hypothesis. Accordingly, the first parameter of any elim rule is the declared variable or number of the hypothesis to be used. The *reduce* rules are the computation rules. The first parameter of a reduce rule is a number that specifies which term of the equality is to be reduced. Among the parameters of some rules are keyword parameters which have the following form:

- `new` $x_1, \cdots, x_n$
  This parameter is used to give new names for hypotheses in the subgoals. In most cases the defaults, which are derived from subterms of the conclusion of the goal, suffice. For technical reasons the same variable can be declared at most once in a hypothesis list, so if a default name is already declared a new name will have to be given. Whenever this parameter is used it must be the case that the names given are all distinct and do not occur in the hypothesis list of the goal.

- `using` T, `over` $z.T$
  These parameters are used when judging the equality of noncanonical forms in types dependent on the principal argument of the noncanonical form. The `using` parameter specifies the type of the principal argument of the noncanonical form. The value should be a canonical type which is appropriate for the particular noncanonical form. The `over` parameter specifies the dependence of the type over which the equality is being judged on the principal argument of the form. Each occurrence of z in T indicates such a dependency. The proof editor always checks that the term obtained by substituting the principal argument for z in T is α-convertible to the type of the equality judgement.

- `at U`i
  The value of this parameter is the universe level at which any type judgements in the subgoals are to be made. The default is `U1`.

7.4 Optional Parameters and Defaults

Each rule will be presented in its most general form. However, some of the parameters of a rule may be optional, in which case they will be enclosed by square brackets (`[]`). If a new hypothesis in a subgoal depends on an optional parameter, and in a particular instance of the rule the optional parameter is not given, that new hypothesis will not be added. Such a dependence is usually in the form of a hypothesis specifically referring to an optional `new` name. The `over` parameter discussed above is almost always optional. If it is not given, it is assumed that the type of the equality has no dependence on the principal argument of the noncanonical form.

The issue of default values for variable names arises when the main term of a goal's conclusion contains binding variables. In general, the default values are taken to be those binding variables. For example, the rule for explicitly showing a product to be in a universe is

H `>>` x`:`A`#`B `in U`i `by intro [new y]`
`>>` A `in U`i
y`:`A `>>` B `in U`i

The rule is presented as if a new name is given, but the default is to use x. All the dependent types follow this general pattern.

For judging the equality of terms containing binding variables the binding variables of the first term are in general the default values for the "appropriate" new hypotheses. Consider the rule (slightly simplified) for showing that a spread term is in a type:

H `>> spread(`e`;`$x, y.t$`) in` T`[`e`/`z`]`
`by intro [over` $z.T$`] using` w`:`A`#`B `[new` u, v`]`
H `>>` e `in` w`:`A`#`B
H`,`u`:`A`,`v`:`B`[`u`/`w`]` `>>` t`[`u, v`/`x, y`]` `in` T`[<`u`,`v`>/`z`]`

Here the new variables default to x, y. If no new names are given and x and y don't appear in H, then the second subgoal will be

H`,`x`:`A`,`y`:`B `>>` t `in` T`[<`x`,`y`>/`z`]`

Again this is the general pattern for rules of this type.

7.5 Hidden Assumptions

For certain rules, we need to be able to control the free variables occuring in the extract term. The mechanism used to achieve this is that of *hidden* hypotheses. A hypothesis is hidden when it is displayed enclosed in square brackets. At the moment the only place where such hypotheses are added is in a subgoal of the set elim rule. The intended meaning of a hypothesis being hidden is that the name of the hypothesis cannot appear free in the extracted term; that is, that it cannot be used computationally. Accordingly, a hidden hypothesis cannot be the object of an `elim` or `hyp` rule. For the rules for which the extract term is the trivial term `axiom`, the extract term contains no free variable references and so all restrictions on the use of hidden hypotheses can be removed. The editor will remove the brackets from any hidden hypotheses in displaying a goal of this form.

7.6 Shortcuts in the Presentation

Several conventions are used to simplify the presentation of the rules.

- Almost all of the rules have the property that the list of hypotheses in a subgoal is an extension of the hypothesis list of the goal. For such rules, we will show only the new hypotheses in the subgoals.
- Trivial extraction terms (that is, those that are just `axiom`.) will not be exhibited.
- The computation rules take an integer i as their first parameter. The conclusion must be of the form t_1`=`$\ldots$`=`t_k `in` T and i is taken to refer to t_i (so $1 \leq i \leq k$). The rule we present only deals with the case $i = 1$ and $k = 2$, but the rule will apply for any $k \geq 1$ and i, with the obvious changes being made.

7.7 Atom

formation

H `>> U`i `ext atom by intro atom`
H `>> atom in U`i `by intro`

canonical

H `>> atom ext "`$\cdots$`" by intro "`$\cdots$`"`
H `>> "`$\cdots$`" in atom by intro`

where '$\cdots$' is any sequence of prl characters.

H >> atom_eq(a;b;t;t') in T by intro
>> a in atom
>> b in atom
a=b in atom >> t in T
(a=b in atom)->void >> t' in T

computation

H >> atom_eq(a;b;t;t')=t'' in T by reduce 1 true
>> a=b in atom
>> t=t'' in T

H >> atom_eq(a;b;t;t')=t'' in T by reduce 1 false
>> (a=b in atom) -> void
>> t'=t'' in T

7.8 Void

formation

H >> Ui ext void by intro void

H >> void in Ui by intro

noncanonical

H,z:void >> T ext any(z) by elim z

H >> any(e) in T by intro
>> e in void

7.9 Int

formation

H >> Ui ext int by intro int

H >> int in Ui by intro

canonical

H >> int ext c by intro c

H >> c in int by intro

where c must be an integer constant.

noncanonical

H >> -t in int
   >> t in int

H >> int ext m *op* n by intro *op*
   >> int ext m
   >> int ext n

H >> m *op* n in int by intro
   >> m in int
   >> n in int

where *op* must be one of +,-,*,/, or mod.

H, x:int,H' >> T ext ind($x; y, z.t_d; t_b; y, z.t_u$) by elim x new z[,y]
   y:int,y<0,z:T[y+1/x] >> T[y/x] ext t_d
   >> T[0/x] ext t_b
   y:int,0<y,z:T[y-1/x] >> T[y/x] ext t_u

Where the optional new name y must be given if x occurs free in H'.

H >> ind($e; x, y.t_d; t_b; x, y.t_u$) in T[e/z]
      by intro [over $z.T$] [new u,v]
   >> e in int
   u:int,u<0,v:T[u+1/z] >> t_d[$u, v/x, y$] in T[u/z]
   >> t_b in T[0/z]
   u:int,0<u,v:T[u−1/z] >> t_u[$u, v/x, y$] in T[u/z]

H >> int_eq($a; b; t; t'$) in T by intro
   >> a in int
   >> b in int
   a=b in int >> t in T
   (a=b in int)->void >> t' in T

H >> less($a; b; t; t'$) in T by intro
   >> a in int
   >> b in int
   a<b >> t in T
   (a<b)->void >> t' in T

computation

H >> ind($nt; x, y.t_d; t_b; x, y.t_u$) = t in T by reduce 1 down
   >> t_d[nt,(ind(nt+1;x,$y.t_d$;t_b;x,$y.t_u$))/x,y] = t in T
   >> nt<0

H >> ind(zt;x,y.t_d;t_b;x,y.t_u) = t in T by reduce 1 base
>> t_b = t in T
>> zt = 0 in int

H >> ind(nt;x,y.t_d;t_b;x,y.t_u) = t in T by reduce 1 up
>> t_u[nt,(ind(nt-1;x,y.t_d;t_b;x,y.t_u))/x,y] = t in T
>> 0<nt

H >> int_eq(a;b;t;t') = t'' in T by reduce 1 true
>> a=b in int
>> t=t'' in T

H >> int_eq(a;b;t;t') = t'' in T by reduce 1 false
>> (a=b in int) -> void
>> t' = t'' in T

H >> less(a;b;t;t') = t'' in T by reduce 1 true
>> a<b
>> t=t'' in T

H >> less(a;b;t;t') = t'' in T by reduce 1 false
>> a<b -> void
>> t'=t'' in T

7.10 Less

formation

H >> Ui ext a<b by intro less
H >> int ext a
H >> int ext b

H >> a<b in Ui by intro
H >> a in int
H >> b in int

equality

H >> axiom in a<b
H >> a<b

7.11 List

formation

H >> Ui ext A list by intro list
>> Ui ext A

H >> A list in Ui by intro
   >> A in Ui

canonical

H >> A list ext nil by intro nil at Ui
   >> A in Ui

H >> nil in A list by intro at Ui
   >> A in Ui

H >> A list ext $h.t$ by intro .
   >> A ext h
   >> A list ext t

H >> $a.b$ in A list by intro
   >> a in A
   >> b in A list

noncanonical

H,x:A list,H' >> T ext list_ind(x;t_b;$u,v,w.t_u$)
        by elim x new w,u[,v]
   >> T[nil/x] ext t_b
   u:A,v:A list,w:T[v/x] >> T[$u.v$/x] ext t_u

H >> list_ind(e;t_b;$x,y,z.t_u$) in T[e/z]
        by intro [over $z.T$] using A list [new u,v,w]
   >> e in A list
   >> t_b in T[nil/z]
   u:A,v:A list,w:T[v/z]
            >> t_u[u,v,w/x,y,z] in T[$u.v$/z]

computation

H >> list_ind(nil;t_b;$u,v,w.t_u$) = t in T by reduce 1
   >> t_b = t in T

H >> list_ind($a.b$;t_b;$u,v,w.t_u$) = t in T by reduce 1
   >> t_u[a,b,list_ind(b;t_b;$u,v,w.t_u$)/u,v,w] = t in T

7.12 Union

formation

H >> Ui ext A|B by intro union
   >> Ui ext A

```
   >> Ui ext B
H >> A|B in Ui by intro
   >> A in Ui
   >> B in Ui
```

canonical

```
H >> A|B ext inl(a) by intro at Ui left
   >> A ext a
   >> B in Ui
H >> inl(a) in A|B by intro at Ui
   >> a in A
   >> B in Ui
H >> A|B ext inr(b) by intro at Ui right
   >> B ext b
   >> A in Ui
H >> inr(b) in A|B by intro at Ui
   >> b in B
   >> A in Ui
```

noncanonical

```
H,z:A|B,H' >> T ext decide(z;x.t_l;y.t_r) by elim z [new x,y]
   x:A,z=inl(x) in A|B >> T[inl(x)/z] ext t_l
   y:B,z=inr(y) in A|B >> T[inr(y)/z] ext t_r
H >> decide(e;x.t_l;y.t_r) in T[e/z]
         by intro [over z.T] using A|B [new u,v]
   >> e in A|B
   u:A, e=inl(u) in A|B >> t_l[u/x] in T[inl(u)/z]
   v:B, e=inr(v) in A|B >> t_r[v/y] in T[inr(v)/z]
```

computation

```
H >> decide(inl(a);x.t_l;y.t_r) = t in T by reduce 1
   >> t_l[a/x] = t in T
H >> decide(inr(b);x.t_l;y.t_r) = t in T by reduce 1
   >> t_r[b/y] = t in T
```

7.13 Function

formation

H >> Ui ext x:A->B by intro function A new x
>> A in Ui
x:A >> Ui ext B

H >> x:A->B in Ui by intro [new y]
>> A in Ui
y:A >> B[y/x] in Ui

H >> Ui ext A->B by intro function
>> Ui ext A
>> Ui ext B

H >> A->B in Ui by intro
>> A in Ui
>> B in Ui

canonical

H >> x:A->B ext $\backslash y.b$ by intro at Ui [new y]
y:A >> B[y/x] ext b
>> A in Ui

H >> $\backslash x.b$ in y:A->B by intro at Ui [new z]
z:A >> b[z/x] in B[z/y]
>> A in Ui

noncanonical

H,f:(x:A->B),H' >> T ext t[f(a)/y] by elim f on a [new y]
>> a in A
y:B[a/x], y=f(a) in B[a/x] >> T ext t

H,f:(x:A->B),H' >> T ext t[f(a)/y] by elim f [new y]
>> A ext a
y:B >> T ext t

where the first of the two rules above is used when x occurs free in B, and the second when it does not.

H >> f(a) in B[a/x] by intro using x:A->B
>> f in x:A->B
>> a in A

equality

H >> f=g in x:A->B ext t
by extensionality [at Ui] [using u:C->D, v:E->F] [new z]
z:A >> $f(z)$=$g(z)$ in B[z/x] ext t
>> A in Ui
>> f in u:C->D
>> g in v:E->F

where, if the "using" terms are not supplied, then x:A->B is used. This rule also applies to unary equalities.

computation

H >> $(\backslash x.b)(a)$ = t in B by reduce 1
>> b[a/x] = t in B

7.14 Product

formation

H >> Ui ext x:A#B by intro product A new x
>> A in Ui
x:A >> Ui ext B

H >> x:A#B in Ui by intro [new y]
>> A in Ui
y:A >> B[y/x] in Ui

H >> Ui ext A#B by intro product
>> Ui ext A
>> Ui ext B

H >> A#B in Ui by intro
>> A in Ui
>> B in Ui

canonical

H >> x:A#B ext <a,b> by intro at Ui a [new y]
>> a in A
>> B[a/x] ext b
y:A >> B[y/x] in Ui

H >> <a,b> in x:A#B by intro at Ui [new y]
>> a in A
>> b in B[a/x]
y:A >> B[y/x] in Ui

H >> A#B ext <a,b> by intro
>> A ext a
>> B ext b

H >> <a,b> in A#B by intro
>> a in A
>> b in B

noncanonical

H,z:(x:A#B),H' >> T ext spread(z;u,v.t) by elim z new u,v
u:A,v:B[u/x],z=<u,v> in x:A#B >> T[<u,v>/z] ext t

H >> spread(e;x,y.t) in T[e/z]
by intro [over z.T] using w:A#B [new u,v]
>> e in w:A#B
u:A,v:B[u/w],e=<u,v> in w:A#B >> t[u,v/x,y] in T[<u,v>/z]

computation

H >> spread(<a,b>;x,y.t) = s in T by reduce 1
>> t[a,b/x,y] = s in T

7.15 Quotient

formation

H >> Ui ext (x,y):A//E by intro quotient A,E new x,y,z
>> A in Ui
x:A,y:A >> E in Ui
x:A >> E[x,x/x,y]
x:A,y:A,E[x,y/x,y] >> E[y,x/x,y]
x:A,y:A,z:A,E[x,y/x,y], E[y,z/x,y] >> E[x,z/x,y]

H >> (u,v):A//E in Ui by intro new x,y,z
>> A in Ui
x:A,y:A >> E[x,y/u,v] in Ui
x:A >> E[x,x/u,v]
x:A,y:A,E[x,y/u,v] >> E[y,x/u,v]
x:A,y:A,z:A,E[x,y/u,v], E[y,z/u,v] >> E[x,z/u,v]

canonical

H >> (x,y):A//E ext a by intro at Ui
>> (x,y):A//E in Ui
>> A ext a

H >> a in (x,y):A//E by intro at Ui
>> (x,y):A//E in Ui
>> a in A

noncanonical

H,u:(x,y):A//E,H' >> t=t' in T by elim u at Ui [new v,w]
v:A,w:A >> E[v,w/x,y] in Ui
>> T in Ui
v:A,w:A,E[v,w/x,y] >> t[v/u] = t'[w/u] in T[v/u]

equality

H >> (x,y):A//E = (u,v):B//F in Ui by intro [new r,s]
>> (x,y):A//E in Ui
>> (u,v):B//F in Ui
>> A = B in Ui
A=B in Ui,r:A,s:A >> E[r,s/x,y] -> F[r,s/u,v]
A=B in Ui,r:A,s:A >> F[r,s/u,v] -> E[r,s/x,y]

H >> t = t' in (x,y):A//E by intro at Ui
>> (x,y):A//E in Ui
>> t in A
>> t' in A
>> E[t,t'/x,y]

7.16 Set

formation

H >> Ui ext $\{x$:$A \mid B\}$ by intro set A new x
>> A in Ui
x:A >> Ui ext B

H >> $\{x$:$A \mid B\}$ in Ui by intro [new y]
>> A in Ui
y:A >> B[y/x] in Ui

H >> Ui ext $\{A|B\}$ by intro set
   >> Ui ext A
   >> Ui ext B

H >> $\{A|B\}$ in Ui by intro
   >> A in Ui
   >> B in Ui

canonical

H >> $\{x{:}A|B\}$ ext a by intro at Ui a [new y]
   >> a in A
   >> $B[a/x]$ ext b
   $y{:}A$ >> $B[y/x]$ in Ui

where all hidden hypothesis in H become unhidden in the second subgoal.

H >> a in $\{x{:}A|B\}$ by intro at Ui [new y]
   >> a in A
   >> $B[a/x]$
   $y{:}A$ >> $B[y/x]$ in Ui

H >> $\{A|B\}$ ext a by intro
   >> A ext a
   >> B ext b

where all hidden hypotheses in H become unhidden in the second subgoal.

H >> a in $\{A|B\}$ by intro
   >> a in A
   >> B ext b

noncanonical

H,$u{:}\{x{:}A|B\}$,H' >> T ext t by elim u
   H, $u{:}A$, [$B[u/x]$], H' >> T ext t

H,$u{:}\{x{:}A|B\}$,H' >> T ext $(\backslash y.t)(u)$ by elim u [new y]
   H,$u{:}\{x{:}A|B\}$,H', $y{:}A$, [$B[y/x]$] >> $T[y/u]$ ext t

where which one of the two above rules is to applied in any instance is determined by whether or not the variable u of the goal actually appears to the user (as opposed to being generated when extraction is done). If it does appear, the first case applies. Note that the subgoals have a hidden hypothesis.

equality

H >> $\{x{:}A\,|\,B\}$ = $\{y{:}A'\,|\,B'\}$ in Ui by intro [new z]
>> A = A' in Ui
$z{:}A$ >> B[z/x] = B'[z/y] in Ui

7.17 Equality

formation

H >> Ui ext $a_1{=}\cdots{=}a_n$ in A by intro equality A n
>> A in Ui
>> A ext a_1
⋮
>> A ext a_n

where the default for n is 1.

H >> ($a_1{=}\cdots{=}a_n$ in A) in Ui by intro
>> A in Ui
>> a_1 in A
⋮
>> a_n in A

canonical

H >> axiom in (a in A) by intro
>> a in A

H,$x{:}T$,H' >> x in T by intro

This rule does not work when T is a set or quotient term, since intro will invoke the equality rule for the set or quotient type, respectively. In any case, the equality rule can be used.

7.18 Universe

canonical

H >> Ui ext Uj by intro universe Uj

H >> Uj in Ui by intro

where $j < i$. Note that all the formation rules are intro rules for a universe type.

noncanonical

Currently there are no rules in the system for analyzing universes. At some later date such rules may be added.

7.19 Tactic

H `>>` C `ext` t `by` *tactic-text*
H_1 `>>` C_1 `ext` t_1
$\vdots$
H_n `>>` C_1 `ext` t_n

The text *tactic-text* must be an ML expression of type `tactic` (see Section 8 for a definition of this). The subgoals are the result of the following procedure. The tactic denoted by *tactic-text* is applied to the degenerate object of type `proof` with hypotheses H, conclusion C, no refinement and an empty list of children.[1] The evaluation of the application must succeed. The result is validation and a proof list. The validation is applied to the proof list. This evaluation of the application must succeed. The result is a proof tree p. The sequent at the root of p must be the goal H>>C. The subgoals H_i>>C are the sequents, in left-to-right order, at the unproved leaves of p. The extraction term t is the extraction computed for p, where the terms t_i are assumed as extraction terms for the corresponding unproved leaves.

7.20 Miscellaneous

hypothesis

H`,`x`:`A`,`H' `>>` A' `ext` x `by hyp` x

where A' is α-convertible to A

thinning

H `>>` A `ext` t `by thinning` $i_1, \ldots, i_k$
H' `>>` A `ext` t

where $i_1, \ldots, i_k$ are hypothesis numbers and where H' is obtained from H by removing the smallest number of hypotheses that includes the named hypotheses and such that the subgoal sequent is closed (that is, every free variable of

[1] Also, before the tactic is applied, the ML variable `new_id_initialized` is assigned the value `false`. This variable is referenced only by functions that generate new identifiers.

the conclusion or of a hypothesis is declared to the left in the sequent). (The rule fails if no such sequent exists.)

sequence

H >> T ext $(\backslash x_1.\cdots(\backslash x_n.t)(t_n)\cdots)(t_1)$
by seq $T_1,\ldots,T_n$ [new $x_1,\ldots,x_n$]
>> T_1 ext t_1
x_1:T_1 >> T_2 ext t_2
$\vdots$
x_1:T_1,...,x_n:T_n >> T ext t

lemma

H >> T ext t[term_of($theorem$)/x] by lemma $theorem$ [new x]
x:C >> T ext t

where C is the conclusion of the complete theorem $theorem$.

def

H >> T ext t by def $theorem$ [new x]
x:term_of($theorem$) = $ext\text{-}term$ in C >> T ext t

where C is the conclusion of the complete theorem, $theorem$, and $ext\text{-}term$ is the term extracted from that theorem. [2]

explicit intro

H >> T ext t by explicit intro t
>> t in T

cumulativity

H >> T in Ui by cumulativity at Uj
>> T in Uj

where $j < i$

[2]This rule introduces very strong interproof dependencies. A proof using this rule depends not only on C but also on the way C is proved.

substitution

H >> T[t/x] ext s by subst at Ui t=t' in T' over $x.T$
>> t = t' in T'
>> T[t'/x] ext s
x:T' >> T in Ui

equality

H >> t = t' in T by equality

where the equality of t and t' can be deduced from assumptions that are equalities over T (or equalities over T' where $T = T'$ is deducible using only reflexivity, commutativity and transitivity) using only reflexivity, commutativity and transitivity.

direct computation

H >> T ext t by compute using S
>> T' ext t

H, x:T, H' >> T'' ext t by compute hyp i using S
H, x:T', H'>> T'' ext t

where S is obtained from T by "tagging" some of its subterms and T' is generated by the system by performing some computation steps on subterms of T, as indicated by the tags. A subterm t is tagged by replacing it by [[*;t]] or [[n;t]], where n is a positive integer constant. The latter tag causes t to be computed for n steps, and the former causes computation to proceed as far as possible. Note that many of the computation rules, such as the one for product, are instances of direct computation.

H >> T ext t by reverse_direct_computation using S
>> T' ext t

H, x:T, H' >> T'' ext t by reverse_direct_computation_hyp i S
H, x:T', H' >> T'' ext t

where S is a legal tagging of T' and T is the result of performing the computations indicated in S.

eval

>> T ext t by eval b thm_1,...,thm_n
>> T' ext t

H, x:T, H' >> G ext t by eval b thm_1,...,thm_n
H, x:T', H' >> G ext t

These rules are efficient special cases of the direct computation rules. For the purpose of the description below, we define a particular evaluation function, call it f, which takes a list l of names of theorems, a boolean value b, and a term t, and returns a term t' that is obtained from t by performing a particular sequence of computation steps. Specifically, evaluation proceeds as follows. If t is non-canonical, evaluate the principal argument(s); if not all the evaluated principal arguments are canonical and of the proper type, then return t with principal arguments replaced by their evaluated forms; otherwise, contract the redex and evaluate the result. If t is canonical, then continue by evaluating the immediate subterms of t that are not within the scope of a binding variable of t. If t is `term_of(`*thm*`)` then return t if b is true and *thm* is in the list l or if b is false and *thm* is not in l, otherwise continue by evaluating the term that `term_of(`*thm*`)` stands for. If t is a variable, then return t.

In the above rules, b is *true* or *false*, each thm_i is identifier, and T' is the result of applying f to the list $[thm_1,\ldots,thm_n]$, b, and the term T.

arith

H `>>` C `by arith at U`i

The `arith` rule 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.

The terms that `arith` understands are those denoting arithmetic relations, namely terms of the form s`<`t, s`=`t `in int` or the negation of a term of this form. As the only equalities `arith` concerns itself with are those of the form s`=`t `in int`, we will drop the `in int` and write only s`=`t in the rest of this description. For `arith` the negation of an `arith`metic relation $s\theta t$ where θ is one of $<$ or $=$ is of the form `(`$s\theta t$`)->void`, which we will write as $\neg s\theta t$. As integer equality and less-than are decidable relations, $s\theta t$ and $\neg\neg s\theta t$ denote the same relation and will be treated identically by `arith`.

The `arith` rule may be used to justify goals of the form

`H >>` C_1 `|` ... `|` 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 type at the level given in the invocation of the rule.

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 `#` $\ldots$ `#` 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.

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

1. Rewrite each relation of the form $\neg s = t$ as the equivalent $s < t \,|\, t < s$. A conclusion C follows from such an assumption if it follows from either $s < t$ or $t < s$; hence `arith` tries both cases. Henceforth, we assume that all negations of equalities have been eliminated from consideration.
2. Replace all occurrences of terms which are not addition, subtraction or multiplication by a new variable. Multiple occurrences of the same term are replaced by the same variable. `Arith` uses only facts about addition, subtraction and multiplication, so it treats all other terms as atomic. At this point all terms are built from integer constants and integer variables using $+$, $-$ and $*$.
3. Rewrite all terms as polynomials in canonical form. The exact nature of the canonical form is irrelevant (the reader may think of it as the form used in high school algebra texts) but has the important property that any two equal terms are identical. Each term now has the form $p + c\theta p' + c'$, where p and p' are nonconstant polynomials in canonical form, c and c' are constants, and θ is one of $<$, $=$ or $\geq$ ($s \geq t$ is equivalent to $\neg t < s$).
4. Replace each nonconstant polynomial p by a new variable, with each occurrence of p being replaced by the same variable. This amounts to treating each nonconstant polynomial as an atom. Now each formula is of the form $z + c\theta z' + c'$. `Arith` now decides whether or not the conclusion follows from the hypotheses by using the total order axioms of $<$, the reflexivity, symmetry, transitivity and substitutivity of $=$, and the following so-called *trivial monotonicity* axioms (c and d are constants).

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

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

For a detailed account of the `arith` rule and a proof of its correctness, see the article by Tat-hung Chan cited in the book.

monotonicity

H `>>` *T* `ext` *t* `by monotonicity` h_1 *op* h_2
A `>>` *T* `ext` *t*

where h_1 and h_2 are numbers of hypotheses that have the form t_1 *rel* t_2, (the possible values of *rel* are described below), and *op* is one of `+`, `-`, `*`. The new hypothesis *A* is computed from *op* and hypotheses h_1 and h_2.

The monotonicity rule performs one instance of an inference based upon "non-trivial" monotonicity of integer arithmetic over `+`, `-`, and `*`. The new hypothesis *A* is computed using tables (which are given below). There is a separate table for each of the three possible values of *op* (there is a table for a fourth operator (/), but the code implementing this part of the rule mysteriously vanished). The rows and columns of these tables are indexed by term schemas. The new hypothesis is the conjunction (that is, independent product) of the formulas which are in the intersection of that row and that column of the table for *op* such that the row and column indices match hypotheses h_1 and h_2, respectively. For example, if h_1 and h_2 are t_1`>`t_2 and t_3`=`t_4, and *op* is `+`, then the new hypothesis will be

t_1`+`t_3`>=`t_2`+`t_4`+1 &` t_1`+`t_4`>=`t_2`+`t_3`+1.`

The monotonicity rule does not make use of definitions, so the above formula will actually appear in terms of Nuprl primitives. The correspondence between the operators that appear in the tables and what is actually used by the monotonicity rule is as follows.

a`>`*b*	*is represented by*	*b*`<`*a*
a`=`*b*		*a*`=`*b* `in int`
a`>=`*b*		*a*`<`*b* `-> void`
a`~=`*b*		(*a*`=`*b* `in int`) `-> void`
a`<=`*b*		*b*`<`*a* `-> void.`

The tables for addition, subtraction and multiplication are given in Figures 7.1, 7.2 and 7.3, respectively.

division

`H >>` *a* `=` *b*`*(`*a*`/`*b*`)+(`*a* `mod` *b*`) by division`
`>>` *b*`~=0`
`>>` *a*`>=0`
`H >>` *a* `=` *b*`*(`*a*`/`*b*`)-(`*a* `mod` *b*`) by division`
`>>` *b*`~=0`
`>>` *a*`<=0`

	z>w	z>=w	z=w	z~=w
x>y	x+z>=y+w+2	x+z>=y+w+1	x+z>=y+w+1 x+w>=y+z+1	error
x>=y	x+z>=y+w+1	x+z>=y+w	x+z>=y+w x+w>=y+z	error
x=y	x+z>=y+w+1 y+z>=x+w+1	x+z>=y+w y+z>=x+w	x+z=y+w x+w=y+z	x+z~=y+w x+w~=y+z
x~=y	error	error	x+z~=y+w x+w~=y+z	error

Figure 7.1: Monotonicity table for +.

	z>w	z>=w	z=w	z~=w
x>y	x-w>=y-z+2	x-w>=y-z+1	x-w>=y-z+1 x-z>=y-w+1	error
x>=y	x-w>=y-z+1	x-w>=y-z	x-w>=y-z x-z>=y-w	error
x=y	x-w>=y-z+1 y-w>=x-z+1	x-w>=y-z y-w>=x-z	x-w=y-z y-w=x-z	x-w~=y-z x-z~=y-w
x~=y	error	error	x-w~=y-z x-z~=y-w	error

Figure 7.2: Monotonicity table for -.

	y>=z	y>z	y=z	y~=z
x>0	xy>=xz	xy>xz	xy=xz	xy~=xz
x>=0	xy>=xz	xy>=xz	xy=xz	error
x=0	xy=xz xy=0	xy=xz xy=0	xy=xz xy=0	xy=xz xy=0
x<=0	xy<=xz	xy<=xz	xy=xz	error
x<0	xy<=xz	xy<xz	xy=xz	xy~=xz
x~=0	error	xy~=xz	xy=xz	xy~=xz

Figure 7.3: Monotonicity table for *.

```
H >> 0 < a/b by division
   >> 0<b & a>=b | b<0 & a<=b
H >> a/b < 0 by division
   >> 0<b & -a>=b | b<0 & -a<=b
H >> a/b = 0 by division
   >> b~=0
   >> a=0
H >> a/b = q & a mod b = r by division
   >> b~= 0
   >> a>=0 & a=b*q+r | a<=0 & a=b*q-r
   >> 0<=r
   >> 0<b & r<b | b<0 & r<-b
```

These rules use the notations given above for the monotonicity rule. One note on "user-friendliness": the rule allows conclusions which match the patterns modulo commutativity over * and + and = . For example, 0 = a/b will "match" the fifth division rule format; and (a/b)*b + (a mod b) = a will match the first pattern. Thus one need not memorize specific details of the patterns, but only remember the "basic format" in order to know when the rule can be successfully invoked.

Chapter 8

The Metalanguage

The programming language ML is a metalanguage for Nuprl's type theory. Its main use is to write programs called *tactics* that automate the proof construction process.

ML is not documented here. The version of ML used in Nuprl is a minor variant of the ML used in the Edinburgh LCF system. This is documented in the book *Edinburgh LCF: A Mechanized Logic of Computation*, M. Gordon, R. Milner and C. Wadsworth, Springer-Verlag LNCS 78, 1979. The language is partially documented in the Nuprl book, which also gives some examples of tactics.

In this section we describe the embedding of Nuprl's logical apparatus within ML, and the structure of the tactic mechanism. Many of the details, such as ML primitives corresponding to the rules of the type theory, and various utilities for accessing libraries and printing, are left to appendices.

In what follows, as well as in Appendix A, a description of an ML function will usually begin with an application of the function to a list of typed formal arguments. For example, the `max` function might be presented as

`max(x: int, y:int): int`. The maximum of `x` and `y`.

This indicates that the type of max is `int->int->int`.

8.1 Nuprl ML versus LCF ML

The ML used by Nuprl has been adapted from the ML of LCF. The changes are as follows.

- All components related to LCF's logic have been removed, including the term-quotation facility.
- New primitive functions and datatypes have been added (these are documented below).
- New facilities for naming constants have been added. In particular, single forward quotes (“’”) are used to delimit the text of Nuprl terms, and a string enclosed in

double backquotes (“‘”) is interpreted as the list of tokens obtained by breaking the string into words according to the occurrences of blanks.

- The names of the functions mapping between an abstract type and its implementation type have an underscore added.

8.2 Basic Types in ML for Nuprl

In specializing the ML language to Nuprl we have made each of the *kinds* of objects that occur in the logic into ML types. These types are abstract in the sense of the abstract type mechanism of ML, except that they are defined in the implementation of ML. Objects of the types can be created or accessed only through the provided functions.

There are four such types: `term`, `declaration`, `rule` and `proof`. The polymorphic equality test `=` of ML has been extended to these types in the following way. Two objects of type `term` are equal if and only if they are α-convertible variants of each other. That is, they are equal if and only if the bound variables can be renamed so that the terms are identical. Unfortunately, equality on proofs does not respect term equality. Two objects of type `proof` are equal if they are *identical*. In particular, two proofs which differ only because of renaming of some bound variables will *not* be equal in ML. Two objects of type `rule` (or `declaration`) are equal if and only if they are identical.

`term`

The type `term` is the type of terms of Nuprl's logic.

`term_kind(t:term): tok`. A token naming the kind of term `t` is. For example, if `t` is the term $2 + 3$ then the result is the token `‘ADDITION‘`. Possible results are: `UNIVERSE`, `VOID`, `ANY`, `ATOM`, `TOKEN`, `INT`, `NATURAL-NUMBER`, `MINUS`, `ADDITION`, `SUBTRACTION`, `MULTIPLICATION`, `DIVISION`, `MODULO`, `INTEGER-INDUCTION`, `LIST`, `NIL`, `CONS`, `LIST-INDUCTION`, `UNION`, `INL`, `INR`, `DECIDE`, `PRODUCT`, `PAIR`, `SPREAD`, `FUNCTION`, `LAMBDA`, `APPLY`, `QUOTIENT`, `SET`, `EQUAL`, `AXIOM`, `VAR`, `BOUND-ID`, `TERM-OF-THEOREM`, `ATOMEQ`, `INTEQ`, `INTLESS`, and `TAGGED`.

Each kind of term has associated with it three functions: a recognizer, a destructor, and a constructor. For example, for addition terms we have the following.

`is_addition_term(t:term): bool`.

`destruct_addition_term(t:term): term#term`. Fails if `t` is not an addition term.

`make_addition_term(a:term, b:term): term`.

The analogous functions for the other term kinds are documented in Appendix A. A constructor will fail if the result would not be a term, such as if `make_universe_term` were applied to a negative number. Nuprl definitions appearing in a term are invisible to ML.

In addition to the individual term constructors, term quotes may be used to specify terms in ML. For example, the ML expression

```
'2+3'
```

is of type `term` and has as its value the term $2 + 3$. Note that these quotation marks are different from those used for tokens in ML. *Warning.* Term quotes can *not* be used in ML *files*. They can only be used in ML library objects or in ML mode.

`rule`

This is the type of (names of) Nuprl refinement rules. The constructors for this type do not correspond precisely to the rules in Nuprl. Refinement rules in Nuprl are usually entered in the proof editor as "`intro`", "`elim`", "`hyp`", etc. Strictly speaking, the notation "`intro`" refers not to a single refinement rule but to a collection of introduction refinement rules, and the context of the proof is used to disambiguate the intended introduction rule at the time the rule is applied to a sequent. There is a similar ambiguity with the other *names* of the refinement rules. In addition to this ambiguity, the various sorts of the rules require different additional arguments. Because rules in ML may exist independently of the proof context that allows the particular kind of rule to be determined, and because functions in ML are required to have a fixed number of arguments, the rule constructors have been subdivided beyond the ambiguous classes that are normally visible to the Nuprl user. For example, there are constructors like `universe_intro_integer`, `product_intro` and `function_intro`, all of which are normally designated in proof editing with `intro`. These rule constructors are given in Appendix A.

There is also a generic constructor that can be used when the proof context is known.

`parse_rule_in_context(x:tok, p:proof):rule`. This function uses the same parser that is used when a rule is typed into the proof editor.

There are no destructors, but there are several marginally useful functions which do a limited amount of rule analysis. The first two are used in the transormation tactic `copy`.

`get_assum_number(r:rule): int`. If `r` is a rule which refers to a single hypothesis, return the number of the hypothesis.

`set_assum_number(r:rule, i:int): rule`. If `r` is as above, modify the rule to refer to the hypothesis numbered `i`.

`rule_kind(r:rule): tok`. The "kind" of `r`. At present this is the internal name of the rule, which is not necessarily the same as what appears in the name of the corresponding rule constructor. The actual names are not documented here.

`declaration`

This is the type of Nuprl bindings that occur in proofs. A declaration associates a variable with a type (term). The declarations in a Nuprl sequent occur on the left of the `>>`.

`construct_declaration(x:tok, T:term): declaration`. A declaration where the variable `x` is associated with the type `T`.

`destruct_declaration(d:declaration): tok#term`.

`id_of_declaration(d:declaration): tok`.

`type_of_declaration(d:declaration): term.`

proof

This is the type of partial Nuprl proofs. Proofs have the structure described in Section 1.1. The related constructors, destructors and predicates are given below. The fact that (non-degenerate) proofs can only be built via the refinement functions guarantees that all objects of type `proof` represent proofs.

`hypotheses(p:proof): declaration list.` The hypotheses of the root sequent of the proof `p`.

`conclusion(p:proof): term.` The term to the right of the turnstile (`>>`) in the root sequent of `p`.

`refinement(p:proof): rule.` The rule at the root of `p`, if one is present (otherwise fail).

`children(p:proof): proof list.` The children of the root of `p`. The function fails if there is no refinement.

`refine(r:rule): tactic.` The type `tactic` is described in the next section. It is defined as the type

```
proof -> proof list # (proof list -> proof).
```

The function `refine(r)` when applied to a proof p applies the rule `r` to the sequent s at the root of p. This either generates a list l of subgoals according to the specifications given in Chapter 7, or, in the case the rule is inapplicable, fails. The result of `refine(r)(p)` is the pair (l, f) where f is a function (called a *validation*) that takes a list pl of proofs, checks whether the list of root sequents is equal to l (failing if not), and if so creates a new proof whose root sequent is the root sequent of p, whose refinement rule is r, and whose children are pl.

The composition of `refine` with `parse_rule_in_context` (see above) can be used to simulate the proof editor, where a string is interpreted as a specification of a rule.

8.3 Tactics

The type of tactics is

```
tactic = proof -> proof list # (proof list -> proof),
```

The component of a tactic that has type `proof list -> proof` is called the *validation*. The basic idea is that a tactic decomposes a goal proof g into a finite (possibly empty) list of subgoals, $g_1, \ldots, g_n$, and also produces a validation v which "jusfifies" the reduction of the goals to the subgoals. The subgoals are degenerate proofs, i.e., proofs which have only the top-level sequent filled in and have no refinement rule or subgoals. We say that a proof

p_i *achieves* the subgoal g_i if the top-level sequent of p_i is the same as the top-level sequent of g_i. The validation, v, is supposed to take a list $p_1, \ldots, p_n$, where each p_i achieves g_i, and produce a proof p that achieves g. (Note that wherever we speak of proof in the context of ML, we mean possibly incomplete proofs.) Because of the combinator language for tactic construction it is rarely necessary to explicitly refer to validations.

Tacticals

Tacticals allow existing tactics to be combined to form new tactics.

`IDTAC`: `IDTAC` is the identity tactic. The result of applying `IDTAC` to a proof is one subgoal, the original proof, and a validation that, when applied to an achievement of the proof, returns the achievement.

tac_1 `THEN` tac_2: The `THEN` tactical is the composition functional for tactics. tac_1 `THEN` tac_2 defines a tactic which when invoked on a proof first applies tac_1 and then, to each subgoal produced by tac_1, applies tac_2.

tac `THENL` *tac_list*: The `THENL` tactical is similar to the `THEN` tactical except that the second argument is a list of tactics rather than just one tactic. The resulting tactic applies *tac*, and then to each of the subgoals (a list of proofs) it applies the corresponding tactic from *tac_list*. If the number of subgoals is different from the number of tactics in *tac_list*, the tactic fails.

tac_1 `ORELSE` tac_2: The `ORELSE` tactical creates a tactic which when applied to a proof first applies tac_1. If this application fails, or fails to make progress, then the result of the tactic is the application of tac_2 to the proof. Otherwise it is the result that was returned by tac_1. This tactical gives a simple mechanism for handling failure of tactics.

`REPEAT` *tac*: The `REPEAT` tactical will repeatedly apply a tactic until the tactic fails. That is, the tactic is applied to the goal of the argument proof, and then to the subgoals produced by the tactic, and so on. The `REPEAT` tactical will catch all failures of the argument tactic and can not generate a failure.

`COMPLETE` *tac*: The `COMPLETE` tactical constructs a tactic which operates exactly like the argument tactic except that the new tactic will fail unless a *complete* proof was constructed by *tac*, i.e., the subgoal list is null.

`PROGRESS` *tac*: The `PROGRESS` tactical constructs a tactic which operates exactly like the argument tactic except that it fails unless the tactic when applied to a proof makes some progress toward a proof. In particular, it will fail if the subgoal list is the singleton list containing exactly the original proof.

`TRY` *tac*: The `TRY` tactical constructs a tactic which tries to apply *tac* to a proof; if this fails it returns the result of applying `IDTAC` to the proof. This insulates the context in which *tac* is being used from failures. This is often useful for the second argument of an application of the `THEN` tactical.

8.4 Print Representation

There are two kinds of functions to convert Nuprl syntactic objects into string representations. One converts objects to ML tokens, and the other builds a representation in a file.

`term_to_tok(t:term): tok`. A string similar to what would appear in a Nuprl window. Note that this may include the display forms of Nuprl definitions, and so there is no inverse function to recover `t` from the string. This remark applies also to the following two functions.

`rule_to_tok(r:rule): tok`.

`declaration_to_tok(d:declaration): tok`.

`print_term(t:term): void`. Print `t` in the command window. This works only in ML mode.

`print_rule(r:rule): void`. Print a representation of `r` in the command window.

`print_declaration(d:declaration): void`.

`latexize_file(input_file: tok; output_file: tok): void`. Create a source file for Latex. The `input_file` must be the result of `mm`, `ppp`, `print_library` or a "snapshot" (OUTPUT). The flag described next must have been set appropriately.

`set_output_for_latexize(flag: bool) : void`. This sets a flag which may be ignored if "special" characters are not used. (Special characters are the ones outside the standard character set that require Nuprl's fonts in order to be printed.) If the output from `mm`, `ppp`, `print_library` or a snapshot command (see below) is to be run through latexize_file then the flag should be set to *true*. If the output is to be treated as standard ascii text then the flag should be set to false. The difference in the two kinds of output is in how special characters are treated. In the first case they are written out using a code (usually a control character); `latex_file` will convert these to the appropriate Latex commands. In the second case, a directly-readable character-string representation is written out. Initially, the flag is true.

`print_library(first:tok, last:tok, file_name: tok): void`. Print the following components of the segment of the current library between objects first and last to the named file:

1. the name and status of every library object,
2. the body of every DEF, ML, and EVAL object, and

3. the extraction (if it currently exists) of any theorem whose name ends with an underscore.

The tokens ‘`first`‘ and ‘`last`‘ may be given as the arguments `first` and `last` and are interpreted to refer to the first and last library objects, respectively.

`mm: tactic`. Print a proof to a file. The next function works similarly to `mm` but produces better looking output. `mm` is the same as IDTAC except that as a side effect it prints out the nodes of its proof argument in the order visited by a depth first traversal of the proof tree. The output is appended to whatever file snapshot output would go to. Indenting is used to indicate depth, and redundant hypotheses are not displayed. Vertical lines (or an approximation thereof) are printed to help the reader keep track of levels, and tick marks (hyphens, actually) are put on the lines to indicate the beginning of a conclusion of a proof node. Given an arbitrary turnstyle (“>>”) in the printout, to the right of it will appear the conclusion of a node of the proof tree. Directly above will be a numbered list of the hypotheses of the proof node which are different from the hypotheses of the nodes parent node. The parent is found by following upward the first vertical line to the left of the turnstyle. Right after the conclusion of the node is the refinement rule, and below that, indented once from the turnstyle, are the children of the node. “INCOMPLETE” in place of a rule indicates that no rule was applied there.

`ppp: tactic`. Pretty-print a Nuprl proof tree.

Print format for proof node is schematically as follows.

```
[...branches....]| 1. [id. (if not NIL)][...Hypothesis term.1........]
[...branches....]|    [............Hypothesis term.1.................]
       .                    .                                  .
       .                    .                                  .
[...branches....]|    [............Hypothesis term.1.................]
       .                    .                                  .
       .                    .                                  .
       .                    .                                  .
[...branches....]| n. [id. (if not NIL)][...Hypothesis term.n........]
[...branches....]|    [............Hypothesis term.n.................]
       .                    .                                  .
       .                    .                                  .
[...branches....]|    [............Hypothesis term.n.................]
[...branches....]|->> [............Conclusion term...................]
[...branches....]?    [............Conclusion term...................]
       .                    .                                  .
       .                    .                                  .
[...branches....]?    [............Conclusion term...................]
[...branches....]? BY [............Refinement rule...................]
[...branches....]? !  [............Refinement rule...................]
       .                    .                                  .
```

```
      .                    .                              .
[...branches....]? !  [............Refinement rule..................]
[...branches....]? !
```

where

1. Branch at ?'s printed only if node is not last sibling.
2. branch at !'s printed only if node has children.
3. term and rule text is folded so as to always stay in the appropriate region.
4. A hypothesis is only printed if it is textually different from the same numbered hypothesis in the parent node.
5. If a hypothesis is hidden, []'s are put around the hypothesis number.
6. The root node does not have a branch entering the node.

`(file_name:tok): tactic`. Set snapshot file to `file_name` then apply `ppp`.

`set_pp_width(i:int): void`. Set the number of columns to print the proof with (default 105).

`complete_nuprl_path(directories: tok list; file_name: tok): tok`. Make a complete pathname from the current "nuprl path prefix", `directories`, and `file_name`. The result is formed by concatenating the path prefix, the strings in the list `directories` and `file_name`, separating with the string "/" (on Unix-like hosts). The main use of this function is in the "load-all" files in the main library directory of Nuprl it provides a relatively host independent way to specify the loading of other ML files. If Nuprl has been installed properly, then the default path prefix will be the root directory of Nuprl.

`set_nuprl_path_prefix(pathname: tok): void`. Set Nuprl's path prefix.

`set_snapshot_file(file_name:tok): void`. Directs subsequent output of the snapshot feature (OUTPUT) to the named file.

8.5 Further Primitives for Terms and Proofs

Most of the ML functions in this section could have been implemented in ML. They are primitive either for efficiency reasons, or because a Lisp implementation (Lisp is the implementation language of both Nuprl and ML) already existed. Many of the functions produce values that depend on the current state of Nuprl's library.

For Terms

`map_on_subterms(f: term -> term; t: term): term.` Replace each immediate subterm of t by its value under f.

`subterms(t: term): term list.` The immediate subterms of t.

`free_vars(t: term): term list.`

`remove_illegal_tags(t: term): term.` Remove all the tags from t that are illegal according to the tagging restrictions of the direct computation rules (see page 172 of the book). The result depends on whether the new versions of the direct computation rules are in force (see above).

`simplify(t:term): term.` Does the same thing as the simplification function used by arith.

`substitute(t: term; subst: term#term list): term.` If the first component of each pair in subst is a variable, then perform on t the indicated substitution.

`replace(t: term; subst: term#term list): term.` Like substitute, except that capture is not avoided.n

`first_order_match(pattern: term; instance: term; vars: tok list): tok#term list.` If the application succeeds, then the result is the most general first-order substitution s, for those variables in vars that are free in pattern, such that s applied to pattern is instance.

`compare_terms(t1: term, t1: term): bool.` True if term t1 is lexicographically less than term t2. False otherwise. This is a total ordering of terms and is useful in normalizing expressions.

`do_computations(t: term): term.` From the bottom up, replaced each tagged term in t by the result of doing the computations indicated by the tag (see page 172 of the book).

`compute(t: term): term.` Head reduce t as far as possible.

`no_extraction_compute(t: term): term#int.` Like compute, except that `term_of` terms are treated as constants, and the number of reductions done is also returned.

`open_eval(t: term; b: bool; thms: tok list): term.` Perform the evaluation defined in the description of the eval rules (see above).

`eval_term(t: term): term.` Apply Nuprl's evaluator to t. The result is the same as would be obtained from entering eval mode (using the eval command) and entering t.

`instantiate_def(def_name: tok; args: term list): term.` Use args as the actual arguments to an instance of the definition `def_name`. This function operates by unparsing the arguments textually plugging them into a definition instance, and reparsing. This is quite inefficient.

`instantiate_trivial_def(def_name: tok; arg: term): term.` The result is equal to arg, and is an instance of the "trivial" definition `def_name`. A trivial definition is one which has a single formal parameter that forms the entire right-hand-side of the template. This function does not involving parsing.

`add_display_form:(adder: term -> (tok # (term list)); t: term): term`. The operation of this function can be defined recursively. Suppose the immediate subterms of t have been replaced by their results under `(add_display_form adder)`, giving t'. If adder fails on t', then return t'. Otherwise, (adder t') is the pair (name,args). Attempt to instantiate the definition named by name using args as the actual parameters. If this succeeds with a result t"=t', then return t", otherwise return t'. The whole procedure involves only one parsing operation.

`remove_display_forms(t: term): term`. Remove all the display forms (i.e., definition instances) from t. The result is equal as a term to t but will be displayed by Nuprl as a term of the base type theory.

For Proofs

`frontier: tactic`. For p a proof, fst (frontier p) is the proof list consisting of the unproved leaves of p.

`subproof_of_tactic_rule(r: rule): proof`. The subproof built by the refinement tactic that is the body of the rule r.

`hidden(p: proof): int list`. The numbers of the hidden hypotheses in the sequent at the root of p.

`make_sequent(dl: declaration list, hidden: int list, concl: term): proof`. The first argument is the hypothesis list, the second is the list of hypotheses which are to be hidden, and the last is the conclusion. The returned proof has incomplete status, of course. The function fails if the the sequent could not ever appear in a Nuprl proof (e.g., if not all variables are declared).

`equal_sequents (p1: proof, p2: proof): bool`. Sequents are equal when the variables of corresponding declarations are identical, the terms of corresponding declarations are alpha-convertible, the same hypotheses are hidden, and the conclusions are alpha-convertible.

`main_goal_of_theorem(thm_name: tok): term`.

`proof_of_theorem(thm_name: tok): proof`.

`extracted_term_of_theorem(thm_name: tok): proof`.

8.6 ML Primitives Related to the Library and State

`make_eval_binding(id: tok; t: term): term`. The result is the same (side effects on the eval environment included) as would be obtained by entering "let id = t;;" in eval mode.

`execute_command_line(line:tok): void`. `line` is a line of text that one could type to the command window to the `P>` prompt. The commands that can be used are: load,

dump, create, delete, jump, move, scroll, down, check, view, save, restore, kill, copystate and exit.

`execute_command(cmd: tok, args: tok list): void.` `cmd` is the name of the command, and `args` is its list of its arguments.

`create_def(name:tok, pos:tok, lhs:tok, rhs:term): void.` `name` is the name of the def, `pos` is the library position (exactly as one would type it to the command window), `lhs` is the left-hand side of the definition (exactly as one would type it into an edit window), and `rhs` is a term which is unparsed to yield the right-hand side of the definition. The free variables of the term are turned into formal parameters (via the addition of angle brackets). The function fails if the set of lhs formal parameters is not equal to the set of rhs formal parameters. The generated rhs has outer parens and parens around all formals added.

`kill_extraction(name:tok): void.` Remove the extraction field from the theorem whose name is `name`.

`arity_of_def(def_name: tok): int.` The number of formal parameters in the definition `def_name`.

`formal_list_of_def(def_name: tok): tok list.` A list of the formal parameters (without angle-brackets) of the definition `def_name`.

`rhs_formal_occurrences_of_def(def_name: tok): tok list.` The list of formal parameters (without angle-brackets) that would be obtained if all text except formal parameters were removed from the right-hand-side of the definition `def_name`.

`create_theorem(name: tok; lib_position: tok; p: proof): void.` A new theorem is created in the library with name name, library position given by `lib_position` (which should be in the same form as would be given to the create command), and whose main goal and proof are given by p.

`create_ml_object(name: tok; lib_position: tok; body: tok): void.`

`kill_extraction(thm_name: tok): void.`

`lib(prefix: tok): tok list.` A list of all names that have prefix as a prefix and that name an object in the current library.

`library():void): tok list.` A list of the names of all objects in the current library, in the same order.

`object_kind(object_name: tok): tok.` One of "DEF", "ML", "EVAL", or "THM".

`rename_object(old_name: tok; new_name: tok): void.`

`object_status(object_name: tok): tok.` One of "COMPLETE", "PARTIAL", "RAW", or "BAD".

`is_lib_member(object_name: tok): bool.`

`restore_library_from_state(state_name: tok): void.` Replace the current library by a copy of the library contained in the named state. No other components of the current state are modified. (For a description of states, see the paragraph on the save and restore commands above.)

`transform_theorem(x:tok, p:proof): void`. The first argument is the name of a theorem, and the second is a proof which is to replace the current proof of the named theorem. The new proof cannot have hypotheses, and its conclusion need not be the same as the current theorem's main goal. This function is useful in conjunction with tactics and the function `proof_of_theorem`.

`apply_without_display_maintenance(f:*->**, a:*): **`. Apply a function to an argument while disabling the display-maintenance procedures. Whenever a substitution into a term is done during the application, little effort will be made to compute a reasonable display form for the term. This means that the term may have all occurrences of Nuprl definitions removed, leaving a term of the base type theory. Using this function can result in a ten-fold speedup for computations (such as symbolic computation) that do alot of substitution.

`set_display_maintenance_mode(on: bool): void`. Display-maintenance procedures become disabled if on is false, and enabled if on is true.

`display_message (message: tok): void`. Displays message in the command/status window.

`set_auto_tactic(tactic_name: tok): void`. Set the auto-tactic to be the ML expression represented by `tactic_name`.

`show_auto_tactic(): tok`. The current setting of the auto-tactic.

8.7 Compatibility with Old Libraries

In order to allow old libraries to be loaded and viewed, it is possible to restore old versions of some of the rules that have been changed. The following ML functions can be used to toggle the rule versions used and to find out which version is in effect. These functions should not be considered to be an "official" part of the system; they are provided only so that old libraries can be viewed.

`make_all_tags_legal(yes:bool): bool`. Make the direct computation rules ignore the tagging restrictions iff yes is true.

`all_tags_legal(foo:void): bool`. True iff the tags are ignored.

`use_old_set_rules(yes:bool): bool`. Use the old set rules iff yes is true.

`old_set_rules_in_use(foo:void): bool`.

`reset_rules(foo:void): void`. Reset the rules to their initial state (new set rules, all tags legal).

8.8 Loading and Compiling ML Code

Although Nuprl provides library objects for ML code, most users write their ML programs using a text editor of their choice and store them in regular files. There are ML functions

(which can be invoked in the command window in ML mode) that load and compile files of ML code. The name of file containing ML code should end in the characters ".ml".

There is not a tremendous difference in the performance of compiled versus uncompiled ML (although it is significant); the main advantage of compiling is that the ML files can subsequently loaded much more quickly (100 times?). Also, compiling an ML file is not too much slower than loading an uncompiled file; in fact, in some Lisps compiling is faster.

If you load a compiled ML file and get an error having to do with a symbol being unbound, it means that the compiled file was compiled in an environment which no longer exists. This usually happens because a change is made to an ML file (`load` always gets the newest version of an ML file, whether or not it is a binary) but a compiled file that depends on it is left unchanged.

`load(file_name: tok; types?: bool): void`. Look for binary or ML versions of the file named `file_name`. The ML version must exist. If the ML version is the newest of the versions, load it; otherwise load the binary version. The names for the versions are obtained by adding to file_name the appropriate extension (e.g. ".ml" for ML files). If `types?` is true, then after loading, a list of the names of all loaded functions, together with their types, is printed out. Note: this printout can blow the control stack in some lisps (e.g. Lucid), possibly corrupting the state, if the printout is large.

`compile(file_name:tok; types?: bool): void`. Compile the ML version of `file_name`. The compiled version is also loaded into the ML state.

Chapter 9

Basic Tactics

A collection of ML files is automatically loaded into Nuprl whenever the system is loaded. Some of these files contain the implementation of the ML functions described in the book. The remaining files contain a fairly large collection of tactics and related functions. This tactic collection should not be regarded as part of Nuprl. Many tactic collections have been developed at Cornell over the last several years; this one was chosen because it is the most recent large collection that has been reasonably documented. It is not, however, free of problems. Some of the tactics have minor bugs, and others could be easily made much more effective.

Serious users of Nuprl will want to build on and modify the basic tactic collection. For those who wish to do so, the relevant files are named in the file `nuprl/ml/load`. Users who want a detailed knowledge of these tactics will also have to consult these files, since this document does not contain a full desription of the tactic collection.

Most of the tactics (and related objects) in the Nuprl collection will be at least mentioned in this document; the ones that are omitted are mostly simple variants of objects that are mentioned, or have turned out not to be very useful in practice. An example of the use of the tactics can be found in the library `nuprl/lib/saddleback/new-lib`. Expanding this library does not require that any extra ML files be loaded.

The tactic collection described in this document is almost identical to the Nuprl 2.0 collection. Many new tactics have been written since Nuprl 2.0 was released, but we have not had the time to package and document these tactics for use elsewhere. We hope to be able to do this in the near future, though. The remainder of this document is the same as the one describing the 2.0 tactics except for a brief section at the end which describes a few additions and fixes.

Most of this document has the form of a user's manual, although many of the descriptions will omit details that are uninteresting but necessary for actual use of the described function. A typical description will be headed by a schematic application that may contain type information. For example, a description of the function `destruct_hyp` might be introduced by

```
destruct_hyp (i:int) (p:proof):  tok#term.
```

This indicates that the function has type `int -> proof -> tok#term`. The parameters (`i` and `p` in the example) will often be referred to in the description.

Many of the tactics described perform some kind of rewriting on a component of a sequent. There are usually several ways to apply a particular kind of rewriting; for example, one can apply it just to the conclusion, or to some hypotheses, or to all hypotheses and the conclusion. Usually, only one of these will be mentioned (usually the one which applies rewriting everywhere). For example, only `Eval` will be described, although there are also tactics `EvalHyp` and `EvalConcl`.

9.1 Definitions

To take full advantage of the tactic collection, the user must follow a simple convention in making definitions. As pointed out in the previous chapter, although Nuprl's definition facility allows considerable flexibility in making notations for terms, it cannot reasonably serve as an abstraction mechanism. A solution to this deficiency is to define objects via extraction. Using this scheme, a complete definition consists of two library objects: a theorem object whose extraction is the defined term, and a definition object which gives a display form for the `term_of` term which denotes the extraction. Sometimes the defining theorem also serves to give a type to the defined term; other times, a separate theorem is used for this. The library objects associated with a definition are related by a naming convention (involving trailing underscores).

There are two styles of definition in this scheme. We will describe these using two simple examples. The first style of definition is exemplified by the definition of a function which takes the maximum of two natural numbers. The definition object for this function is called `N_max`, and its body is

```
max(<m>,<n>) == term_of(N_max_)(<m>)(<n>).
```

The theorem `N_max_` has statement

```
>> N -> N -> N,
```

and is proved by explicitly introducing the desired object, using the tactic invocation

```
(ExplicitI 'λm n.  less(m;n;n;m)' ...).
```

The second style of definition is exemplified by the definition of the length function (for lists). The definition object `length` is

```
|<l>| == term_of(length_)(<l>).
```

The theorem `length_` has statement

```
>> Object
```

and extraction

$$\lambda \texttt{l.}\ \ \texttt{list\_ind(l; 0; h,t,v.v+1).}$$

The type of the theorem extracted from is the trivial type `Object` because it is convenient to have the polymorphism of `length` implicit. (The type `Object` never appears except in these kinds of definitions. See Appendix B for more information on `Object`). In other words, instead of taking as arguments a type and a list over that type, the length function as defined above just takes a list as an argument. The typing lemma for this definition is called `length__` and is

$$\forall \texttt{A:U1.}\ \ \forall \texttt{l:A list.}\ \ \texttt{|l| in N}$$

In summary, an instance of a definition is actually an application of a `term_of` term to some arguments. Definition objects in the library should be regarded as incidental additions that make displayed terms easier to read.

9.2 Terms

There is a large set of functions for dealing with terms. Most of these functions are for analyzing, constructing, or recognizing terms. Only three of them will be described here.

`get_type (`p`:proof) (`t`:term).` The result is a type T for t, such that the autotactic can (almost always, in practice) prove that t is a member of T under the hypotheses of the sequent p. The algorithm that computes T uses a function `g` that computes a type given a term t and a typing environment e (an association between Nuprl variables and types). This function uses a simple recursive algorithm. If t is a variable, the result is obtained by looking up t in e. If t is a canonical term, the result is computed from types for the immediate subterms. This is not alway possible (in which case an exception is raised), as when t is a lambda-abstraction. See the discussion of the membership tactic below for an explanation of why this kind of failure tends not to be a problem in practice. When t is a non-canonical term, the first step is to compute a type T for its principle argument. Head normalization and stripping off of set types is then repeatedly applied to T until a type T' is obtained that is canonical and not a set type. If t is an application $f(a)$, then T' must be a function type x`:`A`->`B (else failure), and the result is $B[a/x]$. If t is of the form `decide(`a`;`u`.`b`,`v`.`c`)`, then T' must be a disjoint union A`|`B, and the result is `g` applied to b and the environment e updated with an association between u and A; or if this fails, an analogous application to c. The `spread` case is analogous. If t is an induction form, then an attempt is made to compute the result just from the base case. Finally, if t is of the form `term_of(`a`)`, then the result is the main goal of the theorem named a.

A special case is when t is an instance of a definition of the second kind described in Section 9.1. In such a case, there is a theorem that is a universally quantified statement ending with a term of the form a `in` A. Matching of a against t is used to compute terms for instantiating the quantified variables appearing in A, and the resulting instance of A is returned as the type of t.

`match_part_in_context` f A t p l. Find a subformula A' of A, and a list of terms that yield t when substituted in A' for the variables that are bound by the universal quantifiers in A whose scope contains A'. The result is the list of terms. The subformula A' is one which can be obtained by descending through conjunctions, through universal quantifiers, through implications via the consequent, and finally by applying the function f, which should be a term destructor. First-order matching is used to compute substitutions for the free variable occurrences of A' that are bound in A. If matching does not provide instantiating terms for all the universal quantifiers encountered on the descent to A', then type information is used. In particular, for each universally quantified variable x which has been instantiated with a term a, a type is computed for a using `get_type` p a, and this type (or certain derived types, if necessary) is matched against the type obtained from the appropriate universal quantifier. After all possible type information has been used, the term list l is used for any further required instantiating terms.

The use of type information is important for definitions that use implicit polymorphism. As a simple example, consider the function `tl` that computes the tail of a list. Suppose that it has the typing lemma

```
∀A:U1.  ∀l:A list.  tl(l) in  A list.
```

In computing the type of a particular application `tl(`s`)`, the matching of `tl(`s`)` against `tl(l)` gives an instantiation only for `l`. If a type for s can be computed, and this type is (or can be reduced to) a list type, then matching against the declared type `A list` for `l` gives an instantiation for `A`. Instantiating the lemma with the computed terms yields a type for `tl(`s`)`.

`tag_redices (`t`:term)`. This tags redex-contractum pairs in t so that direct computation will result in all the contractions being performed. This function can be updated from ML objects in the library (by adding to a global list). This allows the user to make some of the tactics that use reduction take into account definitional extensions to the redex-contractum relation.

9.3 Computation

There are many tactics which are based on the evaluation and direct computation rules. These include the tactics for definition expansion.

`Eval`. Perform evaluation on the conclusion and all hypotheses, expanding `term_of` terms whenever required for computation to proceed.

`EvalOnly` *names*. Like `Eval`, except that terms of the form `term_of(`x`)` are only expanded if x is a member of the list *names*.

`EvalExcept` *names*. Like `Eval`, except that terms of the form `term_of(`x`)` are treated as constants when x is a member of the list *names*.

`EvalSubtermOfConcl` b *names* P. Evaluate a subterm of the conclusion that satisfies the predicate P. The arguments b and *names* have the same purpose as in the evaluation rule.

`CC`. Compute the conclusion to head normal form.

`ComputeEquands`. If the conclusion is of the form a=b `in` A then head-normalize the "equands" a and b, else fail.

`Reduce`. Repeatedly perform reductions of redex-contractum pairs wherever the legal-tagging restrictions permit it.

`HypModComp(`i`:int)`. It can require excessive computation time to demonstrate that a hypothesis is equal (as a type) to the conclusion of a sequent by completely normalizing both terms and then checking that the results are identical. `HypModComp` provides an approximation to complete normalization that is usually much faster. It proceeds by applying reductions to a subterm in the hypothesis and a corresponding subterm in the conclusion. Initially, the subterms are the entire respective terms. At each stage, it applies the smallest number of reductions such that the two resulting subterms are either instances of the same definition, or are in head normal form with the same head. If this is impossible, the tactic fails, otherwise the procedure is repeated on the immediate subterms of the resulting subterms. An important application of this tactic is in the inclusion tactic described below.

There are several tactics that are used for *unfolding* (expanding) and *folding* (the inverse of unfolding) definitions. Because definitions are applications of `term_of` terms to some arguments, unfolding a definition occurrence in a sequent requires using direct computation to replace the appropriate `term_of` term by its denotation and to do reduction steps until all the arguments to the definition instance are substituted into the body of the definition.

`Unfold` *names*. Unfold all definitions whose name is in the list *names*.

`Fold` *names*. Add as many instances of the named definitions as possible. This is not alway easy, since arguments to a definition instance can disappear when the definition is expanded. This is often the case for arguments that are types. For example, the length function given in Section 9.1 could have been defined to be

```
λA. λl.  list_ind(l; 0; h,t,v.v+1),
```

which has type

```
A:U1 -> l:(A list) -> N.
```

To fold a term

```
list_ind(l; 0; h,t,v.v+1)
```

into an application of the length function, the type of l has to be computed.

`UnrollDefs` *names*. For each *name* in *names* and each instance of the definition named *name*, do the following. Unfold the definition, then fail if the result is not a redex that is of the form `list_ind(...)` or `rec_ind(...)`, otherwise contract the redex and attempt to fold it to an instance of the definition *name*. For example, unrolling `|h.t|` gives `1+|t|`.

9.4 Tacticals

A *tactical* in LCF is a function which is used for combining tactics. The basic LCF tacticals have analogues in Nuprl.

T `THEN` T'. Apply the tactic T. If it fails, then fail, otherwise apply T' to the subgoals generated by T. This tactical has several variants which incorporate useful restrictions on the kind of subgoal T' is applied to. T `THENS` T', when applied to a proof p, only applies T' to subgoals that have the same conclusion as p. `THENO` applies its second argument only to goals that have a *different* conclusion. `THENM` and `THENW` apply their second argument only to goals that are not membership goals or well-formedness goals respectively. (A membership goal is one whose conclusion is of the form `>>` t `in` T, and a well-formedness goal is one whose conclusion is of the form `>>` T `in U`i.) Because many tactics have actions that are well-defined except for the generation of "junk" subgoals, the variants are used extensively for combining these tactics (both at the top level in proof construction, and in the construction of other tactics).

`Repeat` T. Apply T until it fails or ceases to make progress. This tactical has variants analogous to the variants of `THEN`.

T `ORELSE` T'. Apply T. If it fails, apply T'.

9.5 Some Derived Rules

Many of the tactics in this section generate unwanted subgoals. These are usually membership goals, or can be proven by some trivial propositional reasoning. In practice they can almost always be proved by the autotactic, and so the tactics will be described as though they did not generate any "junk".

For each primitive inference rule in Nuprl, there is a corresponding tactic. Such tactics are useful partly because they compute many of the parameters required by the corresponding rule. Some of these parameters are easily computed, such as those giving names for new variables, and many are not, such as those which supply types for certain subterms of the goal. Many of these tactics also incorporate computation. For rules which require that a hypothesis or the conclusion be a canonical type, the corresponding tactic will head-normalize the appropriate term before attempting to apply the rule.

`ILeft`. If the conclusion is or computes to a disjoint union, then apply the introduction rule which picks the left disjunct. `IRight` is analogous.

`ITerms` l. If the conclusion is existentially quantified (using product types or set types), then introduce the terms in the list l as witnesses for the existentially quantified variables.

`I`. If the previous three tactics do not apply, and if the conclusion is a canonical type that is not an equality or `void`, then apply the introduction rule for the type.

`EqI`. If the conclusion is of the form t `in` T or t=t' `in` T, where t is not a variable and, in the latter case, t and t' have the same outermost term constructor, then apply the appropriate equality-intro rule. This will often involve computing a type. For example,

when applied to the goal

$$\texttt{>>}\ f\texttt{(}a\texttt{)}\ \texttt{in}\ T\texttt{,}$$

`EqI` must compute a functional type for f in order to apply the equality-intro rule for application. If the type computed for f is $x\texttt{:}A\texttt{->}B$, but $B[a/x]$ is not identical to T, then no equality-intro rule applies. In this case, `EqI` will assert

$$f\texttt{(}a\texttt{)}\ \texttt{in}\ B[a/x]\texttt{.}$$

To prove that this implies the original goal, `EqI` calls `Inclusion`, which is described in Section 9.9.

`EOn` t i. Instantiate with t the universally quantified formula in hypothesis i.

`E` i. If the type in hypothesis i is not universally quantified, apply the appropriate elimination rule.

`Unroll` i. If hypothesis i declares a variable x to be of a list or recursive type, then "unroll" the type. If the type is a recursive type, the main step is to apply the "unroll" rule for recursive types. If it is a list type, then there are two main subgoals. The first is essentially the goal with x replaced by `nil`, and the second is the goal with x replaced by the cons $h\texttt{.}t$ for h and t new variables.

`SubstFor` t. If t is of the form $a\texttt{=}b\ \texttt{in}\ A$, substitute b for a in the conclusion. There is an analogous tactic, `SubstForInHyp`, which works on hypotheses.

There are also slightly generalized versions of some of the elimination rules. For example, `RepeatAndE` breaks up a hypothesis which is a conjunction, making each conjunct into a new hypothesis. The work done by some of these generalizations can be somewhat substantial. As an example, consider the elimination rule for dependent product types. If a hypothesis declaring some variable x to be in a type of the form $y\texttt{:}A\texttt{\#}B$ is eliminated, then the hypothesis list is extended by three new hypotheses:

$$y\texttt{:}A\texttt{,}\ z\texttt{:}B\texttt{,}\ x\texttt{=<}y\texttt{,}z\texttt{>}\ \texttt{in}\ y\texttt{:}A\texttt{\#}B\texttt{,}$$

and in the conclusion `<`y`,`z`>` is substituted for x. The tactic `GenProductE` generalizes this rule to an arbitrary number of repeated products. For an n-fold product, $n+1$ new hypotheses are generated, the last of which is an equation between x and an n-tuple. Simply chaining together an appropriate sequence of applications of the elimination rule will generate n equalities, which must be collapsed using substitution. The tactic uses another method which is slightly better. In either case, new membership subgoals are generated along the way, and so applications of the tactic can be much slower than one might expect.

A useful way to apply elimination rules is with `OnVar`.

`OnVar` x `(`T`:int->tactic)`. If x is declared in the hypothesis list, apply T to that hypothesis. Otherwise, repeatedly apply introduction rules as long as the conclusion is a universal quantification or an implication, or until x becomes declared, in which case apply T to the hypothesis declaring it.

There are several tactics that perform generalization, replacing a term in the conclusion by a new variable.

`LetBe` x a A. If the goal is `>>` B, then the subgoal generated is

$$x{:}A,\ x{=}a\ \texttt{in}\ A\ \texttt{>>}\ B[x/a]$$

(regarding a subgoal `>>` a `in` A as a "junk" subgoal).

`TypeSubterm` a A. If the goal is `>>` b `in` B, then the subgoal generated is

$$x{:}A\ \texttt{>>}\ b[x/a]\ \texttt{in}\ B.$$

`ETerm` a. This is similar to `LetBe`, except that the type A for a is computed, and elimination is done on the resulting new declaration.

There are some simple derived rules for equality reasoning.

`DestructEq` f i. If hypothesis i is of the form $a{=}b$ `in` A, then compute a type A' and add a new hypothesis

$$f\texttt{(}a\texttt{)}\ \texttt{=}\ f\texttt{(}b\texttt{)}\ \texttt{in}\ A'.$$

The term f must be a lambda-abstraction, and should be a destructor function (such as projection from pairs).

`SplitEq` t. If the conclusion is of the form $a{=}b$ `in` A, then generate two subgoals, with conclusions $a{=}t$ `in` A and $t{=}b$ `in` A.

There are two frequently used tactics related to the set type.

`Unhide`. If the conclusion is a squashed type (of the form $\{\, Int \,|A\,\}$ for some type A) then the single subgoal is identical to the goal, except that no hypotheses are hidden (a hidden hypothesis is one that cannot be used until it becomes unhidden—hidden hypotheses are created only by the set-elim rule).

`Properties` l. For each term t in the list l, compute a type for t, head-normalize the type, and if a type of the form $\{x{:}A\,|\,B\}$ is obtained, then add $B[t/x]$ as a new hypotheses.

We end this section with a few miscellaneous tactics.

`AbstractConcl` t. If the conclusion is A, then create a new variable x and produce a subgoal with conclusion

$$\texttt{(}\lambda x.\quad A[x/t]\texttt{)}\ \texttt{(}t\texttt{)}.$$

This is useful in the application of certain lemmas which quantify over predicates, such as those expressing induction schemes.

`BringHyps` l. "Bring" some hypotheses (specified by the numbers in l) to the conclusion side of the sequent, adding them to the old conclusion as antecedents of an implication or as universal quantifiers. As an example, bringing the last hypothesis of

$$x{:}A\ \texttt{>>}\ B$$

results in

$$\texttt{>>}\ x{:}A\texttt{->}B.$$

`NonNegInd` x i. If the type in hypothesis i is (or computes to) a subrange of the integers, then use it to calculate an integer lower bound for the variable x declared by the hypothesis. If the lower bound is non-negative, then do induction on x, using the bound as the base case.

9.6 Chaining and Lemma Application

There is a group of tactics based on the well-known notions of backward and forward chaining. The description of these tactics requires two definitions. For terms A and B, A is an *assumption of* B if: B is of the form x:C->D (which may be degenerate, *i.e.*, an implication) and A is either identical to C or is an assumption of D; or B is of the form C#D and A is an assumption of C or of D. A is a *conclusion of* B if: A and B are identical; or B is x:C->D and A is a conclusion of D; or B is C#D and A is a conclusion of C or of D. As assumption A of B is an antecedent in B of a conclusion A' of B if: B is x:C->D and either A is C and A' is in D or A is an antecedent of A' in D; or B is C#D and either A is an antecedent of A' in C or A is an antecedent of A' in D.

`BackThruHypUsing` l i. Do one backward chaining step using hypothesis i. This uses `match_part_in_context` (described earlier) to match the conclusion C of the sequent against a conclusion of the type A that is hypothesis i. The terms in l are used as additional instantiating terms. If the match is successful, then the tactic generates as subgoals sequents with the same hypothesis list, but with conclusions that are suitble instantiations of assumptions of A that are antecedents in A of the subterm occurrence matched with C.

Several tactics are based on `BackThruHypUsing`.

`BackchainWith` T. This is most concisely described by ML code:

```
let BackchainWith Tactic =
Try Hypothesis THEN
AtomizeConcl THENW
( ApplyToAHyp (λi.  BackThruHyp i THENM BackchainWith Tactic)
  ORELSE Tactic
).
```

`Hypothesis` proves goals where the conclusion is identical to a hypothesis, `AtomizeConcl` repeatedly applies introductions until the conclusion is not a conjunction or function type, and `ApplyToAHyp` T, for T of type `int->tactic`, applies T to each hypothesis in turn until success (*i.e.*, until an application does not fail). A useful instance of this tactic is `BackchainWith Fail`, where `Fail` is the tactic which always fails. This implements a kind of Horn-clause theorem prover (based on depth-first search). It is used by the tactic `Contradiction`, which attempts to show the hypothesis list to be contradictory by asserting `void` and backchaining. Note that `BackchainWith` does not check for looping, although it would be easy to modify it so that it does (in fact the current version does check).

There are two main ways to apply lemmas. One is to apply them explicitly using the tactics described below. The other involves explicitly grouping them according to similarity of usage. The grouping is accomplished by using a naming conventions, as in the association of typing lemmas to definitions, or by using ML objects in the library to update global lists of tactics and lemma names. There are several tactics that refer to such lists; they generally use these lists by trying to apply every tactic in the list or every

lemma named in the list. For example, the tactics `Autotactic`, `Member` and `Inclusion`, described in a later section, each (as a last resort) refer to an assooiated list. Below we give some other examples. This technique is rather *ad hoc*, but it has turned out to be very useful in practice.

`LemmaUsing` *name* l. This is analogous to `BackThruHyp`, except that a lemma (*i.e.*, previously proved theorem), instead of a hypothesis, is "backed through". The tactic `Lemma` can be used when there is no term list l.

`FLemma` *name* l. If the hypotheses whose numbers appear in l can be matched to some of the assumptions of the theorem named by *name*, then a subgoal is generated that has the same conclusion and has as an additional hypothesis the smallest conclusion of the named theorem such that all its antecedent assumptions were matched. As in `match_part_in_context`, types of quantified variables are used to compute further instantiating terms. A typical example of the use of `FLemma` is in the application of a transitivity property. Suppose r has been proved, in a lemma named *foo*, to be transitive, *i.e.*,

$$\forall \texttt{x,y,z:}A.\quad r\texttt{(x,y) => }r\texttt{(y,z) => }r\texttt{(x,z)}.$$

If the first and second hypotheses in some sequent are $r\texttt{(}x\texttt{,}y\texttt{)}$ and $r\texttt{(}y\texttt{,}z\texttt{)}$, then an application of `FLemma`, with arguments *foo* and `[1;2]`, will generate a subgoal which has $r\texttt{(}x\texttt{,}z\texttt{)}$ as a new hypothesis.

`RewriteConclWithLemma` *name*. If the named lemma has an equality as a conclusion, then try to match the left side of the equality against a subterm of the conclusion of the sequent; if successful, substitute for that subterm the appropriate instance of the right side of the equality. This may generate non-trivial subgoals, other than the one resulting from the substitution, since the equality asserted by the lemma may have preconditions. This tactic has several variants.

`Decidable.` This just repeatedly applies tactics from a global list. The members of the list are typically applications of `Lemma` to the name of a theorem that asserts the decidability of some proposition (*i.e.*, that has a conclusion of the form $P \vee \neg P$). A related tactic is `Decide` which, given P, produces subgoals for the cases P and $\neg P$, calling `Decidable` on the subgoal $P \vee \neg P$.

`Assume` P. This is similar to `Decide`, except that it deals with goals of the form $\texttt{?}P$, where $\texttt{?}P$ is defined to be $P\texttt{|True}$. Types of this form are used to simulate ML-style failure in Nuprl. `Assume` P, if successful, generates two subgoals, one where P is a new assumption, and one which is identical to the goal. `Assume` looks for a previously defined function that either produces a proof of P or fails. The two subgoals correspond to the two possible outcomes.

9.7 A Type Constructor

For the purpose of this section, define a *module* to be a function of the form

$$\lambda x_1 \ldots x_m.\quad y_1\texttt{:}B_1\ \texttt{\#}\ \ldots\ \texttt{\#}\ y_{n-1}\texttt{:}B_{n-1}\ \texttt{\#}\ \{y_n\texttt{:}B_n\texttt{|}C\}.$$

that is a member of a type

$$x_1\texttt{:}A_1\ \texttt{->}\ \ldots\ \texttt{->}\ x_m\texttt{:}A_m\ \texttt{->}\ \texttt{U}i\texttt{.}$$

Many data types that arise in programming and in mathematics are modules by this definition (which is nonstandard). To help create and use such parameterized data types, there is a function `create_module`. The inputs to this function are the terms A_i, B_j, and C, a name M for the module, names for Nuprl functions for projecting components of members of the type, a library position at which to begin creating the associated library objects, a preferred name for a typical element of the module, and the universe level i.

Executing the function creates library objects for the following:

- The definition of the module M. This includes a definition object, and a theorem from which the module is extracted. The theorem is usually proved automatically.
- Definitions of n implicitly-polymorphic projection functions. For any members a_1, ..., a_m of A_1, ..., A_m, respectively, the i^{th} projection function selects the i^{th} component from an n-tuple that is a member of $M\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_m\texttt{)}$. The theorems that give the types of the projections are generated, and usually proved, automatically.
- A theorem that states that C, which can be thought of as an axiom about members of the module, is true of the projections of any members of any $M\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_m\texttt{)}$.
- Updating the `tag_redex` function (described above). An ML object is created that has the effect of advising certain tactics that perform reduction to consider an application of a defined projection to an n-tuple to be a redex-contractum pair.
- Updating the generic module eliminator. An ML object is created that has the effect of updating the `ModE` tactic to take account of the new module. This tactic can be applied to any hypothesis whose type is an instance (*i.e.*, application) of a module that has been defined by `create_module`. The tactic works like `GenProductE`, and chooses new variable names that are based on the names of the projection functions.

9.8 Derived Rules via Pattern Theorems

Often one wishes to write a tactic that simply summarizes a pattern of inference with a fixed structure. In such cases, it is often easier to partially prove a "pattern theorem" that contains syntactic variables which get instantiated when the "derived rule" is applied. Nuprl does not directly support this kind of construction of derived rules, but it is possible to simulate it. The method of simulation is somewhat inelegant; it is presented here because it is useful, and because it was used in the formalization of Girard's Paradox (which is contained in the directory `nuprl/lib`).

In stating these pattern theorems, atoms are used as placeholders for irrelevant subterms. The statement of this theorem is

```
>> ∀s:Term0.  "G".
```

Immediately preceding a pattern theorem in the library should be a corresponding ML object. This ML object, when checked, should use the function `set_d_tactic_args` to set certain global variables to contain the arguments that are necessary for the proof of the pattern theorem. During the proof of the pattern theorem, a tactic needing arguments that cannot be computed from the sequent it is applied to uses a special function to access the global variables. For example, a tactic may require terms to instantiate a universal quantifier with; one of the global variables refers to a list of terms from which the tactic may remove the needed number of terms by using the function `get_term_arg`.

The values given to the global variables by the ML object are just placeholders for the actual arguments that will be supplied by the tactic `Pattern` which uses the theorem as a pattern for constructing another proof. `Pattern`, given as input a set of actual arguments, assigns the arguments to the appropriate variables, and then applies, in depth-first order, the rules that occur in the pattern proof tree.

Many of the derived rules that are defined with this mechanism resemble elimination or introduction rules; that is, they analyze either a hypothesis or the conclusion. It is useful to encapsulate such tactics in a pair of tactics `DE` and `DI` (also called `DElim` and `DIntro`). These tactics try to apply members of an associated global list, continuing through the list until success. The user can have these lists updated from the library.

9.9 Type Checking and the Autotactic

As mentioned earlier, the most important single tactic is the autotactic. In this section, the autotactic and related tactics are discussed.

`Inclusion` i. Attempt to prove that a term t is in a type T' assuming that hypothesis i asserts that t is a member of a type T. This tactic will make progress in the following situations.

- One of the members of a user-defined list of tactics succeeds.
- T and T' are (or compute to) universes, with the level of T not greater than that of T'.
- T is a subtype of T'.
- T and T' "almost normalize" to identical terms. `HypModComp`, described in Section 9.3, is used here.
- T and T' are function types x:A`->`B and x:A`->`B', respectively, and `Inclusion` recursively makes progress on the goal

  x:A, y:B `>>` y `in` B'

The last case could be generalized to other type constructors. `Inclusion` is called frequently, mostly by `EqI` (see Section 9.5).

`PolyMember`. Deal with membership goals t `in` T where t is an instance of a definition that is of the second kind mentioned in Section 9.1. The tactic uses matching to compute terms with which to instantiate the typing lemma for the definition. This results in a new hypothesis t `in` T'. If T and T' are not identical, `Inclusion` is applied. The subgoals usually produced by `PolyMember` are to show that the arguments of the definition instance are of the appropriate type. `PolyMember` can also be applied when the conclusion is a binary equality with both sides of the equality being instances of the same definition. In this case, the subgoals will be to prove that the corresponding arguments are equal in the appropriate types.

`MemberI`. Make "one step" of progress on a membership goal t `in` T. What a "step" is depends on the outermost forms of t and T. Generally, the steps are conservative, and if the step that is determined to be applicable fails, no other attempts are made. For some cases, what step to take is obvious. For example, if t is `<`a`,`b`>`, and T is A`#`B, then the step is to apply `EqI` (*i.e.*, to apply the appropriate Nuprl equality-intro rule). In other cases, there are several reasonable alternatives. For example, if the goal is

`>>` f`(`a`)` `in` $\{x$`:`A`|`$B\}$`,`

then `EqI` might succeed. But one can also proceed by analyzing the set-type, getting subgoals

`>>` f`(`a`)` `in` A and `>>` B`[`f`(`a`)/`x`]`.

`MemberI` takes the second alternative (assuming `PolyMember` does not apply). However, if T had not been explicitly a set type, but a term that *computed*, via at least one definition expansion, to a set type, then the first alternative would have been taken. This heuristic has worked well in practice. There are a few other simple heuristics used by `MemberI`. `MemberI` is another one of the tactics that can be updated by the user; the user additions are tried first. Also, as with `PolyMember`, `MemberI` can be applied to binary equalities when the outermost forms of both sides are the same.

`Member`. Reduce the conclusion (without expanding definitions), then repeatedly apply `MemberI` to membership goals where the member is not an induction form. The reason for stopping at induction forms is that it is difficult to guess the types that are necessary to proceed. Proceeding with bad guesses can often lead to false subgoals. The conservative nature of `Member` gives typechecking in Nuprl an interactive character; most of typechecking can be handled automatically, but occasionally the tactic will stop and require the user to make a decision. Fortunately, in practice `Member` is able to automatically prove almost all membership goals that most users would consider trivial.

`Autotactic`. The autotactic cycles through the following, stopping when no further progress can be made.

- Decompose hypotheses that are conjunctions.
- Remove all top level applications of the squash operator in the hypothesis list.

- Apply `Member`.
- Analyze set types in order to expose information for `arith` and then apply it. This fails if `arith` fails.
- Analyze the conclusion (*i.e.*, apply an introduction step) if it is a conjunction or a function-type.
- Apply a member of a user-defined list.

(a second clause dealing with integer arithmetic is omitted). Except for `Member` and the last clause, all the clauses of the autotactic are strongly valid: if a goal is provable, then each subgoal generated by the clause is also provable. This property, the conservative nature of `Member`, and the fact that little backtracking is done, make it feasible to apply the autotactic everywhere desired. For example, users typically call it after almost every top-level application of one of the tactics discussed in Section 9.5.

9.10 An Example From Number Theory

One of the first substantial developments of mathematics carried out in Nuprl was a proof of the fundamental theorem of arithmetic. The complete self-contained library constructed for this theorem contains 113 objects, of which 59 are definitions and 54 are theorems. 36 of the definitions are common to most existing libraries; they are mostly definitions for logical notions. There are 15 definitions dealing with basic list and integer relations and types, and 8 definitions which are particular to the development of the fundamental theorem of arithmetic.

Of the 54 theorems, 20 are associated with definitions in the way described in Section 9.1. Most of the remaining theorems establish simple properties of the defined objects. The fundamental theorem of arithmetic is formalized as the two Nuprl theorems

```
∀n:{2..}.  ∃ l:PrimeFact where eval(l) = n
```

and

```
∀ l1,l2:PrimeFact.  eval(l1) = eval(l2) => l1 = l2 in Fact.
```

The first statement can be read as: *for every integer n greater than or equal to 2 there is a prime factorization that multiplies out (or* evaluates*) to n*. The second can be read as: *any two prime factorizations that evaluate to the same number are the same factorizations.*

The program extracted from the first theorem above is a function mapping numbers to prime factorizations. This program is mostly developed in the proof of the following lemma (the theorem is just an instantiation of the lemma).

```
∀k:N. ∀n,i:{2..} where i≤n & n-i≤k & ∀d:{2..(i-1)}.  ¬(d|n).
    ∃ l:PrimeFact where eval(l) = n
```

This lemma is proved by induction on k and suggests the algorithm used to compute prime factorizations. If we let $P(i, n)$ be

$$i \leq n \ \&\ \forall d{:}\{2..(i\text{-}1)\}.\ \ \neg(d \mid n),$$

$t(i, n)$ be n-i, and $R(n)$ be

$$\exists l{:}\mathtt{PrimeFact}\ \mathtt{where}\ \mathtt{eval}(l) = n,$$

then the lemma is equivalent to (omitting some types)

$$\forall k,n,i.\ \ P(i,n)\ \&\ t(i,n) \leq k \Rightarrow R(n).$$

If $P(i, n)$ and $t(i, n) = 0$ then $n = i$ and n is prime, so the base case of the inductive proof of this lemma follows. For the induction case, assume $P(i, n)$. If $t(i, n) = 0$ we are done, otherwise we compute n' and i' such that $t(i', n') < t(i, n)$ and $P(i', n')$, and use the induction hypothesis. The obvious choices for i' and n' are $i + 1$ and n in the case that i does not divide n, and i and n/i otherwise. In the first case $R(n)$ is proven by using the factorization given by $R(n')$; in the second, i is added to that factorization.

It is possible to express as a theorem in Nuprl a general inductive schema for problems that can be phrased, as in this case, in terms of an "invariant" P, a result assertion R and a bounding function t (although such a schema was not used in our proof of the fundamental theorem of arithmetic). One way of formulating such a schema is the following.

```
∀A,B:U1.  ∀P:(A#B)->U1.  ∀R:B->U1.  ∀t:(A#B)->Int.
  ∀x:A#B. t(x) = 0 & P(x) => R(x.2)
  & ∀x:A#B. P(x) & 0<t(x) =>
      ∃y:A#B. P(y) & 0≤t(y)<t(x) & (R(y.2) => R(x.2))
  & ∀b:B. ∃a:A. 0≤t(<a,b>) & P(<a,b>)
  => ∀b:B. R(b)
```

To apply it to the lemma discussed above, we take A and B to be $\{2..\}$ and P, R and t to be suitable modifications of what was given above.

All of the proofs in the fundamental theorem of arithmetic library were constructed using an early version of the tactic collection that was described in this chapter. This early tactic collection was designed to be generally applicable; none of the tactics were designed to be specifically useful in number theory.

The total time required to complete the library was about forty hours. This includes all work relevant to the effort; in particular, it includes the time spent on entering definitions, on informal planning, on lemma discovery and aborted proof attempts, and on proving all the necessary results dealing with the basic arithmetic operators and relations. The proofs contain a total of 879 refinement steps, most of which were entered manually (some were automatically applied by a primitive analogy tactic).

The fundamental theorem of arithmetic was chosen as a first major experiment with Nuprl because it is widely recognized as a non-trivial theorem in elementary number theory,

because it has interesting computational content, and because mechanical proofs of it have been performed in other systems. It took approximately 8 weeks of effort to prove the theorem in the PLCV system. PLCV is a natural deduction system for reasoning about PL/C programs which has powerful built-in support for arithmetical and propositional reasoning but no tactic mechanism or proof editor. Boyer and Moore also conducted a proof of the fundamental theorem of arithmetic. They do not say how long the effort took, but they do say in their book that a correctness proof of a tautology checker required twelve hours of effort, and that the fundamental theorem of arithmetic was without doubt the hardest theorem proved so far using their system. A comparison with Boyer and Moore's proof is complicated by the fact that they started at a lower level by defining the integers and developing some elementary properties of numbers which are incorporated in Nuprl's `arith` procedure. On the other hand, they also modified their system by adding heuristics for proving theorems in number theory.

9.11 Saddleback Search

in this section we will look at a simple programming problem. The problem, whose solution has been called "saddleback search", is to efficiently find the location of a given integer in a matrix of integers whose rows and columns are sorted in non-decreasing order. The complete solution, including planning, took one afternoon.

The library that was constructed contains nine objects in addition to a set of basic definitions that is shared by most Nuprl libraries. The first of the nine is a simple definition for matrix application:

$$B(i,j) \equiv B(i)(j)$$

This is not precisely as the definition would appear if it were viewed on a Nuprl screen; the parameters here are the italicized identifiers, whereas Nuprl uses angle brackets (*e.g.*, `<B>` instead of B). The second two objects (a theorem and a definition) together give a definition of the proposition that an integer x occurs in an m by n matrix B in a column between i and p and in a row between j and q:

$x \in B\{m,n\}(i:p,\ j:q) \equiv$
$\exists \mathtt{r}:\{0..(m-1)\}.\ \exists \mathtt{s}:\{0..(n-1)\}.$ `where` $B(\mathtt{r},\mathtt{s}) = x$ `&` $i \le \mathtt{r} \le p$ `&` $j \le \mathtt{s} \le q$.

The next two objects in the library define the property of an m by n matrix B being sorted:

`sorted`$(B)\{m,n\} \equiv$
$\forall \mathtt{j}:\{0..(n-1)\}.\ \forall \mathtt{i},\mathtt{p}:\{0..(m-1)\}.$ `i<p` $\Rightarrow B(\mathtt{i},\mathtt{j}) \le B(\mathtt{p},\mathtt{j})$ `&`
$\forall \mathtt{i}:\{0..(m-1)\}.\ \forall \mathtt{j},\mathtt{q}:\{0..(n-1)\}.$ `j<q` $\Rightarrow B(\mathtt{i},\mathtt{j}) \le B(\mathtt{i},\mathtt{q})$.

Following this are two ML objects which contain the definitions of two simple tactics called `RowSorted` and `ColSorted` which are used to apply the fact that a matrix is sorted.

The final two objects in the library are the main theorem and a lemma. The statement of the main theorem is

```
∀m,n:N+.  ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
 ∀x:Int.  x∈B{m,n}(0:(m-1),0:(n-1)) ∨ ¬(x∈B{m,n}(0:(m-1),0:(n-1)))
```

The program extracted from a proof of this theorem is a function that when given inputs of the appropriate type produces an indication of whether or not x occurs in the matrix B together with, in the former case, the pair of integers which give the location of x.

The algorithm implemented by our proof proceeds as follows. Start at the lower left corner of the matrix (*i.e.*, at position $\langle m-1, 0\rangle$). If x is equal to the matrix element b at this position, then we are done. If $x < b$ then since the matrix is sorted x must be smaller than everything in the last row, so we can throw away the last row and repeat the procedure with the smaller matrix. Similarly, if $x > b$, then we can discard the first column and repeat. This algorithm has a simple recursive structure, where a solution for an upper-right block of the matrix can be found in terms of a solution for a smaller block obtained by deleting the bottom row or the leftmost column. This is the motivation for the lemma:

```
∀m,n:N+.  ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
  ∀x:Int.∀k:N. ∀i:{0..(m-1)}.  ∀j:{0..(n-1)} where i+(n-j) = k.
     x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

This can be interpreted as asserting that for every k, we can find whether or not x occurs in any upper-right block of B that has a total of $k-1$ rows and columns.

The proofs that contain the development of the saddleback search algorithm are short enough to permit a complete presentation. The proof of the main lemma is by induction on k, and the first step is shown in Figure 9.1. The tactic used here performs induction on k, using l as a new variable for the induction step. The square brackets around some of the hypotheses in the subgoals indicate that the corresponding hypotheses are *hidden*. A hidden hypothesis cannot be used in any way, and remains hidden through further rule applications until the conclusion has a computationally trivial form (*e.g.*, when it is an equality).

The first subgoal is proved in two steps. The first step, shown in Figure 9.2, is to assert that the hypotheses are contradictory and to expand types which are subsets. The display of this step in Nuprl would show as part of the main goal the hypotheses numbered `1` to `8` shown in Figure 9.1. In the interest of compactness, these have been manually elided (replaced by the string "..."). In addition, the system itself suppresses the display in subgoals of hypotheses that also appear in the goal. The second step, shown in Figure 9.3, is to show that a contradiction can be inferred from the known equalities and inequalities. This is accomplished by using tactics based on the monotonicity rule to add hypothesis `10` to hypothesis `13`, and then to add n to both sides of `13`.

The first step of the proof of the induction step (the second subgoal in Figure 9.1) is shown in Figure 9.4. In this step, we are considering the block of the matrix that

```
* top
>> ∀m,n:N+.  ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
     ∀x:Int.  ∀k:N. ∀i:{0..(m-1)}.  ∀j:{0..(n-1)} where i+(n-j) = k.
          x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (OnVar ‘k‘ (NonNegInd ‘l‘) ...)

1* 1.  m:  N+
   2.  n:  N+
   3.  B: {0..(m-1)}->{0..(n-1)}->Int
   4.  x:  Int
   [5].  sorted(B{m,n})
   6.  i:  {0..(m-1)}
   7.  j:  {0..(n-1)}
   [8].  i+(n-j)=0
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

2* 1.  m:  N+
   2.  n:  N+
   3.  B: {0..(m-1)}->{0..(n-1)}->Int
   4.  x:  Int
   5.  l:  int
   6.  0<l
   7.  ∀i:{0..(m-1)}.  ∀j:{0..(n-1)} where i+(n-j)=l-1.
         x ∈ B{m,n}(0:i, j:(n-1)) ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
   [8].  sorted(B{m,n})
   9.  i:  {0..(m-1)}
   10.  j:  {0..(n-1)}
   [11].  i+(n-j)=l in Int
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

Figure 9.1: By induction.

```
* top 1
...
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (Assert 'False' ...)  THEN (Unsetify ...)

1* 1.  m:  Int
   2.  0<m
   3.  n:  Int
   4.  0<n
   8.  i:  Int
   9.  j:  Int
   11.  0 ≤ j
   12.  j ≤ n-1
   13.  0 ≤ i
   14.  i ≤ m-1
   >> False
```

Figure 9.2: The base case is contradictory.

```
* top 1 1
1.  m:  Int
2.  0<m
3.  n:  Int
4.  0<n
5.  B: {0..(m-1)}->{0..(n-1)}->Int
6.  x:  Int
7.  sorted(B{m,n})
8.  i:  Int
9.  j:  Int
10.  i+(n-j)=0 in Int
11.  0 ≤ j
12.  j ≤ n-1
13.  0 ≤ i
14.  i ≤ m-1
>> False

BY (Mono 10 `+` 12 THEN MonoWithR 13 `+` 'n = n' ...)
```

Figure 9.3: Contradiction follows from known inequalities.

```
* top 2
...
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (Cases ['B(i,j)<x'; 'B(i,j)=x'; 'x<B(i,j)'] ...)

1* 12.  B(i,j)<x
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

2* 12.  B(i,j)=x
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

3* 12.  x<B(i,j)
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

Figure 9.4: The main case split.

```
* top 2 2
...
12.  B(i,j) = x
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (ILeft THENW ITerms ['i';'j'] ...)
```

Figure 9.5: In this case, x has been located.

extends from the top to row i and from column j to the right. The induction hypothesis (hypothesis 7 in the second subgoal in Figure 9.1) asserts the decidability of membership of x in smaller blocks of B. As in the informal description of the algorithm, we do a case analysis on the relation between x and $B(i, j)$. The case where $B(i, j) = x$ is easy; we can justify the left disjunct by providing the coordinates where x occurs. This step is shown in Figure 9.5. For the case where $B(i, j) < x$, we want to discard column j and use the induction hypothesis. There is no point doing this if column j is the last column, however, since in that case x does not occur in the block being considered. Hence we perform the case analysis shown in Figure 9.6.

Consider first the case where $j < n-1$. As shown in Figure 9.7, we apply the induction hypothesis to the smaller block of B obtained by removing column j. Now, x is in the larger block just in case it is in the smaller block (Figure 9.8). The first case is shown in Figure 9.9. If x is in the smaller block, say at position $\langle r, s\rangle$, then it must also be in the larger block at position $\langle r, s\rangle$. The case where x is not in the smaller block is handled in Figure 9.10. We are assuming that x must occur in the larger block, and need to obtain a contradiction. Let r and s be the position of x in the larger block. We need to know whether $\langle r, s\rangle$ is in column j or in the smaller block, so we do the case analysis shown in

```
* top 2 1
...
12.  B(i,j)<x
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (Cases ['j<n-1'; 'j = n-1'] ...)

1* 13.  j<n-1
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

2* 13.  j = n-1
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

Figure 9.6: Is the column to be discarded the last one?

```
* top 2 1 1
...
7.  ∀i:{0..(m-1)}.  ∀j:{0..(n-1)} where i+(n-j)=l-1.
      x ∈ B{m,n}(0:i, j:(n-1)) ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
...
12.  B(i,j)<x
13.  j<n-1
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (InstHyp ['i'; 'j+1'] 7 ...)

1* 14.  x ∈ B{m,n}(0:i, j+1:(n-1)) ∨  ¬(x ∈ B{m,n}(0:i, j+1:(n-1)))
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

Figure 9.7: Try to find x in the smaller block of B obtained by removing column j.

```
* top 2 1 1 1
...
12.  B(i,j)<x
13.  j<n-1
14.  x ϵ B{m,n}(0:i, j+1:(n-1)) ∨  ¬(x ϵ B{m,n}(0:i, j+1:(n-1)))
>> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (OnLastHyp CaseHyp THENL [ILeft;IRight] ...)

1* 14.  x ϵ B{m,n}(0:i, j+1:(n-1))
   >> x ϵ B{m,n}(0:i, j:(n-1))

2* 14.  ¬(x ϵ B{m,n}(0:i, j+1:(n-1)))
   15.  x ϵ B{m,n}(0:i, j:(n-1))
   >> void
```

Figure 9.8: x occurs if and only if it occurs in the smaller block.

```
* top 2 1 1 1 1
...
12.  B(i,j)<x
13.  j<n-1
14.  x ϵ B{m,n}(0:i, j+1:(n-1))
>> x ϵ B{m,n}(0:i, j:(n-1))

BY (OnLastHyp (CHThen (RepeatProductE ``r s``))
    THEN ITerms ['r'; 's'] ...)
```

Figure 9.9: If x occurs in the smaller block, we're done.

the figure. The first case, where $s = j$, is shown in Figure 9.11; a contradiction follows from the fact that column j is sorted (and so $B(r, j) \leq B(i, j)$). In the second case, shown in Figure 9.12, we get a contradiction to hypothesis `14` by showing that x does occur in the smaller block. This finishes the proof of the case where $j < n - 1$.

In the case where $j = n - 1$, we prove (Figure 9.13) that x does not occur by contradiction. If it occurred at position $\langle r, s \rangle$, say, then the fact that column j is column sorted would give us $B(r, j) \leq B(i, j)$, from which a contradiction is immediate.

This completes the proof in the case that $B(i, j) < x$. The case $x < B(i, j)$ is analogous, so it will not be discussed. Except for this analogous branch, the figures we have given cover the entire Nuprl proof of the lemma.

The main theorem follows in two steps from the lemma. The first step, shown in Figure 9.14, is to instantiate the lemma using $m + n - 1$, $m - 1$, and 0 for k, i, and j respectively. The final step (Figure 9.15) is to remove n from the inequality in hypothesis `6`.

For completeness we show in Figure 9.16 one of the two ML objects in the lbirary. This object defines the tactic `ColSorted` which was used in the steps shown in Figures 9.11 and 9.13. This tactic was written to shorten the proof, and its function is to instantiate the property `sorted(B{m,n})` on two positions within a single column. The second ML object defines an analogous tactic `RowSorted`.

```
* top 2 1 1 1 2
...
12.  B(i,j)<x
13.  j<n-1
14.  ¬(x ∈ B{m,n}(0:i, j+1:(n-1)))
15.  x ∈ B{m,n}(0:i, j:(n-1))
>> void

BY (OnLastHyp (CHThen (RepeatProductE ``r s``))
    THEN Cases ['s = j'; 'j<s'] ...)

1* 15.  r:  {0..(m-1)}
   16.  s:  {0..(n-1)}
   17.  s = j
   18.  j ≤ s
   19.  s ≤ n-1
   20.  B(r,s)  =  x
   21.  0 ≤ r
   22.  r ≤ i
   >> void

2* 15.  r:  {0..(m-1)}
   16.  s:  {0..(n-1)}
   17.  j<s
   18.  j ≤ s
   19.  s ≤ n-1
   20.  B(r,s)  =  x
   21.  0 ≤ r
   22.  r ≤ i
   >> void
```

Figure 9.10: x does not occur in the smaller block. Suppose it occurs at $\langle r, s\rangle$ the larger block. We need to know whether $\langle r, s\rangle$ is in column j.

```
* top 2 1 1 1 2 1
...
8.  sorted(B{m,n})
...
12.  B(i,j)<x
13.  j<n-1
14.  ¬(x ϵ B{m,n}(0:i, j+1:(n-1)))
15.  r:  {0..(m-1)}
16.  s:  {0..(n-1)}
17.  s = j
18.  j ≤ s
19.  s ≤ n-1
20.  B(r,s)  =  x
21.  0 ≤ r
22.  r ≤ i
>> void

BY (ColSorted 'j' 'r' 'i' 8 ...)
```

Figure 9.11: The contradiction follows from the fact that column j is sorted.

```
* top 2 1 1 1 2 2
...
12.  B(i,j)<x
13.  j<n-1
14.  ¬(x ϵ B{m,n}(0:i, j+1:(n-1)))
15.  r:  {0..(m-1)}
16.  s:  {0..(n-1)}
17.  j<s
18.  j ≤ s
19.  s ≤ n-1
20.  B(r,s)  =  x
21.  0 ≤ r
22.  r ≤ i
>> void

BY (E 14 THENO ITerms ['r';'s'] ...)
```

Figure 9.12: $\langle r, s\rangle$ must be in the smaller block, contradicting 14.

```
* top 2 1 2
...
8.  sorted(B{m,n})
...
12.  B(i,j)<x
13.  j = n-1
>> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (IRight ...)
   THEN (OnLastHyp (CHThen (RepeatProductE ``r s``)) ...)
   THEN (ColSorted 'j' 'r' 'i' 8 ...)
```

Figure 9.13: When j is the last column of B, x does not occur, since if it did, the fact that column j is sorted would be contradicted.

```
* top
>> ∀m,n:N+.  ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).  ∀x:Int.
      x ϵ B{m,n}(0:(m-1), 0:(n-1))  ∨  ¬(x ϵ B{m,n}(0:(m-1), 0:(n-1)))

BY (Id ...)  THEN
   (LemmaUsing `saddleback_lemma` ['m+n-1'; 'm-1'; '0'] ...)

1* 1.  m:  N+
   2.  n:  N+
   3.  B: {0..(m-1)}->{0..(n-1)}->Int
   4.  sorted(B{m,n})
   5.  x:  Int
   6.  m+n-1<0
   >> void
```

Figure 9.14: Use the main lemma, with $m+n-1$ for the bound, $m-1$ for starting row i, and 0 for the starting column j.

```
* top 1
1.  m:  N+
2.  n:  N+
3.  B: {0..(m-1)}->{0..(n-1)}->Int
4.  sorted(B{m,n})
5.  x:  Int
6.  m+n-1<0
>> void

BY (MonoWithR 6 `-` '0<n' ...)
```

Figure 9.15: Subtract `0<n` from hypothesis 6, getting `m-1<0`, contradicting `m>0`.

```
let ColSorted col row1 row2 i p =
  let n = number_of_hyps p  in
  ( CH i
    THEN (E i)
    THEN InstantiateHyp [col; row1; row2] (n+1)
    THEN Thinning [n+1; n+2]
  ) p
;;
```

Figure 9.16: One of two simple tactics.

Appendix A

Basic ML Constructors and Destructors

In what follows, all universe levels must be strictly positive. Unless otherwise stated the tokens `‘nil‘` and `‘NIL‘` may be used as identifiers to indicate that no identifier should label the new hypothesis.

A.1 Rule Constructors

The constructors are categorized in the same manner as the rules. Each constructor is presented in the form `constructor_name(parameter names): type` where `type` is the type of the constructor. The parameter names are given only to indicate the correspondence between the constructor arguments and the components of the rule. Following are some conventions regarding the parameter names we use.

- `hyp:int` is used to indicate a hypothesis.
- `where:int` is used to indicate which equand of an equality should be reduced in a computation rule.
- `level:int` is used to give a universe level.
- `using:term` is used to indicate a using term.
- `over_id:tok, over:term` are used to indicate an over term. If the `over_id` is the token `‘nil‘`, the over term is ignored.

All parameters corresponding to new identifiers have the same names as the new names given in the corresponding rule, and should be ML tokens. To specify the default identifier for an optional new identifier, give the token `‘nil‘`.

Unlike the rules, each constructor needs a unique name. The general form of the names is *base-type_kind_qualifier*, where *base-type* is a prl type and *kind* is one of `intro`,

elim, equality, formation or computation. So, for example, list_equality_nil is the name of the constructor for the rule which specifies how to show two nil terms are the same in a list type, and int_elim is the name of the constructor for the elim rule for integers. There are some exceptions to this general pattern, the main one being that the name of the constructor for the rule for showing two canonical type terms to be equal in some universe is *type*_equality rather than universe_equality_*type*.

Atom

formation

```
universe_intro_atom:  rule.
atom_equality:  rule.
```

canonical

```
atom_intro(token):  term -> rule.
atom_equality_token:  rule.
```

equality

```
atom_eq_equality:  rule.
```

computation

```
atom_eq_computation(where):  int -> rule.
```

Void

formation

```
universe_intro_void:  rule.
void_equality:  rule.
```

noncanonical

```
void_elim(hyp):  int -> rule.
void_equality_any:  rule.
```

Int

formation

`universe_intro_integer: rule.`

`int_equality: rule.`

canonical

`integer_intro_integer(c): int -> rule.`

`integer_equality_natural_number: rule.`

noncanonical

`integer_intro_addition: rule.`

`integer_intro_subtraction: rule.`

`integer_intro_multiplication: rule.`

`integer_intro_division: rule.`

`integer_intro_modulo: rule.`

`integer_equality_minus: rule.`

`integer_equality_addition: rule.`

`integer_equality_subtraction: rule.`

`integer_equality_multiplication: rule.`

`integer_equality_division: rule.`

`integer_equality_modulo: rule.`

`integer_elim(hyp y z): int -> tok -> tok -> rule.`

`integer_equality_induction(over_id over u v): tok -> term -> tok -> tok -> rule.`

equality

`int_eq_equality: rule.`

`int_less_equality: rule.`

computation

`integer_computation(kind where): tok -> int -> rule.` where kind should be one of ‘UP‘, ‘BASE‘ or ‘DOWN‘.

`int_eq_computation(where): int -> rule.`

`int_less_computation(where): int -> rule.`

Less

formation

```
universe_intro_less: rule.
less_equality: rule.
```

equality

```
axiom_equality_less: rule.
```

List

formation

```
universe_intro_list: rule.
list_equality(level): int -> rule.
```

canonical

```
list_intro_nil(level): int -> rule.
list_equality_nil(level): int -> rule.
list_intro_cons: rule.
list_equality_cons: rule.
```

noncanonical

```
list_elim (hyp u v w): int -> tok -> tok -> tok -> rule.
list_equality_induction (over_id over using u v w): tok -> term -> term ->
tok -> tok -> tok -> rule.
```

computation

```
list_computation(where): int -> rule.
```

Union

formation

```
universe_intro_union: rule.
union_equality: rule.
```

canonical

```
union_intro_left(level):  int -> rule.
union_equality_inl(level):  int -> rule.
union_intro_right(level):  int -> rule.
union_equality_inr(level):  int -> rule.
```

noncanonical

```
union_elim(hyp x y):  int -> tok -> tok -> rule.
union_equality_decide(over_id over using u v):  tok -> term -> term -> tok
-> tok -> rule.
```

computation

```
union_computation(where):  int -> rule.
```

Function

formation

```
universe_intro_function(x term):  tok -> term -> rule.
function_equality(y):  tok -> rule.
universe_intro_function_independent:  rule.
function_equality_independent:  rule.
```

canonical

```
function_intro(level y):  int -> tok -> rule.
function_equality_lambda(level z):  int -> tok -> rule.
```

noncanonical

```
function_elim(hyp on y):  int -> term -> tok -> rule.
function_elim_independent(hyp y):  int -> tok -> rule.
function_equality_apply(using):  term -> rule.
```

equality

```
extensionality(y):  tok -> rule.
```

computation

`function_computation(where): int -> rule.`

Product

formation

`universe_intro_product(x left_term): tok -> term -> rule.`
`product_equality(y): tok -> rule.`
`universe_intro_product_independent: rule.`
`product_equality_independent: rule.`

canonical

`product_intro_dependent(left_term level y): term -> int -> tok -> rule.`
`product_equality_pair(level y): int -> tok -> rule.`
`product_intro: rule.`
`product_equality_pair_independent: rule.`

noncanonical

`product_elim(hyp u v): int -> tok -> tok -> rule.`
`product_equality_spread(over_id over using u v): tok -> term -> term -> tok -> tok -> rule.`

computation

`product_computation(where): int -> rule.`

Quotient

formation

`universe_intro_quotient(A E x y z): term -> term -> tok -> tok -> tok -> rule.`
`quotient_formation(x y z): tok -> tok -> tok -> rule.`

canonical

`quotient_intro(level): int -> rule.`
`quotient_equality_element(level): int -> rule.`

noncanonical

```
quotient_elim (hyp level v w):  int -> int -> tok -> tok -> rule.
```

equality

```
quotient_equality(r s):  tok -> tok ->  rule.
quotient_equality_element(level):  int -> rule.
```

Set

formation

```
universe_intro_set(x type):  tok -> term -> rule.
set_formation(y):  tok -> rule.
universe_intro_set_independent:  rule.
set_formation_independent:  rule.
```

canonical

```
set_intro(level left_term y):  int -> term -> tok -> rule.
set_equality_element(level y):  int -> tok -> rule.
set_intro_independent:  rule.
set_equality_element:  rule.
```

noncanonical

```
set_elim(hyp level y):  int -> int -> tok -> rule.
```

equality

```
set_equality(z):  tok -> rule.
```

Equality

formation

```
universe_intro_equality(type number_terms):  term -> int -> rule.
equal_equality:  rule.
```

canonical

`axiom_equality_equal: rule.`
`equal_equality_variable: rule.`

Universe

canonical

`universe_intro_universe (level): int -> rule.`
`universe_equality: rule.`

Tactic

`tactic(text): tok -> rule.`

Miscellaneous

hypothesis

`hyp(hyp): int -> rule.`

thinning

`thinning(hyp_num_list): int list →rule.`

sequence

`seq(term_list id_list): (term list) -> (tok list) -> rule.` The length of the token list should match the length of the term list.

lemma

`lemma(lemma_name x): tok -> tok -> rule.`

def

`def(theorem x): tok -> tok -> rule.`

explicit intro

`explicit_intro(term): term -> rule.`

cumulativity

```
cumulativity(level):  int -> rule.
```

substitution

```
substitution(level equality_term over_id over):  int -> term -> tok -> term
-> rule.
```

direct computation

```
direct_computation(tagged_term):  term -> rule.
direct_computation_hyp(hyp tagged_term):  int -> term -> rule.
reverse_direct_computation(tagged_term):  term -> rule.
reverse_direct_computation_hyp(hyp tagged_term):  int -> term -> rule.
```

equality

```
equality:  rule.
```

arith

```
arith(level):  int -> rule.
```

monotonicity

```
monotonicity(op hyp1 hyp2):  tok -> int -> int -> rule.
```

division

```
division:  rule.
```

A.2 Term Destructors

Below are the term destructors and their types.
`destruct_universe: term -> int.` The integer is the universe level.
`destruct_any: term -> term.`
`destruct_token: term -> tok.`
`destruct_natural_number: term -> int.`
`destruct_minus: term -> term.`
`destruct_addition: term -> (term # term).` The results are the left and right terms.

`destruct_subtraction: term -> (term # term).`

`destruct_multiplication: term -> (term # term).`

`destruct_division: term -> (term # term).`

`destruct_modulo: term -> (term # term).`

`destruct_integer_induction: term -> (term # term # term # term).` The results are the value, down term, base term and up term.

`destruct_list: term -> term.` Returns the type of the list.

`destruct_cons: term -> (term # term).` Returns the head and tail.

`destruct_list_induction: term -> (term # term # term).` Returns the value, base term and up term.

`destruct_union: term -> (term # term).` Results in the left and right types.

`destruct_inl: term -> term.`

`destruct_inr: term -> term.`

`destruct_decide: term -> (term # term # term).` The result is the value, the left term and the right term.

`destruct_product: term -> (tok # term # term).` The result is the bound identifier, the left term and the right term.

`destruct_pair: term -> (term # term).` Returns the left term and the right term.

`destruct_spread: term -> (term # term).` Returns the value and the term.

`destruct_function: term -> (tok # (term # term)).` The result is the bound identifier, the left term and the right term.

`destruct_lambda: term -> (tok # term).` Returns the bound identifier and the term.

`destruct_apply: term -> (term # term).` Returns the function and the argument.

`destruct_quotient: term -> (tok # (tok # (term # term))).` Returns the two bound identifiers, the left type and the right type.

`destruct_set: term -> (tok # (term # term)).` Returns the bound identifier, the left type and the right type.

`destruct_var: term -> tok.`

`destruct_equal: term -> ((term list) # term).` Returns a list of the equal terms and the type of the equality.

`destruct_less: term -> (term # term)).` Returns the left term and the right term.

`destruct_bound_id: term -> ((tok list) # term).` Returns a list of the bound identifiers and the type.

`destruct_term_of_theorem: term -> tok.` Returns the name of the theorem.

`destruct_atomeq: term -> (term # (term # (term # term))).` Returns the left term, right term, if term and else term.

`destruct_inteq: term -> (term # (term # (term # term))).` Returns the left term, right term, if term and else term.

`destruct_intless: term -> (term # (term # (term # term)))`. Returns the left term, right term, if term and else term.

`destruct_tagged: term -> (int # term)`. The integer returned is zero if the tag is `*`.

A.3 Term Constructors

For each term type there is a corresponding constructor that is the curried inverse to the destructor given above. For example,

```
make_inteq_term: term -> term -> term -> term -> term.
```

Appendix B

"Experimental" Types

Nuprl has been extended with two new type constructors. The first is the type of all canonical objects. This type is not used in an essential way in any existing Nuprl libraries. It is used only to support the "polymorphic" style of definition described in the tactic documentation. It is a trivial matter to incorporate this type into the standard semantic account of Nuprl.

The second type constructor is for recursive types. This should be regarded as experimental since it has not been proven sound. It is highly likely to be sound since it is a slight variant of one which has been rigorously verified.

B.1 The "Object" Type

formation

>> object in Ui BY intro

canonical

>> t in object BY intro [using A]
   [>> t in A]

>> t=t' in object BY intro
   >> t in object
   >> t' in object

where, in the second rule above, if t is canonical then no subgoal is generated, and if it is not canonical, then the "using" term must be present and the subgoal is generated. There are no corresponding rules for n-ary equalities where $n > 2$.

The related ML primitives are as follows.

```
object_equality:  rule.
object_equality_member(t:term):  rule.
```

```
make_object_term:  term.
```

B.2 Simple Recursive Types

The general form of the recursive type constructor presented in the book can be somewhat unwieldy in practice. A simpler form has been implemented which does not allow mutual recursion or recursive definition of type-valued functions.

The new terms are `rec(Z.T)` and `rec_ind(r;h,z.t)`, for `h`, `z`, `Z` variables and `r`, `t`, `T` terms. When closed, `rec(Z.T)` is canonical and `rec_ind(r;h,z.t)` is noncanonical. The term `rec_ind(r;h,z.t)` is a redex with contracta

```
t[r/z; \z.  rec_ind(z;h,z.t)/h].
```

In the book there is a restriction that r must be canonical for the term to be a redex — the implemented version has no such restriction, hence this `rec_ind` form has no principal argument. The direct computation rules do, however, treat `rec_ind` redices as redices.

The proof rules and ML primitives follow.

Formation

```
H >> rec(Z.T) in Ui by intro new Y
Y:Ui >> T[Y/Z] in Ui
```

where no instance of Z bound by rec(Z.T) may occur in the domain type of a function space, in the argument of a function application or in the principal argument(s) of a non-application elimination form.

Canonical

```
H >> rec(Z.T) ext t by intro at Ui
     >> T[rec(Z.T)/Z] ext t
     >> rec(Z.T) in Ui

H >> t in rec(Z.T) by intro at Ui
    >> t in T[rec(Z.T)/Z]
    >> rec(Z.T) in Ui
```

Noncanonical

```
H, x:rec(Z.T), H' >> G ext g[x/y]  by unroll x new y
y:T[rec(Z.T)/Z], y=x in T[rec(Z.T)/Z] >> G[y/x] ext g

H, x:rec(Z.T), H' >> G ext rec\_ind(x;w,z.g[$\backslash$x.void/u])
```

```
                                  by elim x at Ui new u,v,w,z
                     >> rec(Z.T) in Ui
u: rec(Z.T)->Ui, w: (x:{v:rec(Z.T)|u(v)}->G), z: T[{v:rec(Z.T)|u(v)}/Z]
                     >> G[z/x] ext g

H >> rec\_ind(r;h,z.t) in S[r/x]
             by intro over x.S using rec(Z.T) at Ui new u,v,w
       >> r in rec(Z.T)
       >> rec(Z.T) in Ui
u: rec(Z.T)->Ui, v: w:{w:rec(Z.T)|u(w)}->S[w/x], w: T[{w:rec(Z.T)|u(w)}/Z]
       >> t[v,w/h,z] in S[w/x]
```

ML Primitives

The primitives are presented as in Appendix A

```
simple_rec_equality(y):  tok -> rule.

simple_rec_intro(level):  int -> rule.

simple_rec_equality_element(level):  int -> rule.

simple_rec_unroll_elim(level new_ids):  int -> tok list -> rule.

simple_rec_elim(hyp_num level new_ids):  int -> int -> tok list -> rule.

simple_rec_equality_rec_ind(level over_id over using u v w):  int -> tok ->
term -> term -> tok list -> rule.

destruct_simple_rec:  term -> (tok # term).

make_simple_rec_term:  tok -> term -> term.

destruct_simple_rec_ind:  term -> (term # term).

make_simple_rec_ind_term:  term -> term -> term.
```